Algebraic Methods in Language Processing

Proceedings of the Tenth
Twente Workshop on Language Technology

joint with

First AMAST Workshop on Language Processing

A. Nijholt, G. Scollo & R. Steetskamp (eds.)
CIP GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Nijholt, A., Scollo, G., Steetskamp, R.

Algebraic Methods in Language Processing:
Proceedings Twente Workshop on Language Technology 10, joint with 1st AMAST Workshop on Language Processing / A. Nijholt, G. Scollo, R. Steetskamp
Enschede, Universiteit Twente, Faculteit Informatica

ISSN: 0929-0672

trefw.: algebra, categorial grammar, language processing, language technology

© Copyright 1995; Universiteit Twente, Enschede

Book orders:
University of Twente
Ch. Bijron
Dept. of Computer Science
P.O. Box 217
NL 7500 AE Enschede
fax: +31-53-4893503
Email: bijron@cs.utwente.nl

Druk- en bindwerk: Reprografie U.T. Service Centrum, Enschede
PREFACE

TWLT is an acronym of Twente Workshop(s) on Language Technology. These workshops on natural language theory and technology are organised by Project Parlevink (sometimes with the help of others), a language theory and technology project conducted at the Department of Computer Science of the University of Twente, Enschede, The Netherlands. Each workshop has proceedings containing the presented. For the contents of the proceedings of previous TWLT editions, please consult the last pages of this volume.

Previous TWLT workshops.
TWLT1, Tomita’s Algorithm: Extensions and Applications. 22 March 22, 1991.
TWLT4, Pragmatics in Language Technology. 23 September, 1992.
TWLT5, Natural Language Interfaces. 3 and 4 June, 1993.
TWLT6, Natural Language Parsing, 16 and 17 December, 1993.
TWLT7, Computer Assisted Language Learning, 16 and 17 June 1994.
TWLT8, Speech and Language Engineering, 1 and 2 December 1994.
TWLT9, Corpus-based approaches to Dialogue Modelling, 9 June 1995.

This joint workshop is organized in the framework provided by the Algebraic Methodology and Software Technology (AMAST) movement. In this framework four large international conferences have been held and, until now, three workshops (one on Topology and Completion in Semantics, the other two on Real-Time Systems. The program of this first AMAST workshop on language processing has been set up by a Program Committee consisting of A. Nijholt, M. Nivat and T. Rus.

This workshop focussed on algebraic methods in formal languages, programming languages and natural languages. The aim of this workshop was to bring together researchers on formal language theory, programming language theory and natural language description theory, who have a common interest in the use of algebraic methods to describe syntactic, semantic and pragmatic properties of language. The workshop did not concentrate on natural language only. There is interesting use of algebraic methods in programming language processing (compiler construction and development of programming language environments) and (obviously) in formal language theory. Moreover, it is becoming clear that some of the methods developed in these fields can play a role in natural language description and processing.

The workshop took place in the “Logica” complex at the campus of the University of Twente. Just as with the previous workshop programs, there were presentations by a select group of internationally known scientists and other researchers. The general aim was to offer a platform for the presentation of new developments and for the exchange of ideas between people from the various disciplines that play a role.

A workshop is the concerted action of many people. It goes without saying that we are grateful to the authors and the organisations they represent, for their efforts and contributions. But in addition we would like to mention here the people whose work has been less visible during the workshop proper, but whose contribution was evidently of crucial importance. Charlotte Bijron, Alice Hoogvliet-Haverkate and Astrid Henraat took care of the administrative tasks. Finally, we wish to thank the participants for joining the workshop and for contributing to the discussions.

TWLT11, the next workshop in the series, will take place on 19-21 June 1996. Its topic will be Dialogue Management in Natural Language Systems. We hope it will match the success of this and the previous workshops.

December, 1995

Anton Nijholt, Giuseppe Scollo and René Steetskamp
CONTENTS

Algebraic Processing of Programming Languages
Teodor Rus (Iowa City, USA) 1

Polymorphic Syntax Definition
Elco Visser (Amsterdam, NL) 43

Algebraic Specification of Documents
J.C. Ramalho, J.J. Almeida & P.R. Henriques (Braga, Portugal) 55

Multi-layered Pipeline Parsing from Multi-axiom Grammars
Teodor Rus & James S. Jones (Iowa City, USA) 65

Parsing Schemata and Correctness of Parsing Algorithms
Klaas Sikkel (Sankt Augustin, D) 83

A Generic Tabular Scheme for Parsing
François Barthélémy (Paris, F) 99

Parsing Algebraic Power Series Using Dynamic Programming
Frederic Tendeau (INRIA, Rocquencourt, F) 107

Multimodal Linguistic Inference
Michael Moortgat (Utrecht, NL) 113

Computational Minimalism: The Convergence of the Minimalist Syntactic Program and Categorial Grammar
R.C. Berwick (MIT, USA) 143

A Simple Uniform Semantics for Concatenation-Based Grammar
Annius V. Groenink (Amsterdam, NL) 149

Theory of Texts
Grzegorz Rozenberg (Leiden, NL) 161

A Graph Grammar Approach to Graphical Parsing
Jan Rekers (Leiden, NL) & A. Schürr (Aachen, D) 163

Strong Interchangeability and Nonlinearity of Primitive Words
Sándor Horvath (Debrecen, H) 173

Algebraic Methods in Categorial Grammar
Wojciech Buszkowski (Poznan, P) 179

A Variant of a Universal Metagrammar of Conceptual Structures. Algebraic Systems of Conceptual Syntax
Vladimir A. Fomichev (Moscow, Russia) 195

The Method of Rosetta, Natural Language Translation Using Algebras
Theo M.V. Janssen (Amsterdam, NL) 211

Contextual Grammars with Depth-First Derivation
C.M. Martin-Vide, J. Miquel-Verges & Gh. Paun (Tarragona, E) 225

Subword Membership Problem for Linear Indexed Languages
Pál Dömősi (Kossuth University, Debrecen, Hungary) & Jürgen Duske (University of Hannover, Germany) 235

Algebraic Methods for Anaphora Resolution
C. Rico Perez & J.S. Granda (Madrid, E) 239

A Logical Formalism for Intergrammatical Representation
Vincenzo Manca (Pisa, I) 247
Sponsors and Support

We gratefully acknowledge help from:

University of Twente, Enschede

Organization:

Parlevink Project

AMAST
Algebraic Processing of Programming Languages

Teodor Rus
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
rus@cs.uiowa.edu

Abstract

Current methodology for compiler construction evolved from the need to release programmers from the burden of writing machine-language programs. This methodology does not assume a formal concept of a programming language and is not based on mathematical algorithms that model the behavior of a compiler. The side effect is that compiler implementation is a difficult task and the correctness of a compiler usually is not proven mathematically. Moreover, a compiler may be based on assumptions about its source and target languages that are not necessarily acceptable for another compiler that has the same source and target languages. The consequence is that programs are not portable between platforms of machines and between generations of languages. In addition, while a conventional compiler freezes the notation that programmers can use to develop their programs the problem domain evolves and requires extensions that are not supported by the compiler. These problems are addressed by two directions of research in the current language processing technology. One direction enriches the programming environment provided by the conventional compilers with tools that optimize programs according to the architecture of the target machines. The other research direction focuses on a new methodology for programming language design and implementation that accommodates the existing programming tools and at the same time allows programmers to manufacture their own languages and compilers adapted to their own machines and problem domains.

The first research direction further complicates the compiler, which is already too complicated, and does not provide for language extensibility with the problem domain. The second research direction is based on mathematical concepts of programming language and programming language translation that are independent of the computer and computer user and could easily be mastered by programmers.

The research reported in this paper fits within this framework. We first introduce the formal concept of a programming language which captures all three components of a language, the semantics, the syntax, and their association into a communication tool, as mathematical constructs of universal algebra. This concept reveals the compositional structure of programming languages and allows us to look for a natural decomposition of programming languages into simpler objects such as language lexicon, language types, and language constructs. Each language component is however mathematically specified and implemented and the specifications and implementations of these language components are further mathematically integrated into the specification and implementation of the programming language they define. We illustrate this approach of language processing with algebraic compiler generation using as components language independent lexical analyzers, parallel pattern matching parsers, language independent type systems, machine independent target code generation, and parallel program development by the compiler under the direction of the programmer.
1 Introduction

Current technology for compiler construction evolved from the need to release programmers from the burden of writing machine-language programs. That is, a conventional compiler allows programmers to develop their programs using a notation close to their natural languages while the computer is designed to execute programs written in a binary machine language. This technology does not assume a formal concept of a programming language and is not based on a mathematical algorithm that models a compiler. Rather, it consists of a series of well understood transformations performed on the notation used by programmers to express their algorithms, mapping it into the binary notation that represents machine language programs. The lack of a mathematical model for the compiler makes it difficult to integrate these transformations into a software artifact that can be proven correct and is understandable by its user, the programmer. A side effect is that compiler implementation is a difficult task and the correctness of commercial compilers is usually not proven mathematically. Moreover, a conventional compiler may be based on assumptions about its source and target languages that are not necessarily acceptable for another compiler that has the same source and target languages. The consequence is that programs are not portable between platforms of machines and between generations of languages. In addition, while a compiler freezes the notation that programmers can use to develop their programs the problem domain evolves and requires extensions that are not supported by the compiler. The programmer's choice is to use old and sometimes inappropriate tools to solve new problems. Consequently the computation power of the new machines cannot actually be used and the productivity of problem solving is affected. The best example illustrating this situation is provided by the need to exploit parallel machines using sequential languages. This state of the art leads to the paradoxical situation where the compiler developed as a tool to make programming easier becomes a burden for the programming activity.

There are two research directions that address the situation created by the historical evolution of conventional compilers. One of these directions advocates enriching programming environment provided by the conventional compiler with tools that optimize programs according to the architecture of the target machine [BCG+95]. The other direction [Kli95] focuses on the development of a new methodology for language design and implementation that while accommodating the existing programming tools would allow programmers to manufacture their own languages and compilers adapted to their own machines and problem domains. The solution provided by the first research direction further complicates the compiler which is already very complicated. In addition, program optimization tools created within this framework are associated with given languages or family of languages, such as Fortran, and consequently do not provide a solution for language extensibility with the problem domain. The solution provided by the second research direction becomes feasible if the new methodology for language design and implementation is based on mathematical concepts of programming language and programming language translation that are independent of the computer and computer user and could be easy mastered by programmers. Our research fits in this second framework and starts with the observation that programming languages are complex objects, whose compositional structure is hidden by the conventional methodology for language processing. Hence, the first task towards the development of mathematical concepts of programming language and compiler is to find natural decompositions of programming languages into simpler objects that can be mathematically specified and implemented and their specifications and implementations can be further mathematically integrated into the specification and implementation of a specific programming language.

There is already a rather long history of using the universal algebra as a framework for language specification [EM85] and some successful experiments on compiler modeling by algorithms for homomorphism computation [MP67, BL69, Rus83, Rus91]. Therefore, we are seeking the new methodology for language processing in the framework of universal algebras.

The difficulty in using the algebraic methodology for the development of a new technology for language processing resides in the way programming languages evolved as "notations with which people can communicate algorithms to computers and to one another" [AU77] and the manner in which algebraic mechanism have been used to specify such a notation. In other words, while the concept of a programming language evolved as a notation of well understood but unspecified computations the compiler design is based on formal specifications of both the computations expressed by a programming language and the notation used to express these computations. The notation part of a programming language,
henceforth called the *syntax*, was precisely described within the framework of formal languages [HU69]. The computations expressed by the syntax, henceforth called the *semantics*, have been formalized using the domain theory [Sco76, GS90], which is a different framework. These formalizations of syntax and semantics of a programming language allowed us to understand the compilation process and to develop current technology for programming language processing but did not mature into a formal concept of a programming language and into a mathematical model of a compiler. The cause of this may be the lack of a mathematical framework naturally integrating the syntax, the semantics, and the compilation process.

More recently the universal algebra is used as a specification mechanism that integrates semantics and syntax [Wir90] but the compilation process is still controlled by automata capable to recognize the syntax of programming languages. However, we have shown [Rus91] that universal algebra can be used as a framework that integrates programming languages and their processors. This paper builds further on the development of a new methodology for the design of programming languages and their processors that requires no programming activity in the usual sense.

Section 2 of this paper discusses the nature of computations expressed by programming language notations and deduce a formal concept of a machine independent programming language. Using the mathematical concept of a programming language we introduce the algebraic concept of programming language translation as a relationship that ensures the consistency of communication between communicators speaking different programming languages. The semantics and the syntax of a programming language are algebra specified by the same signature of a class of algebras. Hence, the mechanism that bind a semantics to a syntax making up a programming language is provided by various homomorphic mappings between the semantics and the syntax algebras of the language. Further, the language translation is introduced as an embedding of the source language algebras into the target language algebras that preserves the source language algebras as subalgebras of the target language algebras. The implementation mechanism suggested by this framework is the algorithm computing the homomorphisms specified by functions defined on the generators provided by the signature of the source language class of algebras rather than using the automaton generated by this signature. The algebraic model of a programming language suggests the natural decomposition of programming languages into simpler objects such as language lexicon, that corresponds to the notation specification when one studies an algebra, type system, which corresponds to the signature and equations specifying a class of algebras, language constructs, which corresponds to the word algebra of a class of algebras. Section 3 of the paper presents the algebraic specification of a language independent lexicon and the scanner generator from this specification. Section 4 of the paper shows the algebraic mechanism of a language independent type system. Section 5 of the paper discusses the algebraic mechanism that implements a source language into a target language. Finally, Section 6 of the paper is dedicated to the algebraic mechanism that integrates a scanner, a type system, and an algorithm embedding source language constructs into equivalent target language constructs, into an algebraic compiler. The theory we present here is illustrated in the appendices with the partial specification and implementation of a Fortran-90 to C compiler.

2 A FORMAL CONCEPT OF PROGRAMMING LANGUAGE

We start by observing that computations handled by programming languages exist independently of the notation employed to express them. The requirement to formally specify such computations arose with the need to formally prove that a compiler preserves the computations expressed by high-level languages while mapping them into the machine language programs. The mechanism of formalization suggested by the conventional compiler is the operational expression of the high-level language notations as computations performed by the machine destined to perform them. Hence, the data are defined over a universe of memory locations and operations are those performed by the target machine. The mathematical construction usually involved in this formalization does not establish a formal relationship between machine independent computations and machine computations. Rather, computations handled by programming languages are regarded as the computations performed by the machine. This leads to restrictions of machine independent software tool development and thus motivates us to try to formalize computations handled by programming languages as machine independent computations. The general framework for formalization of machine independent computations is provided by the universal algebra
The effective algebras [SHT95] provide both the framework for studying the computability of the machine independent computations and the proof mechanism required by such a formalization.

2.1 Machine Independent Computation

The general schemes used for specifying machine independent computations makes use of three concepts:

1. The environment, $Env$, which consists of a given collection of predefined data types, such as integer, real, boolean, etc., and a given collection of type constructors such as array, function, record, file, etc., that can be used to extend the predefined and defined data types with user defined data types. Note, channels of a reactive application, such as a real-time system, are seen here as data types.

2. The state, $\sigma$, that consists of an assignment of values from $Env$ to elements of a given collection of variables and constants, $V$, of types given in the environment $Env$, that is, $\sigma : V \rightarrow Env$. The state $\sigma : V \rightarrow Env$ characterizes the problem solved by the computation.

3. The state transition that can be expressed by a transformation $\tau : \sigma \rightarrow \sigma'$, such that if $\sigma : V \rightarrow Env$ then $\tau(\sigma) : V' \rightarrow Env'$ where $Env'$ is a new environment, that may coincide with $Env$, and $V'$ is a new set of objects, that may coincide with $V$. A transition $\tau : \sigma \rightarrow \sigma'$ such that $\sigma : V \rightarrow Env$ and $\sigma' : V \rightarrow Env'$ is called internal or silent; a transition $\tau : \sigma \rightarrow \sigma'$ such that $\sigma : V \rightarrow Env$ and $\sigma' : V' \rightarrow Env'$ where $V' \neq V$ or $Env' \neq Env$ is called external or observable. The state transition characterizes the computation (i.e., the algorithm) that solves the problem.

Note that we use the concept of an environment with a different meaning than is used in the conventional compiler design. In both cases the environment provides a mechanism for specifying the universe of discourse of the language. Here we deal with machine independent programming languages and therefore the environment coincides with the collection of machine independent computing abstractions that can be handled through the programming language. A conventional programming language provides high-level notations for machine computations and therefore the environment is the machine memory on which these computations are stored.

Using the concepts of environment, state, and state transition we can define a transition system as a tuple $T = (\Sigma, \Theta)$ where $\Sigma$ is a set of states and $\Theta$ is a set of transitions such that $\forall \tau \in \Theta(\tau : \Sigma \rightarrow 2^\Sigma)$. A computation $C$ over the transition system $T$ is defined as a sequence of states $\sigma_0, \sigma_1, \ldots$ from $\Sigma$ such that

1. $\sigma_0$ satisfies the initial conditions of the solution algorithm of the problem solved by this computation and is called the initial state of the computation.

2. For each $i > 0$, there is a transition $\tau_i \in \Theta$ such that $\sigma_i \in \tau_i(\sigma_{i-1})$.

3. The computation is either transformational, and then there is a $j > 0$ such that for each $k > j$, $\sigma_k = \sigma_j$, or the computation is reactive, and then for each $i$, $\sigma_i \neq \sigma_{i+1}$.

This concept of a computation is still different from the computations expressed by conventional programming languages. There is no implicit or explicit agent to perform a computation implied in the above definition of a computation. However, computations expressed by conventional programming languages are always carried out by a concrete or abstract machine implied in the language definition. To remedy this situation we can further formalize the concept of a machine independent computation within the framework of universal algebra considering the environment $Env$ as a heterogeneous algebra, the state as an agent, $\Pi$, that can perform operations specified in $Env$, and the state transitions as the collection of operations performed by $\Pi$. A computation becomes a process $P = (\Pi, E, S)$ where $E$ is a linguistic expression describing the operations performed by $\Pi$ and $S$ is the status of $\Pi$ with respect to the operations described by $E$ such as executing, thinking, waiting, etc. Thus, a programming language becomes the notation used by $\Pi$ to express computations over the universe of discourse provided by $Env$. This notation is naturally suggested by the signature of the algebra formalizing $Env$ and the agent $\Pi$ performing the operations provided by $Env$. In other words, a programming language is a tuple $[\text{Rus87}]$ $L = \langle \text{Sem}, \text{Syn}, C : \text{Sem} \rightarrow \text{Syn} \rangle$ where
1. Sem is a computing system formally defined by a heterogeneous algebra [BL70, HR76].

2. Syn is the word algebra generated by the signature of the Sem in terms of a given set of constant and variable names.

3. \( \mathcal{L} \) is a partial mapping that associates computing objects (given in Sem) with expressions which represent them (given in Syn) such that there is a homomorphism \( \mathcal{E} : \text{Syn} \rightarrow \text{Sem} \) with the property that for all \( O \in \text{Sem} \) for which \( \mathcal{L} \) is defined the identity \( \mathcal{E}(\mathcal{L}(O)) = O \) holds.

2.2 PROGRAMMING LANGUAGE TRANSLATION

The mappings \( \mathcal{L} \) and \( \mathcal{E} \) involved in the language definition are called [Rus87] the language learning function and the language evaluation function, respectively. Since Sem and Syn are similar universal algebras, \( \mathcal{E} \) is a homomorphism determined by \( \mathcal{L} \) [Rus91]. The consistency of communication between two agents

\[
\begin{array}{cccc}
\text{Sem}_1 & \xrightarrow{\mathcal{L}_1} & \text{Syn}_1 & \xrightarrow{\mathcal{E}_1} & \text{Sem}_1 \\
\downarrow{H_1} & & \downarrow{T_1} & & \downarrow{H_1} \\
\text{Sem}_2 & \xleftarrow{\mathcal{E}_2} & \text{Syn}_2 & \xleftarrow{\mathcal{L}_2} & \text{Sem}_2
\end{array}
\]

Figure 1: Communication consistency diagram.

using the languages \( L_1 = (\text{Sem}_1, \text{Syn}_1, \mathcal{L}_1 : \text{Sem}_1 \rightarrow \text{Syn}_1) \) and \( L_2 = (\text{Sem}_2, \text{Syn}_2, \mathcal{L}_2 : \text{Sem}_2 \rightarrow \text{Syn}_2) \), respectively, to express their computation processes is defined by the commutative diagram in Figure 1. The pair \( (H_1, T_1) : L_1 \rightarrow L_2 \) is called a translator from \( L_1 \) into \( L_2 \); similarly, a pair \( (H_2, T_2) : L_2 \rightarrow L_1 \) that makes this diagram commutative is called a translator from \( L_2 \) into \( L_1 \).

Note that in a communication between two agents using the languages \( L_1 \) and \( L_2 \) respectively, \( \mathcal{E}_1 \) must be computed by the agent speaking the language \( L_2 \) while the \( \mathcal{E}_2 \) must be computed by the agent speaking the language \( L_1 \). From this observation it follows that a communication is well defined in both directions if \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) are consistent and computable. That is, the equalities \( \mathcal{E}_2(T_1(C_1)) = H_1 \) and \( \mathcal{E}_1(T_2(C_2)) = H_2 \) hold whenever the mappings \( \mathcal{E}_1, \mathcal{E}_2, \mathcal{L}_1, \mathcal{L}_2, T_1, T_2, H_1, H_2 \) are defined.

This construction of the computations handled by programming languages shows that the natural decomposition of a programming language into simpler objects is provided by the mechanism of specifying universal algebras where:

- A notation that consists of a set of constant and variable names is first selected. This coincides in programming languages with the language lexicon.

- The names and the constants are further associated with the domain of values that they can assume. This coincides with the typing of the objects of the language universe of discourse.

- A signature is then provided, showing how one can construct new objects of the algebra from the given objects. This coincides with the syntax rules of the language.

- Finally, properties of the algebra thus constructed are specified by a set of equalities called axioms. This coincides with the mechanism of integration of the three components of a programming language, lexicon, type system, and constructs, into the language.

Consequently, we will regard a programming language as a complex object composed of a lexicon, type system, and valid constructs, connected into the language by semantic properties of the universe of discourse.
3 LANGUAGE INDEPENDENT LEXICON

The conventional approach to describing the lexical constructs of a programming language relies on the use of regular expressions [ASU86, HU79a, Tho68, TR84] over sets of characters. Therefore, it seems natural to design scanner generators operating on regular expressions. However, the approaches used in these scanner generators did not lead to the development of language independent scanners that could be easily and conveniently integrated in an algebraic compiler.

In contrast to all previous scanner generators, the lexicon specification for the algebraic compiler is written in terms of primitive lexical constructs such as identifier and number rather than using a given character set taken as the language alphabet. These primitive lexical constructs, called universal lexemes, are found in all programming languages and all of them are generated by regular expressions. In addition, these lexical constructs have structural properties such as the regular expression defining them, the lexemes representing them, the length of their lexemes, etc., that can be used for developing a two level lexicon specification that can be recognized by a two level scanning algorithm called the TwoLev [KR91]. TwoLev operates as a scanner-within-a-scanner. The inner scanner is called the First Level Scanner (FLS) and is efficiently generated from a given set of regular expressions. The unique feature of these regular expressions is that they do not depend upon the programming language that is being implemented, yet the tokens that are returned by the FLS have a clear meaning in every programming language. A TwoLev scanner works exclusively with the lexemes recognized by the FLS, thus allowing us to extend the notion of a regular language as a lexicon specifier to that of a regular language of properties of the primitive lexical entities, called conditions. In this framework, a user developing a scanner can think in terms of higher level constructs which have well-defined semantics rather than thinking about character sets. The specification of a scanner then consists of a set of easy-to-construct equations in the extended regular language of conditions, specifying each construct needed in the language being implemented.

3.1 REGULAR EXPRESSIONS OF CONDITIONS

The primitive lexical entities recognized by the FLS which are universally used by all programming languages can be specified by regular expressions and are grouped into the following classes:

- **I**: identifiers, defined by the regular expression `Letter Letter*`,
- **N**: integer numbers, defined by the expression `Digit Digit*`,
- **W**: white spaces and tabs,
- **U**: unprintable characters (such as newline),
- **O**: other characters (punctuation and separators),

It is important to note that these classes of constructs are universal across all programming languages in the sense that they are used as the building blocks of the actual lexical constructs found in other programming languages. In order to use these lexical items as fundamental entities in the construction of the lexicon of a programming language we characterize them by the attributes `Token` which designate the class, i.e., `Token ∈ {I, N, W, U, O}`, `Lexeme` (abbreviated to `Lex`), which is the actual string of characters identifying the entity of a class, and `Length` (abbreviated to `Len`), which is the number of characters making up a lexeme. Other attributes such as line and column number can be easily added.

The FLS is a deterministic automaton which uses as input a stream of characters in the alphabet of the programming language, and groups them into the above classes common to all programming languages. At each call the FLS returns one tuple of attributes `(Token, Lex, Len)` called a universal lexeme. For example, if a Pascal source text contained the string "var.3", the sequence of universal lexemes returned by repeated calls to the FLS would be: `(I, "var", 3), (O, ",", 1),` and `(N, "3", 1).

Conditions are recursively defined by the following rules:

1. A condition is a relation on the attributes of a universal lexeme; for example, `Token = I, Len <= 8`, and `Lex = "do"` are three conditions.
2. A condition is a logical expression on conditions constructed with the operators or, and and not; for example, \( \text{Token} = I \land (\text{Len} > 3 \lor \text{Lex} = \text{"aa"}) \), is a condition.

Notice: current implementation requires the first relation occurring in a condition to specify the Token attribute, and is called the class specifier. All following relations in that condition (each of which may then specify either the Lex or Len attributes), will refer to the same universal lexeme as the class specifier. Thus, the condition \( \text{Token} = I \land (\text{Len} > 3 \lor \text{Lex} = \text{"aa"}) \) in the example given in rule 2 above specifies an identifier, \( \text{Token} = I \), of length greater than three, \( \text{Len} > 3 \), or whose lexeme is the string \( \text{"aa"} \), \( \text{Lex} = \text{aa} \).

The regular expressions of conditions are constructed from conditions by the usual rules:

1. Any valid condition is a regular expression of conditions.

2. If \( e_1 \) and \( e_2 \) are regular expressions of conditions then \( e_1 | e_2 \) (where \( | \) denotes the choice operation) is a regular expression of conditions.

3. If \( e_1 \) and \( e_2 \) are regular expressions of conditions then \( e_1 \circ e_2 \) (where \( \circ \) denotes the operation of concatenation) is a regular expression of conditions. From now on we will denote the operation of concatenation by a blank.

4. If \( e \) is a regular expression of conditions then \( e^\ast \) (where \( ^\ast \) denotes the Kleene star) is a regular expression of conditions. Its meaning here is the longest possible sequence of universal lexemes such that each element satisfies \( e \).

A language specified by a regular expression of conditions can be recognized by a nondeterministic finite automaton \( \mathcal{NFA} = (Q, C, \delta, q_0, F) \) whose alphabet \( C \) is a set of conditions. For a condition \( e \in C \) and a state \( q \in Q \) the state transition function \( \delta(q, e) = R \) reads: "in state \( q \), evaluate condition \( e \), and if true, then goto states \( R \)." We illustrate the use of regular expressions of conditions with the following specification of a Pascal identifier:

\[
\text{Token} = I \ (\text{Token} = O \text{ and Lex} = \text{"."} | \text{Token} = N \text{ | Token} = I) ^ \ast
\]

3.2 Lexicon Specification by Regular Expressions of Conditions

A lexical entity in a programming language is the lowest level class of constructs in the alphabet of the language that has meaning to the compiler. For example, individual letters or digits generally do not have meaning to the compiler, but the identifiers do. Individual punctuation characters and separators usually have meaning unless they are grouped, as in the Pascal assignment operator \( := \). A lexical entity can represent either an individual construct, as does the operator \( := \), or a class of constructs, as does the class of identifiers. There is a well-established methodology for generating scanners from regular expressions [Tho68, TR84]. We will, however, use regular expressions of conditions rather than conventional regular expressions over some given alphabet to specify a language independent lexicon. This allows us to increase the expressive power of regular expressions while preserving the well-known methodology for scanner generation from regular expressions. Hence, we assume that each lexical entity is specified by a lexicon specification rule of the form \( \text{LHS} = \text{RHS} \) where \( \text{LHS} \) is the token name of the lexeme class specified by the pattern \( \text{RHS} \). For a given programming language, \( \text{PL} \), the lexicon specification rules are collected in the Lexicon Specification File (PL.LSF).

Formally, a lexicon specification rule is an equation that takes the form:

\[
\text{Name} = (\text{Descriptor}) \ [ \ [ (\text{Descriptor}) ] \ [ \ [ \ldots ] \ [ (\text{Context}) ] \ [ (\text{Semantics}) ] \ | \ \text{self};
\]

where \( [ \ [ (\text{Descriptor}) ] \ldots \) denotes any number of optional choices of \( \text{Descriptor} \). The BNF rules specifying \text{Descriptor, Context, and Semantics} are in Figure 2.

\( \text{Condition} \) is a conditional expression and \( \text{REC} \) is a regular expression of conditions as described earlier. The operator \text{any} used in this specification needs special attention. Its meaning depends on the alternative of the \text{Descriptor} in which it occurs. In the first and second alternatives \text{any} is defined
by the equality any = Token ∈ {O, I, N, W}; in the third alternative any is any but c where c is the conditional expression specified by the (REC) in the clause end : (REC); in the fourth alternative any is any but c₁ and any but c₂ where c₁ is the conditional expression specified by (REC) in the clause begin : (REC) and c₂ is the conditional expression specified by (REC) in the clause end : (REC). The clause context : ((REC), (REC)) denotes a list of pairs of regular expressions of conditions specifying the context in which lexical item Name can be found and the clause noncontext : ((REC), (REC)) denotes a list of pairs of regular expressions of conditions specifying the context in which Name cannot be found. (ActList) represents the list of actions to be performed when a lexical construct specified by this equation is recognized. The components using the keywords begin, body, end, context, and noncontext define the syntactical portion of a lexical specification rule, while the keyword action defines the semantic portion. The keyword recursive will allow nested begin and end sequences, as in the definition of Modula-2 comments.

The language specified by lexical equations is usually treated as the first level of valid language constructs of a high-level language. This is achieved by allowing the left-hand sides of lexical equations to be used as terminals in the BNF rules that specify the syntactic constructs of the high-level language.

A collection of equations of the form LHS = RHS where LHS is a token name and RHS is a regular expression of conditions constructed by the rules shown above are used to build a nondeterministic finite automaton, NFA, that will recognize the lexical constructs specified by them. This NFA consumes universal lexemes from the FLS and constructs a tokenized lexeme of the source language. This lexeme is a pair: (Name, Lexeme), where Name is the LHS of an equation and Lexeme is the string resulting from concatenating all of the Lex attributes from the universal lexemes consumed. Thus, a Pascal identifier in the class name id is specified by the equation:

\[ id = \text{Token} = I \ (\text{Token} = O \ \text{and} \ \text{Lex} = "-" \ | \ \text{Token} = N \ | \ \text{Token} = I)^* \]

Notice that many terminals, such as reserved words and punctuation symbols, would be specified by equations defining precisely that terminal. Thus, we allow them to be specified by lexical equations of the form string = self where self represents an equation that will recognize only string. Consider the Pascal assignment operator :=, the lexical equation would be := = self and in this case self represents the equation

\[ \text{Token} = O \ \text{and} \ \text{Lex} = " := " \ \text{Token} = O \ \text{and} \ \text{Lex} = " = " \]

Some constructs in a programming language are more easily defined by specifying a beginning sequence and an ending sequence. Comments and strings can usually be defined in this manner. In order to specify such constructs, we can use the two keywords begin and end. These keywords must be paired. In addition, several choices of beginning and ending sequence can be defined, as is needed to recognize Modula-2 strings using either double quotes or single quotes to open and close them.

Another feature that is necessary in a scanner is the ability to recognize "syntactic sugar" in a construct which does not appear in the final tokenized lexeme. This is best explained by using the example of a Modula-2 string which is delimited by quotes. In order to place a quote in the string itself, the usual convention is to place two quotes together. However, the resulting string should have one quote deleted. This feature is implemented by placing square brackets around an expression, as demonstrated below:

\[ \text{string} = \text{begin: Token} = O \ \text{and} \ \text{Lex} = \ "\" \ \text{body: (any | Token} = O \ \text{and} \ \text{Lex} = \ "\" \ [\text{Token} = O \ \text{and} \ \text{Lex} = \ "\"])^* \]
end: Token = O and Lex = "end"
| begin: Token = O and Lex = "begin"
body: (any | Token = O and Lex = "[" [Token = O and Lex = "]"])*
end: Token = O and Lex = "end"

The context associated with a lexical equation by the keyword context signifies that in order to be recognized by that lexical equation a lexical construct must be matched by the regular expression of conditions from the right-hand side of the lexical equation and must be discovered in the given context. The noncontext associated with a lexical equation by the keyword noncontext signifies that if a lexical construct is matched by the regular expression of conditions in the right-hand side of the lexical equation and is discovered in that context then this lexical construct should not be recognized. Only one or the other needs to be specified, since context and noncontext must be mutually exclusive. Following the keywords context and noncontext is a list of pairs of conditional expressions defining the left and right context or noncontext. We illustrate this facility specifying two lexical constructs called num1 and num2 where num1 must be an integer with the context consisting of a colon preceding it and a comma following it, whereas a number is not num2 in the context of enclosing parentheses.

\[\text{num1 = body: Token=N context: \{ Token=O and Lex=";", Token=O and Lex=";" \}; \]
\[\text{num2 = body: Token=N noncontext: \{ Token=O and Lex="(" Token=O and Lex=")" \};} \]

This provides discrimination of identically defined constructs based on the context only.

The lexical scanner resulting from a lexicon specification by regular expressions of conditions is independent of the compiler of a language that may use this specification as its own lexicon. This scanner reads an input file and tokenizes it accordingly. This facility allows us to use this scanner as the first step of an algebraic compiler which requires the entire source program to be tokenized before the phase of valid construct recognition (see Section 5.3) and their target image generation (see Section 5.4). Therefore this scanner reads its input file and maps it into a data structure called the File of Internal Form, FIF, that may be used by the algebraic compiler and whose structure is given in Figure 3.

![Figure 3: Structure of the FIF.](image)

We implement FIF as a doubly linked list of FIF records that are tuple of the form \((\text{Index}, \text{Root})\) where \text{Index} is the index of the token in the Lexicon data that contains all the left hand sides of the lexicon specification equations and \text{Root} is a data structure that defines the meaning of the token pointed to by the \text{Index}. This could be a lexical name, such as a literal or a variable, stored in the name space, it could be a terminal of the language, or it could be a lexical error. The actions performed by the scanner while constructing a record in the FIF are attached to the lexical equations as semantics information. The default actions performed by the scanner are \text{construct the next FIF record and add it to FIF and do}
not update name space. To change these defaults the semantic specification can use the actions \textit{AddNS}, indicating that the lexeme discovered must be added to the name space, and \textit{NoFIF} indicating that no FIF record is constructed for the lexeme. Other semantics actions can be easily added.

3.3 Implementation of the Scanner

The scanning algorithm reads the input text, tokenizes it according to the lexicon specification, and performs specific actions whenever a lexical item is discovered. The input text is read by the first level scanner, FLS, which is a conventional deterministic finite automaton generated from the regular expressions specifying the universal lexemes. The tokenizing function is implemented by the TwoLev [KR91], which is a non-conventional non-deterministic finite automaton, \textit{NFA}, that reads and evaluates conditions on universal lexemes returned by the FLS rather than reading symbols of an alphabet in order to determine the next state. In a conventional finite automaton, the transition table is indexed by the current state and an alphabet symbol [DeR76, HU79a, Tho68, TR84]. The modified \textit{NFA} implemented by TwoLev uses conditions rather than symbols as the alphabet. The token attribute of the conditions used in the regular expressions of conditions can be used as the second (column) index in the transition table controlling the \textit{NFA}. Therefore, the state transition table of the \textit{NFA} implementing the TwoLev is a two-level table structured as follows:

1. The first level is called the Transition Table, \textit{TT}, and is a two dimensional table whose rows are labeled by the states of the \textit{NFA} and the columns are labeled by the elements of the set \{I, N, W, U, O, \(\epsilon\)\} that are the values of the \textit{Token} attribute of the universal lexemes returned by the FLS (plus the empty transition).

2. Each entry \(TT[q][t]\), \(q \in Q\), \(t \in \{I, N, W, U, O, \epsilon\}\), is a list of transitions. Each transition is a pair <\textit{cond}, \textit{next}> , where \textit{cond} is an index to a record in the second-level table, called the Condition Table, CT, and \textit{next} is the destination of the transition.

The actions performed by the scanner when lexical items are recognized are specified as the semantics attached to the lexical equations and are recognised as the (\textit{ActList}) that follows the reserved word \textit{action}. We assume that they are collected into the Action Table, \textit{AT}.

We developed an automatic implementation of this scanning algorithm using the methodology provided by attribute grammars [ASU86, Paa95]. This implementation contains three phases and is performed by two tools as follows:

Phase 1

Develop the BNF rules that specify the language of lexical equations used as lexical specification mechanism. These rules define an LALR grammar [ASU86, DP81], are the same for every lexicon that one needs to specify with our system, and are given in appendix A. To provide for scanner customization to a particular language we allow each BNF rule to be augmented with one or more functions to be performed by the algorithm that constructs the scanner. That is, each BNF specification rule may have the form \textit{LHS} = \textit{RHS}; \textit{Functions}. These rules are mapped into an extended LALR parse table whose entries are tuples (\textit{Action}, \textit{Function}) where \textit{Action} is the usual LR-action and \textit{Function} is a pointer to the \textit{Functions} provided in the specification. The algorithm used here is a new version of IPC [LR88].

Phase 2

Develop the lexicon specification file, \textit{PL.LSF}, which contains an element of the language specified by the BNF rules at Phase 1. Note, \textit{PL.LSF} is developed following the above rules specifying the syntax of lexical equations. An example of such a file is the \textit{Fortran.LSF} given in appendix B.
Phase 3

A constructor algorithm called ScanGen reads the lexical equations from PL.LSF file and constructs the NFA that recognizes the lexical constructs specified by them. ScanGen reads the lexical equations, recognizes them as valid with respect to the specification language, and maps them into the TT and CT tables required by the TwoLev, and the AT table required by the scanner. ScanGen is controlled by the parse table constructed in Phase 1 and in addition to the usual action performed by an LALR parser it also performs the function associated with the action in the parse table entry. It is this function which indicates that information should be added to one of the table CT, TT, or AT. During its operation ScanGen recognizes conditions, keywords, regular operators, and semantic actions. The conditions are mapped into postfix form and then they are stored in the condition tables CT; regular operators are used to construct the TT table controlling the algorithm performed by the TwoLev; semantic actions are accumulated into the action table AT. We show how these operations are performed by the ScanGen in Figure 4 using the specification of the relations $\leq$ or $\geq$

$$rel = body: \text{Token}=O \text{ and } \text{Lex} = "<"$$
$$(\text{Token}=O \text{ and } \text{Lex} = "=" \text{ or } \text{Token}=O \text{ and } \text{Lex} = ">")$$

The three conditions, called here $c_1$, $c_2$, and $c_3$ are $\text{Token} = O \text{ and } \text{Lex} = "<"$, $\text{Token} = O \text{ and } \text{Lex} = "="$, and $\text{Token} = O \text{ and } \text{Lex} = ">")$, respectively. To build the CT the class specifiers ($\text{Token} = O$ in each case) are used to identify the column of the TT where pointers to the rest of conditions are set. The remainder of the expressions are used to construct the following three postfix expressions: $c_1 : \text{LEX } "<" \text{ EQ } \#$, $c_2 : \text{LEX } "=" \text{ EQ } \#$, $c_3 : \text{LEX } ">" \text{ EQ } \#$, where the operands are the lexeme returned by the FLS, denoted here by LEX, and the constants "$<", "="", ">")$, and the operator is the test for equality, denoted here by EQ. We use $\#$ to mark the end of the postfix form.

![Diagram](image)

Figure 4: a) The modified NFA, b) CT representation of conditions.

Hence, to implement a new scanner with this methodology the user could start with Phase 1, thus re-implementing our system, or use the language of lexical equations as we developed in appendix A and continue with Phase 2, to customize the system for a given language. In either case no programming as usual is required.
The automaton constructed by ScanGen recognizes regular expressions of conditions and consequently the events that determine its transitions are the truth values of some predicates. Therefore, the conventional methodology for the optimization [HU79b] of this automaton is not valid. However, to avoid the flooding of the automaton with  \( \epsilon \) transitions that result naturally from the construction, we apply systematically, during the automaton construction, the transformations in figures 5, 6, 7.

1. When the regular expression \( e_1 \circ e_2 \) is recognized, i.e., a concatenation operator is processed, we apply the transformation in Figure 5.

\[
\begin{array}{c}
\varepsilon \\
\overset{c_1}{\varepsilon} \\
\overset{e_1}{f_1} \\
\overset{e_2}{f_2} \\
\overset{f_2}{f_2}
\end{array}
\quad \Rightarrow
\begin{array}{c}
\varepsilon \\
\overset{c_1}{\varepsilon} \\
\overset{e_1}{f_1} \\
\overset{e_2}{f_2} \\
\overset{f_2}{f_2}
\end{array}
\]

Figure 5: Automaton recognizing \( e_1 \circ e_2 \).

2. When the regular expression \( e_1|e_2 \) is recognized, i.e., a choice operator is processed, we apply the transformation in Figure 6.

\[
\begin{array}{c}
\varepsilon \\
\overset{e_1}{c_1} \\
\overset{e_2}{c_2} \\
\overset{f_1}{f_1} \\
\overset{f_2}{f_2}
\end{array}
\quad \Rightarrow
\begin{array}{c}
\varepsilon \\
\overset{e_1}{c_1} \\
\overset{e_2}{c_2} \\
\overset{f_1}{f_1} \\
\overset{f_2}{f_2}
\end{array}
\]

Figure 6: Automaton recognizing \( e_1|e_2 \).

3. When the regular expression \( e^* \) is recognized, i.e., a Kleene star operator is processed, we apply the transformation in Figure 7.

\[
\begin{array}{c}
\varepsilon \\
\overset{e_1}{c} \\
\overset{e}{f_1} \\
\overset{e}{f}
\end{array}
\quad \Rightarrow
\begin{array}{c}
\varepsilon \\
\overset{e_1}{c} \\
\overset{e}{f_1} \\
\overset{e}{f}
\end{array}
\]

Figure 7: Automaton recognizing \( e^* \).

We use the facility provided by context and noncontext options in the lexical equations specifying the lexicon to reduce the nondeterminism of the automaton performing the scanning. The context and noncontext are provided by the syntax of the source language constructs whose lexicon is specified by the lexicon specification file, \( PL.LSF \). That is, for each lexical equation \( LHS = RHS \) we determine the lexical context that can precede and follow \( LHS \), and the lexical non-context that cannot precede and cannot follow \( LHS \), in the valid constructs of the \( PL \). The smallest set of such pairs are associated as context with the the lexical equation \( LHS = RHS \). The first action of the automaton when it starts the recognition of a new lexical entity determines the context in which it operates and chooses the right sub-automaton to perform. Whether the automaton thus obtained is optimal or not is an open question.

4 LANGUAGE INDEPENDENT TYPE SYSTEM

The collection of types supported by a programming language is usually studied as the lattice of types [Mil78, KU80] and the type system is seen as a procedure of the compiler that determines the type
of a programming language object using the operation that constructs this object and the types of its operands. We consider types as mathematical abstractions which exist independent of the programming language that uses them and which are constructed from given types [Rey85] using algebraic operations. Our goal here is to use the existing type theory for the development of a language independent type system where types are elements of a database and are generated from other types using a given collection of predefined types and type constructors. A compiler can use this database by appropriate queries during language processing, independent of the language it processes. Such a type system is appropriate for both typed and untyped languages. Our motivation for such treatment of the type system is the simplification of a programming language translation and the development of a methodology for machine-independent language design and implementation that is amenable to be learned and used by computer users according to their problem domains.

4.1 Classification of Types

In this paper a type is a tuple \( \text{Type} = (\text{DataCarrier}, \text{Operations}) \) where \( \text{DataCarrier} \) is a set of computing abstractions, such as integers, reals, strings, etc, and \( \text{Operations} \) is a set of operations defined on the objects that belong to the \( \text{DataCarrier} \). The collection of types supported by a programming language can be classified as:

- **Predefined**: which is the set of computing types provided to the programmer by the language definition. Examples of predefined types are integer and real in Pascal.

- **Defined**: which is the set of computing types that programmers define in their programs using a given set of type constructors provided by the language definition. Examples of defined types are arrays and records in Pascal.

Note that a predefined type of one language may be a defined type in another language. For example, the type complex is predefined in Fortran but it could be supported by Pascal or C only as a user defined type. Hence, in order to develop a language independent type system we structure the computation types that a language may assume such that this structure is an algebra built on top of a finite set of predefined types using a given collection of type constructors. We call this algebra a type structure and organize it as a tree called the tree structure of the type system shown in Figure 8, where \( C_k(r_1, \ldots, r_p), \ C_k(s_1, \ldots, s_p), \ C_k(t_1, \ldots, t_p) \) are types constructed by the constructor \( C_k \) using the objects \( (r_1, \ldots, r_p) \), \( (s_1, \ldots, s_p) \), and \( (t_1, \ldots, t_p) \), respectively, as parameters.

![Figure 8: Tree structure of the language types.](image)

Every type supported by a programming language is completely described by a path from the root to a leaf in the tree in Figure 8 and a descriptor specifying the data carrier and the operations of the type represented by the respective leaf. In other words, the root \( TT \) of the tree in Figure 8 is the name of
the type system algebra, leaves of the tree in Figure 8 are the elements of the TT algebra i.e., are types (predefined or defined) or type constructors, and the interior nodes represent type constructors used to construct the types at the leaves. From a mathematical viewpoint we could represent this tree in the form $TT = (D_{TT}; T_1, T_2, \ldots, T_m, C_1, C_2, \ldots, C_n)$ where $D_{TT}$ is a collection of type descriptors, $T_1, T_2, \ldots, T_m$ are given types that can be interpreted as the constants of the algebra $TT$, and $C_1, C_2, \ldots, C_n$ are operators that when applied on some existing types, such as $T_1, T_2, \ldots, T_m$ generate new types in $TT$ as is the case in any other algebra. Figure 9 represents the specification of the algebra $TT$ as initially seen by a compiler.

![Diagram of TT algebra]

Figure 9: The data representation of initial form of TT algebra.

The tree structure of the TT algebra facilitates its efficient implementation as a database. The actual data carriers of the types in the algebra $TT$ are hidden in Figures 8 and 9 by the descriptors given in $D_{TT}$. In other words, for each type in $T \in TT$ there is a descriptor $d_T \in D_{TT}$ that completely defines the data carrier, operations, and pertinent properties of the type $T$.

The predefined types of the language are given to the programmers, i.e., there are no constructors specifying them, therefore the predefined types occur as constants of the type system attached to the root TT of the tree in Figure 8. A Constructor is not an actual type, rather a Constructor is a definition scheme that a programmer uses to define a type. For example, in Pascal array is a type constructor that allows a programmer to define a new type t which is an array of 10 integers as shown below.

$$\text{type } t = \text{array}[1..10] \text{ of integer}$$

Hence, the type $C_k(s_1, \ldots, s_p)$ in Figure 8 is defined using the constructor $C_k$ that takes $(s_1, \ldots, s_p)$ as parameters, where each $s_i$, $1 \leq i \leq p$, is a typed object or a type (predefined or already defined). Further we will treat types and constructors similarly and therefore will describe them using a unique type specification template called the type descriptor.

### 4.2 Type Descriptor

The descriptor a type completely defines the data carrier, operations, and other properties of the type. Since the type system is meant to be used by a compiler, the descriptor of a type should provide all the information needed by the compiler to do the type checking or the type inference of the valid source language constructs and to assure their correct implementation in the target language of the compiler. We use the notation $TD(n)$ to denote the type descriptor of the type at the node $n$ in the type tree $TT$. The structure of a type descriptor is shown in Figure 10.

**Name**

Normally, types used in programming languages are identified by their names. Therefore, the name of a type is a component of its type descriptor. However, the same name may represent different types in different contexts in a programming language construct. For example, in a block structured language, the same type name may identify different types in different blocks.
<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Constructor</th>
<th>Template</th>
<th>Operations</th>
<th>Objects</th>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type name</td>
<td>Type descriptor</td>
<td>definition</td>
<td>0 → nullary operations</td>
<td>unary operations</td>
<td>1 → binary operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>declaration</td>
<td></td>
<td></td>
<td>2 →</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>application</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Kind**

We have structured the types supported by a programming language using a tree whose nodes are either types (predefined or defined) or type constructors. The type descriptor we are constructing here is a mechanism that allows the compiler to handle uniformly these nodes. This is obtained by providing an item of information called *kind* in the type descriptor which is defined by the formula:

\[
TD(n).\text{kind} = \begin{cases} 
\text{predefined}, & \text{if } n \text{ is a predefined type;} \\
\text{defined}, & \text{if } n \text{ is a defined type;} \\
\text{constructor}, & \text{if } n \text{ is a type constructor.}
\end{cases}
\]

**Constructor**

A defined type is always created by a constructor. Therefore, the descriptor is provided with an item of information called *Constructor* that identifies the constructor of that type. That is, \(TD(n).\text{constructor}\) is defined by the formula:

\[
TD(n).\text{constructor} = \begin{cases} 
\text{Null}, & \text{if } n \text{ is a predefined type;} \\
\text{Null}, & \text{if } n \text{ is a type constructor;} \\
C, & \text{if } n \text{ is a type constructed by constructor } C.
\end{cases}
\]

**Template**

As stated before, types are mathematical abstractions. Their data carriers are either sets of mathematical objects, such as the truth values or numbers, for the case of predefined types, or are data structures constructed from the elements of such sets and data carriers of previously constructed types, if they are defined types. In both cases the objects which belong to a type are specified by a template associated with the type descriptor. For predefined types this template specifies either an interval or an enumeration of the objects which belong to the DataCarrier of that type. The interval is usually used when the data carrier of the mathematical type is an infinite set, totally ordered by an order relation; the enumeration is usually used when the data carrier of the mathematical type is finite. For example, we may indicate by \((-\infty, +\infty)\) or \([-2^{31}, 2^{31} - 1]\) the interval of integers supported by a language and by \{0, 1\} the truth values of the boolean type. For the defined types the template of the data carrier shows how the elements of the type are constructed from elements of the component types used by the type constructor. The mechanism that implements this is provided by a parameterized data structure that can be instantiated into an actual object of that type, a macro-operation which allows the compiler to declare (i.e., to generate) an objects of this type, and a macro-operation that allows the compiler to access the components of an object of this type. For example, the template of the data carrier of an array consists of: (1) a macro-operation defining a data structure array_def that specifies the dimension of the array, the lower and upper bounds of the element selectors in each dimension, base location of the array elements, and the type of the elements of the arrays; (2) a macro-operation array_dec that specifies (declare or generate) and array; (3) a macro-operation array_app that accesses an array element of a given array.
Parent and child

The subtype relationship between two types, \( T_1, T_2 \), where \( T_1 \subseteq T_2 \), is recorded in the type descriptor by \( T_1 \) parent pointing to the descriptor of the type \( T_2 \) and \( T_2 \) child pointing to the descriptor of the type \( T_1 \). This allows us to show explicitly the type hierarchy in the \( TT \) database and provides an easy way of implementing type equivalence.

Operations

The type descriptor should have access to the signature of each operation supported by the type, which contains information that specifies the domain, the range, and the target implementation of the operation. The operations supported by programming languages are however heterogeneous. Therefore, the way of assigning uniquely an operation to a type is to assign the operation to its result type, which is unique. Hence, our convention is that operations associated with a type descriptor, \( TD(n) \), are all those operations whose range (i.e., result) is of the type described by the \( TD(n) \). Further, we assume that the implementation of an operation in the target is done through a function associated with the signature of the operation. Since the general form of the signature of an operation \( O \) of arity \( n \) supported by \( TD(n).name \) is \( O : T_1 \times T_2 \times \ldots \times T_n \rightarrow TD(n).name \), our conventions lead to consider the signature of these operations as a list of tuples of the form \((Operator, Type_1, \ldots, Type_n, Function)\), where \( n \geq 0 \). The first element of the tuple is the operation symbol, the last element of the tuple is the implementation of this operation in the target language, the rest of the elements are an ordered set of selectors specifying the types of the operands. If this set is empty, \( Operation \) is a constant implemented in the target by \( Function \). Notice however that the unique result type of an operation could be a type variable whose value can be determined by evaluating a type expression of the type algebra.

Objects

The list of the objects of a given type (constants or variables) which are declared in the program are linked by the compiler to the type descriptor of that type. Program types that are defined by a type constructor are seen here as objects of type the type descriptor used to constructs them and thus are linked to that type descriptor. This link is implemented by providing a field in the type descriptor which will point to a doubly-linked list where all objects of that type are maintained. Since nullary operation are predefined constants of the type, they are linked to this field of the descriptor.

A type system can handle the types of the objects manipulated by programming language computations by using a data representation of the tree in Figure 8 called \( TT \) database. Each entry in the \( TT \) database represents a type and hence contains the type descriptor of that type. \( TT \) has two components, a static component, where all predefined types and type constructors are collected, and a dynamic component, that collects the types defined by the programmer during program development. The static component of the \( TT \) defines the initial contents of the database implementing a language independent type system; the dynamic component of the \( TT \) defines the collection of types defined in a given program. The static component of the \( TT \) is specified by the compiler implementor using an equational language and the dynamic component is constructed by the compiler during program compilation. The initial form of the \( TT \) is the static component of the Figure 11.

4.3 TYPE SPECIFICATION LANGUAGE

The type system of a compiler must be initialized with the predefined types and the type constructors before the compiler can query it and create new entries. A type specification language has been developed for this purpose and is used to specify the predefined types and the type constructors supported by the type system. The type system specification targeted to a given compiler is developed independently of the other components of the compiler and in parallel with them. For a given programming language, \( PL \), the type specification rules are collected in the Type Specification File (\( PL.TSF \)). A compiler generation tool, called TypeInit, reads the \( PL.TSF \) file, analyzes its contents, creates \( TT \) database, and initializes it as shown in Figure 11.
A type specification rule in PL/TSF allows the user to specify all the properties of the types discussed in the previous sections. Therefore, a type specification rule is a reflection of the components of the type descriptor. To make the type specification language convenient to its users we organize type specification

\[
\text{TypeSet} \quad \{ \ [ set_i = (T_{i1}, \ldots, T_{in}) \] \}
\]

\[
\text{TypeSpec} \quad [ \\
\quad \text{<name>} = \text{kind: predefined | defined | constructor} \\
\quad \text{constructor: Null | <constructor name>} \\
\quad \text{descriptor: Null | <descriptor name>} \\
\quad \text{template:} \\
\quad \quad \text{definition} = \{ <\text{macro name}> \} \\
\quad \quad \text{declaration} = \{ <\text{macro name}> \} \\
\quad \quad \text{application} = \{ <\text{macro name}> \} \\
\quad \text{operations:} \\
\quad \quad \{ (\text{operator}, \text{operand1}, \text{operand2}, \ldots, <\text{macro}>) \} \} \\
\quad \text{objects:} \quad \{ <\text{object name}> \{ [, <\text{object name}> \} \} \} \\
\quad \text{parent:} \quad \{ <\text{type name}> \} \\
\quad \text{child:} \quad \{ <\text{type name}> \{ [, <\text{type name}> \} \} \} \]
\]

The complete syntax of the type specification language is shown in the Figure 12. The text in bold should appear as it is, in any order. The components enclosed in curly brackets "{" and "}" are optional and the components in brackets "[" and "]" must appear one or more times. If the value of a keyword is null, that keyword needs not appear in the specification.
The TypeInit creates an empty TT database. Then it reads the PL.TSF file that contains a type specification and for each keyword equation in PL.TSF it constructs a type descriptor and updates TT with this descriptor. We illustrate the usage of the type specification language in Figure 13 with a partial specification of the Pascal types "integer" and "real" and of the array constructor used in Pascal.

\[
\begin{align*}
\text{TypeSet} & \quad S1 = (\text{integer}, \text{real}) \\
\text{TypeSpec} & \quad \text{integer} = \text{kind: predefined} \\
& \quad \text{template: declaration} = \text{int @} \\
& \quad \text{operations:} (+, \text{integer, integer, +}), (-, \text{integer, integer, +}) \\
& \quad \text{parent: real} \\
& \quad \text{real} = \text{kind: predefined} \\
& \quad \text{template: declaration} = \text{float @} \\
& \quad \text{operations:} (==, \text{integer, (float)}), (+, S1, \text{real, +}), (+, \text{real, integer, -}), \\
& \quad (-, S1, \text{real, -}), (-, \text{real, integer, -}) \\
& \quad \text{child: integer} \\
& \quad \text{array} = \text{kind: constructor} \\
& \quad \text{descriptor: array descriptor} \\
& \quad \text{template: definition} = \text{array_def, declaration} = \text{array_dec, application} = \text{array_app}
\end{align*}
\]

Figure 13: Type specification for pseudo Pascal.

Notice that we have grouped the names integer and real into the set S1 in the TypeSet part of the specification in Figure 13. This allows us to reduce the number of operation specification for the predefined type "real". Since we intend to use this specification for a compiler that maps Pascal valid constructs into valid constructs of another language that supports the predefined types "integer" and "real", only the declaration functions "int.dec" and "real.dec" appear in Figure 13. Moreover, we do expect that the operations on integers and reals are denoted by the same symbols in the target language. The implementations of Pascal array definition, Pascal array declaration, and Pascal array application in the target language are explicitly provided in our specification.

Pascal is a strongly typed language where the type "integer" is a subtype of the type "real". Therefore, we can assign an integer to a real but not vice versa. So there is a operation "==" that converts integers to real. When the compiler finds an operation + or -, with one operand of type real and another of type integer, the compiler can query the type system whether such an operation is permissible. The type system looks into the TT database and finds that all + or - operations with one of the operands of type real are of type real. Consequently, the appropriate code can be generated using the functions provided in the signature of these operations. In another context where \( x := y \) is recognized and \( y \) is of type "real" which \( x \) is of type "integer", querying again the type system compiler finds out that no conversion from "real" to "integer" is possible and thus the compiler can declare an illegal operation.

4.4 TYPE SYSTEM INTERFACE

The type system is designed to help the compiler perform the activities related with type checking and type inference. For that, the compiler queries the type system using the interface functions. Interface functions are implementations of given algorithms that find the type of values that can be assumed by valid expressions constructed using the operators provided by the existing types in the type system. These algorithms are suitable whether the language is strongly typed or not. For example, adding the operation (==, real, (int)), which converts a real to an integer, to the specification in Figure 13, does not change the interface to the system. However, the language using the new specification may not remain strongly typed. By adding the above operation to the integer in Figure 13, the integer operations + and - have operands of type S1 and it is up-to the compiler writer to rewrite the conversion functions such that they can either convert both operands to integer and then do the addition or can add or subtract
first the operands and then convert the result to integer. But type checking and type inference methods using the new specification do not change.

Querying and updating the type system is done using an open-ended collection of interface functions. The following are classes of interface functions supported by the current implementation of the type system:

1. Functions that convert between predefined type names and unique type ids.
2. Functions that create new types in the type system.
3. Functions that update the fields of a given type in the type system.
4. Functions that instantiate template and conversion macro-operations.
5. Functions that search the type system for the type of an operation when the type of the operands are given and check if a given operation is supported by a given type.
6. Functions to retrieve descriptors and the data recorded in their fields.

A partial type specification for a Fortran compiler is given in appendix C.

5 LANGUAGE INDEPENDENT CONSTRUCT SPECIFICATION

The programming language constructs used to express computations can be classified in three groups called definitions, declarations, and applications. Definitions are language constructs that allow a programmer to extend the range of computation objects handled by the program. Since we assume that all computations handled by a program are instantiations of some abstract data types, definitions are simply language constructs provided by various type constructors that define new types in terms of the types already available. Declarations are language constructs that allow a programmer to specify the type of a value a constant or a variable can assume by associating lexical names with defined or predefined types. Applications are language constructs that allow a programmer to express computations on objects denoted by constants and variables declared in the program. However, these classes of language constructs are not independent. For example, definitions may have as components other definitions, declarations, and objects constructed by applications (see C function definition); declarations bind names denoting objects constructed by applications to their types in a given context and therefore they may have as components applications and definitions; applications may use declarations, definitions, and other applications as components. This weaving makes the specification of the action performed by the compiler while it recognizes valid source language constructs and maps them into semantically equivalent valid target language constructs a difficult task. But the algebraic compiler is based on mathematical algorithms that operate in a compositional manner. That is, the algebraic compiler recognizes valid source language constructs in terms of the validity of their components and generates equivalent valid target language constructs in terms of the target images of the source language construct components. Therefore, this classification fits naturally the task of the compiler and simplifies it when compositionally performed.

We assume that programming language constructs are specified by equations of the form \( LHS = RHS \) called language specification rules, where \( LHS \) is a lexical name called syntax category and \( RHS \) is a pattern composed of syntax categories and fixed symbols. Examples of such specification rules are BNF rules. Further, we suppose that the class of valid constructs of a programming language is specified by a finite set \( R \) of language specification rules. Each \( r \in R \) is an equation of the form \( \text{lhs}(r) = \text{rhs}(r) \), where \( \text{lhs}(r) \) stands for the left-hand side of the equation \( r \) and \( \text{rhs}(r) \) stands for the right-hand side of the equation \( r \); moreover, \( \text{rhs}(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \) for some \( n \geq 0 \), where \( t_0, t_1, \ldots, t_n \) are fixed strings called terminal symbols or are the empty word, \( \epsilon \), and \( A_1, \ldots, A_n \) are parameters called nonterminals. A specification \( R \) is clean if every nonterminal is a syntax category, i.e., there are no useless specification rules. In other words, for any \( r \in R \) and \( A \) a nonterminal used as a parameter in the \( \text{rhs}(r) \) there is at least one rule \( r' \in R \) such that \( A = \text{lhs}(r') \).
Each specification rule \( r \in R \), \( A_0 = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), has two interpretations by an algebraic compiler, a semantic interpretation and a syntactic interpretation. The semantic interpretation considers nonterminals \( A_i \), \( 0 \leq i \leq n \), used in the rule \( r \) as data types denoted by \([A_i]\), \( 0 \leq i \leq n \), and the rule \( r \) is interpreted as the signature of an operation \( t_0 t_1 \ldots t_n : [A_1] \times \ldots \times [A_n] \rightarrow [A_0] \). The syntactic interpretation considers nonterminals \( A_i \), \( 0 \leq i \leq n \), used in the rule \( r \), as classes of equivalent language constructs denoted by \([A_i]\), \( 0 \leq i \leq n \), where the class of equivalence \([A]\) is defined as follows: \( w \in [A] \) if there exists \( r \in R \) such that \( lhs(r) = A \), \( rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), and \( w = t_0 w_1 t_1 \ldots t_{n-1} w_nt_n \) for some \( w_i \in [A_i] \), \( 1 \leq i \leq n \). The algebraic compiler recognizes the validity of a language construct \( w \in [A] \) in terms of the validity of its components \( w_i \in [A_i] \), \( 1 \leq i \leq n \), using the syntax interpretation of a specification rule \( r \) as an operation of construct well-formation, \( t_0 t_1 \ldots t_n : [A_1] \times \ldots \times [A_n] \rightarrow [A] \).

However, in order to perform the mapping of \( w \) into its target form the algebraic compiler needs to determine the semantic interpretation \( t_0 t_1 \ldots t_n : [A_1] \times \ldots \times [A_n] \rightarrow [A] \) of the rule \( r \) as well. This is a difficult task since there is no one-to-one association between \([A]\) and \([A_i]\), and moreover, the type of the components \( w_i \), \( 1 \leq i \leq n \), of the construct \( w \) identified by the rule \( r \) may change with the context in which \( r \) is used to recognize \( w \) as a valid construct of the language. In a conventional compiler this is known as the type inference problem, where the type of value \( w \) may assume is deduced from the type of values assumed by its components \( w_i \), \( 1 \leq i \leq n \), in the context in which \( w \) is identified by \( r \). The mechanism used by the algebraic compiler to solve problems concerning the type checking and type inference consists of constructing a mapping that associates each syntactic category \( A \) used in \( R \) with a semantic domain \( D_A = \{T_1, T_2, \ldots, T_k\} \) and a syntactic domain \([A]\). As can be seen above, the semantic domain \( D_A \) is a collection of data types whose elements can be denoted by the constructs specified by the rules \( r \in R \) such that \( lhs(r) = A \). The syntactic domain \( A \) is however uniquely determined by the class of equivalence \([A]\), which coincides with the collection of language constructs specified by the rules \( r \in R \) such that \( lhs(r) = A \). We assume that all types used by a program are maintained into a type database and consequently the type inference problem is resolved by an extensive search of the type database, as seen in Section 4.

5.1 DATA STRUCTURES FOR CONSTRUCT RECOGNITION

Different kinds of constructs (definitions, declarations, applications) are treated differently by the compiler. That is, the compiler operates on different data structures when it recognizes the validity of a language construct \( w \) and regenerates it as a target construct \( T(w) \) when \( w \) is a definition, a declaration, or an application. When a definition is recognized the algebraic compiler constructs the type descriptor, \( TD \), of the new defined type and updates the database \( TT \) linking the \( TD \) on the list associated with its constructor. The objects of a declaration are maintained as a double linked list, \( LL \), of data structures called symbols whose components are:

1. A pointer called \( PLink \) that links the symbol with its predecessor on the \( LL \) list.
2. A pointer to the symbol name called \( Name \) in the name space, \( NS \), where the symbol was originally stored as a string by the algebraic scanner, see Section 3.
3. A pointer \( TType \) to the type descriptor in the \( TT \) data base of the type of the symbol.
4. An indicator called \( Kind \) which shows if the symbol is the name of a type, a constant, a variable, or an operator (operation, function, procedure).
5. A pointer to the next object of the \( LL \) list called \( NLink \).

The structure symbol is shown in Figure 14. Hence, when a new construct with the right declaration attribute is recognized, the algebraic compiler adds to the \( LL \) list of that construct the elements declared as new symbols (constants or variables) of the type used in that declaration. Note that a definition is treated as a declaration whose type is the type descriptor of the type constructor used for that definition. The type descriptor of each type in the \( TT \) database has a pointer to the symbol representing its first object. The objects of a type constructor are types. Thus, the compiler can link all elements of a given type on the list of objects of that type maintained by the type system, which survive the execution of the compiler. Hence, this mechanism can easily handle debugging information.
The valid constructs representing applications are identified by the compiler using their specification rules in terms of their components. That is, a valid language construct \( w = t_0 w_1 t_1 \ldots t_{n-1} w_n t_n \) specified by the rule \( r \), \( \text{rhs}(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), is identified by the compiler when it has already identified all the components \( w_i \in [A_i] \), \( 1 \leq i \leq n \), and the context of \( w \) in the text matches the context of \( r \) which is precomputed at compiler generation time [Rus94]. When \( w \) is recognized as a valid construct the compiler reduces its representation in the file of internal form, \( \text{FIF} \), (see Section 3) to a pointer to an abstract parse tree of \( w \). That is, if all \( w_i, 1 \leq i \leq n \), have been reduced to the abstract parse trees \( T_i, 1 \leq i \leq n \), then \( w \) is reduced to the abstract parse tree labeled \( r \) which is either a leaf, when \( \text{rhs}(r) \) contains no nonterminal symbols, or is an interior node of \( \text{arity} n \) where \( n \) is the number of nonterminals that occurs in the \( \text{rhs}(r) \), whose children are the abstract parse trees \( T_i, 1 \leq i \leq n \).

The leaves of the abstract parse tree are labeled by the lexical rule used to identify them (see Figure 3) and are maintained in the doubly linked list of lexical items of the node \( r \) denoted by \( \text{LL}(r) \). Each \( \text{leaf} \in \text{LL}(r) \) has the structure shown in Figure 15 where \( \text{Rule} \) is lexical rule used by the scanner to recognize that leaf. \( \text{Parent} \) is required here by the implementation of the process of transforming leaves into symbols.

The interior nodes of the abstract parsing tree represent specification rules

\[
\text{lhs}(r) = t_0 \ \text{lhs}(r_1) \ t_1 \ldots t_{i-1} \ \text{lhs}(r_i) \ t_i \ldots t_{n-1} \ \text{lhs}(r_n) \ t_n
\]

used to recognize the validity of the portion of the source texts \( t_0 w_1 t_1 \ldots t_{i-1} w_i t_i \ldots t_{n-1} w_n t_n \) where \( w_i \in [\text{lhs}(r_i)], 1 \leq i \leq n \), as seen in figure 16.

The structure of an interior node of the abstract parse tree is given in Figure 17. The \( \text{Scope} \) delimits the portion of the \( \text{LL} \) list of the entire input text that are leaves of the abstract parse tree of the construct represented by this node. \( \text{Macro}_1, \text{Macro}_2, \ldots \) are macro-operations that show the translations of the source language construct whose abstract parse tree is rooted in this node into various target languages.
that may be used by the compiler. $Image_1, Image_2, \ldots$ are the target images of the source language construct whose abstract parse tree is rooted in this node. These target images are available if all the children of the node, represented in Figure 17 by $Child$, have been already generated. Otherwise $Image_1, Image_2, \ldots$, are null pointers. $Rule$ shows the specification rule $r$ that was used to recognize this construct. Finally, $Kind$ and $Type$ are the same as described previously in the case of a leaf or a symbol.

![Diagram of interior nodes structure](image)

Figure 17: Structure of the interior nodes.

The operational manner of the algebraic compiler suggested above requires a family of functions, one function for each nonterminal $A$ used in $R$, denoted $class : [A] \times R \rightarrow \{def, dec, app\}$ which for each $w \in [A]$ and $r \in R$ such that $lhs(r) = A$ decides if $w$ is a definition, a declaration, or an application. A clean language specification would not allow a construct $w \in [A]$ to be of more than one kind. However, conventional languages do allow such constructs. For example, the construct $float x[3] = \{2.5, 2.6, 2.7\}$, which represents the definition \textit{"typedef float array3[3]"}, the declaration \textit{array3 x}, and finally the application $\{x[0] = 2.5; x[1] = 2.6; x[2] = 2.7\}$ at the same time, is valid in C. In addition, various languages support different kind of interweaving of definition, declaration, and application within a single language construct, but all languages support their clean separation. Thus, in order to make the algebraic compiler as universal as possible, the function $class$ must have as range the set of subsets of $\{def, dec, app\}$, i.e., $class : [A] \times R \rightarrow \mathcal{P}(\{def, dec, app\})$, where $\mathcal{P}(X)$ denote the set of all subsets of $X$. That is, $class$ maps each element $w \in [A]$ specified by a rule $r \in R$ into the subset of the kind of constructs it represents.

To implement the function $class$ described above we currently use the finiteness of the set $R$ and associate the property of a construct $w \in [A]$ to be a definition, a declaration, or an application with $A$ itself in a similar way in which attributes are associated with nonterminals in logic attribute grammars [Hed94, Paa95]. This is done by associating with each rule $r \in R$ a subset of the attributes $\{def, dec, app\}$ with the meaning that a construct specified by $r$ may potentially encapsulate all the kind of constructs specified by the attributes associated with $r$. In addition, for each rule $r \in R$, we construct a procedure that transforms any language construct $w$ recognized as being specified by $r$ into a sequence of constructs $w_{def}; w_{dec}; w_{app}$ such that $class(w_{def}, r) = \{def\}; class(w_{dec}, r) = \{dec\}; class(w_{app}, r) = \{app\}$ and $w$ and $\{w_{def}; w_{dec}; w_{app}\}$ are semantically equivalent, i.e., their transition systems specify the same computations. This approach allows us to cleanly separate the action performed by the algebraic compiler when a definition, a declaration, or an application is recognized as a valid source language construct.

### 5.2 Scope and Extent

From a semantic viewpoint the computational objects denoted by source language constructs exist as instances of some data types in time and space. The time existence of these objects is determined by the availability of these language constructs with a given meaning in different states of the transition system represented by the program; the space availability of these objects is determined by the program text where these language constructs are visible. The sequence of transitions where a language construct is available with the same meaning is called the \textit{extent} of that language construct; the program text where a language construct is visible with the same meaning is called the \textit{scope} of that language construct. Neither the extent nor the scope of various language constructs are explicitly provided by their specification rules. Therefore, during the compilation process, the compiler determines both the extent and the scope of the language construct whose validity is currently being recognized based on ad-hoc rules. These rules do not belong to the programming language syntax specification. They are determined by the structure
of the language construct expressed in terms of the kind of components, definitions, declarations, and applications, it takes. For example, in a block structured language a construct defined by a rule of the form \textit{Block = Declaration; Statement} specifies also that the scope of the constants and variables declared in the \textit{Declaration} part is the program text represented by the \textit{Statement} part of the construct. A construct specified by a function definition of the form \textit{Function = Name Parameters Body} may also define the scope of the names used in the \textit{Parameters} part as the text represented by the \textit{Body}. The extent of the constants and variables used in the \textit{Body} may be defined as the life of the process that will perform this function and is implemented by the \textit{activation record} of the \textit{Name}. Consequently, the properties of a language construct that provide information regarding the extent and the scope of the objects manipulated by it are implicitly associated with the left hand side of the specification rules of that language constructs. We associate these properties explicitly with the specification rules in \textit{R} and use the keyword \textit{standard} as the name of a set which contains these attributes (see Section 5.4). That is, if \( r \in R \) and \( \text{scope} \in \text{standard}(r) \) then the program text of a source language construct specified by \( r \) is the scope of all variables and constants defined and declared in that text; if \( r \in R \) and \( \text{process} \in \text{standard}(r) \) then the computation encapsulated in a construct specified by \( r \) represents a process that can perform independently of other processes during program execution and the process life of this process is the extent of all variables and constants defined and declared in that construct.

The implementation of the scope computation rule is based on maintaining an ordered list of lexical names denoted by \textit{LL} associated with every construct recognized by the compiler as a valid component of the program. The elements in the \textit{LL} of a definition or a declaration are symbols denoting types, constants and variables, as discussed in the previous subsection; the elements in the \textit{LL} of an application are lexical items whose types, scope, and extent may not yet be defined; the elements of the \textit{LL} of a definition that has not yet been transformed into a type descriptor are uninterpreted lexical items as well.

As suggested above, we implement the list \textit{LL} by associating it with every node of the abstract parse tree of the source language constructs when these constructs are recognized as syntactically valid components of the program. The structure of the objects in \textit{LL} is given in Figures 14 and 15. Notice however that the \textit{LL} of a declaration is a double-linked list of symbols whose type is already determined by the compiler as shown in Figure 14. Therefore, when a construct specified by a rule \( r \) whose \textit{standard} contains the attribute \textit{scope} is discovered, the \textit{LL} of the application part of this construct is updated using the \textit{LL} of the declaration part, transforming the leaves of the abstract parse tree of the application part into symbols. Let \textit{LL} (dec) and \textit{LL} (app) be the \textit{LL} lists of the declaration and application parts of the construct specified by \( r \). This transformation is actually a procedure for scope computation of the objects in \textit{LL} (dec) and is described by the algorithm in Figure 18. The compositional manner of the

for each \textit{symbol} \in LL(den)  
for each \textit{leaf} \in LL(app)  
  
  if \textit{leaf}.Name = \textit{symbol}.Name  
  
  \{  
  Parent.\text{Leaf.Child} := \textit{symbol};  
  \text{Remove(leaf, LL(den))};  
  \}  

Figure 18: Scope computation algorithm.

compiler operation assures that the scope of a name computed by this procedure is the application part of the smallest valid construct that contains a declaration of that name. Thus, this approach provides a general mechanism for symbol management where the global symbol table used by a conventional compiler is replaced with by a local symbol table \textit{LL} of each construct component.

When all elements in the \textit{LL} (app) of a construct are completely defined, the code generator of the compiler can start expanding the macro operations \textit{Macro1}, \textit{Macro2}, \ldots, thus generating the target images of the current valid construct in parallel with the other activities performed by other components
of the compiler.

The implementation of the exact computation rule is based on mapping each construct specified by rules having the attribute process within the standard of the rule specifying them into objects of type "process" and manipulating these objects as follows:

- The definition of an object of type "process" generates a process data representation of the form \(\langle \text{Name}, \text{ActivationRecord}, \text{Body} \rangle\). This tuple is linked as a new object of the type process in the TT database.

- The declaration of an object of type process, such as a function declaration in C, binds this object in the scope where it occurs to its process definition in the TT. A function (or a procedure) in Pascal is a definition, a declaration, and an application at the same time, and is treated by splitting it in clean components, as we discussed in previous subsection.

- The usage of an object of type process is mapped into an instantiation (in line or call) of its code. At this point a copy of the activation record is updated performing the action of parameter passing, and the process becomes schedulable, providing that its precedence relations are satisfied.

5.3 Compositional recognition of valid language constructs

Now that we know how an algebraic compiler manipulates the type, the scope, and the extent of the objects it handles, we can discuss the algorithm that allows the compiler to recognize the validity of its source language constructs in terms of the validity of their components. We use the term recognition rather than parsing to emphasize the difference between a conventional parsing algorithm, which can parse and analyze one single class of valid language constructs, usually called programs in a programming language, versus the recognizer \(\mathcal{R}\) of an algebraic compiler, which parses and recognizes any valid language construct of the language discovering its abstract parse tree and its class of equivalence. The compositional behavior of this algorithm is based on the capability of the algebraic compiler to handle classes of equivalence of language constructs as well as the constructs themselves. In other words, if \(r \in \mathcal{R}\), where \(r\) has the form \(A_0 = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n\), then the syntactic interpretation of \(r\) as an algebraic operation \(t_0 t_1 \ldots t_n : A_1 \times \ldots \times A_n \rightarrow A_0\) allows \(\mathcal{R}\) to interpret the string \(t_0 A_1 t_1 \ldots t_{n-1} A_n t_n\) as the valid language construct representative of the class of valid language constructs \([A_0]\) specified of the rule \(r\).

Each element of this class has the form \(t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n\) for \(w_i \in [A_i]\), \(1 \leq i \leq n\). Let us denote by \([A]_r\) the class of valid language constructs that belongs to \([A]\) and are specified by the rule \(r\). Then the right hand side of a specification rule \(r \in \mathcal{R}\) is interpreted by \(\mathcal{R}\) as the representative of the equivalence class \([\text{rhs}(r)]_r\).

Now we develop a procedure that for each \(r \in \mathcal{R}\), \(\text{rhs}(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n\), and string \(w = \alpha_0 [A_1] t_1 \ldots t_{n-1} [A_n] t_n\) \(\beta\), allows \(\mathcal{R}\) to decide if the portion \([A_i] t_i \ldots t_{n-1} [A_n] t_n\) of \(w\) is the representative of the class \([\text{lhs}(r)]_r\), i.e., if this portion of \(w\) is specified by the rule \(r\). This procedure is based on structuring the specification rules \(R\) as a language space \([\text{Rus84}]\) which allows us to preprocess the specification \(R\) and to compute the context set, \(C(r)\), and the noncontext set, \(N(r)\), for each \(r \in \mathcal{R}\), which have the following properties:

1. if \((x, y) \in C(r)\) and if \(w = \alpha x t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n y \beta\) is a valid construct of the language then the portion \(t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n\) of \(w\) is specified by the rule \(r\), that is, \(t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n \in [\text{lhs}(r)]_r\).

2. if \((x, y) \in N(r)\) and \(w = \alpha x t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n y \beta\) is a valid construct of the language then the portion \(t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n\) of \(w\) is not specified by \(r\), that is, \(t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n \notin [\text{lhs}(r)]_r\).

Various algorithms that compute the sets \(C(r)\) and \(N(r)\) for each \(r \in \mathcal{R}\) are described in \([\text{Rus83}, \text{Rus88}, \text{Rus94}]\). All these algorithms have as a basis, the computational definitions of the sets \(C(r)\) and \(N(r)\). To introduce these definitions we use the following notations, in addition to the notations used so far: \(V\) is the language vocabulary, i.e., \(V = V_T \cup V_N\), where \(V_T\) is the alphabet of all terminals used in the rules of \(R\) and \(V_N\) is the alphabet of all nonterminals used in the rules of \(R\); \(V^*\) is the collection of valid language constructs specified by the rules in \(R\), i.e., \(V^* = \{ w \in V^* | \exists r \in R, w \in [\text{lhs}(r)]_r \}\); for \(\alpha \in V^*\), \(|\alpha|\) is the length of \(\alpha\) and \(\alpha[k]\) is the \(k\)-th element of the string \(\alpha\).
Now we define the matching operation, $\equiv: V \times V \rightarrow \{true, false\}$ by the equality:

$$x \equiv y = \begin{cases} 
true, & \text{if } x = y; \\
true, & \text{if } x \in V_1, y \in V_n \land x \in [y]; \\
true, & \text{if } x, y \in V_n \land x \in [y]; \\
false, & \text{in all other cases.}
\end{cases}$$

The matching operation $\equiv$ is extended in [Rus94] to operate on strings, $\equiv: V^* \times V^* \rightarrow \{true, false\}$, and is used to provide the following computational definitions of the sets $C(r)$ and $N(r)$:

$$C(r) = \{(x, y) \in V^* \times V^* | \forall w \in V_A^r[[w = \alpha x \gamma y \beta \land \gamma \equiv \text{rhs}(r)] \Rightarrow \gamma \text{ is rhs}(r)]\}$$

$$N(r) = \{(x, y) \in V^* \times V^* | \forall w \in V_A^r[[w = \alpha x \gamma y \beta \land \gamma \equiv \text{rhs}(r)] \Rightarrow \gamma \text{ is not rhs}(r)]\}$$

Let $A(r) = C(r) \cap N(r)$ for each $r \in R$. If $A(r) = \emptyset$ for each $r \in R$ then $C(r)$ and $N(r)$ are disjoint sets and one of these sets (usually the smaller) suffices as the decision mechanism used by the recognizer $R$. If however $A(r) \neq \emptyset$ for some $r \in R$ it means that for each $(x, y) \in A(r)$ there are strings $w \in V_A^r$ of the form $w = \alpha x \text{rhs}(r) y \beta = \alpha' x \text{rhs}(r') y \beta'$ whose process of reduction to their equivalence class can continue either by replacing the occurrence of the rhs(r) in w by the lhs(r) or by the replacement of the occurrence of rhs(r') by lhs(r') where $r \neq r'$. In other words, if $A(r) \neq \emptyset$ the language is ambiguous. Knack [Kna94] has shown that one can determine relations among the rules in R which allow the compositional recognizer of an algebraic compiler to operate as desired even when the language is ambiguous.

One more ingredient is necessary in order to give a formal presentation of $R$. This is an ordering relation on $R$ that allows the algorithm to operate bottom-up. To define this ordering relation, for each $r \in R$ we denote by $Dom(r)$ the set of nonterminals that occur in rhs(r), i.e.,

$$Dom(r) = \{A_i | \text{rhs}(r) = \alpha A_i \beta\}$$

where $\alpha$ and $\beta$ may be $\epsilon$. With this notation we can decompose the specification set $R$ into the subsets $R_0, R_1, \ldots, R_m$, where $m$ is an intrinsic property of $R$, that allow the recognizer to operate in a bottom-up fashion:

1. $R_0 = \{r \in R | Dom(r) = \emptyset\}; \text{lhs}(R_0) = \{\text{lhs}(r) | r \in R_0\}$.
2. For each $i, 1 \leq i \leq m$, $R_i = \{r \in R \setminus \bigcup_{k=0}^{i-1} R_k | Dom(r) \subseteq \bigcup_{k=0}^{i-1} \text{lhs}(R_k); \text{lhs}(R_k) = \{\text{lhs}(r) | r \in R_k\}$.

Using this decomposition of $R$ and the sets $C(r)$, $N(r)$, and $A(r)$, for each $r \in R$, the compositional algorithm $R$ that recognizes constructs specified by $R$ in terms of their components can be formulated as shown in Figure 19. $Err_{\text{lhs}(r)}$ is the error symbol introduced by the recognizer in the input text as soon

$$\begin{align*}
&\mathrm{for} \ j = 0, 1, \ldots, m \\
&\quad \mathrm{for} \ i = 0, 1, \ldots, j \\
&\quad \quad \mathrm{for} \ r \in R_i \\
&\quad \quad \quad \mathrm{for} \ w = \alpha x \gamma y \beta \\
&\quad \quad \quad \quad \mathrm{if} (match(\gamma, \text{rhs}(r)) = \text{true}) \\
&\quad \quad \quad \quad \quad \mathrm{if} (x, y) \notin A(r) \mathrm{and} (x, y) \notin C(r) \mathrm{then} \ w \leftarrow \alpha x \text{lhs}(r) y \gamma; \\
&\quad \quad \quad \quad \quad \mathrm{else} \text{ if} (x, y) \notin A(r) \mathrm{or} (x, y) \notin C(r) \mathrm{then} \mathrm{do} \mathrm{nothing}; \\
&\quad \quad \quad \quad \quad \mathrm{else} \ w \leftarrow \alpha x \text{Err}_{\text{lhs}(r)} y \beta; \\
&\quad \quad \quad \quad \quad \mathrm{else} \ \mathrm{if} \ match(\gamma, \text{rhs}(r)) = \text{err} \ \mathrm{propagate} \ \mathrm{error}; \\
&\quad \quad \quad \quad \quad \mathrm{else} \ \mathrm{do} \ \mathrm{nothing}; \\
&\mathrm{if} \ w \in V_n \mathrm{Accept} \ \mathrm{else} \ \mathrm{Diagnose}(w)
\end{align*}$$

Figure 19: Gliding pattern matching algorithm for valid construct recognition.

as an error is discovered during the parsing activity. The error symbol is propagated in the text analyzed
by the recognizer by the axiom of errors associated with the definition of the matching operation \( \equiv \) in the form

\[
\forall A \in V_n[rhs(r)[i] \in V_n \land w[k] = Err_A \Rightarrow (w[k] \equiv rhs(r)[i] = true)]
\]

and using the predicate \( \text{match}() \) defined by the equality:

\[
\text{match}(\alpha, \beta) = \begin{cases} 
  \text{true}, & \text{if } (\alpha \equiv \beta); \\
  \text{err}, & \text{if } \exists k : 0 \leq k \leq \lambda(\alpha) \land \alpha[k] = Err_A, A \in V_n; \\
  \text{false}, & \text{otherwise.}
\end{cases}
\]

This algorithm is naturally parallel in the sense that all patterns \( rhs(r), r \in R \), can be searched in parallel in the input \( w \). The complexity of this algorithm is \( O(n) \) [Kna94], though depending upon \( R \) the constant of proportionality could be of the order of \( 10^6 \) [Rus88]. The average time of the algorithm for an input string of \( n \) symbols is \( n \cdot \log(n) \) and the worst case is the \( n^2 \).

To bring down the constant of proportionality and to reduce the average and the worst case behavior of \( R \) to linear time with the length of the input string we used the relationship between the rules in the classes \( R_0, R_1, \ldots, R_m \) to compute the constructive and repetition depths associated with the classes \( R_j, 0 \leq j \leq m \). The constructive depth of \( R_j \) is the class \( R_k \), for the smallest \( k \), such that \( \exists r \in R_j \land r' \in R_k \) and \( lhs(r') \in \text{Dom}(r) \). The repetition depth of \( R_j \) is the class \( R_i \), for the smallest \( i \), such that \( \exists r \in R_j \land r' \in R_i \) and \( lhs(r) \in \text{Dom}(r') \). The repetition depth allows us to control the pattern matching operation after a match with \( rhs(r), r \in R_j \) to the class \( R_i \), which is the repetition depth of \( R_j \), rather than to \( R_0 \), and thus we remove the loop by \( i \) from the algorithm. The constructive and repetition depths have been generalized in [Kna94] and lead to a new version of the above algorithm where patterns \( rhs(r) \) do not slide over the input \( w \), rather \( rhs(r) \) are directly positioned over that portion of \( w \) which can be potentially matched by \( rhs(r) \). This is obtained by constructing an ordering relation on \( R \) called \( \prec \) that assures that a construct specified by \( r_1 \) can be a component of a construct specified by \( r_2 \) only if \( r_1 \prec r_2 \). The relation \( \prec \) is defined as follows: \( r_1 \prec r_2 \Leftrightarrow lhs(r_1) \in \text{Dom}(r_2) \). Assume that for each \( r \in R \) we compute

for each \( r \in R_0 \\
  \text{for each } w = \alpha x y \beta \\
   \text{if } (\text{match}(\gamma, rhs(r)) = \text{true}) \\
   \text{if } (x, y) \notin A(r) \land (x, y) \notin C(r) \text{ then } w := \alpha x lhs(r) y \beta; \text{ Push}((r, \lambda(\alpha x)), B) \\
   \text{else if } (x, y) \in A(r) \lor (x, y) \notin A(r) \land (x, y) \in N(r) \text{ then do nothing;}

\text{else } w := \alpha x Err_{lhs(r)} y \beta;
\text{ else if } (\text{match}(\gamma, rhs(r)) = \text{err}) \text{ propagate error} \\
\text{ else do nothing;}
\text{ while } (B \neq \emptyset)
\text{let } (r', p) := \text{Pop}(B);
\text{for each } (r', k) \in \text{Next}(r')
\text{if } (\text{match}(w[p - k \ldots \lambda(rhs(r'))], rhs(r')) = \text{true}) \land w = \alpha x rhs(r') y \beta \\
\text{if } (x, y) \notin A(r') \land (x, y) \in C(r) \text{ then } w := \alpha x lhs(r') y \beta; \text{ Push}((r', \lambda(\alpha x)), B) \\
\text{else if } (x, y) \in A(r') \lor (x, y) \notin A(r') \land (x, y) \in N(r') \text{ then do nothing;}
\text{else } w := \alpha x Err_{lhs(r')} y \beta;
\text{ else if } (\text{match}(w[p - k \ldots \lambda(rhs(r'))], rhs(r')) = \text{err}) \text{ propagate error;}
\text{ else do nothing;}
\text{if } w \in V_n \text{ Accept else Diagnose}(w)

Figure 20: Direct pattern matching algorithm for valid language construct recognition.

the set \( \text{Next}(r) = \{(r', k) | r \prec r' \land rhs(r')[k] = lhs(r)\} \). Now we can organize rules \( R \) in two classes, \( R_0 = \{r \in R | \text{Next}(r) = \emptyset\} \) and \( R_1 = \{r \in R | \text{Next}(r) \neq \emptyset\} \). It is obvious that \( R_0 \cap R_1 = \emptyset \) and moreover, for each rule \( r \in R_0, \text{Dom}(r) = \emptyset \). Using the notation \( w[p..q] \) for the substring \( w[p]w[p+1] \ldots w[q] \) of the string \( w \), and \( B \) for a data structure of type bag, the compositional recognition algorithm \( R \) is described in Figure 20.
The first for loop of $\mathcal{R}$ is a parallel pattern matching algorithm that uses as patterns only rules from $R_0$ that are glided along the input string $w$ and therefore it is called the gliding part of the recognizer $\mathcal{R}$ and is denoted by $R_G$. Whenever a match is found, $R_G$ updates the input string as $\mathcal{R}$ did and in addition collects information in the bag $B$ to be used by the second part of the algorithm. The while loop is a parallel pattern matching algorithm that does not glide patterns over the string. Rather, it positions patterns over those portions of the input string $w$ where a match could potentially be found. Therefore this part of the algorithm is called the jumping part of the recognizer $\mathcal{R}$ and is denoted by $R_J$.

The implementation of this algorithm raises two problems. The first problems occurs when the substitution of a $\text{lhs}(r)$ for an occurrence of the string $\text{rhs}(r)$ in $w$ at position $p$ is required and $\lambda(\text{rhs}(r)) > 1$. Since in this case the string $w$ shortens, each tuple $(r, q) \in B$ where $q > p$, i.e., $q$ is a position in $w$ to the right of the position of $\text{lhs}(r)$, must be updated decreasing $q$ by the distance $q - p$ which corresponds with the shifting of the symbols in $w$, in order that they continue to point to the proper location. The solution is to not shorten $w$, but instead to fill it with a “delete” symbol that indicates that the original symbols have been deleted. Then all pattern-matching performed by $\mathcal{R}$ must skip these delete symbols. This eliminates the need to modify the information in the bag $B$. The second problem is raised by matching $w$ with a $\text{rhs}(r)$ where $\text{rhs}(r)$ contains more than one nonterminal occurring in it. For each nonterminal in $\text{rhs}(r)$, and consequently in $w$, there is one tuple $(r', k) \in B$ where $r' \in \text{Next}(r)$ and $w[k] = \text{lhs}(r)$. Assume that $\text{rhs}(r)$ has $k \geq 2$ nonterminals and a match is found. Then by reducing the matched portion of $w$ to the $\text{lhs}(r)$ there will be $k - 1$ tuples in $B$ pointing to (now nonexistent) nonterminals in $w$. The problem is how to locate and invalidate these tuples. Using the “delete” symbols in $w$ rather than reducing its matched portion the solution is simple: before attempting a match $R_J$ makes sure that the matched position does not contain a deleted symbol. If it does, $R_J$ aborts the match and continues with another tuple.

| $r_0$ | $E = T$   | $C(r_0) = \{(\$, +), (\$, \$)\}$ | $N(r_0) = \{(+, \$), (\$, \$)\}$ |
| $r_1$ | $E = E + T$ | $C(r_1) = \{(), +), (\$, \$)\}$ | $N(r_1) = \{(\$, +), (\$, \$)\}$ |
| $r_2$ | $T = F$   | $C(r_2) = \{(+, \$), (\$, \$)\}$ | $N(r_2) = \{(\$, \$)\}$ |
| $r_3$ | $T = T * F$ | $C(r_3) = \{(+, \$), (\$, \$)\}$ | $N(r_3) = \{(\$, \$)\}$ |
| $r_4$ | $F = id$  | $C(r_4) = \{(\$, \$), (\$, \$)\}$ | $N(r_4) = \{\}$ |

Table 1: Context and noncontext sets of $R_{exp}$.

To illustrate this algorithm we consider the specification $R_{exp}$ of the arithmetic expressions. Table 5.3 shows these rules together with their contexts and noncontexts sets. The symbol “$\$” represents the begin-of-string, when in the left element of a context pair or the end-of-string when found in the right element of the context pair. The behavior of the algorithm is shown in Table 5.3 using $w = id + id * id$

<table>
<thead>
<tr>
<th>Step</th>
<th>$w$</th>
<th>Tuple</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$id + id * id$</td>
<td></td>
<td>${}$</td>
</tr>
<tr>
<td>2.</td>
<td>$F + F * id$</td>
<td>${(1, r_4)}$</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>$F + F * id$</td>
<td>${(1, r_4), (3, r_4)}$</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>$F + F * F$</td>
<td>${(1, r_4), (3, r_4), (5, r_4)}$</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>$T + F * F$</td>
<td>${(1, r_4)}$</td>
<td>${(3, r_4), (5, r_4), (1, r_2)}$</td>
</tr>
<tr>
<td>6.</td>
<td>$T + T * F$</td>
<td>${(3, r_4)}$</td>
<td>${(5, r_4), (1, r_2), (3, r_2)}$</td>
</tr>
<tr>
<td>7.</td>
<td>$T + T * F$</td>
<td>${(5, r_4)}$</td>
<td>${(1, r_2), (3, r_2)}$</td>
</tr>
<tr>
<td>8.</td>
<td>$E + T * F$</td>
<td>${(1, r_2)}$</td>
<td>${(3, r_2), (1, r_0)}$</td>
</tr>
<tr>
<td>9.</td>
<td>$E + T \Delta \Delta$</td>
<td>${(3, r_2)}$</td>
<td>${(1, r_0), (3, r_3)}$</td>
</tr>
<tr>
<td>10.</td>
<td>$E \Delta \Delta \Delta$</td>
<td>${(1, r_0)}$</td>
<td>${(3, r_0)}$</td>
</tr>
<tr>
<td>11.</td>
<td>$E \Delta \Delta \Delta$</td>
<td>${(3, r_0)}$</td>
<td>${}$</td>
</tr>
</tbody>
</table>

Table 2: The behavior of $\mathcal{R}$.

as the input string. The column $w$ of this table contains the form of the string $w$ as it is transformed;
the column $B$ shows the contents of $B$; the column $Tuple$ displays the tuple most recently retrieved from $B$; the deleted symbol is denoted $\Delta$. Steps 1 through 4 are performed by $R_G$ and the steps 5 through 11 are performed by $R_f$. The symbols of $w$ are numbered from 1 to 5 (the 0th and the 6th symbol are the begin-of-string and end-of-string symbol $\$, not shown).

5.4 Compositional code generation

The code generation of an algebraic compiler is based on the assumption that the target language allows us to specify the target image of the source language constructs compositionally, by target macro-operations. This assumption is consistent with our view on programming languages and can be rephrased in algebraic terms by stating that the source language syntax algebra, $Syn_s$, can be embedded into the target language syntax algebra, $Syn_t$, by derived operations [Coh81, BL69]. Derived operations are used as an algebraic mechanism for code generation in an algebraic compiler based on the algorithms for (generalized) homomorphism computation as shown in [Hig64, Rus91]. From a practical viewpoint of engineering an algebraic compiler this statement can be further rephrased in programming terms requiring the target language to be provided with a macro-facility [WC93] that allows the compiler specifier to program the code generation activity of the compiler. Such a facility is obtained by assuming that the target language is provided with semantic macro-operations [Lee90, Rus90, Mad89] and with a macro-processor, $M$, provided with its own arithmetic and logic power.

Macro-operations are parameterized target language constructs associated with the specification rules $r \in R$, $rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$. We use the symbol $\otimes$, accompanied by indices if necessary, to represent formal parameters used by macro-operations, and denote by $M(r)(\otimes_1, \otimes_2, \ldots, \otimes_n)$ the macro-operation that specifies the target representation of the source language constructs of the class $[rhs(r)]$. For a source language construct $w \in [lhs(r)]$, $w = t_0 w_1 t_1 \ldots t_{n-1} w_{n-1} t_n$, $w_i \in \{A_i\}$, $1 \leq i \leq n$, and $T(w_i)$, $1 \leq i \leq n$, the target images of the source language construct components, $w_i$, $1 \leq i \leq n$, the macro-processor $M$ expands the macro-operation $M(r)(\otimes_1, \otimes_2, \ldots, \otimes_n)$ into a target language construct

$$T(w) = M(M(r)(\otimes_1, \otimes_2, \ldots, \otimes_n); (T(w_1), T(w_2), \ldots, T(w_n)))$$

called the target image of $w$, taking $T(w_1)$, $T(w_2)$, $\ldots$, $T(w_n)$ as actual parameters. However, $M$ does not perform a simple text replacement, as it does when it handles syntax macros. Rather, the actions performed by $M$ are semantically controlled in the sense that $M$ assumes that the type, the scope, the extent, and other semantic properties of $T(w_i)$, $1 \leq i \leq n$, are the target representations of the type, the scope, the extent, and other semantic properties of the $w_i$, $1 \leq i \leq n$, and that the type, the scope, the extent, and other semantic properties of $T(w)$ are correctly constructed according to the rules specified in $M(r)(\otimes_1, \otimes_2, \ldots, \otimes_n)$. That is, the target images of the source language constructs of the equivalence class $[lhs(r)]$, $rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$, are expressed in terms of the target images of the construct components of equivalent classes $\{A_1\}$, $\{A_2\}$, $\ldots$, $\{A_n\}$, respectively.

The difficulty in designing a semantic driven target macro-processor to be used as compiler code generator resides in the difference between the manner in which the semantics of source language constructs specified by the rules $r \in R$ is provided in the language specification and the manner in which it is discovered and managed by the conventional compiler. Conventional definitions of programming languages specify the semantics of valid language constructs by interpreting their specification rules as target operations (operational view), thus making the semantics of the source language dependent upon the target language of the compiler. This interpretation is further encapsulated in the compiler as actions (or attributes) to be evaluated by the compiler, when the constructs are recognized. Hence, this approach does not support the development of a machine independent compiler construction technology. The algebraic compiler on the other hand interprets each specification rule $r \in R$ as both a syntactic operator and a computation law constructing the semantic object denoted by the well-formed constructs specified by $r$, independent of any translation of these constructs. Thus, an algebraic compiler is based on a target independent relationship between the syntax and the semantics of the source language constructs. In addition, using the mechanism of derived operations to embed the source in the target, the algebraic compiler establishes a computable relationship between source language constructs and their equivalent target images. Hence, the algebraic compiler provides the mechanism for the development of a machine independent compiler construction technology.
The semantic properties of the source language and target language are uniformly manipulated by the algebraic compiler by associating them with the source and target specification rules as attributes of the equivalence classes of valid language constructs. However, semantic properties of the source language constructs are manipulated by the language recognizer and the semantic properties of their target images are manipulated by the code generator. The attributes expressing semantic properties of the source language constructs are similar with the attributes used by attribute grammars in the conventional compiler technology. But the compositional mechanism used by the algebraic compiler eliminates the need to provide explicitly many of these properties in the compiler specification and provide a framework for formal management of these attributes.

For each \( r \in R \) the attributes of the target language images specified by \( r \) are manipulated according to the syntax rules of the target language and the structure of the macro-operation \( M(r) \) explicitly associated with the source language specification rule \( r \). Thus each source language specification equation \( lhs(r) = rhs(r) \) is transformed into a compiler specification rule of the form \( lhs(r) = rhs(r); M(r) \). To focus on the use of the semantic properties of the target images, the compiler specification rule \( lhs(r) = rhs(r); M(r) \) can be written in the form:

\[
\begin{align*}
lhs(r) &= t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \\
P_1(\varepsilon_0) &= E_1(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) \\
P_2(\varepsilon_0) &= E_2(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) \\
\vdots \\
P_m(\varepsilon_0) &= E_m(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)
\end{align*}
\]

Here \( P_1, P_2, \ldots, P_m \) are semantic properties of the target language images of the constructs specified by \( r \in R \) and \( E_1, E_2, \ldots, E_m \) are well-formed parameterized expressions that are expanded by \( M \) into properties of valid target language constructs. For implementation reasons we use keywords to denote the attributes \( P_1, P_2, \ldots, P_m \). The expressions \( E_1, E_2, \ldots, E_m \) are constructed using three kinds of computation mechanisms supported by \( M \): functions that compute the values of semantic properties they take as arguments, pseudo-operations that allow the compiler constructor to control the actions performed by \( M \) while expanding macro-operations, and target language constructors such as definitions, declarations, applications, making up the target language expressions of the images they represent. Note, this mechanism of compiler specification allows multiple target image specifications to be associated with the same class of source language constructs. That is, the same recognizer \( R \) of the source language can control as many macro-processors \( M_1, M_2, \ldots, M_p \) as necessary. Therefore a complete compiler specification rule has the form

\[
lhs(r) = rhs(r); M_1(r), M_2(r), \ldots, M_p(r)
\]

The expressions \( E_1, E_2, \ldots, E_m \) that make up the body of a macro-operation \( M(r) \) must be computable in terms of the attributes of the target construct components denoted by the formal parameters \( \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n \). Since the target language of a compiler could be itself the source language of another compiler, the semantics attributes of the source language and target language are actually universal semantic properties of valid language constructs that allow both compositional program development by the programmer and compositional program recognition and generation by the compiler. Current programming methodology advocates the compositional program development as a requirement for modular, structured, object oriented, etc., programming, but due to the monolithic view of the program by conventional compiler this is actually not supported by the tools offered to computer users. Trying to develop a complete set of semantics properties to be used for automatic program generation from specification we have classified these properties in two groups: abstract properties and concrete properties.

The abstract properties are those which cannot be computed from the textual representation of the construct. Such properties are type, scope, extent, standard, and mode. Type, scope, and extent have been discussed in Section 5.3. The standard of a construct is an abstraction that allows the compiler specifier to instruct the compiler with respect to the gathering of structural information required for scope and extent computation as well as information pertaining to efficiency matters, such as generating parallel code and controlling the granularity of the parallel processes. The mode of an image is an abstraction that allows the compiler specifier to instruct the compiler with respect to the stage of this
image development (erroneous, partial, total), explicit flow of information among the components, and the portability and reusability of the image in the environment of other images.

The concrete properties are those which can be computed from the textual representation of the construct. Such properties represent resources manipulated by the constructs, objects imported and exported by the construct, and all sort of other textual properties of the construct that depend upon the graphic representation of the target language. For example, when an assembly language is the target such properties could be the section of assembly language code representing an image, machine registers used by that section of code, temporary variables used as labels in that section of code, the result location of the computation specified by the image, entry points, which are labels showing where the execution of the code can start, exit points, which are labels showing where the execution of the code can end. Other target languages may require the specification of other textual properties.

The compiler specification rule in Figure 21 illustrates the code generation of an expression in the assembly language of the IBM RS/6000 system. The attributes are denoted by boldface keywords, functions computing semantic properties are prefixed by $$. pseudo-operations controlling the behavior of macro-processor are prefixed by $$. “.”exit” is a label, and “.”a” is the mnemonic of the addition operation. In conclusion, the activity of code generation of an algebraic compiler is split between compiler specifier,

\[
E = E + T;
\]

standard: \{ Arithmetic-Expression \};
mode: \#if($mode(\text{O1}) = total \land \$mode(\text{O2}) = total) then total else partial;
type: \#if($type(\text{O1}) = $type(\text{O2})) then $type(\text{O1}) else Error;
rep: $\text{code}(\text{O1});$
$\text{code}(\text{O2});$
..exit a $\text{res}(\text{O1}), \text{res}(\text{O2}), \text{res}(\text{O1});$
$\text{FreReg}(\text{O2});$
.#MacEnd;
res: $\text{res}(\text{O1});$
entry: $\text{entry}(\text{O1});$
exit: ..exit

Figure 21: Semantic macro-operation in RS/6000 assembly language.

source language recognizer, and the target macro-processor as follows:

- Compiler specifier associates each source language specification rule \( r \), \( rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), with target language macro-operations, specifying the target images of the source constructs of the class \( [\text{lhs}(r)] \), specified by the rule \( r \) in terms of the target images of the construct components of classes \( [A_1], [A_2], \ldots, [A_n] \), respectively. This means that the compiler specifier programs the activity to be performed by the code generator. For a given programming language, \( PL \), the compiler specification rules are collected in the Compiler Specification File (PLCSF). Appendix D shows a compiler specification the implements the model checking algorithm [CES88], mapping CTL formulas into their satisfiability sets on a given model.

- The algorithm \( R \) recognizes as valid a portion of the source text using the rule \( r, rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), that has associated with it one or more macro-operations \( M(r)(\Theta_1, \Theta_2, \ldots, \Theta_n) \) where \( \Theta_1, \Theta_2, \ldots, \Theta_n \) are formal parameters representing target images of source language constructs of equivalence classes \( [A_1], [A_2], \ldots, [A_n] \), respectively, and calls the code generator to generate the code of the recognized construct.

- The code generator validates the target images \( T(A_1), T(A_2), \ldots, T(A_n) \) of the construct components and calls \( M \) to expand \( M(r)(T(A_1), T(A_2), \ldots, T(A_n)) \). Since \( R \) and \( M \) operate in parallel \( G \) also assures their correct synchronization, whenever necessary.

The compiler specification file, \( PLCSF \), is preprocessed by the language analysis system, \( LAS [RM, \text{Rus94}] \), that validates the structure of the compiler specification rules it contains, maps the macro-
operations to their internal form, computes the context, noncontext, and ambiguity sets, and finally reorganizes the compiler specification rules in PLCSF into the table of universal scheme of operations, TUSO. Each rule \( r \in \text{PLCSF} \) has an entry in TUSO which specifies completely the rule, its class for the recognizer, its successors, the macro-operations associated with it, and the context, noncontext, and ambiguity sets, as seen in Figure 22.

<table>
<thead>
<tr>
<th>Flag(r)</th>
<th>lhs(r)</th>
<th>rhs(r)</th>
<th>Next(r)</th>
<th>Macro(r)</th>
<th>C(r)</th>
<th>N(r)</th>
<th>A(r)</th>
</tr>
</thead>
</table>

Figure 22: Structure of a TUSO entry.

When the recognizer \( R \) implements the direct matching algorithm the entry \( Flag(r) \) is defined by

\[
Flag = \begin{cases} 
G, & \text{if the rule is used by } R_G; \\
J, & \text{if the rule is used by } R_J. 
\end{cases}
\]

When the recognizer \( R \) implements the gliding matching algorithm \( Flag(r) \) shows the class to which \( R \) belongs and \( Next(r) \) shows the repetition coefficient associated with \( r \). TUSO is further the heart of the algebraic compiler \( C = (R, \mathcal{G}, \mathcal{M}) \).

## 6 Integrating Components of an Algebraic Compiler

Let \( L_s = (\text{Sem}_s, \text{Syn}_s, \mathcal{L}_s : \text{Sem}_s \rightarrow \text{Syn}_s) \) and \( L_t = (\text{Sem}_t, \text{Syn}_t, \mathcal{L}_t : \text{Sem}_t \rightarrow \text{Syn}_t) \) be two programming languages, where \( L_s \) is a high-level language called the source language, and \( L_t \) is another language (machine language is not excluded) called the target language. For historical reasons the translator \( (H_s, T_s) : L_s \rightarrow L_t \) is called a compiler. The mapping \( T_s : \text{Syn}_s \rightarrow \text{Syn}_t \) is the syntax mapping also called the translator component of the compiler, and the mapping \( H_s : \text{Sem}_s \rightarrow \text{Sem}_t \) is the semantics component of the compiler. From a compiler implementation viewpoint, the semantics component of the compiler must exist and assure the commutativity of the communication diagram in Figure 1 and the translator component must exist, assure the commutativity of the communication diagram, and must be computable. Computer users use \( \text{Syn}_t \) to express their computing needs which are data and algorithms operating on this data in \( \text{Sem}_t \). When performed, these algorithms solve the particular problems for which they were designed. In order to use a machine, on which the language \( L_t \) is implemented, to perform the algorithms written in \( L_s \), a compiler \( (H_s, T_s) : L_s \rightarrow L_t \) that preserve the meaning of algorithms in \( L_s \) into \( L_t \) must be implemented. As observed above, the semantics mapping component \( H_s \) of this compiler is an abstraction that embeds the algebra \( \text{Sem}_s \) into the algebra \( \text{Sem}_t \); the translator component \( T_s \) of this compiler must be computable and must embed effectively the algebra \( \text{Syn}_s \) into the algebra \( \text{Syn}_t \) assuring the commutativity of the communication diagram in Figure 1. In other words, the images of \( \text{Sem}_s \) and \( \text{Syn}_s \) through \( H_s \) and \( T_s \), must be subalgebras of \( \text{Sem}_t \) and \( \text{Syn}_t \) respectively, isomorphic with the source algebras \( \text{Sem}_s \) and \( \text{Sy}_n \). The algebraic methodology for implementing the mapping \( T_s \) has been discussed in previous sections. Here we will show that \( H_s \) exists and that the components of the mapping \( T_s \) can be formally integrated into the algorithm performed by the algebraic compiler.

### 6.1 Components of a Conventional Compiler

There is no relationship between \( \text{Sem}_s \) and \( \text{Sem}_t \) and \( \text{Syn}_s \) and \( \text{Syn}_t \), respectively, in general. Therefore, the conventional design and implementation of the translator \( T_s \) consists of a sequence of the following transformations:

1. A source language parser \( P \) that recognizes valid programs \( p \in \text{Syn}_s \) and maps them into an intermediate form that can be further mapped into the machine language is designed. Usually \( P \) is an automaton generated by the context-free grammar defined by the specification rules of \( \text{Syn}_s \) and the intermediate form of \( p \in \text{Syn}_s \) is a tree, \( t_p \), whose nodes are labeled by semantics information such that \( t_p \) can be evaluated using the target machine.
2. A code generator $\mathcal{G}$ that for each $p \in \text{Syn}_n$ takes as data the intermediate form $t_p$ generated by $\mathcal{P}$ and the instruction set of the target machine and generates a machine language program $m_p$ evaluating $t_p$.

3. A collection of supporting tools such as preprocessors, lexical analyzers, semantics analyzers, type checkers, program restructures, optimizers, resource allocators, etc, that act on $t_p$ and $m_p$ performing all sort of checks, optimizations, restructuring, and resource management.

The algorithms performing the transformations $\mathcal{P}, \mathcal{G}$, and their supporting tools have been developed on different mathematical bases and evolved rather independently of each other. Therefore, it is difficult to integrate them into a unified framework that would allow us to truly automate the compiler construction and to prove the correctness of the resulting compiler. In addition, the usage of a context-free grammar to specify the well-formed constructs in $\text{Syn}_n$, restricts the class of source language valid constructs recognized by $\mathcal{P}$ to just one, the axiom (or the starting symbol) of the grammar which coincides with the program. In turn, this affects the degree of interaction between $\mathcal{P}$ and the programmer as well as the capability of $\mathcal{P}$ to support incremental development of programs. While we can adapt the parser $\mathcal{P}$ to recognize other constructs of $\text{Syn}_n$ as valid, the only executable construct is the program. But the incremental development of programs requires both program development as well as program execution to be incremental.

6.2 Algebraic Model of the Compiler

Limitations of the conventional methodology for the design and implementation of a compiler become more visible in the context of the dynamic evolution of the problem domain and the new parallel machines designed to tackle grand challenge problems raised by current science and technology. These limitations motivate the search for a unified framework that integrates all components of a compiler and frees the compiler from the restrictions imposed by context-free grammars. Since programming languages, as defined in this paper, are algebraic objects, we use the universal algebra as the framework for the development of an algebraic methodology for compiler design and implementation.

The algebraic methodology for designing a compiler $(H_t, T_t) : L_t \rightarrow L_t$, requires compiler implementer to establish first a mathematical relationship between $L_t$ and $L_t$ and then to use this relationship to define $H_t$ and $T_t$ as algebraic objects. One way of establishing such a relationship is by representing the operations defining the signature of $\text{Sem}_n$ and $\text{Syn}_n$ as derived operations in the algebras $\text{Sem}_n$ and $\text{Syn}_n$, respectively, such that the mappings $\mathcal{L}_t$ and $\mathcal{E}_t$ used in the language definition are preserved. Since $T_t$ is the only tangible component of the compiler we focus further on the implementation of this transformation by derived operations. For that we make the following assumptions about the source language $L_t$ and the target language $L_t$ of the compiler:

1. $L_t$ is specified by a finite set of specification rules, $\text{Spec}$, and is denoted by $L(\text{Spec})$. These specification rules can be split into three disjoint classes called lexicon specification rules, organized in the lexicon specification file $L.LSF$, type specification rules, organized in the type specification file, $L.TSF$, and construct specification rules, organized in the compiler specification file, $L.CSF$.

2. $L_t = (\text{Sem}_t, \text{Syn}_t, \mathcal{L}_t : \text{Sem}_t \rightarrow \text{Syn}_t)$ is provided with a macro-processor, $\mathcal{M}$, that allows compositional specification of its valid constructs by derived operations in $\text{Syn}_t$; semantic macro-operations discussed in Section 5.4 are examples of compositional specification of target language constructs by derived operations. Therefore the terms derived operation and macro-operation are further interchangeable.

3. There are two general transformations denoted $\text{Sem}$ and $\text{Syn}$ that map a finite set of specification rules, $\text{Spec}$, into two similar algebras $\text{Sem}(\text{Spec})$ and $\text{Syn}(\text{Spec})$ and allow the construction of the mappings $\mathcal{L}_{\text{Spec}} : \text{Sem}(\text{Spec}) \rightarrow \text{Syn}(\text{Spec})$, and $\mathcal{E}_{\text{Spec}} : \text{Syn}(\text{Spec}) \rightarrow \text{Sem}(\text{Spec})$ such that $\mathcal{E}_{\text{Spec}}$ is a homomorphism and the equality $\mathcal{L}_{\text{Spec}} \circ \mathcal{E}_{\text{Spec}} = \mathcal{I}_{\text{Sem}}$ holds, where $\mathcal{I}_{\text{Sem}}$ is the identity on $\text{Sem}(\text{Spec})$; that is, $\text{Sem}$ and $\text{Syn}$ specify the language $L(\text{Spec}) = (\text{Sem}(\text{Spec}), \text{Syn}(\text{Spec}), \mathcal{L}_{\text{Spec}} : \text{Sem}(\text{Spec}) \rightarrow \text{Syn}(\text{Spec}))$. $\text{Syn}$ is the universal construct in algebra that maps a signature into the ground-terms algebra generated by that signature; $\text{Sem}(\text{Spec})$ is determined by the language designer.
4. There is an operator \( \mathcal{D} : \text{Spec} \rightarrow \text{Syn} \) that maps each specification rule, \( r \in \text{Spec} \), into a derived operation of the target language, \( \mathcal{D}(r) \), such that for each source language construct \( w_r \) specified by \( r \) the equality \( H_t(E_{\text{Spec}}(w_r)) = E_t(D(r)(w_r)) \) holds.

5. For a specification \( \text{Spec} \), a target language \( L_t \), and a derived operator \( \mathcal{D} : \text{Spec} \rightarrow \text{Syn} \), the tuple \( \text{LS} = (\text{Spec}, \text{Sem}_t, \text{Syn}_t, L_{\text{Spec}}, E_{\text{Spec}}) \) is called the source language specification and the tuple \( \text{CS} = \{ (r, \mathcal{D}(r)| r \in \text{Spec} \} \) is called the compiler specification of a compiler that maps the language specified by \( \text{LS} \) into the given target language \( L_t \).

That is, assumptions (1) through (5) tell us how to specify an algebraic compiler such that the commutativity of the diagram in Figure 23 implied in this specification provides the correctness proof of the compiler thus specified.

![Compiler specification diagram](image)

**Figure 23: Compiler specification diagram.**

Each rule \( r \in \text{Spec} \) is assumed to be a BNF rule having the general form \( r : A \rightarrow t_0A_1t_1 \ldots t_{n-1}A_nt_n \) where \( A, A_1, \ldots, A_n \) are parameters called nonterminals and \( t_0, t_1, \ldots, t_n \) are fixed symbols called terminals. The left hand side of the rule \( r \) is \( lhs(r) = A \) and the right hand side of the rule \( r \) is \( rhs(r) = t_0A_1t_1 \ldots t_{n-1}A_nt_n \). From an algebraic point of view, \( r \) is interpreted as the signature of an operation \( t_0t_1 \ldots t_n : A_1 \times \ldots \times A_n \rightarrow A \). The transformation \( S_{\text{em}} \) interprets the parameters \( A, A_i, i = 1, \ldots, n \), as data types \([A], [A_i], 1 \leq i \leq n\), of the language environment, \( Env \), and the rule \( r \) as a heterogeneous operation \( t_0t_1 \ldots t_n : [A_1] \times \ldots \times [A_n] \rightarrow [A] \). The transformation \( S_{\text{yn}} \) interprets the parameters \( A, A_i, 1 \leq i \leq n \), as classes of valid constructs \([A], [A_i], 1 \leq i \leq n\), of the language called syntax categories and the rule \( r \) as a heterogeneous operation that takes as data valid constructs \( w_i \) of syntax categories \([A_i], 1 \leq i \leq n\), and builds as the result valid constructs \( w \) of syntax category \([A] \) by the rule \( t_0t_1 \ldots t_n(w_1, \ldots, w_n) = t_0w_1t_1 \ldots t_{n-1}w_nt_n \). The syntax categories defined by BNF rules \( r \) of the form \( r : A \rightarrow t \) are called lexical elements of the language. We assume that the finite set of syntax categories used in \( \text{Spec} \) are valid constructs in \( S_{\text{yn}}(\text{Spec}) \). Consequently, for each BNF rule \( r \), \( rhs(r) \) is a valid construct of syntax category \( lhs(r) \).

### 6.3 The Algebraic Compiler

The algebraic methodology for the implementation of the translator component \( T_s \) of a compiler \( (H_s, T_s) : L_s \rightarrow L_t \) consists of two steps. In the first step the following parallel activities are performed using their formal supporting tools:

- Develop the file \( L.TSF \) where the predefined types of the source language \( L_s \) and the type constructors supported by \( L_s \) are specified. The file \( L.TSF \) is mapped by \( TypeInit \) into the database \( L.TT \).
- 34 -

- Develop the file LLSF where the lexicon of the language \( L_a \) is specified by regular expressions of conditions in terms of the properties of the universal lexemes as seen in Section 3. Run the ScanGen on LLSF to obtain a scanner that maps any source language construct into the FIF shown in Figure 3.

- Develop the file LCSF which contains the specification of the valid source language constructs of the language \( L_t \) and run the language analysis system, LAS, on LCSF to validate it and to compute context, noncontext, and ambiguity sets. In parallel with this activity, for each specification rule in LCSF design a macro-operations that express in the language \( L_t \) the computation expressed by this rule in \( L_a \), as shown in Section 5.4. When LCSF is validated, context, noncontext, and ambiguity sets are computed, and each rule in LCSF has its target macro-operation associated with it, run again LAS, on LCSF to generate the table of universal scheme of operations, TUSO, whose entries are seen in Figure 22.

- Develop integrating procedures of the data structures LTT, Lexicon, and TUSO. These procedures examine the consistency of the Lexicon data constructed by ScanGen with the types names in LTT generated by TypeInit, and the lexical elements identified by LAS in the file LCSF.

The second step consists of running the integrating procedures develop above on the data structures LLexicon, LTT, and TUSO, on which all components of the compiler operate. These procedures are applied repeatedly until the data structures LLexicon, generated by ScanGen, LTT, generated by the TypeInit, and TUSO generated by LAS, are consistent.

The translator component of the compiler consists of the algorithm \( \mathcal{R} \), called the recognizer, the algorithm \( \mathcal{G} \), called the generator or the synthesizer, and the algorithm \( \mathcal{M} \), called the macro-processor, discussed in Section 5, that operate on the data structure FIF, are controlled by the data structure TUSO, and are mathematically integrated as follows:

1. The recognizer \( \mathcal{R} \) recognizes the source language constructs \( w = t_0 w_1 t_1 \ldots t_{n-1} w_n t_n \) specified by the BNF rules \( r : A_0 \rightarrow t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), where each \( w_i \) is a valid source language construct of syntax category \( A_i \), \( 1 \leq i \leq n \), and maps it into its syntax category \( A \). This is performed bottom-up, starting with the rules \( r \) such that \( \text{Dom}(r) = \text{emptyset} \), and continues according to the order relation defined by LAS on the rules in the LCSF file.

2. Let \( \mathcal{G}(w_i) \) be the target language images of the components \( w_i \), \( 1 \leq i \leq n \), of the construct \( w \) recognized by \( \mathcal{R} \) and \( D(r) \) be the target macro-operation (i.e., the derived operation) associated with the specification rule \( r \). Then the image of \( w \) in \( L_t \) is obtained by expanding the macro-operations \( D(r) \) using the images \( \mathcal{G}(w_i) \), \( 1 \leq i \leq n \), as the actual parameters, that is, \( \mathcal{G}(w) = \mathcal{M}(D(r)(\mathcal{G}(w_1), \ldots, \mathcal{G}(w_n))) \).

3. Let \( P \) be an input to the translator \( T_z \) of the compiler \( (H, T_z) : L_z \rightarrow L_t \). Assume that \( P \) has been tokenized by the scanner into the initial form in FIF (see Section 3). That is, each valid lexical component in \( P \) was tokenized by the scanner to its lexical name \( N_k \) and is associated with its target image denoted by \( s_k \), as seen in Figure 3, i.e., \( P \) has the following form in FIF:

\[
P = s_0 N_0 : @_0 s_1 N_1 : @_1 \ldots s_i N_i : @_i \ldots s_m-1 N_m : @_m s_m
\]

Here \( s_0, s_1, \ldots, s_m \) are empty words or are components of operator names \( t_0 t_1 \ldots t_n \) determined by rules \( r \in LCSF, r : A \rightarrow t_0 A_1 t_1 \ldots t_{n-1} A_n t_n \), used to generate \( P \) as a ground term of the algebra \( S_y_n(Spec) \), and \( N_0, N_1, \ldots, N_m \) are lexical tokens specifying constants and variables. During the process that recognizes \( P \) as a valid source language construct and that maps the recognized construct into the target image \( T_z(P) \), \( \mathcal{R} \), \( \mathcal{G} \), and \( \mathcal{M} \) interact using the source language specification rules \( r \) and the macro-operations \( D(r) \) specified by the operator \( \mathcal{D} \), by the following protocol:

(a) For each tuple \( (r : A \rightarrow t_0 A_1 t_1 \ldots t_{n-1} A_n t_n, D(r)) \) \( \text{GenR} \) interprets the \( \text{rhs}(r) \) as a pattern to be searched in \( P \) ignoring the target images embedded in \( P \). When an occurrence of the \( \text{rhs}(r) \) is discovered in \( P \), i.e., \( P = \alpha x t_0 A_1 t_1 \ldots t_{n-1} A_n t_n y \beta \), and the context \((x, y)\) surrounding \( \text{rhs}(r) \) in \( P \) determines that this portion of \( P \) is specified by \( r \), then that portion of \( P \) can be replaced by the \( \text{lhs}(r) \) preserving the syntactic validity of the \( P \), i.e., \( P \) is transformed into \( P' = \alpha x \text{lhs}(r) y \beta \).

This operation is denoted by \( \mathcal{R}(r) \).
(b) For each tuple \( (r : A \rightarrow t_0A_1 \ldots t_{n-1}A_nt_n, D(r)) \), \( \mathcal{G} \) interprets \( \text{rhs}(r) \) as the name of the derived operation \( D(r) \). Therefore, when \( \mathcal{R} \) determines that a portion of \( P \) can be replaced by the \( \text{lhs}(r) \), \( \mathcal{G} \) calls the macro-processor \( \mathcal{M} \) to expand the macro-operation \( D(r) \). Let \( \mathcal{G}_0 \) be the resulting code thus generated. Then \( \mathcal{G} \) associates the parameter \( \mathcal{G}_0 \) with the \( \text{lhs}(r) \) creating the record \( \text{lhs}(r) : \mathcal{G}_0 \). This operation is denoted by \( \mathcal{G}(r) \).

(c) When \( \mathcal{M} \) is called by \( \mathcal{G} \) with the arguments \( (r,t_0A_1 : \mathcal{G}_1t_1 \ldots t_{n-1}A_n : \mathcal{G}_nt_n) \) it expands \( \mathcal{D}(r)(\mathcal{G}_1, \ldots, \mathcal{G}_n) \) into the target image \( \mathcal{G}_0 \) of source syntax category \( \text{lhs}(r) \) and constructs the tuple \( \text{lhs}(r) : \mathcal{G}_0 \) that can be used by \( \mathcal{G} \) to replace the portion \( t_0A_1 : \mathcal{G}_1t_1 \ldots t_{n-1}A_n : \mathcal{G}_nt_n \) of the input \( P \). This operation performed by \( \mathcal{M} \) is denoted by \( \mathcal{M}(r) \).

The relationship between components \( \mathcal{R}, \mathcal{G}, \) and \( \mathcal{M} \) of the algebraic compiler while performing a transformation of the input text is shown in Figure 24.

\[
P = \alpha z t_0A_1 : \mathcal{G}_1t_1 \ldots t_{n-1}A_n : \mathcal{G}_nt_n y \beta
\]

**Figure 24**: The integration of the components of an algebraic compiler.

The algorithm performed by the translator \( T_r \) consists of a sequence of transformations \( T^0, T^1, \ldots, T^m \) of the source text as described by (a), (b), (c) above and shown in Figure 24. Each \( T^i, 1 \leq i \leq m \), takes source text already transformed by \( T^0, T^1, \ldots, T^{i-1} \), and applies the operation \( \mathcal{R}(r) \rightarrow \mathcal{G}(r) \rightarrow \mathcal{M}(r) \), \( r \in \mathcal{P}(LCSF) \), where \( \mathcal{P}(LCSF) \) is a partition of the compiler specification rules defined in Section 5.3. Note that \( \mathcal{R}(r), \mathcal{G}(r), \) and \( \mathcal{M}(r) \) may use (in parallel) all specification rules. The correctness of the algorithm performed by the translator \( T_r \) is assured by the unique extension lemma [BL69, Rus91] that can be expressed here as follows:

Let \( \text{Sym}_0 \) be the set of free generators of \( \text{Sym}_1 \). Then any function \( \mathcal{E}_0 : \text{Sym}_0 \rightarrow \text{Sem}_r \) can be uniquely extended to the homomorphism \( \mathcal{E} : \text{Sym}_1 \rightarrow \text{Sem}_r \).

The function \( \mathcal{E}_0 \) is implemented by \( \text{ScanGen} \) and \( \text{TypeInit} \). Further, the computability of the homomorphism \( \mathcal{E}_0 : \text{Sym}_1 \rightarrow \text{Sem}_r \) is assured by the finite-specification assumption and by the algorithm that is able to identify the generators of a given specification [Rus91]. That is, \( \text{Sym}_1 \) is the ground-term algebra specified by the finite set of BNF rules \( r \in \text{Spec} \) interpreted as signatures of heterogeneous operations. Thus, it is obvious that \( \text{Sym}_1 \) has a finite set of free generators. These are the constants and variables defined by the nullary operations in \( \text{Spec} \). They are associated with their token names in the data structures \( \text{FIF} \) on which the compiler operates. Furthermore, let \( A \rightarrow t \in \text{Spec} \) specifying the constant \( t \) of syntax category \( A \). From an algebraic viewpoint the function \( \mathcal{L} \) maps \( A \) into the data type \( [A] \) and the constant symbol \( t \) into \( t_0 \in [A] \). Therefore, we can define the image of \( t \) through \( \mathcal{E}_0 \) by the equality \( \mathcal{E}_0(t) = t_0 \). Then \( \text{Sym}_0 = \{ \text{rhs}(r) | r \in R_0 \} \) where \( R_0 = \{ t \in \text{Spec} | \text{Domain}(r) = \emptyset \} \). Further, \( \forall r \in R_0 \land \text{rhs}(r) \in \text{Sym}_0 \) we define \( \mathcal{E}_0(\text{rhs}(r)) = \mathcal{L}(\text{rhs}(r)) \). Having in view that the operator \( \mathcal{D} : \text{Spec} \rightarrow \text{Sym}_1 \) maps constants defined in \( \text{Sym}_0 \) into their images in \( \text{Sym}_1 \) and makes the diagram in Figure 28 commutative, and the unique extension of the function \( \mathcal{E}_0 \) to the homomorphism, the algorithm performed by \( T_r \) carry out the computation performed by \( \mathcal{E}_0 \) in \( \text{Sym}_1 \) as an equivalent computation performed by \( \mathcal{E}_1 \) in \( \text{Sym}_1 \).
contributed to the design and the implementation of the tools supporting the language independent lexicon development and language independent construct specification; Eric Van Wyk contributed to the design and implementation of the tools supporting language independent construct specification and used these tools to implemented the CTL model checking algorithm; Anil Dutt Goje contributed to the design and implementation of the language independent type system.

REFERENCES


A  Rules for the LSF specification language

(LSF) = (Specs);
(Specs) = (Cseq) (Eq);
(Cseq) = "case_sensitive" "=" (YesNo) ";";
(YesNo) = "yes" | "no";
(Eq) = (Eq) | (Eq) (Eq);
(LHS) = "=" (Desc) | (LHS) "=" (Desc) (Semantics) | (LHS) "=" (Desc) (Context) | (LHS) "=" (Desc) (Context) (Semantics);
(Desc) = (Delims) | (Body); end_RE
(Delims) = "self"; do_self_split
(Body) = "where" ";" (Expr);
(Expr) = (Expr1); (Expr) ";" (Expr1); RE.do_choice
(Expr1) = (Expr2); (Expr1) (Expr2); RE.do-compose
(Expr2) = (Expr3); (Expr3) ";"; RE.do_star
(Expr3) = "any" | "(" (Expr) ")" | (Cond); RE.do_cond
(Cond) = (ClassSpec); | (ClassSpec) "and" (AExpr) | ";" (Cond)
(ClassSpec) = (Class); | "token" "=" (Class);
(Class) = "integer" | "a" | "f" | "w" | "u" | "c"; hold_class
(AExpr) = (AExpr1); | (AExpr) "or" (AExpr1); do_cond_or
(AExpr1) = (AExpr2); | (AExpr1) "and" (AExpr2); do_cond_and
(AExpr2) = "lex" (Rel) "#string"; lex_string
(AExpr2) = "len" (Rel) "#int"; len_int
(AExpr2) = "lin" (Rel) "#int"; line_int
(AExpr2) = "col" (Rel) "#int"; column_int
(AExpr2) = "(! (AExpr) ")" | "not" (AExpr2);
(Cond) = "true" | "false" | "true" | "false"; hold_relation
(Context) = "context" ";" (CxtList); | "context" ";" (CxtList);
(CxtList) = (LftRt); | (LftRt) ";" (CxtList);
(LftRt) = "(" (LftRt) ")"; (RtExpr) ");"
(RtExpr) = "none" | "bool" | (Expr);
(Semantics) = "action" ";"; "adds"; | "action" ";"; "nolist"

B  Partial specification for FORTRAN lexicon

comment = begin: Tok = I ∧ Lex = "C" ∧ Col = 1
body: ( any ) *
end: Tok=U ∧ Lex=10;
action: noif
seq_num = body: Tok=N ∧ Col > 72
action: noif
label = Tok=N ∧ Col < 6
action: adds
power_op = Tok=O ∧ Lex="*" Tok=O ∧ Lex="*"
concat = Tok=O ∧ Lex="/" Tok=O ∧ Lex="/"
rel_op = ( Tok=O ∧ Lex="" Tok=I ∧ Lex="EQ" Tok=O ∧ Lex="" )
          | ( Tok=O ∧ Lex=" " Tok=I ∧ Lex="NE" Tok=O ∧ Lex="" )
C

PARTIAL TYPE SPECIFICATION FOR A FORTRAN TO C COMPILER

TypeSet
S1 = (real, double-precision, complex), S2 = (integer, real),
S3 = (integer, double precision, complex), S4 = (integer, real, double precision),
S5 = (integer, real, complex), S6 = (integer, real, double precision, complex)

TypeSpec
integer = kind: predefined
template: declaration = int @
operations: (=, S1, (int)), (+, integer, +), (-, integer, -), (+, integer, integer, +)
(-, integer, integer, -), (*, integer, integer, *), (/ integer, integer, /), (**, integer, integer, exp())
parent: real

real = kind: predefined
template: declaration = float @
operations: (=, S3, (float)), (+, real, +), (-, real, -), (+, S2, S2, +), (-, S2, S2, -),
(*, S2, S2, *), (/ S2, S2, /), (**, S2, S2, exp())
parent: double-precision
child: integer
double-precision = kind: predefined
template: declaration = double @
operations: (=, S5, (double)), (+, double-precision, +), (-, double-precision, -), (+, S4, S4, +),
(-, S4, S4, -), (*, S4, S4, *), (/ S4, S4, /), (**, S4, S4, exp())
child: real
complex = kind: predefined
template: definition = typedef struct C-definition ComplexDef
declaration = ComplexDef @, application = @ @1 @ @2 -> @1 @ @2
operations: (=, S4, (ComplexDef)), (+, complex, unaryadd()), (-, complex, unarysub()),
(+, S6, S6, complexadd()), (-, S6, S6, complexsub()), (*, S6, S6, complexprod()),
(/, S6, S6, complexdiv()), (**, S6, S6, complexexp())
logical = kind: predefined
D THE SPECIFICATION OF THE MODEL CHECKER ALGORITHM

CTLform =$\equiv$ "$u$" ["$u$" CTLform "$u$"

let $Z$, $Z'$ be sets;
$Z := \emptyset$ ; $Z' := \emptyset$ ;
while ($Z \neq Z'$) do $Z := Z'$ ; $Z' := Z' \cup \{ r \in S | r \in \@1 \land \text{successors}(r) \subseteq Z \}$ od
$\@0 := Z$ ;

CTLform =$\equiv$ "$e$" ["$e$" CTLform "$e$"

let $Z$, $Z'$ be sets;
$Z := \emptyset$ ; $Z' := \emptyset$ ;
while ($Z \neq Z'$) do $Z := Z'$ ; $Z' := Z' \cup \{ r \in S | r \in \@1 \land \text{successors}(r) \cap Z \neq \emptyset \}$ od
$\@0 := Z$ ;

CTLform =$\equiv$ "ax" Form ;
$\@0 := \{ r \in S | \text{successors}(r) \subseteq \@1 \}$ ;

CTLform =$\equiv$ "ex" Form ;
$\@0 := \{ r \in S | \text{successors}(r) \cap \@1 \neq \emptyset \}$ ;

CTLform =$\equiv$ Form ;
$\@0 := \@1$ ;

Form = Form "or" Expr ;
$\@0 := \@1 \cup \@2$ ;

Form = Expr ;
$\@0 := \@1$ ;

Expr = Expr "and" Term ;
$\@0 := \@1 \cap \@2$ ;

Expr = Term ;
$\@0 := \@1$ ;

Term = "not" Fact ;
$\@0 := S \setminus \@1$ ;

Term = Fact ;
$\@0 := \@1$ ;

Fact = "(" CTLform ")" ;
$\@0 := \@1$ ;

Fact = "ap" ;
$\@0 := P(\ ap )$ ;

Fact = "true" ;
$\@0 := S$ ;

Fact = "false" ;
$\@0 := \emptyset$ ;
POLYMORPHIC SYNTAX DEFINITION
(Extended Abstract)

Eelco Visser
Programming Research Group, University of Amsterdam,
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
email: visser@fwi.uva.nl, http://adam.fwi.uva.nl/~visser/

ABSTRACT

Context-free grammars can be used in algebraic specification instead of first-order signatures to define the structure of algebras. The rigidity of these first-order structures enforces a choice between strongly typed structures with little genericity or generic operations over untyped structures. Two-level signatures provide a better balance between genericity and typing. Two-level grammars are the grammatical counterpart of two-level signatures. The paper discusses generic polymorphic syntax definition in context-free grammars and two-level grammars and investigates the problems for the practical usage of two-level grammars as signatures in algebraic specification formalisms.

1 INTRODUCTION

Languages are algebras. A sentence, program or expression in a language is an object of its algebra. The constructs for composition of expressions from smaller expressions and the operations that interpret, translate, transform or analyze expressions are the operations of the algebra. Algebraic specifications describe algebras. Algebraic specifications are finite structures that describe the sorts of the algebra, its operations and the relations between the operations. Because a specification is finite there is always more than one algebra that fits the description; a specification can not exactly describe the intended algebra, but can only approximate it. Likewise, not all algebras can be described by algebraic specifications. Grammars describe languages. Grammars are finite structures that describe the syntactic categories of a language and the sentences of its categories. Like algebraic specifications, grammars only approximate the language they intend to describe.

First-order algebraic specifications consist of a first-order signature and a set of equations over the terms generated by the signature. A first-order signature consists of a finite set of sorts and a finite number of operations over those sorts. Context-free grammars and first-order signatures generate the same class of algebras; Goguen et al. (1977) show that parse trees or abstract syntax trees can be considered as terms over a signature and that the language of terms over a signature can be described by a context-free grammar. This correspondence is exploited in several algebraic specification formalisms by allowing the use of signatures with mix-fix operators (Putatsugi et al., 1985; Bidoit et al., 1989) or even arbitrary context-free grammars (Heering et al., 1992) instead of just prefix function signatures. This provides concrete notation for functions and constructors in data type specifications and it enables definition of operations on programming languages directly in their syntactic constructs.

The rigidity of first-order signatures and context-free grammars makes it difficult to generically describe properties of an algebra. For example, an algebra with lists of integers and lists of strings can be specified with a first-order signature by declaring a sort LI (list of integers) and a sort LS (list of strings) and by defining operations like the empty list, cons, head, tail and concatenation on both sorts. However, if these list sorts have the same properties independent of the contents of the lists for some, or all, operations, this can not be expressed in a first-order specification. Similarly, if for both list sorts an operation exists that applies a function to each element of a list, this can not be expressed in a generic way in a first-order specification.

A higher type algebra (Meinke, 1992b) is an algebra with an algebraic structure imposed on the set of sorts. These sort operators are interpreted as functions from collections of carrier sets to collections of carrier sets. For instance, the sorts LI and LS above can be seen as sorts constructed from the sorts I (integer) and S (string) by the
sort operator $L$. In such algebras more generic statements about (classes of) objects of the algebra can be made. For example, one can say that the tail function has type $Lx$ to $Lz$ for $z$ equal to $I$ or $S$. One could say that higher type algebras provide a higher resolution in the sort space of an algebra. Algebraic specifications in higher types (e.g., Poiné, 1986; Meinke, 1992a; Hearn and Meinke, 1994) describe higher type algebras by means of two (or more) levels of signatures. Each level specifies the sort operations for the next level. The terms over the signature at one level are the sorts of the signature at the next level. If variables are allowed in terms, polymorphic functions, uniformly ranging over many sorts, can be declared. The grammatical counterpart of higher type algebraic specifications with two-levels are two-level grammars. The connections are summarized by the diagram in figure 1.

In this paper we discuss polymorphic syntax definition by means of context-free grammars and two-level grammars. Section 2 contains a review of first-order signatures, context-free grammars and their correspondence and gives some examples of generic programming with context-free grammars. Section 3 defines two-level grammars and illustrates how these can be used for polymorphic syntax definition. Section 4 defines the parsing problem for two-level grammars and section 5 discusses several open problems.

2 Signatures and Grammars

In this section we review the use of context-free grammars in the algebraic specification of languages and data-types. 2.1–2.4 are roughly based on Goguen et al. (1977).

2.1 Definition A many-sorted signature $\Sigma$ is a pair $(S(\Sigma), F(\Sigma))$ where $S(\Sigma) \subseteq S$ is a set of sort names and $F(\Sigma) \subseteq O \times S(\Sigma)^*$ a set of function declarations (with $S$ and $O$ some sets of symbol names and operation names, respectively). We write $f : \tau_1 \rightarrow \tau_2$ if $(f, \tau_1 \tau_2) \in F(\Sigma)$ for some $\tau_1 \in S(\Sigma)^*$ and $\tau_2 \in S(\Sigma)$. $\Sigma \cup \tau$ is the extension of a signature $\Sigma$ with a $S(\Sigma)$ indexed family of sets of variables $V(\tau)$. The class of all signatures is denoted by SIG. The $S(\Sigma)$ indexed family $\text{Tree}(\Sigma)$ of well-formed terms over signature $\Sigma$ is defined by the inference rules in the left part of table 1 such that $t \in \text{Tree}(\Sigma)(\tau)$ iff $\Sigma \vdash t : \tau$.

A $\Sigma$-algebra $A$ is an $S(\Sigma)$ indexed family of carrier sets $A(\tau)$ and an assignment of each $\Sigma$ function $f : \tau_1 \rightarrow \tau_2$ to an $A$ function $f_A : A(\tau_1) \rightarrow A(\tau_2)$ such that $f_A(a) \in A(\tau_2)$ if $a \in A(\tau_1)$. $\text{Alg}(\Sigma)$ denotes the collection of all $\Sigma$-algebras. Note that $\text{Tree}(\Sigma)$ is an initial algebra in $\text{Alg}(\Sigma)$; there is a unique homomorphism $h : \text{Tree}(\Sigma) \rightarrow A$ for any $A \in \text{Alg}(\Sigma)$.

2.2 Definition A context-free grammar $G$ is a pair $(S(G), P(G))$ with $S(G) \subseteq S$ a set of symbols and $P(G) \subseteq S(G)^+ \times$ a set of productions. We write $\tau_1 \rightarrow \tau_2$ for a production $\tau_1 \tau_2 \in P(G)$. Note that productions are reversed in order to make them look like function declarations in a signature (conventionally a production $\tau_1 \rightarrow \tau_2$ is written as $\tau_2 \rightarrow \tau_1$ or $\tau_2 := \tau_1$). $G \cup V$ is the extension of a grammar with variables. A symbol $\tau_2 \in S(G)$ is a terminal symbol in $G$ (term($G$)($\tau_2$)) if there is no production $\tau_1 \rightarrow \tau_2 \in P(G)$. The class of all context-free grammars is denoted by CFG. The $S(G)$ indexed family $\text{Tree}(G)$ of parse trees over
symbol: \[ \tau \in S(\mathcal{G}) \]
\[ \mathcal{G} \vdash \tau : \tau \]

terminal symbol: \[ \tau \in S(\mathcal{G}), \text{ter}(\mathcal{G})(\tau) \]
\[ \mathcal{G} \vdash_{\text{ter}} \tau : \tau \]

variable: \[ x \in \mathcal{V}(\tau) \]
\[ \Sigma \cup \mathcal{V} \vdash x : \tau \]
\[ f : \tau_1 \rightarrow \tau_2 \in \mathcal{F}(\Sigma), \Sigma \vdash t : \tau_1 \]
\[ \Sigma \vdash f(t) : \tau_2 \]

application: \[ p = \tau_1 \rightarrow \tau_2 \in \mathcal{P}(\mathcal{G}), \mathcal{G} \vdash t : \tau_1 \]
\[ \mathcal{G} \vdash [p\ t] : \tau_2 \]

empty list: \[ \Sigma \vdash e : \epsilon \]
\[ \mathcal{G} \vdash e : \epsilon \]

concatenation: \[ \Sigma \vdash t_1 : \tau_1, \Sigma \vdash t_2 : \tau_2 \]
\[ \Sigma \vdash t_1 t_2 : \tau_1 \tau_2 \]
\[ \mathcal{G} \vdash t_1 t_2 : \tau_1 \tau_2 \]

| Table 1: Tree construction rules for signatures (left) and context-free grammars (right). |

Grammar \( \mathcal{G} \) is defined by the inference rules in the right part of table 1 such that \( t \in \text{Tree}(\mathcal{G})(\tau) \) iff \( \mathcal{G} \vdash t : \tau \). The set \( \text{Tree}(\mathcal{G}) \) includes partial parse trees that have non-terminal symbols as leaves. The set \( \text{Tree}_{\text{ter}}(\mathcal{G}) \) contains only parse trees with terminal symbols as leaves and is defined by using the rule labeled ‘terminal symbol’ in table 1 instead of the rule labeled ‘symbol’.

Partial parse trees are useful, for example, to formalize replacement behavior of structure editors and to formalize sentential forms, strings containing non-terminals. It is a matter of perspective whether one uses partial parse trees or terminal trees only. For semantics we usually do not intend to have an extra constant for each sort.

The similarity of the two sets of term construction rules in table 1 suggest that the tree structures generated by signatures and context-free grammars are isomorphic.

2.3 Proposition There are mappings \( \text{sig} : \text{CFG} \rightarrow \text{SIG} \) and \( \text{grm} : \text{SIG} \rightarrow \text{CFG} \) such that \( \text{Tree}(\text{sig}(\mathcal{G})) \equiv \text{Tree}_{\text{ter}}(\mathcal{G}) \) and \( \text{Tree}_{\text{ter}}(\text{grm}(\Sigma)) \equiv \text{Tree}(\Sigma) \).

Proof. Take the definitions of \( \text{grm} \) and \( \text{sig} \) in table 2. It is clear that \( i_{\text{grm}} \) and \( i_{\text{sig}} \) are isomorphisms.

Context-free grammars can thus be used to specify algebras. Parse trees are the canonical representations of the elements of these algebras.

However, we use grammars to represent and manipulate these structures as strings.

2.4 Definition The language \( L(\mathcal{G}) \) generated by a context-free grammar \( \mathcal{G} \) is the \( S(\mathcal{G}) \) indexed family of strings such that \( L(\mathcal{G})(\tau) = \text{yield}(\text{Tree}(\mathcal{G})(\tau)) \), where the function \( \text{yield} : \text{Tree}(\mathcal{G}) \rightarrow S(\mathcal{G})^* \) is defined by

\[ \text{yield}(\tau) = \tau \]
\[ \text{yield}(\epsilon) = \epsilon \]
\[ \text{yield}([p\ t]) = \text{yield}(t) \]
\[ \text{yield}(t_1 t_2) = \text{yield}(t_1) \cdot \text{yield}(t_2) \]

and applied to a set of trees denotes the pointwise extension to sets. The parse function \( \Pi(\mathcal{G}) : S(\mathcal{G})^* \rightarrow \mathcal{P}(\text{Tree}(\mathcal{G})) \) maps a string of symbols to a sub-family of \( \text{Tree}(\mathcal{G}) \) such that \( \Pi(\mathcal{G})(w)(\tau) = \{ t \in \text{Tree}(\mathcal{G})(\tau) \mid \text{yield}(t) = w \} \).

2.5 Discussion Note that \( L(\mathcal{G}) \) is an element of \( \text{Alg}(\mathcal{G}) \) in which all trees with the same yield are identified. In case of ambiguous grammars this is usually not intended. Disambiguation methods are used to map strings to the correct tree. With such methods algebraic properties do not apply to the strings used to denote trees. For example, (in arithmetic) the composition of the strings \( x \) and \( y - z \) does not correspond with the composition of their trees, i.e., \( x - (y - z) \), but with \( (x - y) - z \), which have usually different semantic interpretations. In the sequel we will assume that we are dealing with such grammars that we can use strings to denote trees. In examples we use a simple method for disambiguation by priority and associativity declarations.
\[
S(\text{grm}(\Sigma)) = S(\Sigma) \cup \{ f \mid f : \tau_1 \rightarrow \tau_2 \in F(\Sigma) \} \cup \{(,)\}
\]
\[
P(\text{grm}(\Sigma)) = \{ f(\tau_1) \rightarrow \tau_2 \mid f : \tau_1 \rightarrow \tau_2 \in F(\Sigma) \}
\]
\[
i_{\text{grm}} : \text{Tree}(\Sigma) \rightarrow \text{Tree}_{\text{ter}}(\text{grm}(\Sigma))
\]
\[
i_{\text{grm}}(f(t)) = [f(\tau_1) \rightarrow \tau_2 \ f(i_{\text{grm}}(t))]
\]
\[
i_{\text{grm}}(t_1\ t_2) = i_{\text{grm}}(t_1)\ i_{\text{grm}}(t_2)
\]
\[
S(\text{sig}(\Gamma)) = S(\Gamma)
\]
\[
P(\text{sig}(\Gamma)) = \{ p : \tau_1 \rightarrow \tau_2 \mid p = \tau_1 \rightarrow \tau_2 \in P(\Gamma) \} \cup \{ \tau : \tau \ \text{ter}(\Gamma)(\tau) \}
\]
\[
i_{\text{sig}} : \text{Tree}(\Gamma) \rightarrow \text{Tree}(\text{sig}(\Gamma))
\]
\[
i_{\text{sig}}(\tau) = \tau
\]
\[
i_{\text{sig}}(p\ t) = p(i_{\text{sig}}(t)) \quad \text{where} \quad p = \tau_1 \rightarrow \tau_2
\]
\[
i_{\text{sig}}(t_1\ t_2) = i_{\text{sig}}(t_1)\ i_{\text{sig}}(t_2)
\]

Table 2: Translation of signatures to grammars and from grammars to signatures.

2.6 Discussion As concrete syntax for grammars in examples we adopt the style of the syntax definition formalism SDF (Heering et al., 1992). The basic syntax of grammars in this formalism is given by the following module CFG-Syn, which is itself an SDF definition:

```
module CFG-Syn

sorts Symbol Production Grammar

context-free syntax

Symbols "->" Symbol -> Production
"sorts" Symbol
"syntax" Production* -> Grammar
```

It is clear that trees over this grammar correspond to CFGs as defined above, i.e., there is an isomorphism \text{Tree}(\text{CFG-Syn})(\text{Grammar}) \rightarrow CFG. We will use some extra ingredients in addition to this basis. Context-free and lexical syntax indicate two separate classes of productions in a grammar. The symbols in the left-hand side of a context-free production can be separated by layout symbols (whitespace and comment) while lexical symbols are not separated by layout. Literals are strings of characters between double quotes that denote the symbol consisting of the characters without the double quotes. When typeset, the characters of a literal might be changed. For instance, in the grammar above, the literal "->" is written \(\rightarrow\) in an actual production. Character classes are enumerations of sets of characters, e.g. [a-z] denotes the lowercase letters. Literals and character classes used in the grammar are implicitly declared as symbols. We also use modules to name and later refer to grammars. A formal algebraic specification of these features as extensions to CFG-Syn is defined in Visser (1995).

2.7 Example We study a typical example of a generic untyped language defined to support very general operations. The cost of this generality is a leaky grammar that defines much more sentences than are actually intended; terms over the language have to be ‘type’ checked to verify their consistency.

The following grammar of generic applicative terms (ATerms) is defined by Klint (1994) to represent parse trees and abstract syntax trees over arbitrary grammars. Literal strings are the basic terms, \([T_1\ T_2]\) denotes application of \(T_1\) to \(T_2\), ‘nil’ denotes the empty list and \(T_1\ ;\ T_2\) denotes the concatenation of \(T_1\) and \(T_2\).

```
module ATerms

imports Literals

sorts ATerm

context-free syntax

Literal \rightarrow ATerm
"[" ATerm ATerm "]" \rightarrow ATerm
ATerm ":;" ATerm \rightarrow ATerm \{right\}
"nil" \rightarrow ATerm
"(" ATerm ")" \rightarrow ATerm \{bracket\}
```

The following proposition shows how this language can be used to represent parse trees over arbitrary grammars. (Note that we use the concrete syntax of ATerms to represent elements of \text{Tree}(\text{ATerms}).)

2.8 Proposition For any CFG \(\Gamma\), there is a homomorphism \(\gamma_{\rightarrow}\) : \text{Tree}(\Gamma) \rightarrow \text{Tree}(\text{ATerms})\) such that \text{Tree}(\Gamma) is isomorphic with its \(\gamma_{\rightarrow}\) image in ATerms, i.e., \(\text{Tree}(\Gamma) \cong \gamma_{\rightarrow}(\text{Tree}(\Gamma))\).

**Proof.** Given some CFG \(\Gamma\) define the homomorphisms \(\gamma_{\rightarrow}\) and their (partial) inverses \(\downarrow\) as in table 3. Now we have that, for any \(t \in \text{Tree}(\Gamma)\), \(\downarrow \gamma_{\rightarrow}(t) = t\) and \(\downarrow\) is a homomorphism of type \(\gamma_{\rightarrow}(\text{Tree}(\Gamma)) \rightarrow \text{Tree}(\Gamma)\). Therefore, \(\text{Tree}(\Gamma) \cong \gamma_{\rightarrow}(\text{Tree}(\Gamma))\). \(\square\)
Table 3: Translation of parse trees to ATerms and back.

As a result, any sentence in a context-free language can be represented as a string in the fixed language of ATerms. For example, the parse tree for the string 'not false' according to the usual grammar for Boolean expressions is translated as follows:

\[(\neg \text{BOOL} \rightarrow \text{BOOL}) \rightarrow \text{not } ([\text{false} \rightarrow \text{BOOL}] \text{ false})\]

\[\left[\text{"prod" } [\text{"sym" } [\text{"sym" } \text{ not}]];\right.\]
\[\left. [\text{"sym" } \text{ BOOL}];\right.\]
\[\left[\text{"sym" } \text{ not}];\right.\]
\[\left[\text{"prod" } [\text{"sym" } [\text{"sym" } \text{ false}]];\right.\]
\[\left. [\text{"sym" } \text{ BOOL}];\right.\]
\[\left[\text{"sym" } \text{ false}]);\right.\]

The resulting string does not only have a fixed syntax, it is also self-descriptive. The function \(\downarrow\) can derive \(G\) from the ATerm it decodes. With this encoding we can define very generic operations on parse trees like substitution, unification and searching of subtrees that are not language specific.

However, the disadvantage of this scheme is that \(\downarrow\) is a partial function. In other words, there are (many) ATerms that are not encodings of parse trees, e.g. \(\text{"abc" } \text{ "def"}\) is a syntactically correct ATerm but is not an element of \(\text{\texttt{Tree}(G)}\) for any \(G\). Therefore, programs that manipulate ATerms encoding parse trees have to 'type' check the terms they receive and have to preserve well-formedness of the terms they process and construct.

2.9 Example Another example, based on Visser (1993), of a generic untyped language is the language of applicative expressions of combinatory logic extended with lists. Where the previous example defined a data format, this example defines an untyped functional programming language with higher-order functions in a first-order algebraic context.

module CTerms
sorts CTerm
context-free syntax

CTerm CTerm \rightarrow CTerm \{left\}
"(" CTerm "\)" \rightarrow CTerm \{bracket\}
"[" CTerm "\),"]" \rightarrow CTerm
CTerm "++" CTerm \rightarrow CTerm \{assoc\}
map \rightarrow CTerm

variables

\[\text{[fx]} \rightarrow \text{CTerm}\]
\[\text{[xy]} \rightarrow \text{CTerm}\]

equations

\([x^*] ++ [y^*] = [x^*, y^*]\]
map \(f [] = []\)
map \(f [x, x^*] = [f x] ++ \text{map } f [x^*]\)

Such a definition works well as long as sensible terms are considered. However, \(([] \text{map})\), the empty list applied to the function map, is also a syntactically correct term, but does not have a clear interpretation. We would rather forbid this term on the basis of some typing rule without losing the genericity of the term structure.

3 Two-Level Grammars

Context-free grammars provide either a strongly typed but rigid syntactic structure or a generic but untyped structure. Two-level grammars provide a method for polymorphic syntax definition that supports definition of generic structures with type constraints. Two-level grammars have been
defined in various guises after the original formulation for the definition of the syntax of Algol68 in van Wijngaarden et al. (1976). (See also Skonneker and Kurtz (1995) for a definition of Van Wijngaarden grammars and some examples.) Here we introduce a definition that is straightforwardly formulated as two levels of context-free grammars.

3.1 Definition A two-level grammar $\Gamma$ is a pair $(G_1, G_2)$ of context-free grammars such that the sorts of $G_2$ are terms, possibly with variables, over $G_1$, i.e. $S(G_2) \subseteq \text{Tree}(G_1 \cup V_1) \cup S$. A two-level grammar $\Gamma$ is an abbreviation for a, possibly infinite, context-free grammar $[\Gamma]$ that is derived from $\Gamma$ by taking all substitutions of symbols and productions of $G_2$ as follows: $S([\Gamma]) = \{ \sigma(\tau) \mid \sigma \in S(G_2), \sigma : V_1 \rightarrow \text{Tree}(G_1 \cup V_1) \}$ and $P([\Gamma]) = \{ \sigma(\tau_1) \rightarrow \sigma(\tau_2) \mid \tau_1 \rightarrow \tau_2 \in P(G_2), \sigma : V_1 \rightarrow \text{Tree}(G_1 \cup V_1) \}$.

The well-formedness of a parse tree over a two-level grammar can be determined by expanding a finite number of productions. Define $\tau \in S(\Gamma)$ iff $\Gamma \vdash \tau$ ($\tau$ is a sort) and $t \in \text{Tree}(\Gamma)(\tau)$ iff $\Gamma \vdash t : \tau$ ($t$ is a term of sort $\tau$), where the inference relations are defined by the rules in table 4. In these rules $\sigma$ is a substitution of sort variables, i.e. the extension to Tree($G_1 \cup V_1$) of a family of functions of type $V_1(\tau) \rightarrow \text{Tree}(G_1 \cup V_1)(\tau)$.

It is straightforward to define two-level signatures analogously to two-level grammars and extend the translations from context-free grammars to signatures and vice versa accordingly. Meinke (1992a) gives a calculus of equations over terms and types. Visser (1996) gives an algebraic specification of a type checker for programs with multi-level signatures.

The productions in the second level of a two-level grammar are in fact production schemata that uniformly describe sets of context-free productions in the same way as polymorphic functions in a framework like ML (Milner, 1978) describe collections of functions.

We observe that the two ways of defining the terms generated by a two-level grammar are equivalent.

3.2 Proposition $\Gamma \vdash t : \tau$ iff $[\Gamma] \vdash t : \tau$

Proof. (Sketch) Check that a derivation with inference rules for two-level grammars corresponds with a context-free derivation with the expanded two-level grammar and vice versa. Comparing the rules in tables 1 and 4 one sees that the differences are in the application rule and the new substitution rule. These differences correspond to the substitution in the definition of $[\Gamma]$.

3.3 Discussion According to the definition above, trees over the first grammar are used as sorts in the second grammar. However, if we write such grammars, we want to use strings instead of trees, i.e., $S(G_2) \subseteq L(G_1)$ instead of $S(G_2) \subseteq \text{Tree}(G_1 \cup V_1) \cup S$. This entails that the syntax of two level grammars is not fixed, the syntax of the symbols of the second level is determined by the first level. To parse a two-level grammar we first have to parse the first grammar
with the normal parser for our context-free grammar formalism in order to construct a parser for the second-level grammar. Note that we use the same, SDF style, notation for productions and modules at both levels.

3.4 Claim Van Wijngaarden (VW) grammars (van Wijngaarden et al., 1976) are isomorphic to two-level grammars as defined above.

3.5 Claim Definite Clause Grammars (Pereira and Warren, 1980) are two-level grammars with a fixed first level that defines an untyped domain of terms that can be used as grammar symbols.

3.6 Claim Two-level grammars are isomorphic to two-level signatures as defined in Poigné (1986) and Hearn and Meinke (1994) in the same way that context-free grammars are isomorphic to first-order signatures.

3.7 Example In this example we show how to define regular sort operators (list sorts), use them to define a fragment of the programming language Pico, define polymorphic list constructors and define some generic operations on these constructors. We start by defining the first level grammar that defines the syntax of Sorts: module Sorts introduces the sort Sort, module Pico-Sorts introduces several Sort constants that are specific to Pico and module Regular-Operators defines the sort constructors "?" (optional), "*" (list), "+" (non-empty list), {}+ (list with separator), {}+ (non-empty list with separator).

module Sorts

sorts Sort

variables

[A-D] → Sort

module Pico-Sorts

imports Sorts

context-free syntax

Sort "?" → Sort
Sort "*" → Sort
Sort "+" → Sort
"{" Sort Sort "}" → Sort
"{" Sort Sort "}" → Sort

equations

A* = A+?
{A B}* = {A B}+?

The equations in the last module indicate that the "*" operators are merely abbreviations. Now we can use expressions over the Sort language as sorts in a second level grammar, just as if it were a normal context-free grammar—{Stat ";"}+ and Exp are both grammar symbols.

module Pico

imports Regular-Syntax, Pico-Sorts, Types

expressions

sorts Var Type Var-Type Exp Stat Decl

{Var-Type ";"}+ Decl? {Stat ";"}+

context-free syntax

Var ";" Type → Var-Type
"declare" {Var-Type ";"}+ ";" → Decl
Var ";"= Exp → Stat
"while" Exp "do" Stat → Stat
"begin" Decl? {Stat ";"}+ "end" → Stat

We can now proceed by defining the productions for the rest of the symbols, for instance, {Stat ";"}+ can be defined as follows to denote a list of Stats separated by ";"s:

context-free syntax

→ {Stat ";"}+
{Stat ";"}+ → {Stat ";"}+
Stat → {Stat ";"}+
{Stat ";"}+ ";" {Stat ";"}+ → {Stat ";"}+

At this point we have normal context-free grammars with user-definable sort syntax. However, we can do better by providing polymorphic productions for the constructors corresponding to the regular sorts. An optional A? is either empty or A, a non-empty list A+ is either an A or the concatenation of two A+’s, a non-empty list of A’s separated by B’s is either an A or two lists concatenated by a B, and, according to the equations in module Regular-Operators, a possibly empty list A* is an optional non-empty list A+?. The productions for statement lists above are instantiations of productions in the following module:
module Regular-Syntax
imports Regular-Operators
sorts A

class context-free syntax

→ A?
A → A?
A → A+
A+ A+ → A+ {assoc}
A → {A B}+
{A B}+ B {A B}+ → {A B}+ {assoc}

Now that we have a polymorphic definition of list construction we can also define polymorphic functions over lists. For instance, the length function that computes the number of elements of a list can be generically defined by the following specification (assuming some appropriate specification of integers):

class context-free syntax

"length" "(" A+ ")" → Int

variables

"a" → A
"a" [12] "+" → A+

equations

length() = 0
length(a) = 1
length(a_1 a_2) = length(a_1) + length(a_2)

3.8 Example Another example of a Sort constructor is the arrow ⇒ that we can use to construct function sorts:

class context-free syntax

Sort "=>" Sort ⇒ Sort {right}

A term of sort A ⇒ B, i.e. a function from A to B, can be applied to a term of sort A. The higher-order function 'map' takes as argument a function and a list and applies the function to each element of the list.

context-free syntax

A ⇒ B A "map" "(" A ⇒ B "," A* ")" ⇒ B*

variables

[f] → A ⇒ B

equations

map(f, ) = map(f, a) = f a
map(f, a_1 a_2) = map(f, a_1) map(f, a_2)

If we want to pass the functions length and map themselves as arguments to some higher-order function we need to define the combinators associated to the prefix functions as follows:

class context-free syntax

"length" A⇒Int
"map" (A⇒B)⇒(A⇒B)

equations

map f a* = map(f, a*)
length a* = length(a*)

These examples show that two-level grammars provide (1) user-definable syntax for grammar symbols, i.e. type constructors, and (2) generic definition of productions, i.e. polymorphic functions and constructors over data types.

3.9 Example In the example above we defined two versions of the functions length and map, one mix-fix version for normal use and a combinator version to pass as argument to other functions. Another example of this combining of mix-fix notation with combinator notation is the convention in functional programming languages to write a binary operator in brackets (section) when it is used as a combinator. For example, the addition operator + on natural numbers and its section (+) are defined as

class context-free syntax

N "+" N ⇒ N
"(+)") N⇒N⇒N

equations

(+) x y = x + y

We could express this more generically by the following grammar in which (A B) ⇒ C represents the type of binary operators with left argument of type A, right argument of type B and result of type C:

class context-free syntax

"+" A (A B) ⇒ C B
"(" (A B) ⇒ C ")" A ⇒ B ⇒ C

variables

"⊕" (A B) ⇒ C

equations

(⊕) x y = x ⊕ y

3.10 Discussion In the last grammar we had to change the original production N "+" N ⇒ N.
It would be interesting to express the derivation of the syntax of the 'section' from the original production by rules like

\[
\frac{A \, \text{"+"} \, A \, \rightarrow \, A}{\text{"(+)"} \, \rightarrow \, A \, \Rightarrow \, A} \quad \frac{A \, \text{"L"} \, B \, \rightarrow \, C}{\text{"(L)"} \, \rightarrow \, A \, \Rightarrow \, B \, \Rightarrow \, C}
\]

Another application of such production derivation rules is to subtypes. If \( A \) is injected into \( B \) by the production \( A \rightarrow B \), then we could say that \( A \) is a subtype of \( B \). It would then be natural to have \( A^+ \) as subtype of \( B^+ \), i.e. if we have a list of \( A \)'s, we want to consider it also as a list of \( B \)'s. That is to say that the injection \( A \rightarrow B \) derives the injection \( A^+ \rightarrow B^+ \). This is like a theorem derivable from the productions for lists; a list of \( A \)'s can be derived in two ways: by first injecting the \( A \)'s into \( B \)'s and then constructing a list or by first constructing a list and then injecting it into \( B \). This last operation is however not automatically generated by the list grammar. This could again be expressed by production derivation rules like the following:

\[
\frac{A \rightarrow B}{A^+ \rightarrow B^+} \quad \frac{f(A) \rightarrow B}{f(A^+) \rightarrow B^+} \quad \frac{\tau_1 A \tau_2 \rightarrow B}{\tau_1 A^+ \tau_2 \rightarrow B^+}
\]

The first rule derives an injection between lists from an injection. The second rule is like the map function: if there is a function \( f \) from \( A \) to \( B \), then \( f \) can be lifted to apply to lists of \( A \)'s resulting in a list of \( B \)'s. The last rule generalizes the previous two: instead of an injection or a prefix function an arbitrary context \( \tau_1 \tau_2 \) can be lifted to lists.

4 Parsing

A parser for a two-level grammar maps strings to sets of parse trees.

4.1 Definition The type of a parse tree is defined as:

\[
\text{type}([(\tau_1 \rightarrow \tau_2) \, t]) = \tau_2 \\
\text{type}(\tau) = \tau \\
\text{type}(\epsilon) = \epsilon \\
\text{type}(t_1, t_2) = \text{type}(t_1) \text{ type}(t_2)
\]

A parser for a two-level grammar \( \Gamma \) can be defined by

\[
\Pi(\Gamma)(w) = \{ t \mid \epsilon \ast w \Rightarrow^* t \ast \epsilon \}
\]

where the relation \( \Rightarrow^* \) between pairs \( t \ast w \) of trees and strings of symbols is defined as:

\[
\begin{align*}
t_2 \in \Pi(\Gamma_1)(w_1) \quad &\quad t_1 \ast w_1 \ast w_2 \Rightarrow^* \tau \quad t_1 \ast t_2 \ast w_2 \\
p = \text{type}(t_2) \rightarrow t_2 \in \Pi(\Gamma_2) \\
t_1 \ast t_2 \ast w \Rightarrow^* \tau \quad t_1 \mid [p \, t_2] \ast w
\end{align*}
\]

4.2 Proposition \( \epsilon \ast w \Rightarrow^* t \ast \epsilon \) \iff \text{yield(yield(t)) = w and } \Gamma \vdash t \)

4.3 Proposition It is not possible to produce terminating parsers for arbitrary two-level grammars.

Proof. Sintzoff (1967) shows that every enumerable set can be defined by means of a two-level grammar. □

All parsing strategies have problems with two-level grammars. The production for application \( A \Rightarrow B \, A \rightarrow B \) causes a loop in top down prediction; if some symbol \( \tau \) is predicted, this production is applicable (\( B \) matches any symbol of sort Sort), but then \( A \Rightarrow \tau \) is predicted and the application production qualifies again. Top-down prediction plays an important role in a 'bottom-up' algorithm like Earley's to prevent unnecessary steps. However, a pure bottom-up parser should have no trouble with this production. It is not clear how to define a left-corner parser for two-level grammars since the left-corner relation is not finite. A bottom-up parser has problems with productions like \( A \rightarrow A^+ \). The \( A \rightarrow A^+ \) production generates infinitely many parse trees for any tree generated by the grammar; it is a function that takes an argument of any sort and makes it into a singleton list, to which the function can be applied again and so on.

4.4 Proposition There are two-level grammars \( \Gamma \) and strings \( w \) such that \( \Pi(\Gamma)(w) \) contains infinitely many non-unifiable trees.

Proof. Take the grammar with lists. The production \( p = A \rightarrow A^+ \) produces for any tree \( t \) of sort \( A \) the trees \( \{ [A \rightarrow A^+ \, t] \, [A \rightarrow A^+ \, t] \, [A \rightarrow A^+ \, t] \, [A \rightarrow A^+ \, t] \} \).

This means that a string can have infinitely many different types that are not instantiations of a single simple principal type in the sense of Danas and Milner (1982). It is also clear that this infinite set of parse trees cannot be compacted...
by the techniques known from parse forests for context-free grammars. Nevertheless, it should be possible to automatically construct parsers for many practical grammars using some kind of tabular parsing method as have been defined for context-free grammars (e.g., Tomita, 1985), or for DCGs and Prolog programs (Pereira and Warren, 1983; Warren, 1992). In particular it would be interesting to derive parse forests as in Tomita (1985) instead of just parse trees. To be able to parse according to the two-level grammars that we saw in the examples in the previous section we need a compact characterization of sets of parse trees generated by productions like $A \to A+$, which is different from the parse forests known from context-free parsing, because packing also has to take place at the level of sorts.

Finite Instantiations It seems that for a great number of 'natural' grammars like the ones above it should be possible to construct parsers. For example, for the Pico grammar above we can take all instantiations of the generic list productions that are needed for the list sorts used in that grammar and obtain a finite context-free grammar. As a result normal context-free parsing techniques apply. This approach is used in Visser (1995) to define the semantics of an extension of normal context-free grammars with regular operators. The following conjecture expresses conditions under which it is expected that this approach can be applied.

4.5 Definition (i) A production is variable preserving if $\text{Var}(r_1) \subseteq \text{Var}(r_2)$ for each $r_1 \to r_2 \in P(G_2)$

(ii) The productions of a two-level grammar are descending if there is no chain $r_n s_n t_n \cdot w \Rightarrow r \circ \cdots \Rightarrow r_1 s_1 t_1 \cdot w \Rightarrow r_0 \cdot w$ such that $s_0$ is more general than $s_n$, i.e. there is a substitution $\sigma$ with $\sigma(s_0) = s_n$

4.6 Conjecture The sub-grammar $[\Gamma]_r$ of a two-level grammar $\Gamma$ consisting of all productions reachable from some ground symbol $r$ is finite if all productions of $G_2$ are variable preserving and descending.

Naturally, this construction is not applicable to all interesting grammars; the production for application with $\Rightarrow$ sorts is not variable preserving and is not descending.

Note: The length function above is not variable preserving but only a finite number of instantiations are needed for the Pico grammar. This could be called the renaming property; a finite number of copies of the generic grammar are needed; this could also be achieved through renaming.

5 Open Problems and Future Work

We summarize the problems that stand in the way of the use of two-level grammars and discuss some possible further work.

For a certain class of two-level grammars we are only interested in a finite subset of its expansion. The two-level aspect is then used as an abbreviation mechanism. Is it possible to characterize this class of two-level grammars? Is it decidable whether the projection with respect to some ground symbol $r$ is finite?

To use functions defined by means of productions as arguments to higher-order functions it is necessary to refer to the 'function name' of such productions. This function name can be defined explicitly as another production and the application of the function to the appropriately typed arguments can be reduced to the original form. However, it would be better to derive the function name in a uniform way or at least to generically describe such a derivation for certain classes of productions as suggested by the production derivation rules in section 3.

A characterization of the 'principal types' for two-level grammars is needed to compactly express the parse trees for grammars with productions like $A \to A+$. The methods of parse forests for generalized context-free parsing do not seem to apply here.

In Visser (1996) the syntax, type assignment and semantics of a functional programming language with multi-level signatures is defined algebraically. These signatures differ from the two-level signatures defined above in that the application operator is generalized; not only a function can be applied to a term but an arbitrary term. It is not immediately clear how these multi-level signatures correspond to grammars.

If the genericity available in the second level is also needed at the first level, two-level grammars can be extended to three or more levels. The disadvantage of such multi-level grammars (including two-level grammars) is that the syntax defined at level $n$ is only available at level $n+1$ leading to copies of the same syntax at multiple levels. Therefore, it might be an idea to general-
ize multi-level grammars by collapsing all levels into a single level leading to reflexive grammars. The sorts of a reflexive grammar are trees over itself. It is not clear whether there are necessarily finite characterizations of the trees over such grammars. The syntax of reflexive grammars is extensible and parsing techniques for extensible languages are not very well developed. Furthermore, it seems that grammars for extensible languages need some way to talk about the parse trees they generate in order to interpret these as new productions, which calls for a mechanism like reflexive grammars.

6 Conclusions

In this paper we have discussed the application of context-free and two-level grammars to polymorphic syntax definition. We gave a simple definition of two-level grammars and several examples of their usage. Two-level grammars provide (1) user-definable syntax for grammar symbols (type constructors), (2) generic definition of productions (polymorphic functions and constructors of data types) and (3) support for combining generic structures with type constraints. Before two-level grammars can be used as signatures of algebraic specifications a number of open problems have to be solved.

The general conclusion of this paper is that the extension of algebraic specification formalisms like OBJ or ASF+SDF with polymorphism, while keeping user-definable syntax leads necessarily to two-level grammars. Likewise, the extension of polymorphic signatures with user-definable syntax leads to two-level grammars. In this paper we have shown how the integration of algebraic specification with user-definable syntax and polymorphism can be materialized.

Acknowledgments I thank Mark van den Brand, Arie van Deursen, Jan Heering, Jasper Kamperman and Paul Klint for many discussions on this subject. Mark, Arie and Jasper commented on drafts of this paper. The research for this paper is supported by project 612-317-420 of the Dutch Organization for Scientific Research (NWO).

References


Algebraic Specification of Documents

José Carlos Ramalho  
jcr@di.uminho.pt

José João Almeida  
jj@di.uminho.pt

Pedro Henriques  
prh@di.uminho.pt

Departamento de Informática  
Universidade do Minho  
Braga – Portugal

ABSTRACT

It is becoming normal that a document should serve several purposes. However, the majority of available text processors is purpose-oriented, reducing the necessary flexibility and reusability of documents. Some waste of time arises from adapting the same text to each different purpose, when this task could be done automatically (from the first version of the document) with an appropriate system.

This communication highlights the guidelines to build a system to solve the above problem. Such a system should be an algebraic based environment and provide facilities for:

- Document type definitions;
- Definition of Function over document types;
- Document definitions as algebraic terms.

This approach (rooted in the tradition of constructive algebraic specification models), will allow for an homogeneous environment to deal with operations such as merging documents, converting formats, extracting portions of documents, and some other unusual operations like mail reply and literate programming.

We intend to build on CAMILA (a specification language and prototyping environment developed at Universidade do Minho, by the Computer Science group) developing the above mentioned system as one of its extensions.

1 INTRODUCTION

A document is a collection of pieces of text — a pure character string — organized according to a specific structure. It contains a message to be delivered (to someone), and its structure is defined in order to enhance some special parts of that message, and in general, to improve its transmission process.

Document processing means transforming a given document in order to produce another document (with a different structure or with the same organization expressed in a different markup language) or to execute some reactive action. As it can be seen, the definition above includes tasks such as text formatting, translation, interpretation, automatic reply to message, literate programming, and so on. Therefore document processor is no more than a typical language processor.

Algebraic programming is an approach to (computer) problem solving based on the definition of operations that perform data transformations according to a given signature (a family of sorts and a family of operators with a well defined domain and co-domain). A data element (to be processed) is a piece of information with a specific structure; this is specified associating to the entity a set (type) amounting to the carrier of the sort it belongs to. An operation formally describes as function or relation on the carriers the way some data elements are transformed into a specific result.

These definitions suggest that a document can be thought of as a data element and document processing as an algebraic operation.

Thus we propose to apply the algebraic specification method to document processing. The key idea of this approach is the definition of a document type — every document must have an associated type, predefined or undefined. Each processing task is specified as an operator (a function) defined over document types, and a document can be expressed as a term of the underlying term algebra.
These concepts — document types, functions and documents — are discussed in detail in the next section (sec. 2). The architecture of the algebraic system we envisage to develop and its interface is described in section 3. In section 4 we illustrate our proposal with two well-known examples. As usual the paper is concluded with some final remarks and prospect for future work (sec. 5).

It should be noticed that this is a preliminary research paper in the sense that we will not present results of a finished work, but propose instead some ideas and principles to be further investigated and developed by our research team.

2 THE PROPOSED ALGEBRAIC APPROACH

The problem of document definition has been a worrying one for a long time.

Whoever uses a computer to carry out the tasks involved in document production wants:

Easy manipulation of documents
Some common requirements being:

- subdocument extraction
- structural document translation

Furthermore manipulating tools should be correct and reusable.

One of the main concerns in document production is:

Document correctness
It is difficult to define correctness. We can think of

- structural correctness — have the right components on account of text purpose. It is important to have an explicit structure visible in the document (in order to allow for automatic structure recognition).
- invariant
- Define properties (in a formal way) of any kind to be satisfied by the document.

Another important concern, that arises whenever somebody has to deal with some documents based on the same text is:

Document reuse
Ex. a dictionary can be printed. However, the dictionary’s definition should not be too much tied to pagination, because that would disable the possibility of using it for other operations as an electronic hyper-text.

To achieve this it is necessary:

- separate a document from the details of final “views”

2.1 DOCUMENT TYPE DEFINITIONS

The document type definition is a way to define properties to be verified by document instances. Therefore it is a step to the notion of correctness.

In CAMILA (Barbosa and Almeida, 1995) (a model-based algebraic specification, briefly introduced in appendix A) a type definition involves the definition of:

- the carrier set of its sort
- an invariant (a boolean-valued function that restricts the carrier set cardinality to cope with semantics requirements).

Ex: 1 [the electronic mail document] (see a complete example in a next section). The carrier set definition is:

\[
\begin{align*}
\text{mail} : & \quad \text{header} : \text{id} \rightarrow \text{string} \quad (1) \\
& \quad \text{body} : \text{string-seq} \quad (2)
\end{align*}
\]

(1) - mapping between id(entifier) and strings.
(2) - a list of strings (lines)

The following invariants guarantee that message has a no null contents:

\[
\text{inv_mail(m) = body(m) != "" \land header(m)[subject] != "";}
\]

In order to be consistent with this model, a document has to be “structurally” correct, and guarantee the invariant.

The structure of a document is also a very good guide to build translations to/from other formats/models, manipulation functions and browsers of documents.
2.2 Function Definition Over Document Types

The basic collections of functions associated to CAMILA type constructors (e.g., union, intersection of two sets, domain or range of binary relations, application or overwrite of finite functions, etc.) are available as primitive functions in the language. So are the propositional connectives and quantifiers.

The specification of new functions over document types, plays a very important role in all the process. It is a way to:

- describe document manipulation as translations between different formats (internal or external)
- describe the behavior (or intended behavior) of existing tools
- discuss future tools and (document) types
- build documentation of tools and formats

The definition of a function in CAMILA includes:

- definition of a precondition (a predicate over input parameter and state) that has to be evaluated to true
- definition of a returned value
- update of the state variable

Function specifications are an important step in the definition of system correctness.

If its precondition is true, in order to be correct, a function must:

- have the right signature
- guarantee that the invariant of the returned value type evaluates to true
- guarantee that the invariant of the computed state variable evaluates to true.
- guarantee that the precondition of every function used is true

The availability of all the repertoire of CAMILA operators and the guidelines offered by the type model greatly simplifies the task of defining a new function.

3 The Algebraic System

It is quite clear that an algebraic system is a little far from what really is document electronic interchange. This entails the need of one more layer intended to establish the bridge between the algebraic system and the outside world of documents. A format (or set of formats) must be chosen as the input and output of this layer and consequently of the system. This format should not have any character set dependencies and should be easy to parse and generate. This layer will incorporate a parser/translator for the chosen formats.

![Diagram](image)

Figure 1: System Architecture

Fig. 1, gives a more detailed idea of the intended system.

f1 stands for a CAMILA function that receives two documents as arguments and produce a new one.

f2 CAMILA function that transform one document in another.

Whenever we want to do external processing using an external tool (accepting a format FMT1 and building a document in format FMT2), we have to write an `exportFMT1` and a `importFMT2`
functions (see `texedit` in mail example and in Fig. 1).

Looking at the actual scene there are some strong candidates like \texttt{LaTeX} (Lamport, 1986), Word or SGML (Sperberg-McQueen and Burnard, 1994), to be considered as an input/output format to/from our system. On the other hand, a closer look to those formats shows that Word is not a good choice because it has not a visible structure and its format (Microsoft copyright) is not well known to the public.

Both SGML and LaTeX have a visible structure, are widely used, and there are plenty of tools able to process documents written in their format.

Comparing this two one major difference comes up. LaTeX is too much tied up to format and typographic aspects, and SGML is not. Besides that SGML has the following advantages:

- it is an ISO standard (ISO 8879).
- it is not concerned with formatting aspects and is fully data independent.
- its only concern is the textual structure of a document.
- its use is spreading rapidly, and there are many commercial and public-domain tools \texttt{RITA} (D. Cowan and D. V. Smit, ), \texttt{CoST} (Harto, 1994) and \texttt{sgmlpl} (Megginson, 1995b; Megginson, 1995a) available to create and process SGML documents.

Therefore, for the time being, SGML is the base format chosen to communicate with the outside world (this will not eliminate the possibility of adding other formats formally).

3.1 SGML AS INPUT AND OUTPUT

SGML, abbreviation for "Standard Markup Language", is a meta-language to define descriptive markup languages which specifies the structure of a particular kind of document. The markup language does not specify how the document is to be processed or printed, it only specifies its structural elements and the relations between them. For example, a markup language could specify the lines and stanzas of a poem, but not the type of font or size to be used when printing or displaying the document.

Using SGML it is possible to specify the structure of a certain kind of document creating a Document Type Definition (DTD). Documents that obey that structure are classified as being of that type. This way, any SGML document belongs to a class/type of documents.

When creating a DTD, structural elements and the way they are related are specified. Then when writing a document according to a specific DTD, we decorate it with start tags ("<tag>") and end tags ("</tag>") delimiting the structural elements, as, for example, the following mail message:

Ex: 2 [electronic mail]

```xml
<mail>
  <header>
    <from> jcr@di.uminho.pt </from>
    <to> epl@di.uminho.pt </to>
  </header>
  <body>
    This is only a tutorial example to be used in this article...
  </body>
</mail>
```

The correspondent DTD may be specified as:

Ex: 3 [Mail DTD]

```xml
<!ELEMENT mail (header, body)>
<!ELEMENT header (from, to)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Assuming SGML as the standard format for input/output to/from the system brings some problems. We need to establish a correspondence function between SGML elements and the Abstract Data Types of the Algebraic System. To do this, we must ensure that SGML specification language is self-contained in Camila data models.

In this case, we will establish a correspondence
function between SGML and Camila Data Models.

3.1.1 SGML ↔ Camila Data Models

SGML is a very simple language, structure oriented. So it should not be difficult to create a correspondence between each one of its features and Camila data models.

An SGML specification is composed by a series of ELEMENT declarations. Each ELEMENT corresponds to a structural element of the document and is defined as text or as being a combination of other elements. SGML as a few operators to specify relations between elements:

<table>
<thead>
<tr>
<th>Expression</th>
<th>SGML features</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y</td>
<td>element x followed by element y</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>x and y in any order</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>either x or y</td>
</tr>
<tr>
<td>x*</td>
<td>element x 0 or more times</td>
<td></td>
</tr>
<tr>
<td>x+</td>
<td>element x 1 or more times</td>
<td></td>
</tr>
<tr>
<td>x?</td>
<td>element x 0 or 1 time</td>
<td></td>
</tr>
</tbody>
</table>

Given the variety of Camila data models it is easy to see that for each case we can have more than one correspondence. For example, the following mapping could represent a translation scheme:

<table>
<thead>
<tr>
<th>Translation Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGML</td>
</tr>
<tr>
<td>x, y</td>
</tr>
<tr>
<td>x &amp; y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>x*</td>
</tr>
<tr>
<td>x+</td>
</tr>
<tr>
<td>x?</td>
</tr>
</tbody>
</table>

The above scheme is poor in some respects. For example x+ is being mapped into X-seq but this list should have one or more element. This can be defined through an invariant. The translation to Camila besides converting the types should add the necessary invariants to each case.

4 SOME EXAMPLES

In order to illustrate some of the advantages of the proposed approach, we present two example.

4.1 Mail

In this example we specify what a unix mail message is. Next the specified structure is used in some real processing.

To define the document type that describes a unix mail message we could write the following Camila specification:

```plaintext
MODEL mail
use "txt.cam"

TYPE
  header = SYM -> ANY;
  mail   = h:header
          b:TXT;
  env    = user:SYM
          date:ANY;
ENDTYPE

STATE e:env;
  e <= env(’joao’,”today”); /initial state/

Mail is composed by header and body. The body is simply text. The header is a finite function from symbol to anything, where symbol is a token, in this case pertinent tokens are: to, from, cc, ...

Now we can write some functions over that type reflecting our knowledge about the behaviour of mail messages. For example, it may be stated that a mail should have a from field and its body not to be empty, to be considered correct. This can be written in Camila as the following invariant:

```plaintext
inv_mail(a)=
  'from in dom(h(a)) \/
  (b(a) != "" \/
  h(a)['subj'] != "");
```

Following we specify a mail reply:

```plaintext
func reply(a:mail):mail
returns
  mail([’to’ -> h(a)['from'],
        ’subj’ -> strcat("re:","h(a)['subj']"),
        ’from’ -> user(e),
        ’date’ -> date(a),
        ’cc’ -> h(a)['cc']],
    < "In the last episode you said:" ;
    <strcat(" ",x) | x <= b(a)>>);
```

To finish this example we reproduce a mail session in Camila (putting things to work):

```plaintext
ex<mail([’to’ => ’joao’,
          ’from’ => ’peter’,
          ’subj’ => ”Test of the system”,
          ’cc’ => ’jcr’],
```
< "dear Joao",
   "good luck with this" > ;

re_ex <- reply(ml);

the document "re.ex" now has the value:

mail( ['to' => 'peter,
       'from' => 'joao,
       'subj' => "Re: Test of the system",
       'cc' => 'jcr'],
   < "In the last episode you said:
   "> dear Joao",
   "> good luck with this" > )

Now it is necessary to edit the body of the mail
in order to continue the message. The function
txtedit will do that task by:

* writing the STR-seq body to a file (txtsave)
* calling an external editor (Ex. vi)
* reading back a text (txtload) (using an external
txt2cam format translator)

```haskell
func txtedit(txt:TXT):TXT
returns do( txtsave(".tmp",txt),
   sh("vi .tmp")),
   txtload(".tmp");

func txtload(name:STR):TXT
returns
let(f=popen(strcat("txt2cam ",name),"r"),
   t=realff(f),
   x=pclose(f)) in t;
```

Now it possible to edit re.ex body:

re_ex.b <- txtedit(b(m2));

4.2 LITERATE PROGRAMMING

In this section a naive literate programming
(Knuth, 1992) system is described.1

The main idea is to have a document type lpt
(literate programming type) that is a list of ele-
ments which can be:

* titles (of document(tit) or section(sec))
* association of identifiers(id) with pro-
grams(pro)
* program(pro) – sequences of strings(STR) or
  program references(id)
* straight text strings(STR)

That document contains a program (to be ex-
tracted with getprog function) and a textual doc-
ument (to be extracted with getlatex) typically
a manual describing the program implementation
and including the program.

MODEL lpt
TYPE
lpt = ele-seq;      list of elements
   ele = STR | pro | defi | id | sec | tit;
   pro=STR | id=seq;    program with id
defi = i : id;      id definition
   v : pro;
   id = SYM;         identifier
   sec = STR;        section title
tit = STR;         document title
ENDTYPE

Let ex be an example document (built using
the implicit constructors of the language)2:

ex <- <
   tit("Example of literate prog"),
   sec("Stack - FAQ"),
   defi('main, <"main()
   "},
   "int S[20]: sp=0",
   'pop ,
   'push >),

   sec("pushing elements"),
   "to push elements",
   "you can use this function:",
   defi('push, <"void push(int x)",
   "{S[sp++]=x;3}">),

   sec("popping elements"),
   "not yet available",
   defi('pop, <"int pop(x)",
   "{/#to be continued/}"">);

Next we define the function getprog whose
purpose is to extract a program(prog) from a lit-
erate programming text(lpt).

In the first step an index is built (function
mkindex). The function explode is defined to
make the recursive substitution of identifiers(id).

```haskell
TYPE
prog = STR-seq;       (prog with no id)
index = id -> ele-pro;
```

1The examples complete code (including the auxil-
ary functions not presented here because of space con-
straints) and other case studies can be obtained from the authors.

2A more WEB-like notation could be used based on a
webget translator (easily built in PERL).
ENDTYPE

func mkindex(t:list): index
return [i(x) -> v(x) | x<t : is-defi(x)];

func getprog(t:list): prog
return explode('main,mkindex(t);

func explode(i:id, d: index): prog
pre i in dom(d)
returns CONG(
  < if(is-id(x)-> explode(x,d),
    else -> <x> ) |x <- d[i]>>)

Let pex be the program extracted from ex:

pex <- getprog(ex);

would assign to pex

main(){...}
int S[20]; sp=0
int pop(x)
{ /* to be continued */
void push(int x)
{S[sp++]=x;}

To extract the document part(latex) of the literate programming text, we have to define the document type latex:

/*@ micro Latex */
ltx
  d : documentclass
    t : tit
    s : section-seq
section
  t : sec
  v : (STR | verbatim)-seq
documentclass = SYM
verbatim = STR-seq;
func getlatex(t:list):latex
returns
if (t is<-t:sec->latex('article,
  ti,
  getsecList(ta));

....

To create the latex part of ex:

latex_ex <- getlatex(ex);

would assign to latex_ex

latex(  
article,
  tit( Example of literate prog ),
  section(   
    sec( Stack FAQ ),
    < verbatim( < main
      main(){...}
      int S[20]; sp=0
      pop
    push )>>)
  section(   
    sec( pushing elements ),
    < to push elements
    you can use this function:
    verbatim( < push
    void push(int x)
    {S[sp++]=x;} )>>)
  section(   
    sec( popping elements ),
    < not yet available
    verbatim( < ...

5 CONCLUSION

Along this paper we have discussed an approach to document processing we intend to develop further: define document types and specify document manipulations under an algebraic system. Types are described using the usual abstract data models plus a predicate that establishes type invariants. Documents are created, and processed as instances of a given type by means of function application. Those functions with type models define an algebra and documents can then be thought as algebraic terms.

Our proposal is based on the use of the algebraic system CAMILA, a general purpose constructive specification language and an environment for building and running program prototypes.

With this approach we gain in simplicity and conciseness. Moreover, we think that three other obvious advantages emerge from this method: the reusability of types and functions; the proofability of correctness, based on type invariant checking and validation of function calls (with respect to its signature); the refinement guidelines.

Two examples —definition and manipulation of Unix mail messages and literate programs— were presented for illustration of our approach, its style and its power.

One of our concerns in the near future is to search for similar proposals and compare them with ours in order to improve these ideas.

The application of the method to other realistic

---

3In order to be useful, this example should also include a generate function that produce the actual \LaTeX syntax from the camila latex document type.
documents and operations also deserves a deeper investigation.

Another topic we intend to research carefully is the use of SGML as the a description format to input, or output, documents into, or from, our algebraic system.

The long term aim is to develop an automatic, or semi-automatic, translation process based on the systematic analysis of document types.

REFERENCES


A CAMILA : A BRIEF INTRODUCTION

Parts of this appendix come from (Barbosa and Almeida, 1995) lecture notes where a more detailed overview of CAMILA can be obtained from that document.

A.1 CAMILA PHILOSOPHY AND EVOLUTION

From school physics we got used to a basic problem solving strategy: create a mathematical model, reason on it, calculate a solution. The CAMILA approach is an attempt to make such a strategy available at the software engineering level. Based on a notion of formal software component it encompasses a set-theoretic notation, a prototyping environment, fully connectable to external applications and equipped with communication facilities, and an inequational refinement calculus.

CAMILA aims to be both a learning tool for Computer Science students and a working tool for software engineers. At the first level it provides
a smooth way to programming. At the second a rigorous way to develop complex systems and to promote the use of formal methods in software industry.

CAMILA\textsuperscript{4} was originally devised as a collection of interrelated support tools for teaching different parts of the Computer Science and Software Engineering curricula. The project affilliates itself, but is not restricted to, to the research in exploring Functional Programming as a rapid prototyping environment for formal software models, whose origin can be traced back to P. Henderson's me too (Henderson, 1984).

In the way, some new theoretical and technological results — namely a component classification and reification calculus and a notion of connectable high-level prototyping environment — were achieved and incorporated in the project.

As a working tool for software engineers it offers a simple set-theoretic notation and a fully connectable environment. As a learning tool supporting a Computer Science curriculum, it aims to be easy to understand and use, and to stimulate a kind of abstract and compositional reasoning which paves the way to sound methodological principles.

The CAMILA platform is organized around 5 main components:

- An executable (functional) specification language directly based on naive set theory.

- An inequational calculus (Oliveira, 1990; Oliveira, 1992) — SETS — for refining and classifying software formal models. In particular it enables the synthesis of target code programs by transformation of the initial specifications.

- A flexible rapid prototyping kernel which bears "full citizenship" at C/C++ programming level (C may call CAMILA services and CAMILA may also invoke external C functions). It is available at both Unix, LINUX and MS/DOS operating systems and may provide services under X-WINDOWS or as a WINDOWS 3.1 DLL. Furthermore the prototyping environment provides a set of communication facilities to animate systems built by composition of independent and concurrent software components.

- A formal software components repository which catalogues available models and a compositional notation based on "software-circuit" diagrams (a shorthand for some piece of mathematics), suggestively resembling the conventional hardware notation.

- An approach to the specification and generation of structural Human-Machine Interfaces, independent of but mirroring the application semantics.

The CAMILA approach to programming technology claims to provide a smooth way to teaching and using (constructive) formal methods in software engineering. Its roots on functional prototyping of information models (Henderson, 1984) has already been referred. Similar motivations may be found either in the research on formal specification methods, such as VDM, Z, RAISE (Haxthausen, 1990), COLD-K (Feijs and Jonkers, 1992) or LARCH (Guttag and Horning, 1993), or on functional programming languages such as ML (Harper and Mitchell, 1986) or MIRANDA (Turner, 1986).

In contrast with the former group one could stress the lighter notation of CAMILA, borrowed from set theory, and the direct correspondence to the prototyping language. But what is, to our knowledge, new is the associated calculus for model reasoning and refinement. On the other hand, CAMILA lacks the sophisticated interface and documentation management features available, for instance, in RAISE.

CAMILA, or at least its prototyping language, may also be compared with other functional languages which achieved a high degree of clarity and expressive power. Although some features of more elaborated languages (eg, effective polymorphism) are absent in CAMILA, we would point out as original features CAMILA's flexibility in being fully connectable to other "galaxies" of the computation universe and easily suited to different application domains.

A.2 THE CAMILA LANGUAGE

A CAMILA specification is a set of software components. Each one is a model that includes type, function and state definitions.

\begin{verbatim}
Model --> Model id  
  TypeDef  
  FunDef  
  StataDef  
  ENUMODEL
\end{verbatim}

\textsuperscript{4}CAMILA is named after a Portuguese 19th-century novelist — Camilo Castelo-Branco (1825 - 1890) — whose immense and heterogeneous writings, deeply rooted in his own time experiences and controversies, mirrors a passionate and difficult life.
Where a type definition has the following form:

\[
\text{TypeDef} \rightarrow \text{TYPE} \\
( \text{id} = \text{TypeModel} ) \\n\text{ENDTYPE}
\]

The basic data type models predefined in CAMILA are:

<table>
<thead>
<tr>
<th>Data Models</th>
<th>CAMILA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets</td>
<td>X-set</td>
</tr>
<tr>
<td>Lists</td>
<td>X-list</td>
</tr>
<tr>
<td>Finite functions</td>
<td>X \rightarrow Y</td>
</tr>
<tr>
<td>Binary Relations</td>
<td>X \leftrightarrow Y</td>
</tr>
<tr>
<td>Alternatives</td>
<td>[X]</td>
</tr>
<tr>
<td>Tuples</td>
<td>T :: X : A</td>
</tr>
<tr>
<td>Integers</td>
<td>INT</td>
</tr>
<tr>
<td>Strings</td>
<td>STR</td>
</tr>
<tr>
<td>Tokens</td>
<td>SYM</td>
</tr>
<tr>
<td>Universe</td>
<td>ANY</td>
</tr>
</tbody>
</table>

CAMILA also provides some other primitive types which do not bear a direct mathematical correspondence but are inherent to its programming environment.

The basic collections of functions associated with CAMILA type constructors (eg, intersection or union of two sets, domain or range of binary relations, application or overwrite of mappings, concatenation of sequences and reduce operators, structure definition by enumeration or comprehension, etc.) are available as primitive functions in the language. So are the propositional connectives and quantifiers. To exemplify, a synopsis of some collections is presented above in the form of tables showing the CAMILA syntax, a brief informal description and the corresponding set theoretic notation.

(Finite) Functions \( \rightarrow X \rightarrow Y \)

<table>
<thead>
<tr>
<th>CAMILA</th>
<th>Description</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom(f)</td>
<td>Domain</td>
<td>dom(f)</td>
</tr>
<tr>
<td>ran(f)</td>
<td>Co-domain</td>
<td>ran(f)</td>
</tr>
<tr>
<td>f[x]</td>
<td>Application</td>
<td>f[x]</td>
</tr>
<tr>
<td>f/s</td>
<td>Dom. restriction</td>
<td>f/s</td>
</tr>
<tr>
<td>f/s</td>
<td>Dom. subtraction</td>
<td>f/s</td>
</tr>
<tr>
<td>f ∨ g</td>
<td>Overwrite</td>
<td>f ∨ g</td>
</tr>
<tr>
<td>[ ... ]</td>
<td>Map. enum.</td>
<td>[...]</td>
</tr>
<tr>
<td>[x→s</td>
<td>x&lt;s; p]</td>
<td>Map. compreh.</td>
</tr>
</tbody>
</table>

Sequences \( \rightarrow X-seq \)

<table>
<thead>
<tr>
<th>CAMILA</th>
<th>Description</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd(s)</td>
<td>Head</td>
<td>hd s</td>
</tr>
<tr>
<td>tl(s)</td>
<td>Tail</td>
<td>tl s</td>
</tr>
<tr>
<td>nth(i,s)</td>
<td>Elem. by pos.</td>
<td>s(i)</td>
</tr>
<tr>
<td>s r</td>
<td>Concatenation</td>
<td>s r</td>
</tr>
<tr>
<td>&lt;x&gt;s</td>
<td>Appending</td>
<td>&lt;x&gt;s</td>
</tr>
<tr>
<td>CONC(s)</td>
<td>concatenation</td>
<td>s_1 s_2 ... s_n</td>
</tr>
<tr>
<td>elems(s)</td>
<td>Set of elements</td>
<td>{ x</td>
</tr>
<tr>
<td>inds(s)</td>
<td>Domain</td>
<td>dom s</td>
</tr>
<tr>
<td>plusq(s,z)</td>
<td>overwrite</td>
<td>s f</td>
</tr>
<tr>
<td>&lt;e</td>
<td>x&lt;s:p&gt;</td>
<td>Seq. compreh.</td>
</tr>
<tr>
<td>o-oric(e,s)</td>
<td>Distribut. form</td>
<td></td>
</tr>
</tbody>
</table>

A function definition has the following form:

\[
\text{FunDef} \rightarrow \text{FHeader FPredCond FState FBody} \\
\text{FHeader} \rightarrow \text{FUNC fid (ParamList) : typeid} \\
\text{FPredCond} \rightarrow \text{PRE CondExp} \\
\text{FState} \rightarrow \text{STATE exp} \\
\text{FBody} \rightarrow \text{RETURNS Exp}
\]

Finally, a state definition is written according to the rule:

\[
\text{StateDef} \rightarrow \text{STATE sid : typeid}
\]
Multi-layered Pipeline Parsing from Multi-axiom Grammars

Teodor Rus and James S. Jones
Department of Computer Science
University of Iowa, Iowa City, Iowa 52242
rus@herky.cs.uiowa.edu

Division of Science and Mathematics
Graceland College, Lamoni, Iowa 50140
jsjones@graceland.edu

Abstract

Multi-axiom grammars have been introduced as an alternative to the single-axiom context free grammars and the all-axiom algebraic grammars for programming language specification. Multi-axiom grammars eliminate some of the limitations of the context-free grammars used for programming language specification and implementation. However, prior algebraic notions regarding multi-axiom grammars such as subgrammar, primitive subgrammar, quotient grammar, and grammar/language hierarchy, were based on limited use of non-axioms. In this paper we use the mechanism of non-axioms to redefine some of these concepts and to develop a naturally parallel algorithm for parsing programming languages specified by multi-axiom grammars. This algorithm is convenient for large program development because it can recognize any phrase in the language and thus facilitates incremental development of programs. The algorithm is efficient and can use the computation power of the parallel machines because it distributes the parsing task among a collection of smaller parsers associated with individual language layers. The algorithm is called a PHRASE parser, an acronym for its actions: Pass, Halt, Reduce, Accept, Shift, and Error.

1 Introduction

Context-free grammars (CFG) are the de facto tools for conventional programming language specification and implementation [8]. They are convenient because they are equivalent with various kinds of finite automata for which there are algorithmic methods that map specification rules into the program that implements the respective automaton. The viable-prefix property [2] implemented by these algorithms restrict their behavior to consume their input in a strictly left-to-right order. Moreover, the single-axiom nature of context-free grammars forces these algorithms to regard their input as a monolithic data specified by the axiom (often program) of the specification grammar. Hence the phrase structure of the language is difficult to handle with context-free grammar based parsers. However, conventional parsing methods [1, 2] divide the parsing task into two-layers, the lexical layer, which uses specification tools and parsing mechanisms which are not single axiom by nature, and syntax layer, which is single axiom by nature. It is tempting to unify the specification of both layers using a unique context-free grammar, G, where a subset of specification rules specify the lexicon of the language and define a subgrammar $G'$ of $G$. But this $G'$ is not the same class of object as $G$, and to make it so would require violating the subset relation by adding a unique start symbol and new rules to it. In short, it is difficult to operate with algebraic notions using context-free grammars.

The algebraic grammars (AG) are similar to CFG except that every variable, or non-terminal, of the language is an axiom [4, 7, 12, 13]. The multi-axiom nature of AGs frees us to consider any phrase of the language as meaningful and acceptable, and provides a natural framework for the definition of algebraic notions, like subgrammar. The primary limitation with the AG is that the language designer cannot use non-terminals as variables for unimportant parts of phrases without elevating such constructs to the same
importance as an acceptable phrase. This limitation results in a smaller class of languages specified by AGs. For instance, the language denoted by the regular expression $a^*b^*c$ cannot be specified by an AG because one cannot avoid using a non-terminal that generates some string in $a^*b^*$, $b^*$, or $b^*c$.

The Multi-axiom grammar (MAG) is a generalization of CFG and AG [16] because it gives the language designer additional flexibility to specify non-axioms. The result of this flexibility is that the class of languages specified is restored to the class of context free languages as illustrated in Figure 1. These

![Figure 1: Classes of languages compared](image)

relations between the context free languages (specified by CFGs), the algebraic languages (specified by AGs), and the multi-axiom languages (specified by MAGs) are easy to show.

In [16], algebraic notions for MAGs were defined and a language layering model based on them was presented (generalizing the conventional two-layered view). The role of non-axioms was limited to the initial (lexical) layer, but now we have expanded the role of non-axioms for all the layers. In Section 2 of this paper we define the algebraic notions and prove the necessary properties that assure a clean layering of a multi-axiom language on a maximum number of layers. The algorithms that operate on a given multi-axiom grammar and split it in subgrammars that specify the language layering are treated in Section 3. Two different dimensions of the parsing process are considered:

- **Parsing by layer** is a one-pass (left-to-right) algorithm to recognize phrases, embedded in the input stream, that are associated with a particular language layer.

- **Multi-pass parsing** is an algorithm that manages multiple applications of layered parsing to ultimately recognize the entire input as a meaningful phrase of the language.

Parsing by layer is incidental to the multi-pass parsing process, whose goal is to recognize any stand alone phrase. Section 4 shows how to construct a multi-axiom LR parser, called a PHRASE parser, for each layer of the language. The interaction between these layers can be tightly coupled (as in a master-slave arrangement) or loosely coupled (as in a pipe arrangement). Section 5 presents the latter, using a collection of PHRASE parsers in pipelined fashion. Results of the current implementation are sketched in Section 6. Conclusions and current research are provided in Section 7.

2 Algebraic Properties of Multi-Axiom Grammars

A multi-axiom grammar is a generalization of the single-axiom context-free grammar and the all-axiom algebraic grammar.

**Definition 1** A multi-axiom grammar (MAG), $G$, is the quadruple $G=(NV, TV, P, AX)$ where $NV$ is the finite set of non-terminals, $TV$ is the finite set of terminals, $P$ is the finite set of productions (or rules), and $AX$ is the set of axioms. The relations among these are: $NV \cap TV = \emptyset$, $AX \subseteq NV$, and $P \subseteq NV \times (NV \cup TV)^*$. For simplicity and without loss of expressive power a multi-axiom grammar is, by definition, free of useless symbols.

**Related notations:** $V$ is shorthand for $NV \cup TV$ (the vocabulary) and $AX$ stands for $NV - AX$ (the non-axioms). For $r \in P$, $r = (A, \alpha)$, the notation $A \rightarrow \alpha$ is generally used; $A$ is the left hand side of $r$ and is denoted by $lhs(r)$ and $\alpha$ is the right hand side of $r$ and is denoted by $rhs(r)$. If $lhs(r) \in AX$ then $r$ is an axiom rule, otherwise $r$ is a non-axiom rule. Conventional notions of parse tree and derivation apply.
as does the familiar notation $\Rightarrow$ for derivation step, $\Rightarrow^*$ for derivation, and $\Rightarrow^+$ for derivation step by rule $r \in P$. If needed, the notations $\Rightarrow^*$ and $\Rightarrow^+$ identify the grammar associated with the derivation.

**Definition 2** Given a multi-axiom grammar, $G = (NV, TV, P, AX)$, a multi-axiom language (MAL) is the set of strings, $L(G)$, defined by the equality $L(G) = \{ \alpha \in TV^* | A \Rightarrow \alpha, A \in AX \}$.

Context-free grammars and algebraic grammars can be reformulated as special cases of multi-axiom grammars. Thus a CFG is written as $(NV, TV, P, \{S\})$ and an AG is written as $(NV, TV, P, NV)$. Hereafter, the term grammar will mean multi-axiom grammar unless otherwise indicated. The familiar notion of sentential form is extended for MAG to mean any derived string from an axiom and a sentence is any derived terminal string from an axiom. The term phrase will be an alias for sentence, since the latter may be confusing because of conventional single axiom usage.

**Notational Conventions:**

1. $G$ denotes $(NV, TV, P, AX)$. Subscripts and superscripts are consistently applied, for example $G_i$ denotes $(NV_i, TV_i, P_i, AX_i)$, $AX_i$ denotes $NV_i' - AX_i'$, and $V_i$ denotes $NV_i \cup TV_i$.

2. $G_{empty} = (\emptyset, \emptyset, \emptyset, \emptyset)$ is the empty grammar and $L_{empty} = L(G_{empty}) = \emptyset$.

3. Unless stated otherwise, $a$ and $b$ represent terminals, $A$ and $B$ are non-terminals (typically axioms), $M$ and $N$ are non-axioms, lower case Greek letters are $V$ strings, $X$ and $Y$ are elements in $V$, and $x$ and $y$ are used in various ways.

4. $[A]$ denotes the set of terminal strings derivable from the non-terminal $A$ in $G$. $[A]$ denotes the same when the respective grammar is obvious. If $A \in AX$, then $[A]$ is called a syntax category of phrases, otherwise $[A]$ is simply a set of intermediate constructs.

**Definition 3** Let $G$ and $G'$ be multi-axiom grammars. $G'$ is a subgrammar of $G$, $G' \preceq G$, iff $AX' \subseteq AX$, $AX' \subseteq \bar{AX}$, $TV' \subseteq TV$, and $P' \subseteq P$. $G'$ is a proper subgrammar of $G$, $G' \prec G$, iff $G' \preceq G$ and at least one of $AX' \neq AX$, $AX' \neq \bar{AX}$, $TV' \neq TV$, or $P' \neq P$ holds.

Whenever $G' \preceq G$, it is easy to show that a proper subset relation between their axiom, non-axiom, or terminal sets can only occur if a proper subset relation exists between their rule sets. Otherwise $G'$ or $G$ would be in violation of the multi-axiom grammar definition and Definition 3 would not apply. Thus a simpler version of the proper subgrammar definition is:

$$G' \text{ is a proper subgrammar of } G, \quad G' \prec G, \quad \text{if and only if } G' \preceq G \text{ and } P' \subset P$$

This version of Definition 3 is more intuitive since one expects a proper subgrammar of a grammar to have fewer rules than the grammar has. It is easy to see that $G' \preceq G$ implies $L(G') \subseteq L(G)$ but $G' \prec G$ does not imply $L(G') \subset L(G)$.

The next two definitions distinguish between two types of non-determinism when dealing with phrases. The first is to be expected (in languages with phrases that have different meanings in different contexts for instance). The second is to be avoided.

**Definition 4** Given a MAG $G$ and a phrase $\alpha \in L(G)$, we say that $\alpha$ is an overloaded phrase with respect to $G$ if it has two parse trees with different root axioms. This is equivalent to saying that $\alpha$ is a member of multiple syntax categories in $L(G)$. $G$ is an overloaded grammar if it specifies overloaded phrases.

**Definition 5** Given a MAG $G$ and a phrase $\alpha \in L(G)$, we say that $\alpha$ is an ambiguous phrase with respect to $G$ if it has two different parse trees with the same root axiom. $G$ is an ambiguous grammar if it specifies ambiguous phrases.

The definitions which follow are the basis for layering the language hierarchically. In these definitions we use the concept that a non-axiom reaches an axiom in the following sense: A non-terminal, $N$,
reaches a symbol \( X \) if \( X \) occurs in some derivation starting at \( N \); if \( N \) reaches the symbol \( X \) then \( X \) is said to be reachable from \( N \). Ambiguous grammars or grammars with non-axioms which can reach axioms complicate any clear notion of language hierarchy. Therefore we use non-axioms to specify local, intermediate constructs so it makes sense not to have them reach axioms. It is trivial to show that constraining MAGs to be free of such non-axioms does not reduce the class of languages specified. The definitions and algorithms throughout the remainder of this paper presume that the grammars are clean (see Definition 6), unless stated otherwise.

**Definition 6** \( G \) is a clean grammar if it is not ambiguous and has no non-axioms which reach axioms.

**Lemma 1** Given that \( G' \prec G \) and \( G \) is clean, then \( G' \) is clean.

**Proof**: Let \( G' \prec G \) in which \( G \) is clean. Seeking a contradiction, suppose \( G' \) is ambiguous. There must be an ambiguous phrase, \( \alpha \), with respect to \( G' \), having same rooted but different parse trees with respect to \( G' \). Since all symbols and rules in \( G' \) are in \( G \), then these trees are parse trees for \( \alpha \) with respect to grammar \( G \). Thus \( \alpha \) is ambiguous with respect to \( G \), so \( G \) is ambiguous, contradicting the main premise that \( G \) is clean. Thus, \( G' \) must be unambiguous. It follows that \( G' \) has no non-axioms which reach axioms by the properties of Definition 3. That is, any \( N \xrightarrow{\Delta} \alpha \Lambda \beta \) implies \( N \xrightarrow{\triangle} \alpha \Lambda \beta \). \( N \in AX \) implies \( N \in AX' \), and \( A \in AX' \) implies \( N \in AX \). So, if \( G' \) had this problem then so would \( G \), making it not clean. Hence, by Definition 6, \( G' \) is clean.

**Definition 7** Given \( G \), the set of primitive symbols of \( G \) is \( AX \cup TV \).

**Definition 8** Given that \( G' \prec G \) and \( N \in NV \), we say that \( N \) is completely defined in \( G' \) if every derivation, \( N \xrightarrow{\Delta} \alpha \), has an identical derivation \( N \xrightarrow{\triangle} \alpha \). That is, if \( N \xrightarrow{\Delta} x \xrightarrow{\Delta} y \xrightarrow{\triangle} \alpha \) then \( r \in P' \).

**Definition 9** \( G' \) is a primitive subgrammar of \( G \), \( G' \prec_{p} G \), if the following conditions hold:

1. \( G' \prec G \)
2. \( \forall r \in P \), if \( \text{lhs}(r) \in AX \) and all symbols in \( \text{rhs}(r) \) are primitive then \( r \in P' \)
3. \( \forall N \in AX', N \) is completely defined in \( G' \)

**Definition 10** The language \( L(G') \) generated by a primitive subgrammar \( G' \) of the multi-axiom grammar \( G \) is called a primitive sublanguage of the language \( L(G) \). \( L(G') \prec_{p} L(G) \).

The differences between the definitions given so far and those in [16] are that a no useless symbols property is now explicitly required for MAGs (Definition 1), a subset relation with respect to non-axioms is now a condition for subgrammars (Definition 3), and non-axioms now have a constraining influence on primitive subgrammars (Definition 9). Definitions 7 and 8 are new ones used in Definition 9. Other changes are simplifications that result from these (the no useless symbols property accounts for many). One effect of the changes is that \( G \) is always a primitive subgrammar of itself, therefore, calling \( G' \) a proper primitive subgrammar of \( G \) will mean \( G' \neq G \) and \( G' \prec_{p} G \). The smallest primitive subgrammar of \( G \) (i.e., the one with the smallest \( P' \subset P \)) is another useful notion. It can be shown that the smallest primitive subgrammar of \( G \) cannot be empty unless \( G = G_{\text{empty}} \).

**Example**: Consider the expression grammar \( G = \{(E, T, Add, Sym, Num), \{+, -, (,), d\}, P, \{E, T\}\} \) and three \( P' \subset P \) specifications in Figure 2. Figure 2a is the smallest primitive subgrammar of \( G \) since the \( T \)-rules must be in \( G' \) by property 2 of Definition 9 and the others by property 2. Figure 2b shows another, and larger, primitive subgrammar. In Figure 2c, \( G' \) is not a primitive subgrammar because it violates all three properties of the definition: \( T \rightarrow Sym Num \) is not in \( G' \) (violating property 2), \( Num \) is incomplete (violating property 3), and \( G' \) is not even a MAG and thus not a subgrammar (violating property 1) because \( Add, Sym, +, - \) are unreachable from an axiom (i.e., useless). Before leaving this example, one observes that the rules for a given non-axiom are either all in the primitive subgrammar or none of them are, but the rules for a given axiom may be split between \( P' \) and \( P - P' \). Also observe that if \( Add \) were an axiom then \( Add \rightarrow + \) would have to be in \( G' \), which would establish "+" to be a
recognizable phrase in $L(G')$. Hence, the language designer effects what constitutes the smallest primitive subgrammar (and sublanguage) by specifying which non-terminals are axioms and which are non-axioms.

**Definition 11** Let $G' \preceq_p G$. For any $x \in L(G)$ we call $\alpha$ a reduced form of $x$, in symbols $x \models \alpha$, with respect to $G'$ and syntax category $[A]_G$ if and only if for some $k \geq 0$ there exist $s_0, \ldots, s_k \in TV^*$ and $A_1, \ldots, A_k \in AX'$ such that $\alpha = s_0 A_1 s_1 \ldots s_{k-1} A_k s_k$, $x = s_0 x_1 s_1 \ldots s_{k-1} x_k s_k$, and $A \xrightarrow{\alpha}^x \alpha \xrightarrow{x}^x$.

**Definition 12** Let $\alpha$ be a reduced form of $x$ with respect to $G'$ and $[A]_G$. We call $\alpha$ a most reduced form (MRF) of $x$, in symbols $x \models_m \alpha$, if and only if $x \models \beta \models \alpha$ for every $\beta$ which is a reduced form of $x$ with respect to $G'$ and $[A]_G$.

**Observation:** The most reduced form has the same form as that expressed in Definition 11 except that $k$ is minimal. Existence and uniqueness for MRFs is assured when working with clean grammars. In addition, the MRF of $x$ cannot be $x$ itself, and therefore $k > 0$, when the grammars are clean.

**Lemma 2** Given that $G' \preceq_p G$ and $G$ is clean, if $x \in L(G)$ then for each $[A]_G$ in which $x$ belongs there exists a unique, $\alpha$, such that $x \models_m \alpha$.

**Proof:** Let $G' \preceq_p G$, $G$ is clean, and $x \in L(G)$. Choose any $[A]_G$ in which $x$ belongs. Since $G$ is clean, $x$ is not ambiguous and so it has a unique parse tree rooted by $A$. Any derivation $A \xrightarrow{\alpha}^x \alpha$, in which $\alpha \xrightarrow{x}^x$, corresponds with a top down traversal of the parse tree in which traversing proceeds only as far as the nodes corresponding to the symbols in $\alpha$. Since this parse tree has a finite number of traversals, there are a finite number of reduced forms of $x$, $x$ being one of them (by Definition 11). To show there is a most reduced form of $x$, it is sufficient to show that for any two reduced forms, a reduced form which is identical or more reduced than each of them exists. Suppose $\alpha_1$ and $\alpha_2$ are arbitrary reduced forms. Thus, $A \xrightarrow{\alpha_1} \alpha_1 \xrightarrow{x}^x$ and $A \xrightarrow{\alpha_2} \alpha_2 \xrightarrow{x}^x$. Consider a top down traversal of this parse tree in which traversing proceeds only as far down as the nodes corresponding with the elements of $\alpha_1$ or $\alpha_2$. This traversal corresponds with some $A \xrightarrow{\alpha_0} \alpha_0$. Since the untraversed subtrees below the $\alpha_1$ and $\alpha_2$ elements apply rules from $G'$ only, the same is true for $\alpha_0$. Hence, $A \xrightarrow{\alpha_0} \alpha_0 \xrightarrow{x}^x$. Since $A \xrightarrow{\alpha_0} \alpha_0 \xrightarrow{x}^x$, $\alpha_0$ is a mix of substrings from $\alpha_1$ and $\alpha_2$ is has the proper form for Definition 11, and since $A \xrightarrow{\alpha_0} \alpha_0 \xrightarrow{x}^x$ it is a reduced form of $x$. Furthermore, $\alpha_0 \xrightarrow{\alpha_0} \alpha_1$ and $\alpha_0 \xrightarrow{\alpha_0} \alpha_2$ since the traversal can continue downward from $\alpha_0$ to the $\alpha_1$ or $\alpha_2$ nodes, respectively, to construct these derivations. So $\alpha_0$ is more reduced. Therefore, among the finite set of reduced forms there is a most reduced form, $\alpha$, such that $\mathbf{x} \models_m \mathbf{\alpha}$. This $\alpha$ is also unique, since if $\beta$ is another MRF, then by definition $x \models \beta \models_m \mathbf{x}$ and $\mathbf{x} \models \alpha \models_m \mathbf{\beta}$, and so $\alpha = \beta$ and uniqueness holds.

**Corollary 3** Given that $G' \preceq_p G$ and $G$ is clean, if $x \in L(G)$ is not an overloaded phrase with respect to $G$ then there is a unique $\alpha$ such that $x \models_m \alpha$.

**Proof:** By Lemma 2 there is a unique MRF for each syntax category containing $x$, but in this case $x$ belongs to only one syntax category so there is only one such MRF.

**Corollary 4** Given that $G' \preceq_p G$ and $G$ is clean, if $G$ is not overloaded then there is exactly one MRF with respect to $G'$ for any $x \in L(G)$.

**Proof:** Extend Corollary 3 to all $x \in L(G)$, none of which are overloaded.
3 Tokenization and the Language Hierarchy

Conventional parsing is a two-layered approach in which the syntax analyzer processes a stream of tokens generated by the lexical analyzer. The syntax grammar specifies a considerably reduced language because each token represents a potentially infinite category of lexemes which have been factored out of the overall language. This idea is formalized for the multi-axiom grammars and languages using the algebraic notions from the previous section.

Formally, a clean grammar, $G$, is used to specify a language in all of its detail and a primitive subgrammar, $G' \prec_p G$, is used to specify a primitive sublanguage of $L(G)$. This is a generalization on using a lexical subgrammar since any primitive subgrammar is allowed. If $G$ is not overloaded, then each $x \in L(G)$ can be represented by a unique most reduced form of $x$ with respect to $G'$ (Corollary 4). Therefore, the members of $L(G)$ can be divided into equivalence classes sharing common most reduced forms (MRFs). The equivalence class for $x \in L(G)$ can be represented by the expression

$$s_0 [A_1] s_1 \ldots s_{m-1} [A_m] s_m$$

where $x \models_m s_0 A_1 s_1 \ldots s_{m-1} A_m s_m$ for certain $A_1, \ldots, A_m \in AX'$ and fixed strings $s_0, \ldots, s_m \in TV^*$. If we replace string $x$ by $s_0 \$A_1 \$s_1 \ldots \$s_{m-1} \$A_m \$s_m$ in which each $\$A_i$ denotes a token representing the category $[A_i]$, then we call this tokenizing the string. Tokenizing all strings of the language simplifies the entire language and is called tokenizing the language. The tokenization process which factors out the lower language from the overall language is a grammar operation, denoted by $G/G'$, which computes the quotient of $G$ by $G'$. To assist with this, an auxiliary function with respect to a given grammar, $G$, is defined by Algorithm 1. The function $reachable_G(X)$ takes some $X \subseteq AX$ and returns the transitive closure of non-axioms which can be reached using only the non-axiom rules.

Algorithm 1 (The reachable function of $X$)

1. $N:=X$ (parameter $X$ is a subset of $AX$)
2. for each $N \in N$ do
   for each $N \rightarrow \alpha M \beta \in P$ do
     if $M \in AX$ then $N:=N \cup \{M\}$
3. Return $N$

*a*-when $N$ changes the outer loop is extended

Algorithm 2 (Quotient grammar construction)

1. $G_q := G_{\text{empty}}$ (i.e., $NV_q := \emptyset, TV_q := \emptyset, P_q := \emptyset, AX_q := \emptyset$)
2. for each rule $r$ in $P-P'$ do
   add $r$ to $P_q$ and if $r$ is an axiom rule then also add $\text{lhs}(r)$ to $AX_q$
   add all terminals and non-terminals occurring in $r$ to $TV_q$ and $NV_q$, respectively
3. for each axiom rule $A \rightarrow \alpha$ in $P'$ do
   add $A \rightarrow \$A$ to $P_q$ and add $\$A$ to $TV_q$
   add $A$ to both $NV_q$ and to $AX_q$
4. $X:=NV_q-AX_q$ (i.e., $X$ is the set of non-axioms in $G_q$ thus far)
5. for each $N$ in $reachable_G(X)$ do
   for each rule $N \rightarrow \alpha$ in $P'$ do
     add $N \rightarrow \alpha$ to $P_q$ and add $N$ to $NV_q$
     add all terminals and non-terminals occurring in $\alpha$ to $TV_q$ and $NV_q$, respectively

Given $G$ and $G'$ such that $G' \prec_p G$, the quotient grammar, $G_q = G/G'$, is computed by Algorithm 2. It does so by factoring out of $G$ the primitive subgrammar axiom rules (step 2), adding tokenized replacements (step 3), and copying whatever non-axiom rules are needed so that $G_q$ is valid (steps 4-5). A non-axiom rule can occur in both $G_q$ and $G'$ whereas an axiom rule cannot (axiom rules are partitioned
between \( G_q \) and \( G' \). Clearly, \( G_q \) specifies the same language, albeit simplified by tokenization, as did grammar \( G \). We say that \( G_q \) and \( G' \) specify adjacent layers of the language, with \( G' \) specifying the lower layer.

The formal notion of tokenization by factoring out a lower layer of the language is more complex with overloaded grammars since an overloaded phrase has several MRFs and so its tokenized string is not fixed by this algorithm. In this case, the contexts \([15]\) associated with the rules play a part in the factoring process (implemented by parsing mechanisms described later).

This completes the methodology for dividing a language into two layers. What follows is an iterative method for a full multi-layered stratification.

It can be shown that the Algorithm 2 preserves the clean property, i.e., if \( G \) and \( G' \) are clean then so is \( G_q/G' \). Thus, when a proper primitive subgrammar of \( G_q \) is specified the layering process can continue over and over until there is no proper primitive subgrammar. Formally, one begins with a given \( G \) which specifies the language and a \( G' \) \( \prec_q \) \( G \) which specifies the initial primitive sublanguage. Then one computes the sequence \((G_0, G'_0), (G_1, G'_1), \ldots, (G_m, G'_m)\) where \( G_0 = G, \ G'_0 = G', \ G'_i \prec_q G_0 \) and for \( 1 \leq i \leq m, G_i = G_{i-1}/G'_{i-1} \), where \( G_i, 1 \leq i \leq m \), is the smallest primitive subgrammar of the grammar \( G_i \) and \( G_m = G'_m \). Thus, assuming that \( G'_0 \) is given, the Algorithm 3 performs a total layering of the language by automating the selection of all but the initial primitive subgrammar. This is achieved by

**Algorithm 3 (Grammar hierarchy construction)**

1. \( G_0 := G; \ G'_0 := G' \); \( X := A \bar{X} \cup TV \); \( i := 0 \)
2. while \( P_i \neq P_i \) do
   2.1 \( i := i + 1 \)
   2.2 \( G_i := G_{i-1}/G'_{i-1} \) (by Algorithm 2)
   2.3 \( X := X \cup AX'_{i-1} \)
   2.4 for each axiom rule \( A \rightarrow \alpha \) in \( P_i \) do
      add \( A \rightarrow \alpha \) to \( P'_i \) and add \( A \) to both \( NV'_i \) and to \( AX'_i \)
      add terminals and non-terminals occurring in \( \alpha \) to \( TV'_i \) and \( NV'_i \), respectively
   2.5 \( Y := NV'_i - AX'_i \) (i.e., \( Y \) is the set of non-axioms in \( G'_i \) thus far)
   2.6 for each \( N \) in \( \text{reachable}_{G'_i}(Y) \) do
      for each rule \( N \rightarrow \alpha \) in \( P_i \) do
      add \( N \rightarrow \alpha \) to \( P'_i \) and add \( N \) to \( NV'_i \)
      add terminals and non-terminals occurring in \( \alpha \) to \( TV'_i \) and \( NV'_i \), respectively

Algorithm 3 since step 1 initializes \( X \) to be the set of primitive symbols and steps 2.4-2.6 compute a \( G'_i \) which is the smallest primitive subgrammar of \( G_i \), with the property that tokenized axioms from previous layers are also treated as primitive symbols (step 2.3). Thus, a maximum number of layers beyond layer 0 are computed. Each step 2 iteration defines a subsequent layer, computing its grammar and primitive subgrammar pair, \((G_i, G'_i)\). Step 2.2 computes \( G_i \), which specifies the original language tokenized by layers 0 to \( i-1 \). Step 2.3 adds newly tokenized axioms to the set \( X \). Step 2.4 selects the minimal set of axiom rules to meet property 2 of the primitive subgrammar definition using the expanded set, \( X \), of primitive symbols. Step 2.6 selects the minimal set of axiom rules necessary to meet property 3 of the primitive subgrammar definition. Termination occurs when no proper primitive subgrammar exists (i.e., \( G'_i = G_i \)). Termination is guaranteed if at least one additional axiom rule is included in each iterated primitive subgrammar computed (which is assured because \( X \) will become larger with each iteration).

The sample grammar in Figure 3 is used for illustration here and later. \( G \) specifies a language of arithmetic expressions and \( G' \) specifies primitive data items embedded in such expressions. The first iteration of Algorithm 3 on the given \( G \) and \( G' \) is illustrated in Figure 4. Figure 5 illustrates the remaining iterations. Figure 6 illustrates how this process partitions the original set of axiom rules. Each partition of the axiom rules is indexed by a grammar layer in correspondence with the language layer it represents. Figure 6 also shows the non-axiom rules that are needed in each layer.
\( G = \{ (E, T, F, Id, Cn, Sym, Num, Add, Mul), \{ +, *, (,), a, d, \}, P, \{ E, T, F, Id, Cn \} \} \)

\( P = \{ E \rightarrow E \ Add \ T, \ Add \rightarrow +, \ E \rightarrow T, \ T \rightarrow T \ Mul \ F, \ Mul \rightarrow *, \ T \rightarrow F, \ F \rightarrow (E), \ F \rightarrow Id, \ F \rightarrow Cn, \ Id \rightarrow a \ Id, \ Id \rightarrow a, \ Cn \rightarrow Sym \ Num, \ Num \rightarrow d \ Num, \ Num \rightarrow d, \ Sym \rightarrow -, \ Sym \rightarrow c \} \)

\( G' = \{ (Id, Cn, Sym, Num), \{ a, -, d \}, P', \{ Id, Cn \} \} \)

\( P' = \{ Id \rightarrow a \ Id, \ Id \rightarrow a, \ Cn \rightarrow Sym \ Num, \ Num \rightarrow d \ Num, \ Num \rightarrow d, \ Sym \rightarrow -, \ Sym \rightarrow c \} \)

Figure 3: Example multi-axiom grammar and primitive subgrammar

![Figure 4: First iteration of loop in Algorithm 2](image)

(a) \( G \) and \( G' \) as given  
(b) \( G_1 = G_0 / G'_0 \) computed  
(c) \( G'_1 \) computed

Figure 4: First iteration of loop in Algorithm 2

![Figure 5: Second, third, and forth iterations in Algorithm 2](image)

Figure 5: Second, third, and forth iterations in Algorithm 2

<table>
<thead>
<tr>
<th>Axiom Rules</th>
<th>Non-axiom Rules</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E \ Add \ T )</td>
<td>( Add \rightarrow + )</td>
<td>4</td>
</tr>
<tr>
<td>( F \rightarrow (E) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( Mul \rightarrow * )</td>
<td>3</td>
</tr>
<tr>
<td>( T \rightarrow T \ Mul \ F )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F \rightarrow F )</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>( F \rightarrow Id )</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( F \rightarrow Cn )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Id \rightarrow a \ Id )</td>
<td>( Num \rightarrow d \ Num )</td>
<td>0</td>
</tr>
<tr>
<td>( Cn \rightarrow Sym \ Num )</td>
<td>( Sym \rightarrow c )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Original rules by layer
4  Parsing by Layer (Phrase Parsing)

A PHRASE parser for layer \(i\) can be constructed using \(G'_i\) (computed by Algorithm 3). PHRASE is an acronym for the parsing actions involved: *Pass, Halt, Reduce, Accept, Shift, Error*. It differs significantly from an earlier model called SHARE (*Shift, Halt, Accept, Reduce, Error*) [16], mostly due to the expanded role for non-axioms. PHRASE parser construction is similar to conventional LR construction [1], but each table is smaller than one monolithic LR table since it is based on only the primitive subgrammar rules. The larger grammar provides the context information to reduce conflicts, however grammar \(G_i\) is not powerful enough to provide this information. The original grammar \(G_0\) is. That is, \(G'_i\) specifies phrases that can occur in a larger context, and \(G_0\) contains all the information needed regarding this context.

One begins by computing a collection of item sets associated with \(G'_i\) (for layer \(i\)). As in conventional LR(0) construction, each item is a rule \(r\) with a scan point, denoted by "\(*"\), within the \(\text{rhs}(r)\). Algorithm 4 specifies this construction. The algorithm (including closure and goto operations) is identical to conventional LR(0) construction with one exception. That is, building the initial item set (state 0) begins by inserting an unscanned item associated with every axiom rule of the subgrammar. For this reason it is called *multi-axiom* LR(0) *item set construction*. The item sets computed represent states and the goto function values represent transitions of a finite state machine (FSM). The FSM steps through viable prefixes of phrases in \(G'_i\) (that is, rightmost sentential forms of phrases) until a handle exists on the stack to be reduced. To illustrate, the FSM constructed by Algorithm 4 on \(G'_5\) (see Figure 5) is shown in Figure 7.

PHRASE parsing is table-driven, bottom-up parsing associated with a given layer, \(i\). The level \(i\) parser is denoted by \(P_i\) and its parse table is \(T_i\). \(P_i\) tokenizes any member of \(L(G_i)\) based on the primitive subgrammar, \(G'_i \prec \rho G_i\). The parser scans the input from a left-to-right and, given the current state and current input at each point, chooses an action from the table for making its next parsing decision. The parser may decide to *pass* the input symbol to the output stream, *halt* the process if all is finished, *reduce* the \(\text{rhs}(r)\) on top of the stack and replace it with \(\text{lhs}(r)\) for some \(r \in P'\), *accept* the current phrase on the stack and output its token, *shift* the input onto the stack, or otherwise handle an *error* situation. In this paper a parse table is assumed to be one entity, not split into separate *action* and *goto* tables (of course it can be implemented as two tables). A variety of approaches exist for constructing and using a PHRASE table. This paper details a simplified method which has been implemented for PHRASE parsing and briefly discusses a more general method. In this simplified version, *accept* parser actions are not explicit in the table, but rather implicit in the *reduce* entries. A parse table entry will be a set of actions, so the parser must choose one if there are several. Multiple actions require conflict resolution by the parser.

*Follow* and *Precede* sets are used to avoid table conflicts and make simple context decisions. Their definitions are given below since they differ slightly from conventional ones (in particular, axioms are included). The symbol "\(\$$\) denotes the begin or end file marker.

\[
\begin{align*}
    \text{Follow}_{G}(N) &= \{ X \in (TV \cup AX \cup \{\$$\}) \mid A \xrightarrow{\beta} \alpha NXB \text{ for some } A \in AX \} \\
    \text{Precede}_{G}(N) &= \{ X \in (TV \cup AX \cup \{\$$\}) \mid B \xleftarrow{\beta} \alpha AXB \text{ for some } A \in AX \}
\end{align*}
\]

The grammar subscript will usually be dropped with the understanding that the original grammar, \(G = G_0\), is the basis for computation. Whenever axioms from \(G_0\) occur in *Follow* and *Precede* sets, they actually represent their token counterparts occurring in \(G_i\), for \(i > 0\). The axiom \(A\), for instance, represents the terminal \(\sharp A\) which can occur in the I/O streams. The relevant *Follow* and *Precede* sets computed for the example \(G'_5\) is shown in Figure 8. There are symbols in these sets that \(P_3\) will not see if parsing at lower layers has been done. For instance, \(a\) and \(d\) will have been consumed by \(P_0\). The presence of these symbols gives maximum flexibility for multi-pass parsing later on, allowing layer-by-layer parsing to occur in the wrong order without error.

With all this in view, Algorithm 5 constructs the PHRASE parse table, \(T\), which maps any member of \(S \times V_5\) to a set of parse actions (\(V_5\) denotes \(V \cup \{\$$\}\)). *Follow* sets are used in step 4.4 to eliminate some *shift/reduce* and *reduce/reduce* conflicts. The algorithm implements alias usage of an axiom, \(A\), for its token, \(\sharp A\), and the comments in steps 4.1 and 4.2 show how this idea reduces the table size. The effect
Algorithm 4 (Multi-axiom LR(0) item set construction)
Given a primitive subgrammar, $G'$, construct item sets $S = \{S_0, S_1, \ldots, S_n\}$ as follows:

1. $S_0 := \text{closure}(\{ [A \rightarrow \bullet \alpha] \mid A \rightarrow \alpha \text{ is an axiom rule in } G' \})$
2. $n := 0; S := \{S_0\};$
3. for each $S_i \in S$ do
   for each item $[A \rightarrow \alpha \bullet X \beta] \in S_i$ do
     $S := \text{goto}(S_i, X)$
     if $S \not\in S$ then
       $n := n + 1; S_n := S; S := S \cup \{S_n\};$

![Diagram of Algorithm 4](image)

Figure 7: Multi-axiom LR(0) item sets for $G'_3$

of this on the FSM diagram in Figure 7 is to eliminate all tokenized rule items and to prune $S_2$, $S_3$, $S_4$, and $S_5$ from the diagram. The constructed PHRASE table, $T_0$, for the example is given in Figure 9. For illustration, the table is divided into sections: terminals (including $\$)$, axioms, and non-axioms. $Sh_i$ denotes the shift action and a transition to state $i$. $R_i$ denotes the reduce action based on rule $i$ (in this example, rules 1-3 are $E \rightarrow T$, $T \rightarrow T \bullet \text{Mul} \bullet$, and $\text{Mul} \rightarrow \bullet$, respectively). Numerals in non-axiom columns are goto states following a reduction (Algorithm 5 shows $\text{Shift}_k$ being entered, but only the $k$ is relevant in non-axiom columns). Obviously, table reduction methods could be implemented (identical columns folded together for instance). Since axiom columns double for token columns, the shift actions have dual meaning. For instance, the $Sh_8$ in column $F$ means either of two things depending on the situation at parse time: (1) shift input token $\#F$ and go to state 8; or (2) go to state 8 after a reduction by an $F$-rule. In either case the effect is the same.

Algorithm 6 details the parsing process. It references the given parse table, $T$, and manages the Stack (with the usual PUSH, POP, and TOP operations) in order to output the tokenized form of the input. Each Stack element has three components and the first two, the state and symbol, are the same as in LR parsing. The third is the most reduced form derivative (MRFderiu) of the second component, $X$.

<table>
<thead>
<tr>
<th>$x \in NV'_3$</th>
<th>$\text{Precede}_{G'_3}(x)$</th>
<th>$\text{Follow}_{G'_3}(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>${$, }$</td>
<td>${$, +, }$</td>
</tr>
<tr>
<td>$T$</td>
<td>${$, +, }$</td>
<td>${$, +, *}$</td>
</tr>
<tr>
<td>$\text{Mul}$</td>
<td>${T, F, }, Id, Cn, a, d}$</td>
<td>${F, (, Id, Cn, a, d, -}$</td>
</tr>
<tr>
<td>$F$</td>
<td>${$, +, *}$</td>
<td>${$, +, }$</td>
</tr>
<tr>
<td>$Id$</td>
<td>${$, +, *}$</td>
<td>${$, +, }$</td>
</tr>
<tr>
<td>$Cn$</td>
<td>${$, +, *}$</td>
<td>${$, +, }$</td>
</tr>
</tbody>
</table>

Figure 8: Precede and Follow sets for $G'_3$ non-terminals
Algorithm 5 (PHRASE parse table construction)
Given $G$ and $G'$, where $G' \prec_p G$, construct the parse table, $T$, as follows:

1. Construct item sets $S = \{S_0, S_1, \ldots, S_n\}$ by Algorithm 4
2. $T[S_i, X] := \emptyset$ (empty set) for each $S_i \in S$ and each $X \in V_q$
3. $T[S_0, \$] \leftarrow \text{Halt}$ ($\$\$ denotes EOF and $\leftarrow\$$ denotes set insertion)
4. for each item set $S_i = S_0, \ldots, S_n$ do
   for each item $i$ of $S_i$ do
      if $i$ is in the form $A \rightarrow \alpha \cdot X \beta$ then break
      Note: eliminate row for $S_i$ due to alias use of $A$ for $\alpha \cdot X$
      else if $i$ is in the form $A \rightarrow \alpha \cdot X \beta$ then break
      Note: eliminate row for $S_i$ due to alias use of $A$ for $\alpha \cdot X$
      else if $i$ is in the form $A \rightarrow \alpha \cdot X \beta$ then
         $T[S_i, X] \leftarrow \text{Shift}_k$ (where $S_k = \text{goto}(S_i, X)$)
      else
         if $X \in \text{Follow}(A)$ then $T[S_i, X] \leftarrow \text{Reduce}_{A \rightarrow \alpha}$
         else $T[S_i, X] \leftarrow \text{Error}$
5. for each $T[S_i, X]$ that is empty do
   5.1 if $i = 0$ and $X \in AX \cup TV$ then $T[S_0, X] \leftarrow \text{Pass}$
   5.3 else $T[S_i, X] \leftarrow \text{Error}$

<table>
<thead>
<tr>
<th>State</th>
<th>Terminals</th>
<th>Axioms (tokens)</th>
<th>nonAx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$$ \quad + \quad -$</td>
<td>$a \quad d \quad -$</td>
<td>$E \quad T \quad F \quad Id \quad Ch$</td>
</tr>
<tr>
<td>1</td>
<td>$R_1 \quad R_1 \quad Sh_7 \quad R_1$</td>
<td>$P \quad Sh_1 \quad P \quad P \quad P$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$R_2 \quad R_2 \quad R_2 \quad R_2$</td>
<td>$Sh_8 \quad R_3 \quad R_3 \quad R_3$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$R_3 \quad R_3 \quad R_3 \quad R_3$</td>
<td>$R_3 \quad R_3 \quad R_3$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Parse table for $G_3$ (empty cells are Error actions)

It is defined as the smallest derivation (i.e., fewest derivation steps) beginning with $X$ and ending in a string free of non-axioms (thus it is $X$ itself unless $X$ is a non-axiom). The $\text{MRFder}$ enables the parser to flush the contents of the stack in a form suitable for output (since non-axioms cannot be output). Whenever the input is valid and the parser reaches an action calling on it to flush the stack, then at that point the $\text{MRFder}$ of $X$ is that portion of the entire input's MRF which is derived by $X$. To maintain this the algorithm identifies the symbol itself as the $\text{MRFder}$ component to be pushed on Stack, except when a reduce on a non-axiom rule, $N \rightarrow \alpha$, occurs. In this case, the $\text{MRFder}$ of all symbols in $\alpha$ are concatenated to yield the $\text{MRFder}$ of $N$. The flush operation shown in Algorithm 6 flushes the stack by writing each $\text{MRFder}$ string on the stack, from bottom to top, and resets Stack to empty. Of course, any time this algorithm outputs an axiom (by Output or Flush) it is its respective token which is actually written.

In our simplified implementation of PHRASE parsing, the one described here, a reduce table entry represents any of three actions. First, it can be a reduce action when a standard goto transition to the next state exists (always the case with non-axiom rule reductions). Second, it can be an accept action when no goto state exists. Third, it can abort the reduction and pass the stack contents to the output when the Precede context is not appropriate for reduce nor accept. Implicit in each $\text{Reduce}_{A \rightarrow \alpha}$ action for an axiom, $A$, is a potential Accept$_A$ action, since the $A$ goes into the $\text{MRFder}$ and its respective token will be output when the stack is flushed, if not removed by further reduction. Some left context information is maintained by the stack itself (i.e., the states), but whenever the stack gets emptied there is none. In this implementation a simple lookback function identifies the most recent symbol written to the output (the begin file marker initially). This is compared with the appropriate Precede set. Multi-axiom
Algorithm 6 (PHRASE parsing process)

1. Create empty Stack of (State, Symbol, MRFderiv)
   Note: Top of empty Stack is (S₀, null, null)
2. let x be the first symbol in the input stream
3. repeat forever
   case Action = $T[\text{Top}_{\text{state}}, x]$ of
      Action = Pass:
      if empty Stack then
         OUTPUT x and get next input, x
      else
         FLUSH the Stack
      Action = Halt:
      FLUSH the Stack and exit loop
      Action = Reduce$_{A \rightarrow 
      \alpha}$:
      POP $|\alpha|$ elements from Stack and
      $Z :=$ concatenation of all popped MRFderiv strings
      if $A \in AX$ then
         PUSH $(\text{goto}(\text{Top}_{\text{state}}, A), A, Z)$ onto Stack
      else if non-empty Stack or else lookback $\in$ Precede$(A)$ then
         if $\text{goto}(\text{Top}_{\text{state}}, A)$ is defined then
            PUSH $(\text{goto}(\text{Top}_{\text{state}}, A), A, A)$ onto Stack
         else Note: this is the Accept action
            FLUSH the Stack and then OUTPUT A
      else
         OUTPUT Z
      Action = Shift$_{z}$:
      PUSH $(s, z, z)$ onto Stack
      get next input, x
      Action = Error:
      handle error (or treat like Pass)

*there should be only one (otherwise resolve conflict)
*output each MRFderiv from bottom to top of Stack and empty it

LR(0) item set construction requires that both left and right contexts be consulted if a reasonable class of grammars can be handled. Figure 10 shows stack activity for a sample input to $P_3$. When $T$ is the only stacked symbol and when $\text{"\textquotedblright}$$"$ is the input, the parser must decide whether to reduce or pass. If “$\text{"\textquotedblright}"$ is the preceding symbol it reduces to $E$ since “$\text{"(E)}$" can occur in an MRF. If “$+$” is the preceding symbol it passes because “$\text{"+E}"$ is not in any MRF.

Because simple Follow and Precede sets are used to avoid conflicts, this parsing method is called Simple Multi-axiom LR (SMLR) parsing (somewhat analogous to conventional SLR parsing).

4.1 WHEN SMLR PARSING DOES NOT WORK

If the use of simple Follow sets in Algorithm 5 is unable to prevent conflicts in the table (i.e., multiple actions) or the use of simple Precede sets in Algorithm 6 does not properly distinguish between a reduce and pass situation, then SMLR parsing will not work and a more powerful mechanism must be employed. PHRASE parse table conflicts can be resolved automatically during parsing using powerful context information about potential conflict points. Preprocessing of the grammar is done to compute sets of contexts [14, 15] which may be found around a category of phrases and are associated with the derivation rules specifying such phrases. For a specification rule, $r$, Context$(r)$ is a set of pairs of variable-length strings $(x, y)$ of length $m, n$ respectively, referred to as $(m, n)$ contexts, that determine whether or not the rule $r: M \rightarrow \omega$ has been used in a derivation to generate a phrase of the form $\alpha x w y \beta$ from a syntactically valid phrase of the form $\alpha M y \beta$. That is, from the parsing viewpoint the pair
Input = $T \ast (\$T + \$t)$

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift$_1$</td>
<td>0</td>
<td>$T$</td>
</tr>
<tr>
<td>Shift$_2$</td>
<td>0 $T$ 1 *</td>
<td>$T$</td>
</tr>
<tr>
<td>Reduce$_{Mul\rightarrow}$</td>
<td>0 $T$ 1 * 7</td>
<td>(</td>
</tr>
<tr>
<td>Pass</td>
<td>0 $T$ 1 $Mul$ 6</td>
<td>(</td>
</tr>
<tr>
<td>Pass</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>Shift$_3$</td>
<td>0</td>
<td>$T$</td>
</tr>
<tr>
<td>Reduce$_E\rightarrow$</td>
<td>0 $T$ 1</td>
<td>$+$</td>
</tr>
<tr>
<td>Accept$_E$</td>
<td>0 $E$ (to goto)</td>
<td>$+$</td>
</tr>
<tr>
<td>Pass</td>
<td>0</td>
<td>$+$</td>
</tr>
<tr>
<td>Shift$_4$</td>
<td>0</td>
<td>$T$</td>
</tr>
<tr>
<td>Reduce$_E\rightarrow$</td>
<td>0 $T$ 1</td>
<td>(</td>
</tr>
<tr>
<td>Pass (so reduce)</td>
<td>0 $T$ 1</td>
<td>(</td>
</tr>
<tr>
<td>Pass</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>Halt</td>
<td>0</td>
<td>$S$</td>
</tr>
</tbody>
</table>

Output = $T \ast (\$E + \$T)$

Figure 10: $P_3$ activity on sample input

$(x, y) \in Context(r)$ has the property that whenever a syntactically valid input string has the form $\alpha x \omega y \beta$ it can be reduced to $\alpha x M y \beta$ preserving its syntactic validity.

If a conflict is associated with rules $r_1: A\rightarrow\alpha$ and $r_2: B\rightarrow\alpha$ for a non-ambiguous grammar, then rules $r_1$ and $r_2$ will have different contexts associated with them. During parse table construction, when a reduce/reduce conflict is identified, the contexts identifying syntax categories $[A]$ and $[B]$ are computed and attached to the table at the conflict entry. During a parse the parser makes use of this additional precomputed information by means of the LookAround method, a formal means of conflict resolution that consists of two actions, LookAhead and LookBack. Let $Top(Stack)$ be the rhs(r) on the top of the stack for some r when a conflict point is reached. The parser uses LookAhead and LookBack to examine the current context of $Top(Stack)$ and compare it with the precomputed Context(r) for all conflicting r to determine the rule r. The LookAhead action examines the next symbols in the input while LookBack examines the previous symbols on the stack. For each (m, n)-context in Context(r) at most n lookahead and m lookback symbols must be examined. The LookAhead is the same as with any LR(k) parser, is well understood and needs no further comment. We explain the LookBack action, by which the stack history (i.e., conceptually the stack extended further downward by the elements written to the output) is examined, as follows:

If a grammar contains the productions $A\rightarrow\alpha$ and $B\rightarrow\alpha$, then a reduce/reduce conflict may appear. When such a conflict is identified, the grammar is examined and productions that contain A or B in their right-hand sides (that is, productions of the form $X\rightarrow\beta A\delta$ and $Y\rightarrow\beta B\delta$) are located. If the input text belongs to the language, then the stack history must contain some $\beta'$ or $\beta''$. By determining which $\beta$ is in the stack history, the conflict can be resolved.

In computing the LookBack context, only as much of the context as is needed to resolve a conflict is computed and recorded. In the above explanation, the set of last symbols of each $\beta'$ and the set of last symbols of each $\beta''$ are initially computed and their intersection is taken. If the intersection is empty, the conflict is resolvable by a LookBack(1) action. If the intersection is not empty, LookBack(2) is computed only for those symbols in the intersection. The process continues until the conflict is resolvable for all contexts. In this way the $k$ involved in any LookBack action is always the smallest possible.

Shift/reduce conflicts fall into two categories. If the conflict is based on the two items $[A\rightarrow\alpha]c\omega$ and $[B\rightarrow\alpha]c\beta$, then the conflict is resolvable by a LookBack action, like the above situation. If the conflict is based on the two items $[A\rightarrow\alpha]c\omega$ and $[A\rightarrow\alpha]c\beta$, then the conflict is resolvable by a LookAhead action.

In essence the LookBack action is a pattern-matching algorithm that uses the string represented by the entire stack history as the pattern to be matched. Because this includes a portion of the input string
already parsed, the pattern string may contain tokens. Furthermore, since each token is a placeholder for a syntax category of the grammar, it can represent a large number of sub-phrases of varying length as portions of the input string. Thus, the LookBack action performs a context check using a finite context that may represent a potentially infinite number of strings of potentially infinite length in the text. Therefore, it is extremely powerful.

To summarize, the LookAround method makes use of the complete state of the parsing machine. The entire stack history for a given pass, if necessary, can be examined in order to resolve a parsing conflict, or additional input symbols can be inspected, if needed. Note that this is not the same as an LR(k) parser for an arbitrary k. In such a parser the entire k-lookahead table is computed and used by the parsing machine. In the LookAround method additional context necessary for performing the actions LookBack which examines the stack history and LookAhead which examines the input, is computed and recorded only at points of conflict. Such points of conflict are found in the parse table building process. The number of LookBack or LookAhead contexts are determined by each separate conflict, and only the contexts necessary to resolve a conflict are computed. All LookBack or LookAhead computation is done in the parse table construction phase, and only such additional information as is needed is attached to the parse table.

The PHRASE parser with LookAround, then, is a variable k parser rather than a fixed k parser. At each point in the parse table the smallest k necessary to provide a deterministic parse is computed, and only that necessary information is retained. Usually this k is 0. Only a small number of entries need a k equal to or greater than 1, and then the k is still a small number. Such a parser can be efficiently implemented.

5 Multi-pass Parsing

Multi-pass parsing utilizes the collection of PHRASE parsers, \( P_0, P_1, \ldots, P_m \), for the grammar \( G \) hierarchy to recognize the primary phrase which the entire input represents. Each parser, \( P_i \), is identical (Algorithm 6) except for the different parse table it uses, \( T_i \) (Algorithm 5). Each PHRASE parser operates independently, it can process input symbols as they arrive, and it can output symbols as it makes accept and pass actions. As a consequence of this and the fact that each parser scans its input in the same direction (left to right), they can be pipeline parallelized for speed. This process is illustrated in Figure 11.

![Figure 11: Multi-layered pipeline parsing](image)

The diagram shows loops back to previous stages. In the conventional LR parser, such loops are contained within the parser as one observes in the FSM of item sets which usually has many cycles. We have fragmented the parse tables and so there may be cycles in the pipeline as well as within the individual parsers. Not every parsing stage needs to have a loop back to a previous stage, nor does such a loop need to return to the start. The nature of the pipeline loops depends on the characteristics of the stratified grammar and the characteristics of a particular input.

A brute force algorithm for multi-pass parsing is given in Algorithm 7. The UNIX pipe operator symbol, "|", is used to denote piping of the input through a series of parsers. This algorithm will result in many useless passes over the input by individual PHRASE parsers (i.e., those that leave the input unchanged). The algorithm terminates when every one of the \( m+1 \) passes in an iteration is useless, which is an inefficient, but correct, way of knowing that the input has been tokenized as much as it can be. Ideally we want to predict which PHRASE parsers will be useful and only include those in the pipeline operation. For instance, given the input "\( d + (a - dd) * aa * dd \)" for the sample grammar, the brute
Algorithm 7 (Multi-pass pipeline parser)
Given a sequence of parsers $P_0, P_1, \ldots, P_m$, parse input as follows:

\[
\text{repeat forever; } \\
\text{pipe input through } P_0 P_1 \ldots P_m \text{ transforming it to output} \\
\text{if input=output then exit loop} \\
\text{else let input=output}
\]

force sequence $P_0 P_1 P_2 P_3 P_4$, $P_0 P_1 P_2 P_3 P_4$, can be reduced to $P_0 P_1 P_2 P_3 P_4$. The table in Figure 12 shows this result.

<table>
<thead>
<tr>
<th>Parser</th>
<th>Parser Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial input</td>
<td>$d + (a + -dd) * aa * dd$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>$%Ca + (%Id + %Ca) * %Id * %Ca$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$%F + (%F + %F) * %F * %F$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$%T + (%T + %T) * %F * %F$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$%E + (%E + %T) * %F * %F$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$%E + %F * %F * %F$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$%E + %T * %F * %F$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$%E$</td>
</tr>
</tbody>
</table>

Figure 12: Multi-pass parsing of an expression

As an alternative to pipeline loops, one might implement multi-pass parsing by spawning as many instances of these parsers as needed to achieve maximum parallelism. Thinking of each parser as a filter process, piping can be naturally implemented as illustrated on the same multi-pass example input in Figure 13.

Input
$d + (a + -dd) * aa * dd$

Figure 13: Piping through many PHRASE parsers
During preprocessing of the grammar, repetition coefficients [9, 14] can be computed and used to avoid needless passes. Research in this area has been with algebraic grammars and non-LR parsing mechanisms [9], so more study is required to apply it here.

6 Results

We have implemented the simplified PHRASE parser described in this paper as a feature of a menu driven system called MAGLAB (Multi-Axiom Grammar LAB environment). MAGLAB allows one to open a grammar file, get information about it, modify its axiom, non-axiom, and subgrammar specifications, and observe the effects of all this on the quotient grammar, the grammar hierarchy, the item sets, the parse tables, and on the parsing activity for any input file. All of the illustrations in this paper are results shown with this system. Piping has not been implemented, but the independent operation of the individual parsing layers has. The user can specify, after each pass, which layer of the language to process next. We have implemented PHRASE parsing so that an error action is handled as a pass action. The results have been that the relation output = input always holds true and each $P_i$ operates more like a filter that consumes only that portion of the input which can be accepted and leaving the rest unchanged.

The current implementation uses a micro-scanner as an initial pass to get word strings that are presumed to be atomic in the BNF. Other lexemes are specified by the BNF rules. In practice, some form of micro-scanner is needed at the bottom layer, but whether this can be limited to a mere character classifier remains to be seen. In any case, we have found that a distinction needs to be made between rules whose right hand sides allow whitespace and those that do not (i.e., lexeme rules). So far this has not been done in our experiments.

A clear and careful analysis of multi-pass parsing with PHRASE parsers is premature at this point. The operation of the stack in PHRASE parsing differs from classical LR parsers, and implementation issues which affect its efficiency are still evolving. For instance, the pass action results in the stack being flushed and the various ways this is handled effects the overall analysis since flushing implies that needless reductions may have occurred and symbols may be reconsidered several times more. The current implementation flushes the entire stack, but the costs and benefits of partial flushing (i.e., re-mapping the remaining stack of state numbers) is being explored. One thing is clear, however, that pipeline parallel parsing can take place in a manner not done before.

7 Conclusions

The language designer can use multi-axiom grammars as the specification tool for programming languages. In doing so the designer can approach the task of language design as both algebraist and systems engineer. The algebraist identifies which variables of the overall specification are axioms, i.e., the specifiers of the true phrases of the language, and considers how the choice of axioms and axiom rules affect the language hierarchy. The systems engineer uses axioms and non-axioms to achieve readability and modularity in the language specification, and non-axioms become tools for localized (or hidden) intermediate constructs.

Dividing the language into layers allows us to divide the task of a parser of the language into many subtasks, reducing its complexity. This language layering model also suggests that new languages with sublanguage features of existing languages can be incrementally developed by reusing subgrammar specifications. Since this model provides a basis to simplify the design of the parser, it suggests that new language translators can be incrementally developed as well.

The approach described in this paper, therefore, is quite different from conventional parsing because:

- any phrase of the language can be recognized
- a many layered approach is used
- the same specification tool is used to specify each language layer
- the same mechanism is used to parse each layer
- the stages of parsing are independent so that a loosely coupled pipeline can be implemented
Areas for future study:

Grammars with non-axioms that reach axioms are worth further consideration since there may be good reasons to give language designers that freedom. To do so, however, one must reconsider the algorithm for grammar hierarchy construction and develop a clear concept of what constitutes the next incremental layer at each iteration in that situation. Treating all non-axioms as primitive symbols is a poor assumption in this case.

There may be benefits in having the language designer explicitly specify the choice of primitive subgrammar for each layer rather than making it be the smallest primitive subgrammar.

Handling error actions like pass actions has proved to be convenient. Therefore, alternatives to conventional error handling may be useful to explore, such as an error checking mechanism which monitors the overall multi-pass parsing process separate from and parallel with the PHRASE parsers.

REFERENCES


Parsing Schemata and Correctness of Parsing Algorithms

Klaas Sikkel

GMD – Forschungszentrum Informationstechnik,
FIT.CSCW, Schloß Birlinghoven,
53757 Sankt Augustin, Germany,
sikkel@gmd.de

ABSTRACT

Parsing schemata provide a high-level formal description of parsers. These can be used, among others, as an intermediate level of abstraction for deriving the formal correctness of a parser. A parser is correct if it duly implements a parsing schema that is known to be correct.

In this paper it is discussed how the correctness of a parsing schema can be proven and how parsing schemata relate to some well-known classes of parsers, viz. chart parsers and LR-type parsers.

1 INTRODUCTION

Parsing schemata were introduced in [Sik93] as a framework for high-level description of parsing algorithms, both parallel and sequential. A parsing schema abstracts from implementation details of an algorithm like data structures and control structures. A prime application of this framework is the analysis of relations between different parsing algorithms by studying formal relations between their underlying parsing schemata. For a concise introduction, see [Sik94].

Here we concentrate on correctness, an aspect of parsing schemata that has not been treated very extensively so far. Formal correctness proofs are easier for schemata than for algorithms, simply because there is much less to prove. A general proof method for parsing schemata will be introduced and illustrated with examples.

We do not dwell on implementation details of parsing algorithms, but discuss some general relations between parsing schemata and some well-known classes of algorithms.

Parsing schemata are informally introduced in Section 2 and formalised in Section 3. While this includes a notion of correctness of a schema, it is not clear how to proceed when one wants to prove a given schema correct. This subject is dealt with in Section 4.

How chart parsers relate to parsing schemata is the subject of Section 5. Parsers based on pushdown automata are treated in Section 6.

In Section 7 we briefly review the relation between context-free parsing and unification grammars, which have become the predominant grammar formalism in natural language processing. Conclusions follow in Section 8.

2 Parsing schemata

We introduce the general idea of a parsing schema by means of a few informal examples. A more rigorous treatment follows in Section 3.

The following conventions apply throughout this article:

A context-free grammar is a 4-tuple \( G = (N, \Sigma, P, S) \), with \( N \) a set of nonterminal symbols, \( \Sigma \) a set of terminal symbols, \( P \) a finite set of productions, and \( S \in N \) the start symbol. Furthermore, \( N \cap \Sigma = \emptyset \). We write \( V \) for \( N \cup \Sigma \).

We write \( A, B, \ldots \in N \) for nonterminals; \( a, b, \ldots \in \Sigma \) for terminals; \( X, Y, \ldots \in V \) for arbitrary variables; \( \alpha, \beta, \ldots \in V^* \) for strings of arbitrary variables; \( \varepsilon \) for the empty string. The letters \( i, j, \ldots \) denote nonnegative integers.

We write \( A \rightarrow \alpha \) for a production \( (A, \alpha) \) in \( P \). The relation \( \Rightarrow \) on \( V^* \times V^* \) is defined by \( \alpha \Rightarrow \beta \) if there are \( \alpha_1, \alpha_2, A, \gamma \) such that \( \alpha = \alpha_1 A \alpha_2, \beta = \alpha_1 \gamma \alpha_2 \) and \( A \rightarrow \gamma \in P \).

The class of context-free grammars is denoted by \( CFG \). An subclass of \( CFG \) is \( CNF \), the class of grammars in Chomsky Normal Form. If \( G \in CNF \) then \( P \) contains productions of the form \( A \rightarrow BC \) and \( A \rightarrow \alpha \) only.
A very simple parsing algorithm is the so-called CYK algorithm \cite{Kas65, You77}, called after Cocke, Younger and Kasami. It is restricted to grammars in Chomsky Normal Form.

Assume that we have some grammar \( G \in \mathcal{CNF} \) and a string \( a_1 \ldots a_n \) to be parsed. The CYK algorithm recognizes items \([A,i,j]\) that satisfy \( A \Rightarrow^* a_{i+1} \ldots a_j \).

The canonical way to implement this is to use a triangular matrix \( T \) with cells \( T_{i,j} \) for all applicable value pairs of \( i \) and \( j \). Recognition of an item \([A,i,j]\) is denoted by adding an entry \( A \) to \( T_{i,j} \). If we have \( a = a_j \) and \( A \Rightarrow a \in P \) then \( A \) can be added to entry \( T_{i-1,j} \). If we have \( B \in T_{i,k}, C \in T_{k,j} \) and \( A \Rightarrow BC \in P \) then \( A \) can be added to \( T_{i,j} \). The CYK algorithm gives an obvious control structure to make sure that all items are recognized that can be recognized.

It is worth noting that the output of the algorithm is not a parse tree, or a collection of parse trees. The output of the CYK algorithm (abstracting from its canonical data structure) is a set of items

\[ \{[A,i,j] \mid A \Rightarrow^* a_{i+1} \ldots a_j \} \]

The string is correct if and only if \([S,0,n]\) is in this set. Moreover, if the string is correct, a parse forest or a particular (e.g. leftmost) parse can be constructed fairly easy from the items in this set. If we have \([S,0,n]\) then there must be \( B, C, \) and \( k \) such that \( S \Rightarrow BC \in P \) and \([B,0,k]\) and \([C,k,n]\) have been recognized as well. So, in a strict sense, CYK is not a parser but a recognizer enhanced with information that facilitates parse tree construction. It is common practice to call this a parser as well, and most parsers discussed in the remainder of this article will be of the same nature.

The way in which the CYK algorithm recognizes items for a given grammar \( G \in \mathcal{CNF} \) and string \( a_1 \ldots a_n \) can be denoted by a logical deduction system, called a parsing system.

**Example 2.1 (CYK)**

Firstly, we define a domain of items

\[ \mathcal{I}_{CYK} = \{[A,i,j] \mid A \in \mathcal{N} \land 0 \leq i < j \} \]

One could restrict \( \mathcal{I} \) to items with \( j \leq n \), of course, but there are some advantages in choosing the domain of items independent of the given sentence. Secondly, we need a set of so-called hypotheses\(^1\)

\[ H = \{[a,i-1,i] \mid a = a_i \land 1 \leq i \leq n \} \]

that represent the string. Thirdly, we need inference rules. We specify an inference rule by a set of deduction steps that covers all instances of inferences. A set of inference rules, therefore, can be denoted by the union of corresponding sets of deduction steps. For CYK we define:

\[ D^{(1)} = \{ [a,i-1,i] \mid A \Rightarrow a \in P \}, \]

\[ D^{(2)} = \{ [B,i,j],[C,j,k] \mid [A,i,k] \mid A \Rightarrow BC \in P \}, \]

\[ D_{CYK} = D^{(1)} \cup D^{(2)} \]

As with the domain \( \mathcal{I} \), we have not bothered to restrict the deduction steps to items with \( j \leq n \). The parsing system \( \mathcal{P}_{CYK} \) for \( G \) and \( a_1 \ldots a_n \) is defined by the triple \( (\mathcal{I}, H, D) \).

A parsing schema CYK is a generalization of \( \mathcal{P}_{CYK} \) to arbitrary strings and arbitrary grammars in \( \mathcal{CNF} \). One can see a parsing schema as a function that yields a parsing system for a given grammar and a given string over the alphabet of that grammar.

The CYK algorithm has the disadvantage that it is restricted to grammars in Chomsky Normal Form. A similar algorithm for arbitrary context-free grammars has been discovered by Earley \cite{Ear68, Ear70}. Different variants of Earley's algorithm exist. First we investigate the one that is closest to CYK, the bottom-up Earley parser.

**Example 2.2 (bottom-up Earley)**

An Earley item has the form \([A \Rightarrow \alpha, \beta, i, j]\), with \( A \Rightarrow \alpha \beta \in P \). The bottom-up Earley parser recognizes the item set

\[ \{[A \Rightarrow \alpha, \beta, i, j] \mid \alpha \Rightarrow^* a_{i+1} \ldots a_j \} \]

for some \( G \in \mathcal{CFG} \) and \( a_1 \ldots a_n \in \Sigma^* \). A recognized item denotes partial recognition of a production. If \( \beta = \varepsilon \), we have recognized a full production—and hence the left-hand side \( A \) corresponding to \([A,i,j]\) in the CYK case. Partially recognized productions can be expanded by "moving the dot rightwards", i.e. recognizing the symbol behind the dot. How to organize this and

\(^1\)Whether the hypotheses are included in the domain of items or not does not really matter. It will turn out to be more convenient to define a separate set of hypotheses.
store the results does not concern us here. We only specify the domain of items, the hypotheses and the deduction steps. For some grammar $G$ and string $a_1 \ldots a_n$ we specify a parsing system $P_{\text{buE}}$ by:

\[
I_{\text{buE}} = \{ [A \to \alpha \beta, i, j] \mid A \to \alpha \beta \in P \land 0 \leq i \leq j \};
\]

\[
H_{\text{buE}} = \{ [a, i - 1, i] \mid a = a_i \land 1 \leq i \leq n \};
\]

\[
D_{\text{Init}} = \{ \vdash [A \to \gamma, i, i] \},
\]

\[
D_{\text{Scan}} = \{ [A \to \alpha \alpha \beta \beta, i, j], [a, j, j + 1] \mid \vdash [A \to \alpha \alpha \beta \beta, i, j + 1] \},
\]

\[
D_{\text{Compl}} = \{ [A \to \alpha \beta \beta, i, j], [B \to \gamma, j, k] \mid \vdash [A \to \alpha \beta \beta, i, k] \},
\]

\[
D_{\text{buE}} = D_{\text{Init}} \cup D_{\text{Scan}} \cup D_{\text{Compl}}.
\]

Deduction steps $D_{\text{Init}}$ are needed to start the deduction of further valid items, hence we have no antecedents. In the definition of $D_{\text{Init}}$ there is no need to state explicitly that $A \to \gamma \in P$ is required, as the deduction steps are only meaningful for items drawn from $I$ and $H$.

$D_{\text{Scan}}$ and $D_{\text{Compl}}$ conform to the scan and complete steps of Earley's algorithm. In Figure 1 it is sketched how the complete step produces an item representing a larger partial parse from two known partial parses.

![Figure 1: The complete step](image)

Earley's original algorithm is more restrictive in the items it recognizes. It makes use of top-down filtering. That is, the recognition of a production is started only if there is a need to do so. Only if we have an item $[A \to \alpha \beta, i, j]$ there is a need to start recognizing a nonterminal $B$ that produces $a_{j+1} \ldots a_k$ for some $k$. Top-down filtering reduces the number of recognized items, but also reduces the possibilities for parallel processing. Earley's algorithm is essentially left-to-right.

Example 2.3 (canonical Earley)

The parsing system $P_{\text{Earley}}$ for a given context-free grammar $G$ and string $a_1 \ldots a_n$ is defined by $I$ and $H$ as in $P_{\text{buE}}$ (cf. Example 2.2) and by $D_{\text{Earley}}$ as follows:

\[
D_{\text{Init}} = \{ \vdash [S \to \gamma, 0, 0] \},
\]

\[
D_{\text{Pred}} = \{ [A \to \beta \beta, i, j], [B \to \gamma, j, k] \mid \vdash [A \to \beta \beta, i, k] \},
\]

\[
D_{\text{Scan}} = \{ [A \to \alpha \beta \beta, i, j], [a, j, j + 1] \mid \vdash [A \to \alpha \beta \beta, i, j + 1] \},
\]

\[
D_{\text{Compl}} = \{ [A \to \beta \beta, i, j], [B \to \gamma, j, k] \mid \vdash [A \to \beta \beta, i, k] \},
\]

\[
D_{\text{Earley}} = D_{\text{Init}} \cup D_{\text{Scan}} \cup D_{\text{Compl}} \cup D_{\text{Pred}}.
\]

The Earley parsing system for $G$ and $a_1 \ldots a_n$ yields the following set of recognized items:

\[
\{ [A \to \alpha \beta, i, j] \mid \alpha \Rightarrow^* a_{i+1} \ldots a_j \land S \Rightarrow^* a_1 \ldots a_k \gamma \text{ for some } \gamma \}.
\]

3 A FORMAL TREATMENT

Parsing systems and parsing schemata are formally introduced in Section 3.1 and 3.2, respectively. Various types of relations between parsing schemata are discussed in 3.3. Section 3.4 reviews the nature of items and introduces a concept of parsing schema correctness.

3.1 PARSE SYSTEMS

Definition 3.1 (parsing system)

A parsing system $P$ for some grammar $G$ and string $a_1 \ldots a_n$ is a triple $P = (I, H, D)$, in which:

- $I$ is a set of items\(^2\), called the called the domain or the item set of $P$;
- $H$ is a finite set of items called the hypotheses of $P$;
- $D \subseteq \mathcal{P}_{\text{fin}}(I \cup H)\times I$ is a set of deduction steps.

Note that $H$ need not be a subset of $I$. $\mathcal{P}_{\text{fin}}$ in the above definition denotes the powerset restricted to finite sets. As a more convenient notation for deduction steps, we write $\eta_1, \ldots, \eta_k \vdash \xi$ rather than $([\eta_1, \ldots, \eta_k], \xi)$. Furthermore if we

\(^2\)From the usual set notation $\ldots \mid \ldots \ldots$ we omit the second part if there are no further constraints on the elements that comprise the set. It should be evident and will be formally stated in Section 3.1) that only items are used that relate to productions of the grammar $G$.\(^3\) Here we treat 'item' as an undefined basic concept. A discussion about the nature of items follows in Section 3.4.
have \( Y = \{\eta_1, \ldots, \eta_k\} \), we may also write \( Y \vdash_\xi \) as an abbreviation for \( \eta_1, \ldots, \eta_k \vdash \xi \).

To be formally correct, however, we make a distinction between the set of deduction steps \( D \) and the inference relation \( \vdash \) on \( p_{hf}(H \cup I) \times I \). We want the inference relation to have the following conventional property:

if \( \eta_1, \ldots, \eta_k \vdash x \) holds, then also 
\( \eta_1, \ldots, \eta_k, \xi \vdash x \) for any \( \xi \).

Therefore we define \( \vdash \) as the closure of \( D \) under addition of antecedents to an inference:

**Definition 3.2 (inference relation \( \vdash \))**

Let \( P = (I, H, D) \) be a parsing system. The relation \( \vdash \subseteq p_{hf}(H \cup I) \times I \) is defined by

\[
Y \vdash \xi \text{ if } (Y', \xi) \in D \text{ for some } Y' \subseteq Y.
\]

**Definition 3.3 (deduction sequence)**

Let \( P = (I, H, D) \) be a parsing system. We write \( \mathcal{I}^+ \) for the set of non-empty, finite sequences \( \xi_1, \ldots, \xi_j \), with \( j \geq 1 \) and \( \xi_i \in I \) (\( 1 \leq i \leq j \)).

A deduction sequence in \( P \) is a pair

\[
(Y; \xi_1, \ldots, \xi_j) \in p_{hf}(H \cup I) \times \mathcal{I}^+,
\]

such that \( Y \cup \{\xi_1, \ldots, \xi_{i-1}\} \vdash \xi_i \) for \( 1 \leq i \leq j \).

As a practical informal notation we write

\[
Y \vdash \xi_1 \vdash \ldots \vdash \xi_j
\]

for a deduction sequence \((Y; \xi_1, \ldots, \xi_j)\).

**Definition 3.4 (\( \vdash^* \))**

For a parsing system \( P = (I, H, D) \) we define the relation \( \vdash^* \) on \( p_{hf}(H \cup I) \times I \) by

\[
Y \vdash^* \xi \text{ if } \xi \in Y \text{ or } Y \vdash \ldots \vdash \xi.
\]

**Definition 3.5 (valid items)**

For a parsing system \( P = (I, H, D) \) the set of valid items is defined by

\[
\mathcal{V}(P) = \{\xi \in I \mid H \vdash^* \xi\}.
\]

We do not make a distinction between semantic validity (usually denoted \( \models \xi \)) and syntactic provability (i.e. \( H \vdash^* \xi \)).

### 3.3 Relations between Parsing Schemata

Various types of relations between parsing schemata can be defined.

A parsing schema \( P_2 \) is a refinement of a schema \( P_1 \) if

- single deduction steps in \( P_1 \) correspond to sequences of deduction steps in \( P_2 \) (and, most likely, \( P_2 \) contains additional items for the additional intermediate results);
- single items in \( P_1 \) correspond to multiple items in \( P_2 \).
A parsing schema $P_2$ is called an extension of a schema $P_1$ if it is defined for a larger class of grammars. A generalization is a combination of refinement and extension (in which either relation can be the identity relation).

We will give a simple, informal example. For formal definitions and a proof of the transitivity of these relations (nontrivial for refinement), see [Sik93, SN96].

**Example 3.11**

$buE$ is a refinement of CYK. This relation is established as follows:

- Firstly, the CYK items $[A, i, j]$ are replaced by final Earley items $[A \to \alpha, i, j]$. Note that a single CYK item is replaced, in general, by a set of Earley items, as there can be different productions with left-hand side $A$. $D^{(a)}$ does now contain deduction steps of the form $[B \to \beta, i, j], [C \to \gamma, j, k] \vdash [A \to BC, i, k]$.

- Secondly, we introduce the remaining Earley items, split each CYK deduction step into the appropriate set of scan and complete steps, and add init steps as in $buE$.

- With the above two simple transformations we have obtained a parsing system $P_{buE}$ for each grammar in CNF and each string. As a trivial third step, we extend the schema to the entire class of context-free grammars.

Refinement means that more deduction steps have to be performed, in order to find all valid items. This is useful when it leads to a qualitative change in the algorithm. $buE$ is more useful than CYK because it handles a larger class of grammars within the same complexity bounds. For a more mundane quantitative improvement, one can try to reduce the number of deduction steps needed to obtain all valid items. This is called filtering. One can differentiate between

- static filtering: deleting irrelevant items and deduction steps;
- dynamic filtering: adding antecedents ( Эти условия ) to deduction steps;
- step contraction: replacing sequences of steps by single steps.

Again, see [Sik93, SN96] for formal definitions.

**Example 3.12**

Earley is obtained from $buE$ by applying a (dynamic) filter, as follows. Deduction steps $\vdash [B \to \beta, i, i]$ are replaced by deduction steps $[A \to \alpha, B \gamma, h, i] \vdash [B \to \beta, i, i]$, except for the remaining init steps in $P_{Earley}$.

### 3.4 Correctness of Parsing Schemata

In order to define a notion of correctness, some understanding of the nature of items is needed. We have seen two kinds of items so far, there are other parsing algorithms that involve different kinds of items. What, exactly is an item?

An item lists a set of constraints on a (partial or complete) parse tree. Recognition of an Earley item $[A \to \alpha \beta, i, j]$ means: There is some tree that has a root labelled $A$ with children labelled $\alpha \beta$ (concatenated from left to right). Moreover, the nodes labelled $\alpha$ are the roots of subtrees that yield $a_{i+1} \ldots a_j$ whereas the nodes labelled $\beta$ are leaves, cf. Figure 2.

![Figure 2: A partially specified tree](image)

One way to interpret an item is to identify it with a set of trees, viz., all trees that satisfy the constraints stated in the item. This approach is taken in [Sik93]. Pursuing this line of thought, an item set is defined by a congruence relation on a set of trees with respect to the deduction relation.

A rather more simple approach is to regard an item as a partial specification of a tree. We assume that there is some general item specification language and that all items used in practical algorithms are (efficient notations for) specific instances of this specification language. We will not further formalize this, because in all practical cases it is abundantly clear what is meant by the various types of items.

Before we define correctness, there are two regularity properties on item sets that have to be stated explicitly.

Firstly, we have tacitly assumed that there is a clear separation between final items, denoting
completed parse trees, and intermediate items, denoting partial, not yet completed trees.

It is possible—but admittedly rather artificial—to contract mixed items that denote a combination of both types. Consider, for example, a grammar in Chomsky Normal Form that has productions \( A \rightarrow SC \) and \( A \rightarrow BC \), with \( S \) and \( B \) not occurring anywhere else in the right-hand side of a production. For the recognition of \( A \), therefore, it is irrelevant whether \([S, i, j]\) or \([B, i, j]\) has been recognized. So we could replace these two items by a single item \([S, B, i, j]\). But then we have a problem with the item \([S, B, 0, n]\). If this item is recognized, it is unclear whether it denotes the existence of a parse tree.

Secondly, we assume that for each parse tree of a sentence, this parse tree conforms to the partial specification of some item in \( \mathcal{I} \).

**Definition 3.13 (semirulearity)**

A parsing system \( P = (I, H, D) \) for a grammar \( G \) and string \( a_1 \ldots a_n \) is called semiruleary if \( I \) does not contain mixed items and each parse tree of \( a_1 \ldots a_n \) conforms to the specification of some item in \( I \).

A parsing schema \( P \) for a class of grammars \( \mathcal{G} \) is semiruleary if \( P(G)(a_1 \ldots a_n) \) is semiruleary for all \( G \in \mathcal{G} \) and all \( a_1 \ldots a_n \in \Sigma^* \).

**Definition 3.14 (correct final items)**

We write \( \mathcal{F}(P) \subseteq \mathcal{I} \) for the set of the final items of a parsing system \( P \) for a grammar \( G \) and a string \( a_1 \ldots a_n \).

A final item is correct if there is a parse tree for \( a_1 \ldots a_n \) that conforms to the specification expressed by this item. We write \( \mathcal{C}(P) \subseteq \mathcal{F}(P) \) for the set of correct final items of \( P \).

**Example 3.15 (final and correct final items)**

1. \( \mathcal{F}(P_{CYK}) = \{ [S, 0, n] \} \);
2. \( \mathcal{C}(P_{CYK}) = \{ [S, 0, n] \} \) if \( a_1 \ldots a_n \in L(G) \), \( \mathcal{C}(P_{CYK}) = \emptyset \) if \( a_1 \ldots a_n \notin L(G) \);
3. \( \mathcal{F}(P_{bue}) = \mathcal{F}(P_{Earley}) = \{ [S \rightarrow \alpha, 0, n] \mid S \rightarrow \alpha \in P \} \);
4. \( \mathcal{C}(P_{bue}) = \mathcal{C}(P_{Earley}) = \{ [S \rightarrow \alpha, 0, n] \mid \alpha \Rightarrow^* a_1 \ldots a_n \} \).

---

**Definition 3.16 (correctness)**

A semiruleary parsing system \( P \) is sound if \( \mathcal{F}(P) \cap \mathcal{V}(P) \subseteq \mathcal{C}(P) \), i.e., all valid final items are correct.

A semiruleary parsing system \( P \) is complete if \( \mathcal{F}(P) \cap \mathcal{V}(P) \supseteq \mathcal{C}(P) \), i.e., all correct final items are valid.

A semiruleary parsing system is correct if \( \mathcal{F}(P) \cap \mathcal{V}(P) = \mathcal{C}(P) \), i.e., it is sound and complete.

A semiruleary parsing schema \( P \) is sound/complete/correct for a class of grammar \( \mathcal{G} \) if \( P(G)(a_1 \ldots a_n) \) is sound/complete/correct for all \( G \in \mathcal{G} \) and \( a_1 \ldots a_n \in \Sigma^* \).

How to show that CYK, bue, and Earley are indeed correct semiruleary parsing schemata will be discussed in the next section.

4 Correctness Revisited

We have introduced a formal notion of correctness, following [Sik93] in a simplified form. We must have such a notion, so as to be able to state that certain useful schemata are formally correct.

But in order to prove correctness, however, the notions introduced in Section 3.4 are not of much help. For a given parsing system \( P \), it is generally trivial to identify \( \mathcal{F}(P) \) and to establish semirulearity of \( P \). Given \( \mathcal{V}(P) \), furthermore, it generally trivial to establish \( \mathcal{F}(P) \cap \mathcal{V}(P) = \mathcal{C}(P) \).

The real problem is to establish \( \mathcal{V}(P) \). In the previous sections we have dodged the issue and tacitly assumed that \( \mathcal{V}(P) \) is known somehow. For the given examples this was not entirely untrue, because the algorithms for which we gave schemata are well known from the parsing literature. We will now repair this omission and discuss a general method to establish the contents of \( \mathcal{V}(P) \). It should be noted that this method is not suitable for automation, because the essential steps require some ingenuity: the criteria for a successful solution are clear, but how to find a successful solution is not stated.

4.1 A Proof Method

Usually, we start with some educated guess of which items are valid and which items are not. The task is to turn this educated guess into a firm proof. This is accomplished as follows.

- Define a set of viable items \( \mathcal{V} \subseteq \mathcal{I} \) for a parsing system \( P \) for an arbitrary \( G \in \mathcal{G} \) and \( a_1 \ldots a_n \in \Sigma^* \). There is only one right choice: the set \( \mathcal{V}(P) \) itself. But at this point we only...
guess what \( \mathcal{V}(P) \) is. It has to be proven that \( \mathcal{V}(P) = W \).

- Show the soundness\(^5\) of all deduction steps, i.e., \( \mathcal{V}(P) \subseteq W \). To that end, it suffices to show that for all \( \eta_1 \ldots \eta_k \vdash \xi \in D \) with \( \eta_i \in H \cup W \), \( 1 \leq i \leq k \), it holds that \( \xi \in W \).

- Construct a so-called deduction length function (dlf) on \( W \). Dlf’s will be defined shortly. Let \( d \) be such a dlf. Then the completeness, i.e., \( W \subseteq \mathcal{V}(P) \), is proven by induction on \( d(\xi) \). From the assumption that items \( \eta \) with \( d(\eta) < m \) are valid, it has to be proven that all \( \xi \) with \( d(\xi) = m \) are valid.

- \( \mathcal{V}(P) = W \) follows from the two previous steps. Generalization from parsing systems to parsing schemata is straightforward as usual.

The difficulty is the choice of an appropriate dlf. A function \( d \) that is guaranteed to be right is \( d(\xi) = m \) if there is a deduction sequence \( H \vdash^m \xi \) (hence the name “deduction length function”). But it is not at all clear a priori how to do this without using induction in the definition of \( d \) that would lead to a circularity in the proof.

We define a dlf in such a way that completeness follows automatically.

**Definition 4.1 (dlf)**

Let \( P \) be a parsing schema, \( W \subseteq I \) a set of items. A function \( d : H \cup W \to N \) is a dlf if

(i) \( d(h) = 0 \) for \( h \in H \)

(ii) for each \( \xi \in W \) there is some \( \eta_1 \ldots \eta_k \vdash \xi \in D \) such that \( \{\eta_1 \ldots \eta_k\} \subseteq W \) and \( d(\eta_i) < d(\xi) \) for \( 1 \leq i \leq k \).

**Proposition 4.2**

Let \( P \) be a parsing schema, \( W \subseteq I \) a set of items, \( d : H \cup W \to N \) a dlf. Then \( W \subseteq \mathcal{V}(P) \). \( \square \)

**Example 4.3 (\( \mathcal{V}(P_{CYK}) \))**

For a parsing system \( P_{CYK} \) for arbitrary \( G \in CNF \) and \( a_1 \ldots a_n \in \Sigma^* \) we define

\[
W = \{ [A,i,j] \mid A \Rightarrow^* a_{i+1} \ldots a_j \}
\]

and a function \( d \) by

\[
d([A,i,j]) = j - i
\]

\(^5\)Soundness means, in this section, soundness of a parsing system with respect to the postulated set of valid items \( W \). This is stronger than then soundness with respect to final items, as stated in Definition 3.16. The same holds for completeness.

Then \( d \) is a dlf. The soundness of \( P_{CYK} \) with respect to \( W \) is trivial, hence \( \mathcal{V}(P_{CYK}) = W \).

**Example 4.4 (\( \mathcal{V}(P_{EB}) \))**

For a parsing system \( P_{EB} \) for arbitrary \( G \in CNF \) and \( a_1 \ldots a_n \in \Sigma^* \) we define

\[
W = \{ [A \rightarrow \alpha \beta, i, j] \mid \alpha \Rightarrow^* a_{i+1} \ldots a_j \}
\]

and a function \( d \) by

\[
d([A \rightarrow \alpha \beta, i, j]) = \min \{ \mu + j - i \mid \alpha \Rightarrow^* a_{i+1} \ldots a_j \}
\]

Then \( d \) is a dlf. Note that we take the minimum in case there are different ways to recognize an item. Again, the soundness is trivial, hence \( \mathcal{V}(P_{EB}) = W \).

### 4.2 Earley is Correct

For the canonical Earley algorithm, represented by the parsing schema \( P_{Earley} \) for an arbitrary grammar \( G \) and string \( a_1 \ldots a_n \), it is not immediately clear how to define a dlf. Therefore we examine in some detail how the Earley algorithm proceeds through a sentence.

![Figure 3: The Earley tree walk](image)

As an exemplary case, we consider a grammar that produced only a single sentence with a single parse tree:

\[
S \rightarrow NP \ VP, \ NP \rightarrow *_{d} *_{n}, \ VP \rightarrow *_{v} NP.
\]

In Figure 3 it is shown that the algorithm performs a tree walk through the parse tree. Every arrow corresponds to (the recognition of) a single item. Predict steps correspond to downward arrows, e.g.

\[
[S \rightarrow NP \ VP, 0, 0] \vdash [NP \rightarrow *_{d} *_{n} 0, 0] \quad (1)
\]

Scan steps only move the dot over a terminal, as in
\[ [NP \to \ast d \ast n, 0, 0], [\ast d, 0, 1] \]
\[ \vdash [NP \to \ast d \ast n, 0, 1], \]
\[ [NP \to \ast d \ast n, 0, 1], [\ast n, 1, 2] \]
\[ \vdash [NP \to \ast d \ast n, 0, 2]. \]  
(2)

Complete steps correspond to upward arrows:
\[ [S \to NP \ast V P, 0, 0], [NP \to \ast d \ast n, 0, 2] \]
\[ \vdash [S \to NP \ast V P, 0, 2] \]  
(4)

and so on.

The soundness of \( P_{\text{Earley}} \) with respect to \( W \) is trivial as usual. For the completeness we define a function \( d : W \to \mathbb{N} \) by
\[ d([A \to \alpha \cdot \beta, i, j]) = \min\{\pi + 2\lambda + 2\mu + j \mid S \Rightarrow^* \delta A \gamma \]
\[ \delta \Rightarrow^\lambda a_1 \ldots a_i \land \]
\[ \alpha \Rightarrow^\mu a_{i+1} \ldots a_j \} \].

Note that \( d \) is indeed properly defined on \( W \).

It remains to be checked that condition (ii) in Definition 4.1 holds for all \( \xi \in W \). We distinguish the following cases:

- \( \xi = [A \to \alpha a \cdot \beta, i, j + 1] \):
  Then \( \eta = [A \to \alpha a \cdot \beta, i, j] \in W \) and \( \zeta = [a, j, j + 1] \in H \). Moreover, \( d(\zeta) = 0 \) and \( d(\eta) = d(\xi) - 1 \).

- \( \xi = [A \to \alpha B \cdot \beta, i, k] \):
  Let \( \alpha \Rightarrow^\mu a_{i+1} \ldots a_j \) and \( B \Rightarrow^\gamma \gamma' \approx a_{j+1} \ldots a_k \) with minimal \( \mu, \gamma \).
  Then \( \eta = [A \to \alpha B \cdot \beta, i, j] \in W \), \( \zeta = [B \to \gamma, j, k] \in W \), and \( d(\eta) < d(\zeta) = d(\xi) - 1 \).

- \( \xi = [B \to \gamma, j, j] \):
  Let \( S \Rightarrow^\pi A_1 \ldots a_j \gamma' \), and let \( \delta, \gamma'' \approx \pi, \lambda, \mu \) such that \( S \Rightarrow^\pi \delta A \gamma'' \).
  Then \( \eta = [A \to \alpha B \cdot \beta, \alpha \Rightarrow^\mu a_{i+1} \ldots a_j] \approx a_{i+1} \ldots a_j \) with \( \pi + 2\lambda + 2\mu \).
  Then \( \eta = [A \to \alpha B \cdot \beta] \in W \), \( \eta \vdash \xi \), and \( d(\eta) = d(\xi) - 1 \).

Hence we conclude
\[ \mathcal{V}(P_{\text{Earley}}) = \{ [A \to \alpha \cdot \beta, i, j] \mid \alpha \Rightarrow^* a_{i+1} \ldots a_j \land S \Rightarrow^* a_1 \ldots a_4 \gamma \}. \]

\[ \square \]

### 4.3 SLR(1) is Correct

There is a close relation between Earley-type algorithms and LR parsers. This will be discussed in Section 6. An LR(0) parser is in fact an implementation of the parsing schema Earley. We will now introduce a schema that employs look-ahead. This constitutes another kind of filtering.

Recognition of an item does not need to take place if the next symbol in the string cannot logically follow, given the context of the item. For the sake of convenience, we augment the grammar with an end-of-sentence marker \( \$ \), and a new start symbol \( S' \).

---

Note, however, that LR-type parsers—also generalized LR-parsers like Tomita’s algorithm—make some restrictions on the class of grammars that can be used.
Definition 4.6 (augmented grammar)
For each grammar $G \in \mathcal{CFG}$ we define an augmented grammar $G'$ by

\[
N' = N \cup \{S'\}, \\
\Sigma' = \Sigma \cup \{\$\}, \\
P' = P \cup \{S' \rightarrow S\}, \\
G' = (N', \Sigma', P', S')
\]

with $\{S', \$\} \cap V = \emptyset$.

For a string $a_1 \ldots a_n$ an augmented grammar has only a single final item $[S' \rightarrow S, S, 0, n]$.

Furthermore, we define the function $\text{FOLLOW}(A) = \{a \mid \exists \alpha, \beta : S' \Rightarrow^* \alpha A \alpha \beta\}$.

Schema 4.7 (SLR(1))
The parsing schema SLR(1) is defined by a parsing system $P_{\text{SLR}(1)}$ for any $G \in \mathcal{CFG}$ and for any $a_1 \ldots a_n \in \Sigma^*$ by

\[
\begin{align*}
\mathcal{T}\text{Compl} &= \{[A \rightarrow \alpha \beta, i, j] \mid A \rightarrow \alpha \beta \in P' \land 0 \leq i \leq j\}; \\
H &= \{[a, i-1, i] \mid a = a_i \lor 1 \leq i \leq n\} \cup \{[\$, n, n+1]\}; \\
\mathcal{D}\text{Init} &= \{ \vdash [S' \rightarrow SS, 0, 0]\}; \\
\mathcal{D}\text{Pred} &= \{[A \rightarrow \alpha \beta, i, j] \vdash [B \rightarrow \gamma, j, j]\}; \\
\mathcal{D}\text{Scan} &= \{[A \rightarrow \alpha \beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha \beta, j, j+1]\}; \\
\mathcal{D}\text{Compl} &= \{[A \rightarrow \alpha \beta, i, j], [B \rightarrow \gamma, j, k], [a, k, k+1] \vdash [A \rightarrow \alpha \beta, i, k] \mid a \in \text{FOLLOW}(B)\}; \\
\mathcal{D}_{\text{SLR}(1)} &= \mathcal{D}\text{Init} \cup \mathcal{D}\text{Pred} \cup \mathcal{D}\text{Scan} \cup \mathcal{D}\text{Compl}.
\end{align*}
\]

Note that it is possible to exploit the look-ahead more efficiently, for example by using $a \in \text{FIRST}([B \text{FOLLOW}(A)])$, rather than $a \in \text{FOLLOW}(B)$ to filter irrelevant items. Also, one could apply a filter to the scan steps. The schema has been defined such that incorporates exactly the same look-ahead that is used in the construction of an SLR(1) parsing table.

Next, we will establish $\mathcal{V}(P_{\text{SLR}(1)})$ with the method introduced above. We define a set

\[
\mathcal{W} = \mathcal{W}\text{Scan} \cup \mathcal{W}\text{Compl} \cup \mathcal{W}\text{Pred} \cup \mathcal{W}\text{Init}
\]

by

- $[A \rightarrow \alpha \beta, i, j+1] \in \mathcal{W}\text{Scan}$ if
  - $S' \Rightarrow^* a_1 \ldots a_i A \gamma \$, 
  - $A \Rightarrow^* a_{i+1} \ldots a_j$;

- $[A \rightarrow \alpha \beta, i, j] \in \mathcal{W}\text{Compl}$ if there is some $j < k$ such that
  - $S' \Rightarrow^* a_1 \ldots a_i A \gamma \$, 
  - $A \Rightarrow^* a_{i+1} \ldots a_j$,
  - $B \Rightarrow^* a_{j+1} \ldots a_k$,
  - $a_{k+1} \in \text{FOLLOW}(B)$;

- $[A \rightarrow B \alpha \beta, i, j] \in \mathcal{W}\text{Compl}$ if
  - $S' \Rightarrow^* a_1 \ldots a_i A \gamma \$, 
  - $A \Rightarrow^* a_{i+1} \ldots a_j$,
  - $B \Rightarrow^* e$,
  - $[A \rightarrow \alpha \beta, i, j] \in \mathcal{W}$,
  - $a_{j+1} \in \text{FOLLOW}(B)$;

- $\mathcal{W}\text{Init} = \{[S' \rightarrow SS, 0, 0]\}$;

- $[C \rightarrow \gamma, t, j] \in \mathcal{W}\text{Pred}$ if there is some
  - $[A \rightarrow \alpha \beta, i, j] \in \mathcal{W}\text{Init} \cup \mathcal{W}\text{Scan} \cup \mathcal{W}\text{Compl}$
  - such that $B \Rightarrow^*_\text{rm} C\delta$ for some $\delta$
  - (where $\Rightarrow^*_\text{rm}$ denotes rightmost derivation).

Note that the recursion in the definition of $\mathcal{W}$ only relates to nullable symbols (i.e., $B$ such that $B \Rightarrow^* e$). These have to be taken proper care of. Consider, for example a grammar $G$ defined by productions $S \rightarrow ABA$, $S \rightarrow CBC$, $A \rightarrow a$, $B \rightarrow c$ and an input string $ac$. Then $[S \rightarrow ABA, 0, 1] \notin \mathcal{W}$. There is a deduction step $[S \rightarrow ABA, 0, 1], [B \rightarrow c, 1, 1], [c, 1, 2] \vdash [S \rightarrow ABA, 0, 1] \in D$, but this is never activated because $[S \rightarrow ABA, 0, 1] \notin \mathcal{W}$. Completion (or, in LR-terminology, reduction) of the first $A$ is blocked by the look-ahead $c$.

The function $d$ on $\mathcal{W}$ is similar to the one defined on $P_{\text{Earley}}$:

\[
d([A \rightarrow \alpha \beta, i, j]) = \\
\min \{\pi + 2\lambda + 2\mu + j \mid S \Rightarrow^* \delta A \gamma \land \delta \Rightarrow^* a_1 \ldots a_i \land A \Rightarrow^* a_{i+1} \ldots a_j\}.
\]

In order to prove that $\mathcal{V}(P_{\text{SLR}(1)}) = \mathcal{W}$ we have to prove:

(i) for all $\eta_1 \ldots \eta_k \vdash \xi \in D$ with $\eta_i \in H \cup \mathcal{W}$,
  \begin{align*}
  1 \leq i \leq k, & \text{ it holds that } \xi \in \mathcal{W} \text{ (soundness);} \\
  & \text{ (i).}
\end{align*}
(ii) for each \( \xi \in \mathcal{W} \) there is some \( \eta_1 \ldots, \eta_k \vdash \xi \in D \) such that \( d(\eta_i) < d(\xi) \) for \( 1 \leq i \leq k \).

(df).

(i) can be verified straightforwardly. For a proof of (ii) we do not have to worry about the \( d \)-values of the items this time. All deduction sequences in \( \mathcal{P}_{\text{SLR}(1)} \) do exist in \( \mathcal{P}_{\text{Earley}} \) as well. The difference is that less items are recognized. The point that must be clarified is, whether for any \( \xi \in \mathcal{W} \) a deduction step \( \eta_1 \ldots, \eta_k \vdash \xi \) can be found with all \( \eta_i \in \mathcal{W} \). The reader may verify that this follows straightforwardly in all cases. Note that the rightmost condition in \( B \Rightarrow_m C \delta \) is essential: it prevents derivations of the type \( B \Rightarrow^* \alpha C \delta \Rightarrow^* C \delta \) in which \( \alpha \) rewrites to \( e \).

Hence, we have shown that \( \mathcal{V}(\mathcal{P}_{\text{SLR}(1)}) = \mathcal{W} \) as defined above.

\[\Box\]

4.4 Other Applications

Establishing the correctness of CYK and buE was trivial, but Earley and SLR(1) required some ingenuity. This raises the question how the proposed proof method "scales up" to more complicated schemata that are not generally known to be correct from the parsing literature. Rather more involved examples are given in [SA92, Sik93, SA96], where correctness of (parsing schemata for) Left-Corner (LC) and Head-Corner (HC) parsers is established using the same technique. A more detailed treatment of schemata for LC and HC parsers would require a lot of space, hence the interested reader is referred to the cited publications. Finding a deduction length function is not more difficult than the Earley case (for LC and HC it is easier, in fact). An analysis of how a single parse tree is processed by the algorithm suggests an appropriate function \( d \). The schemata LC and HC have different kinds of items and more kinds of deduction steps, hence the proof that \( d \) is indeed a \( df \) requires checking quite a large number of different cases. But each of these cases is straightforward as in the above examples. In sum, more elaborate parsing schemata require hardly more complicated proofs. We claim that the predictive Head-Corner parser proposed [SA92, Sik93, SA96] is the only HC parser that has ever been formally proven correct.

Head-Corner parsers do not process a sentence from left to right but start with the "most interesting" parts. In each production some right-hand side element is designated as head. Evaluation of a production always starts with evaluation of the head. Head-first evaluation can speed up the parser when the information known from the head avoids the evaluation of dead ends that would have been explored otherwise. But the fact that the processed part of a sentence is not contiguous burdens the parser with a rather more complicated administration. Other work on Head-Corner parsing can be found in [Kay89, SS89, BN93, Mol95, A&za96].

5 Chart Parsers

So far we have only discussed parsing schemata. Although we will not dwell on specific parsing algorithms in detail, we discuss some general classes of parsing algorithms in this section and the next one.

Parsing schemata are a generalization of chart parsers [Kay80, Kay82, Win83]. From the view that has been unfolded in the previous sections, we can see a chart parser as the canonical implementation of a parsing schema.

A chart parser employs two data structures: An agenda, containing items that will be actively used to search for new items that can be recognized, and a chart, storing the items that need no further attention. The general chart parsing algorithm is shown in Figure 5. An Earley chart parser, for example, is initialized with items \([a, i-1, i]\) on the chart and \([S \rightarrow \gamma, 0, 0]\) on the agenda. The control structure of the chart parser guarantees that the final chart, which is reached when the agenda is empty, contains \( \mathcal{V}(\mathcal{P}) \). It needs no further elaboration that if a parsing schema is correct, then also the chart parser for this schema is correct.

In the most general form a chart parser is not particularly efficient. In order to speed up parsing, the chart and agenda have to be enhanced with data structures that allow efficient searching and storing of relevant items. How this can be done for various kinds of chart parsers is beyond the scope of this article.

6 Pushdown Automata

An important class of parsing algorithms is based on the notion of a Pushdown Automaton (PDA). A fundamental theorem in formal language theory states that the class of languages accepted by (nondeterministic) PDA's is equal to the class of languages generated by context-free languages.
program chart parser
begin
  create initial chart and agenda;
  while agenda is not empty
    do
      delete some (arbitrarily chosen)
      current item from agenda;
      for each item that can be recognized
      by current in combination with
      other items in chart
        do
          if item is neither in chart
          nor in agenda
            then add item to agenda
          fi
        od
    od
end.

Figure 5: The chart parser algorithm

6.2 FROM SCHEMATA TO PDA'S

The question arises how PDA-based algorithms like LR and parsing schemata are related to each other. To that end, we will transform the schema Earley (or, to be precise, an uninstantiated parsing system for some grammar $G$) to a PDA, and argue that its correctness (in the sense of Section 3.4) is preserved.

We use a somewhat opportunistic definition of a PDA, that is tuned towards the description of parsers. This is not unusual, however. See, e.g., [Oud93] and [Ned94] for similar definitions of PDA’s designed to model parsing algorithms. For the sake of brevity we will only consider recognition and not dwell on how the PDA can be augmented to a push-down transducer that yields a parse tree as a side result of recognizing a string.

A PDA is defined by means of a deduction relation on instantaneous descriptions, also called configurations. A configuration consists of a stack and the remainder of the input. It is a “snapshot” of a PDA at work. From a given configuration, the PDA may move to another configuration, as laid down in rules that take into account the (top part of the) stack and the (beginning of the) remaining input. We write $\varphi, \psi, \ldots \in I^*$ for stacks and parts of stacks. Configurations are denoted as pairs $(\varphi, w) \in (I^* \times \Sigma^*)$.

Deduction rules for configurations are usually defined by means of a finite transition table. We will not demand this in the definition; our first PDA actually requires an infinite transition table, but this inconvenience will be eliminated in the second PDA.

Definition 6.1 (PDA)
Let $G$ be a context-free grammar and $G'$ its augmented grammar according to Definition 4.6. A pushdown automaton $\Pi$ for $G$ is a quadruple $(I, \xi_0, F, D)$ in which

- $I$ is a set of items;
- $\xi_0 \in I$ a start item;
- $F \subseteq I$ a set of final items;
- $D \subseteq (I^* \times \Sigma^*) \times (I^* \times \Sigma^*)$ a set of deduction steps.

Definition 6.2 (acceptance)
Let $\Pi$ be a PDA for some grammar $G \in CFG$. A string $a_1 \ldots a_n \in \Sigma^*$ is accepted by $\Pi$ if

$$(\xi_0, a_1 \ldots a_n, \$) \vdash^*(\varphi, F, \$)$$

1 A grammar is called hidden left-recursive if $A \Rightarrow^+ \alpha \beta$ and $\alpha \Rightarrow^+ \epsilon$. This term has been coined in [NS93].
for some $\varphi \in \Sigma^*$ and $\xi \in \mathcal{F}$.

**Definition 6.3 (\(\Pi_{\text{Earley}}\))**

The PDA \(\Pi_{\text{Earley}}\) is defined for a grammar \(G \in \mathcal{CFG}\) by

\[
I_{\text{Earley}} = \{(A \rightarrow \alpha \beta, i, j) | A \rightarrow \alpha \beta \in P', \}
\]
\[
\xi_0 = \{s' \rightarrow s S, 0, 0,\}
\]
\[
\mathcal{F} = \{[s' \rightarrow s S, 0, n] | n \geq 0,\}
\]
\[
D_{\text{Pred}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j], v) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][B \rightarrow \gamma, j], v)\},
\]
\[
D_{\text{Sh}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j], aw) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][A \rightarrow \alpha \beta B, \beta, i, j + 1], w)\},
\]
\[
D_{\text{Re}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j][B \rightarrow \gamma, j, i], \ldots [B \rightarrow \gamma, j, k], w) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][A \rightarrow \alpha \beta B, \beta, i, k], w)\},
\]
\[
D_{\text{Earley}} = D_{\text{Pred}} \cup D_{\text{Sh}} \cup D_{\text{Re}}.
\]

We have replaced the terms **scan** and **complete** by **shift and reduce**, respectively, the latter one being more usual to describe LR-type parsers. It should be clear how these operations relate to each other. The PDA \(\Pi_{\text{Earley}}\) recognizes the same items for \(a_1 \ldots a_n\) as the parsing schema \(\mathcal{P}_{\text{Earley}}\).

One of the advantages of restricting the notion of correctness to final item in Section 3.4 is that we can relate correctness of parsing schemata and pushdown automata straightforwardly.

**Proposition 6.4 (equivalence of Earley(G) and \(\Pi_{\text{Earley}}\))**

Let \(\Pi_{\text{Earley}}\) be the PDA for some grammar \(G \in \mathcal{CFG}\) and \(\mathcal{P}_{\text{Earley}}\) the parsing system for \(G\) and some string \(a_1 \ldots a_n \in \Sigma^*\). Then \(\Pi_{\text{Earley}}\) recognizes \(a_1 \ldots a_n\) if and only if \(C(\mathcal{P}_{\text{Earley}}) \neq \emptyset\).

**Proof.** Straightforward. \(\square\)

Next, we observe that the position markers in the items in \(\Pi_{\text{Earley}}\) do not have much relevance. Position markers were necessary in a parsing schema, in order to know which part of the sentence an item relates to. But with all the previous items stacked, and the remainder of the sentence given, this information can be discarded.

So we can simplify the PDA. The next PDA, surprisingly, defines an LL(0) parser (this is elaborated further in, e.g., [OS92, Oud93]).

**Definition 6.5 (\(\Pi_{\text{LL(0)}}\))**

The PDA \(\Pi_{\text{LL(0)}}\) is defined for a grammar \(G \in \mathcal{CFG}\) by

\[
I_{\text{LL(0)}} = \{(A \rightarrow \alpha \beta, w) | A \rightarrow \alpha \beta \in P', \}
\]
\[
\xi_0 = \{s' \rightarrow s S, 0, 0,\}
\]
\[
\mathcal{F} = \{[s' \rightarrow s S, 0, n] | n \geq 0,\}
\]
\[
D_{\text{Pred}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j], v) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][B \rightarrow \gamma, j], v)\},
\]
\[
D_{\text{Sh}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j], aw) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][A \rightarrow \alpha \beta B, \beta, i, j + 1], w)\},
\]
\[
D_{\text{Re}} = \{(\varphi[A \rightarrow \alpha \beta B, \beta, i, j][B \rightarrow \gamma, j, i], \ldots [B \rightarrow \gamma, j, k], w) \vdash (\varphi[A \rightarrow \alpha \beta B, \beta, i, j][A \rightarrow \alpha \beta B, \beta, i, k], w)\},
\]
\[
D_{\Pi_{\text{LL(0)}}} = D_{\text{Pred}} \cup D_{\text{Sh}} \cup D_{\text{Re}}.
\]

**Proposition 6.6 (equivalence of \(\Pi_{\text{Earley}}\) and \(\Pi_{\text{LL(0)}}\))**

Let \(\Pi_{\text{Earley}}\) and \(\Pi_{\text{LL(0)}}\) be PDAs for some grammar \(G \in \mathcal{CFG}\). Then \(\Pi_{\text{LL(0)}}\) recognizes a string \(a_1 \ldots a_n \in \Sigma^*\) if and only if \(\Pi_{\text{Earley}}\) recognizes \(a_1 \ldots a_n\).

**Proof.** Trivial. \(\square\)

The transformation from LL(0) to LR(0) need not be spelled out. See, e.g., [OS92, Oud93] for a more detailed treatment. Two further transformations are needed:

- Items are extended from dotted productions to sets of dotted productions. For each item, its closure is computed by inserting all the dotted rules that can be obtained with predict steps. Hence the predict rule can be eliminated from \(D\). Moreover, one item less is popped from the stack in a reduce step.

- A full-fledged LR(0) PDA is obtained by combining items. If the dotted productions \([A \rightarrow \alpha X \beta], [A' \rightarrow \alpha' X \beta'], \ldots\) are contained in a single item, then the item set contains another item

\[
\text{closure}(\{(A \rightarrow \alpha X \beta, [A' \rightarrow \alpha' X \beta'], \ldots\})
\]

The algorithm for the construction of the set of LR(0) items can be found in any textbook on compiler construction, e.g., [ASU86].
There is much more to say about the intricacies of constructing LR-type parsers but this is not the place to do so. An issue that has to be avoided is nontermination. On the—still abstract—level of PDA’s this issue does not exist. In an implementation, however, it has to be avoided that a parser gets stuck in an infinite loop. Chart parsers are more robust, because by definition (cf. Figure 5) each item is recognized only once.

6.3 Chart Parsers vs. PDA’s

We have sketched how the uninstantiated parsing system Earley(\(G\)) can be transformed into an LR(0) PDA. The transformation is bidirectional. The parsing schema SLR(1) has in fact been derived by an analogous transformation in reverse direction. For other LR-type algorithms, an underlying parsing schema can be derived in similar fashion.

Recalling that chart parsers are in fact canonical implementations of parsing schemata, we have sketched a general relation between these two seemingly different parsing paradigms. This does not come as a surprise, though, the close relation between Earley’s algorithm and Tomita’s algorithm has been known for considerable time and was investigated in detail in [Sik91].

7 Beyond Context-Free Grammars

The parsing schemata framework has been specified for context-free grammars, but it can easily be extended to other grammar formalisms as well.

Unification-based grammars are the predominant class of grammar formalisms in current computational linguistics. The interested reader is referred to [Shi86] for a simple introduction and [Shi92] and [Car92] for a thorough treatment of unification logics. Parsing schemata for a simple kind unification grammar have been defined in [Sik93]. A parsing schema for a unification-based ID/LF grammar is described in [Mor95].

Unification grammars treat syntactic and semantic information in a uniform manner. One can reduce the role of syntax and consider syntactic category as a feature like any other. Indeed there seems to be a trend that less and less information is stored in the context-free backbone of a grammar—i.e., the cat feature in a feature structure—because various syntactic properties can be expressed more elegantly by other kinds of feature constraints. A typical example is subcategorization of verbs: all verbs have syntactic category verb; constraints on the various kinds of objects that a verb can take are denoted in the subcat feature of the particular verb.

Independently of each other, Nagata [Nag92] and Maxwell and Kaplan [MK93] have pointed out that this is convenient for writing natural language grammars, but that it has repercussions on parsing efficiency. Context-free parsing is much more efficient than feature structure unification. Hence it is not surprising that the experiments reported in [Nag92] and [MK93] show that the efficiency of unification grammar parsing can be increased by retrieving an (implicit) context-free backbone from a unification grammar that covers more than just the cat feature and using this context-free part for syntactic analysis.

Therefore, context-free parsing remains to be of importance to natural language analysis.

8 Conclusions

Parsing schemata provide a general framework for description, analysis and comparison of parsing algorithms, both sequential and parallel. Data structures, control structures and (for parallel algorithms) communication structures are abstracted from. This framework constitutes an intermediate, well-defined level of abstraction between grammars (defining what valid parses are) and parsing algorithms (prescribing how to compute these).

Correctness proofs are easier at this level of abstraction, simply because there is less to prove. The correctness of an algorithm can be derived by showing that is a correct implementation of a schema that is known to be correct.

A general method to prove the correctness of a parsing schema has been introduced, and illustrated with various examples. Also, we have shown how parsing schemata are related to two important classes of parsing algorithms, viz., chart parsers and pushdown automata. A chart parser can be regarded as the canonical implementation of some parsing schema. A PDA can be obtained from a parsing schema—and reversed—with a straightforward transformation that preserves the correctness.
REFERENCES


[Shi86] Shieber S.M. (1986): An Introduction to Unification-Based Approaches to Grammar. CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford University, Stanford, Ca.


A Generic Tabular Scheme for Parsing

François Barthélemy *
CNAM Paris
barthe@cnam.fr

1 INTRODUCTION

Tabulation is an important aspect of parsing and recognition. Many grammar classes use the key concept of syntax tree. Parsing and recognizing strings from the corresponding language involve building a parse-tree, which is usually done using a stack machine, some kind of push-down automaton.

Stack operations usually use only a few top-most elements of the current stack which implies that the other elements may be temporary forgotten. This property can be used to evaluate stack-based computations with tabular techniques in order to gain computation sharing: Complex computations are split into elementary ones represented by items that are stored in a table (i.e. tabulated) and that can be reused in different contexts. Rules are given to produce new items from tabulated ones. A subsumption test is done to avoid retabulating redundant items. Rules are applied until a fixed-point is reached.

This general scheme encompasses various tabular algorithms appearing in the literature, such as [1, 6, 11, 7] in Parsing, [10] in Abstract Interpretation, magic-sets [2] in Deductive Databases and [5] in Constraint Logic Programming. These algorithms have sometimes been designed independently. Each one required non-trivial adequacy proofs. However, they are based on common principles.

Tabulation results in computation sharing, by computing the common parts of alternative parse-trees only once. A spectacular evidence of this computation sharing is given by the fact that context-free tabular parsers build in cubic time a finite representation of an infinite forest of parse trees.

Tabular parsing is common for context-free parsing. But what about classes of more powerful grammars: Definite Clause Grammars, Tree-Adjoining Grammars, Lexical-Functional Grammars, Unification Grammars, HPSGs, and many others? Tabulation is designed independently for each class and each parsing algorithm, although parsing for these different grammars share some common properties. In this paper, we try to express tabulation abstractly, for any stack-based computation. This abstract scheme is useful both to design a new tabular simulation of an existing stack-based parsing algorithm and to prove the equivalence of such an algorithm with its simulation.

This paper is related to two previous ones [3] [4]. We try here a new, more intuitive formulation of the problem.

The next section is devoted to an expression of stack-based computations. The third one deals with simple tabulation. Then we address more complex matter: using a subsumption mechanism together with tabulation. An example is given in the fifth section. The conclusion explains that this paper gives only incomplete answers to the problem it addresses, and gives some directions we plan to explore.

2 GENERALIZED PUSH-DOWN AUTOMATA

We need formal definitions of stacks and stack-based computations to define precisely how to tabulate such a computation. Several equivalent definitions are possible. We choose the most intuitive, corresponding to usual definitions of stacks and Push-Down Automata.

Definition 2.1 Stack
A stack over a domain D is a finite, possibly empty, sequence of elements from D. The first element of the sequence is called the top of the stack. The following operations apply on stacks:
• the push operation consists in adding an element on top of the stack.
• the pop operation consists in removing one element from the top of the stack.
• the consult operation consists in looking at the top of the stack.

No other operation is allowed.

Definition 2.2 Height
The height of a stack is the number of elements of the sequence.

Notation: we note \( [a, b, c] \) a stack, the top being on the left-hand side. We note \([\;]\) the empty stack and \([a, b]E\) any stack having \(a\) on the top, \(b\) as second element, and \(E\) is a stack. We note \(S(D)\) the set of stacks over the domain \(D\).

Definition 2.3 Generalized Push-Down Automaton
A Generalized Push-Down Automaton is a 5-tuple \((D_1, D_2, T, I, F)\) where:

• \(D_1\) and \(D_2\) are arbitrary sets. The elements in \(D_1\) are called stack symbols, those in \(D_2\) are called states.

• \(T\) is a set of partial functions from \(S(D_1) \times D_2\) to \(S(D_1) \times D_2\) such that:
  
  - \(\forall f \in T, [(a_0, \ldots, a_n), m] \in \text{dom}(f) \Rightarrow \forall b_1, \ldots, b_i \in D_1, [(a_0, b_1, \ldots, b_i, m) \in \text{dom}(f)]\)
  
  - \(\forall f \in T, \forall [(a_0, \ldots, a_n), m] \in \text{dom}(f), 0 \leq n, f([(a_0, \ldots, a_n), m]) = (\xi, h_f(a_0, m))\) where \(h_f\) is any function from \(D_1 \times D_2\) to \(D_2\)
  
  - Each function applies the same transformation on the stacks of all the configurations in its domain, and this transformation is one of the following:
    * no transformation: \(f(\xi, s) = (\xi, s')\)
    * a push: \(f([(a_0, \xi), m] = [(g_f(a_0, m), a_0, \xi), h_f(a_0, m))]\) where \(g_f\) is an arbitrary function from \(D_1 \times D_2\) to \(D_1\)
    * a pop: \(f([(a_0, \xi), m] = ([\xi], h_f(a_0, m))]\)

The functions in \(T\) are called transitions. Depending on the effect of their application on the stack, they are called respectively Nop, Push or Pop transitions.

• \(I\) is a subset from \(S(D_1) \times D_2\) whose elements are called the initial configurations. The height of the stacks of these configurations is exactly 1.

• \(F\) is a subset from \(S(D_1) \times D_2\) whose elements are called the final configurations. The height of the stacks of these configurations is exactly 1.

The restriction on the height of the stack in initial and final configuration does not result in any loss of generality. It is not really important, it is a detail that simplifies some forthcoming definitions. Similarly, the restriction on the changes on the stack by transition application is not necessary, but simplifies the description of tabulation.

We call state all the data used by the automaton that are not stacked. We give this name by analogy with finite-state automata, and push-down automata. The states in our generalization are not ranked in a finite domain. In particular, they include some information about the string, that does not appear explicitly as separate information. If we want to translate a PDA into a GPDA, the domain of states of the GPDA is the Cartesian product of the state and string domains of the PDA.

A Generalized Push-Down Automaton (GPDA) resembles a Push-Down Automaton (PDA), with two differences. The first difference is that \(D_1\) and \(D_2\) are not supposed finite. This allows the storage of arbitrary pieces of information in the stack. The second difference is that transitions may perform complex computations. The only limitation is that they respect the basic restrictions imposed by the stack: their domain and their result depend only on its top, not on the other components. A transition may only result in a push or a pop, or it must leave the stack unchanged.

Definition 2.4 Configuration, GPDA execution, success
Let \(G = (D_1, D_2, T, I, F)\) be a GPDA.

• A configuration of \(G\) is a couple \((\xi, s)\) such that \(\xi \in S(D_1)\) and \(s \in D_2\).

• An execution of \(G\) is a finite sequence \(c_0, c_1, \ldots, c_n\) of configurations from \(S(D_1) \times D_2\) such that:
  - \(c_0 \in I\)
  - \(\forall i, 0 \leq i < n, 3f \in T\) such that \(f(c_i) = c_{i+1}\)
3 Tabulation

3.1 Intuitive Presentation

Some tabular procedures have been defined to simulate computation of many different stack-base computations. The computation is split into small parts often called items, stored in a table (a storage structure). Each computation step consists in producing a new item by using one or several items already stored in the table. The process goes on until a fixed-point is reached.

We would like to create a generic tabular procedure for all the generalized push-down automata. We suppose that the granularity of the simulation is given by the transition application, i.e., the transition application corresponds to an item creation in the simulation, a reasonable assumption.

The minimal information needed by the transition to compute a new state (and for some of them, a new stack symbol to be pushed), comes from the current top and state, as shown by the definition 2.3. So items must contain these pieces of information. It is sufficient to simulate the push-transitions and the nop-transitions.

Pop-transitions are more tricky to perform, since the current top is removed, you have to know about the second element of the stack. This second element was once on the top. Items must also contain a way to retrieve the former top of stack. This is achieved by a sort of pointer to another item.

To increase the sharing between items, it is useful to implement the pointers in such a way that the same pointer gives access to several items satisfying a certain property. Part of the information needed to retrieve these items is in the pointer and part is in the dereferencing function. The less information stored in the pointer, the more the item is likely to represent many configurations. But if the pointer does not discriminate enough, it will point to irrelevant items.

We call logic pointer the pointer used in items that may be more than a simple address. For example, for rich computation domains, the pointer describes how to propagate information down into the stack through variables when popping.

The following formal descriptions should help clarify any misunderstandings.

3.2 Tabular Procedure

Definition 3.1 Approximation, logic pointer, simulation domain, item
Let \( G = (D_1, D_2, T, I, F) \) be a GPDA.

- an approximation \( \mathcal{A} \) is a function from \( D_1 \times D_2 \) to a domain \( A \), called the domain of logic pointers. We impose that \( A \) contain at least one symbol noted \( \perp \) such that \( \perp \) is not in the image of \( D_1 \times D_2 \) by \( \mathcal{A} \).
- \( \langle D_1, D_2, A \rangle \) is called the simulation domain of \( G \) and \( \mathcal{A} \). An item is an element of a simulation domain.

Definition 3.2 Simulated transitions
Let \( G = (D_1, D_2, T, I, F) \) be a GPDA and \( A \) a set of logic pointers. For each transition \( t \) in \( T \), we define a simulated transition \( \text{Sim}(t) \) from \( \langle D_1, D_2, A \rangle \) to \( \langle D_1, D_2, A \rangle \) as follows:

- if \( t \) is a nop-transition,
  - \( \forall \langle [o]E, s \rangle \in S(D_1) \times D_2, \langle [o]E, s \rangle \in \text{dom}(t) \iff \langle \forall a \in A, ([o, s], a) \in \text{dom}(\text{Sim}(t)) \rangle \)
  - \( \langle \xi, a \rangle \in \text{dom}(\text{Sim}(t)) \Rightarrow \text{height}(\xi) = 1 \)
  - \( \forall \langle [o], s \rangle, \langle \xi, a \rangle \in \text{dom}(\text{Sim}(t)), \text{Sim}(t)(\langle [o], s \rangle, a) = \langle [o', s'], (o, s), a \rangle \)
    where \( \langle o', s' \rangle = t([o], s) \)

- if \( t \) is a push-transition,
  - \( \forall \langle [o], s \rangle \in S(D_1) \times D_2, \langle [o], s \rangle \in \text{dom}(t) \iff \langle \forall a \in A, ([o, s], a) \in \text{dom}(\text{Sim}(t)) \rangle \)
  - \( \langle \xi, a \rangle \in \text{dom}(\text{Sim}(t)) \Rightarrow \text{height}(\xi) = 1 \)
  - \( \forall \langle [o, s], a \rangle \in \text{dom}(\text{Sim}(t)), \text{Sim}(t)(\langle [o, s], a \rangle) = \langle [o', s'], (o, s), a \rangle \)
    where \( \langle o', s' \rangle = t([o], s) \)

- if \( t \) is a pop-transition,
  - \( \forall \langle [o, s], (o', s') \rangle \in S(D_1) \times D_2, \langle [o], s \rangle \in \text{dom}(t) \iff \forall x \in D_1 \times D_2, ([o, s], x), a \in \text{dom}(\text{Sim}(t)) \)
  - \( \langle \xi, a \rangle \in \text{dom}(\text{Sim}(t)) \Rightarrow \text{height}(\xi) = 2 \)
  - \( \forall \langle [o_1, s_1], (o_2, s_2), a \rangle \in \text{dom}(\text{Sim}(t)), \text{Sim}(t)(\langle o_1, s_1 \rangle, (o_2, s_2), a) = \langle [o_2, s'], a \rangle \)
    where \( \langle o_2, s' \rangle = t([o_1, s_1], a) \)

In other words, the simulation of a transition acts like the original transitions except that it applies on simulation domains, and that its domain is limited to fixed-height stacks. Note that
the transition simulations are not transitions as defined in 2.3. In particular, the condition on the domain does not hold.

**Definition 3.3 Cut, glue**

Let $G = (D_1, D_2, T, I, F)$ be a GPDA, $A$ a set of logic pointers and $\hat{A}$ an approximation from $D_1$ to $A$.

- **Cut**: it is a function from $(S(D_1 \times D_2) \times A)$ to $(S(D_1 \times D_2) \times A)$ defined on all the couples $(\xi, a)$ such that $\text{height}(\xi) = 2$.
  
  $\text{cut}([([0, 1, s_1]), ([0, 2, s_2]), a]) = ([0, 1, s_1], A([0, 2, s_2]))$

- **Glue**: it is a function from $(S(D_1 \times D_2) \times A)^3$ to $(S(D_1 \times D_2) \times A)$
  
  $\equiv \text{height}(\xi_1) = 1$ and $\text{height}(\xi_2) = 1$ and $\text{height}(\xi_3) = 1$
  
  $\text{cut}([([0, 1, s_1]), ([0, 2, s_2]), a]) = ([0, 1, s_1], A([0, 2, s_2]), a_2)$

**Definition 3.4 Tabular Procedure**

- The complete set of items for a GPDA $G = (D_1, D_2, T, I, F)$, a logic pointer domain $A$ and two functions glue and cut is the smallest set $E$ such that:
  
  - $([\epsilon], s) \in I \Rightarrow ([\epsilon], s, \bot) \in E$
  
  - $E$ is closed under the simulated transitions from $\text{Sim}(T)$ and the cut and glue operations.

- A tabular procedure is an algorithm that computes the complete set of items.

Obviously, $E$ contains only items with stacks of height 1 or 2. We call them respectively 1-items and 2-items.

Figure 1 shows the computation system we use to design tabular procedures.

We use the special symbol $\bot$ from logic pointers to avoid glue being applied on some items. This allows us to note $< T, A >\{([\epsilon], s), \bot\} \Rightarrow c$ the fact that $c$ is reachable from a state $([\epsilon], s)$ without popping. The complete set $E$ may be defined by $E = \{c | < T, A > \in \text{Sim}(I) \Rightarrow c\}$ where $\text{Sim}(I) = \{([\epsilon], s), \bot\}$ in the case of initial configurations and $([\epsilon], s) \in I$.

A tabular procedure is completely defined by giving an approximation $A$. But we are interested in tabular procedures to simulate direct execution of a GPDA, and more precisely, to simulate the complete set of possible executions of the GPDA.

We have to state a correspondence between items and configurations.

**Axiom 3.1 Strong equivalence**

$\forall([0, 1, s_1], a_1), ([0, 2, s_2], a_2) \in \text{ITEMS}$ such that

$\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 1, s_1], a_1)\text{ and}$

$\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 2, s_2], a_2)$,

$\hat{A}(0, 1, s_1) = a_2 \Rightarrow$

$\langle T, A \rangle \{([0, 1, s_1], \bot)\} \Rightarrow ([0, 2, s_2], ([0, 1, s_1], \bot))$

**Theorem 3.1** Let $G = (D_1, D_2, T, I, F)$ be a GPDA, and $\hat{A}$ an approximation for $D_1$. If the strong equivalence axiom holds then the following hold too:

- **Soundness**: for all items $i = ([0, 1, s_1], \ldots, [0, n, s_n]), a$ such that

  $\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 1, s_1], \ldots, [0, n, s_n], a)$,

  there exists an execution of $G$ ending in a configuration $([0, 1, \ldots, 0, n, s], s_1)$ and either

  $x = [0, \xi']$ with $\exists([0, \xi'], s_1)$ in the execution such that

  $A(o, s) = o$ or $\xi = \bot$ and $s = \bot$

- **Completeness**: for all configurations $c = ([0, 1, 2], \bot)$ such that $c$ appears in an execution of $G$,

  $\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 1, 2], a)$

The proof is a simple induction. The axiom 3.1 is used to ensure the soundness of the glue operation.

There is a weaker notion of equivalence that relies on the correctness of the only 1-items. The 2-items may have no counterpart in the configurations as far as they are not used to produce 1-items (by a pop-transition). 1-items are sufficient to represent all the possible configurations and to retrieve final ones, since final configurations are of height 1. It is even possible to describe tabulation without 2-items by systematically composing pushes with a cut application, and glue with pops.

**Axiom 3.2 Weak equivalence**

$\forall([0, 1, s_1], a_1), ([0, 2, s_2], a_2) \in \text{ITEMS}$ such that

$\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 1, s_1], a_1)\text{ and}$

$\langle T, A \rangle \Rightarrow \text{Sim}(I) \Rightarrow ([0, 2, s_2], a_2),$

$\hat{A}(0, 1, s_1) = a_2 \Rightarrow$

$\langle T, A \rangle \{([0, 1, s_1], \bot)\} \Rightarrow ([0, 2, s_2], ([0, 1, s_1], \bot))$ or there is no transition $t \in T$ such that

$\text{Sim}(I) = \{([0, 2, s_2], ([0, 1, s_1], \bot)) \text{ and } \hat{A}(0, 1, s_1) = a_2 \}$

**Theorem 3.2** Let $G = (D_1, D_2, T, I, F)$ be a GPDA, and $\hat{A}$ an approximation for $D_1$. If the weak equivalence axiom holds then the following hold too:
\[ \frac{t \in T}{\langle T, A \rangle \{c_1\} \vdash c_2} \]

\[ \frac{cut(c_1) = c_2}{\langle T, A \rangle \{c_1\} \vdash \text{cut} \ c_2} \]

\[ \frac{\text{glue}(c_1, c_2) = c_3}{\langle T, A \rangle \{c_1, c_2\} \vdash \text{glue} \ c_3} \]

\[ \frac{\langle T, A \rangle \{c_1, \ldots, c_n\} \vdash c}{\langle T, A \rangle \{c_1, \ldots, c_n\} \vdash \text{cut} \ c} \]

\[ \frac{\langle T, A \rangle S \vdash c}{\langle T, A \rangle S \cup S' \vdash c} \]

\[ \frac{\langle T, A \rangle \{c_1, \ldots, c_n\} \vdash c}{\langle T, A \rangle \{c_1, \ldots, c_n\} \vdash \text{cut} \ c} \]

\[ \frac{\langle T, A \rangle S \vdash c_1}{\langle T, A \rangle S \cup \{c_1\} \vdash c} \]

\[ \frac{\langle T, A \rangle \vdash c}{\langle T, A \rangle \vdash \text{cut} \ c} \]

Figure 1: computation rules

- **Soundness:**
  for all item \( i = ([o, s], a) \) such that
  \( \langle T, A \rangle \text{Sim}(l) \vdash \{[o, s], a\} \), there exists an execution of \( G \) ending in a configuration \(([o][\xi], s)\) and either \( \xi = [o'][\xi'] \) with \( A(o', s) = a \) or \( \xi = [] \) and \( a = \bot \)

- **Completeness:**
  for all configurations \( c = ([o_1][\xi], s) \) such that
  \( c \) appears in an execution of \( G \), \( \exists a \in A \) such that
  \( \langle T, A \rangle \{\text{Sim}(l)\} \vdash \{[o_1], s, a\} \)

The strong and weak equivalence axioms are quite tricky to prove because of the quantification over computable items. It is easier (but more restrictive) to prove the property for all items in the domains, and not just those computable by the simulation.

4 ORDERING THE DOMAIN

For some rich stack domains, it is possible to increase computation sharing by using a subsumption mechanism. A partial order is given, and, provided the transitions respect this order, most general solutions are reached when using only most general items.

Examples of the use of subsumption: in logic programming (DCGs), only most general answers are computed using the instance relation; in hierarchical typing such as that used by some object oriented languages, one is mainly interested in the principal type (the most general one). There is also an order relation between feature structures.

A tabular procedure uses subsumption as follows: when a new item is computed, it is compared to all the items already computed. If there exists a smaller item, the new one is simply ignored. If not, it is stored in the table and all the greater items are removed from the table.

The conditions for using such an order: monotonicity of the operations (simulated transitions and glue and cut) and change in the definition of glue.

The change of glue comes from the fact that items on which glue applies may not be stored in the table, because there exist more general items. These more general items may not be in the domain of glue. We define a new glue:

**Definition 4.1** Extended glue
Let ITEMS be a simulation domain, and glue the operation on this simulation domain, let \( \leq \) be a
pre-order on ITEMS. We define a subsumption compatible operation glue≤ by:

\[
\forall i_1, i_2 \in ITEMS, \\
glue≤(i_1, i_2) = \{i_2|\exists i_1', i_2' \in ITEMS, \\
i_1 \leq i_1', i_2 \leq i_2', (i_1', i_2') \in dom(glue), \\
i_3 = glue(i_1', i_2')\}
\]

The glue≤ operation returns an item set because there exist ordered domains with no greatest lower bound. In that case, all the greatest lower bounds are to be considered. In the definition, we do not explicitly state that only the smaller it≤ 3 are sufficient because it is not so easy to express. The definition is sound anyway, because only the smaller elements in the set computed by glue≤ will be ultimately stored in the table, due to the subsumption mechanism.

**Axiom 4.1 Subsumption relation**

Let ≤ be a pre-order on items such that 2-items and 1-items are incomparable. ≤ is called a subsumption relation for a simulation if the following hold:

- \(\forall i_1, i_2 \in ITEMS, \forall i \in T, i_1 \leq i_2 \text{ and } i_2 \in dom(Sim(t)) \Rightarrow i_1 \in dom(Sim(t)) \text{ and } Sim(t)(i_2) \leq Sim(t)(i_2)\)
- \(\forall i_1, i_2 \in ITEMS, i_1 \leq i_2 \text{ and } i_2 \in cut \Rightarrow i_1 \in dom(cut) \text{ and } cut(i_2) \leq cut(i_2)\)
- \(\forall i_1, i_2, i_1', i_2' \in ITEMS, i_1 \leq i_1', i_2 \leq i_2' \text{ and } (i_1, i_2) \in dom(glue≤) \Rightarrow (i_1', i_2') \in dom(glue≤) \text{ and } glue≤(i_1', i_2') \leq glue≤(i_1, i_2)\)

We state the equivalence between the subsumption computation and the stack automaton through the equivalence of tabular procedures with and without subsumption. In the following, we denote \(\vdash\) the computation without subsumption and \(\vdash≤\) the corresponding computation with subsumption.

The soundness proof of equivalence requires an additional axiom. We first give an actually stronger property than the axiom we need, because it is simpler to understand.

**Axiom 4.2 Overly strong property**

\[
\forall (i_1, s_1, a), i \in ITEMS \text{ such that } ((i_1, s_1, a), i) \in dom(glue), \\
<T, A> \{\{i_1, s_1, _\downarrow\} \vdash≤ i,\}
\]

The property needed for a strong equivalence is similar, but restricted to the items of the simulation domain that are greater than the items computable from sim(I) by \(\vdash≤\).

**Axiom 4.3 Strong equivalence**

\[
\forall (i_1, a), i \in ITEMS \text{ such that } ((i_1, a), i) \in dom(glue), \\
\exists i' \in ITEMS \text{ such that } i' \leq (i_1, a) \text{ and } i \leq i' \text{ and } <T, A> \text{ sim(I)} \vdash≤ i' \text{ and } <T, A> \text{ sim(I)} \vdash≤ i''
\]

It is also possible to give a weaker property that ensures the soundness of the only 1-items, as we did in the previous section.

**Theorem 4.1 Equivalence**

Let \(G = (D_1, D_2, T, I, F)\) be a GPDA, \(A\) an approximation for \(D_1\), and ≤ a subsumption relation (i.e., the axiom 4.1 holds). The following hold:

- **Soundness:** \(\forall i \in ITEMS \text{ such that } <T, A> \text{ sim(I)} \vdash≤ i, \exists i' \in ITEMS \text{ such that } <T, A> \text{ sim(I)} \vdash≤ i' \text{ and } i' \leq i\)
- **Completeness:** \(\forall i' \in ITEMS \text{ such that } <T, A> \text{ sim(I)} \vdash≤ i', \exists i \in ITEMS \text{ such that } <T, A> \text{ sim(I)} \vdash≤ i \text{ and } i \leq i'\)

The proof is again made by induction. The axiom 4.3 is needed to ensure that there is a counterpart of the items produced by glue≤ in the \(\vdash\) calculus.

## 5 Example

We show as an example the adaptation of Earley's algorithm [6] to Definite Clause Grammars recognition [9].

Definite Clause Grammars (DCGs) are an extension of Context-Free Grammars where arguments are added to non-terminals. These arguments take values in first-order terms. Recognition and parsing make use of first-order unification. For more details about DCGs, see [8].

A DCG \(G\) is a collection of rules \(a_0 \to a_1, a_2, \ldots, a_n\), where the \(a_i\) are first-order terms possibly with variables or terminals between quotes. The set of variables is supposed infinite.

First, we give a recognizer in the form of a generalized push-down automaton. The stack symbols are rules from the grammar where a dot is added somewhere in the right-hand side and all the variables are renamed by fresh variables. The state is composed of the remaining of the input string (the part that has not been scanned yet),
and an answer substitution. In the following, we note \( A, B, \ldots \) the atoms non-terminal, \( a, b, \ldots \) the terminals, \( \alpha, \beta, \ldots \) the strings of terminals and non-terminals, \( v, w, \ldots \) the strings of terminals, \( \theta \) the renaming substitutions, and \( \sigma, \ldots \) the answer substitutions.

- predictor: \( t_1([A \rightarrow \alpha \bullet B \beta][\epsilon], w, \sigma) = ([C \rightarrow \gamma] \theta, A \rightarrow \alpha \bullet B \beta[\epsilon], w, \sigma.mgu(B\sigma, C\theta)) \) where \( C \rightarrow \gamma \) is a rule of the grammar and \( C \) is unifiable with \( B \).

- scanner: \( t_2([A \rightarrow \alpha \bullet a\beta][\epsilon], aw, \sigma) = ([A \rightarrow \alpha a \bullet \beta][\epsilon], w, \sigma) \)

- completer:
  \[
  t_2(C \rightarrow \gamma, A \rightarrow \alpha \bullet B \beta[\epsilon], w, \sigma) = \\
  ([A \rightarrow \alpha B \bullet \beta][\epsilon], w, \sigma.mgu(B\sigma, C\theta))
  \]

The set of initial configuration is
\[
\{[[S' \rightarrow S], w, \epsilon d]\} \text{ where } S \text{ is the axiom, } \epsilon d \text{ is the identity substitution, and } S' \text{ a new symbol. The set of final configurations is } \{[[S' \rightarrow S'], \epsilon, \sigma]\} \text{ where } S \text{ is the axiom and } \epsilon \text{ the empty string.}
\]

Now, we have to choose an approximation \( A \) in order to define a simulation domain for a tabular procedure. Several choices are available. To retrieve the usual items used for Earley's algorithm, we define \( A(A \rightarrow \alpha \bullet \gamma, w, \sigma) = w \).

The 1-items are then composed of a stack containing one dotted rule, a substitution and two strings: the one in the state and the one from the approximation. In usual formulations of Earley's algorithm, these two strings are encoded by two integers that are pointers to some positions in the input string. The information is the same, only the encoding differs.

A subsumption order is usable with this automaton, defined using the usual instance relationship between first-order terms:

- \( ([\sigma], \sigma, v, w) \leq ([\sigma'], \sigma', v', w') \) if \( \exists \delta \) substitution, such that \( \sigma \delta = \sigma' \delta \theta \) where \( \theta \) is a renaming, and \( v = v' \) and \( w = w' \).

- \( ([\sigma_1, \sigma_2], \sigma, v, w) \leq ([\sigma_1', \sigma_2'], \sigma', v', w') \) if \( \exists \delta \) a substitution such that \( \sigma_1 \delta = \sigma_1' \sigma' \theta \) and \( \sigma_2 \delta = \sigma_2' \sigma \theta \) (where \( \theta \) is a renaming) and \( v = v' \) and \( w = w' \).

It is possible to show that the strong equivalence axiom 3.1 does not hold, but the weak equivalence 3.2 does.

6 Where the paper finishes too soon

This paper describes work in progress. In sections 3 and 4 we specify the properties we needed to perform the equivalence proof (strong and weak equivalence axioms). These properties are a little tricky to prove. We believe that it is possible to give some easier properties, for instance properties on the transitions of the automata. This paper gives a precise formulation of the problem. It is a first step, and we hope to have more results in the near future.

References


a survey of the formalism and a comparison
with augmented transition networks. *Artifi-

ing as deduction. In *Proc. of the 21st An-
nual Meeting of the Association for Com-
putational Linguistic*, pages 137–144, Cam-
bridge (Massachusetts), 1983.

of Third Int. Conf. on Logic Programming*,

context-free parsing algorithm. *Computa-
Parsing with Algebraic Power Series using Dynamic Programming

preliminary version

Frédéric TENEAU
INRIA-Rocquencourt, BP 105, F 78153 Le Chesnay CEDEX
E-mail: Frederic.Tendeau@inria.fr

Abstract

The use of ambiguous context-free grammars raises a complexity problem for parsing: an exponential (or even infinite, in case of cyclic grammars) number of parse-trees are to be considered. Dynamic programming techniques reduce this complexity down to polynomial. When decorating trees with contextual informations, the problem arises again. The formal power series formalism is very interesting for its generic description of the parsing paradigm. We show how to apply dynamic programming techniques to computations in the abstract semiring of the power series, independently from any representation, i.e. without considering it as describing boolean (recognition), forests (parsing), or any decoration domain with more practical purposes (probabilities, constraints...).

1 Introduction

The theory of formal power series for algebraic languages has been introduced by Chomsky and Schützenberger in [CS]. It generalizes the notions of recognition, parsing, stochastic parsing etc. by describing the computations executed from a $\Sigma^*$ input to a domain having a semiring structure: when the semiring is boolean, the power series can represent the recognition (a word does or does not belong to the language); when the semiring is the set of forests (described below), the power series represent the parsing (the set of derivation trees is given for any given input string in $\Sigma^*$). Kech and Salomaa describe in [KS] algebraic systems of equations on power series and study their solutions. It will be useful because parsing amounts to producing such systems and solving them.

In 1970, Earley proposed in [Ear] a parsing algorithm for context-free languages achieving a cubic (time and space) worst complexity for all CF-languages. In 1974, Lang, in [Lg74], showed that dynamic programming techniques applied to any pushdown automaton implementing a parsing strategy lead to cubic worst complexity parsing algorithms for CF-languages, stating the independence between the strategy and its dynamic programming interpretation.

Parsing, in a formal power series framework, consists in finding the coefficient of a sentence in the considered semiring. This framework is base upon a semiring structure, which is precisely what is required to invoke dynamic programming techniques. Hence we propose to apply dynamic programming to algebraic power series in order to obtain low complexity parsers that yield the coefficient of the input string w.r.t. the decorated grammar.

We show that the parse-forest semiring $F$ can be considered as canonical, in the way that if one has the parse forest of a string, then its coefficient in any other semiring $A$ is obtained by replacing, in an expression of $F$, $F$'s composition operations by $A$'s, and $F$'s values of the grammar rules by $A$'s.

2 Basics

In the sequel, we assume that we have a finite set of non-terminal $N = \{N_1, ..., N_n\}$, a context-free grammar $G = (\Sigma, N, \mathcal{R}, N_0)$, a semiring $\langle A, \oplus, \times, 0, 1 \rangle$ — simply noted $A$ —, a string $x \in \Sigma^*$, and a mapping $\phi: \mathcal{R} \rightarrow A$ such that $\forall R \in \mathcal{R}, \phi(R) \neq 0$. This mapping represents decorations of the grammar rule in $A$.

The set $V = \Sigma \cup N$ is called the vocabulary.
The relation \( \text{derive} \) is denoted \( \implies \), its transitive closure \( \implies^* \), and its reflexive transitive closure \( \implies^= \). The language defined by \( G \) is \( \mathcal{L}(G) = \{ w \in \Sigma^* \mid N_i \implies^* w \} \). A non-terminal \( N \) is said \emph{productive} if there exists \( w \in \Sigma^* \) s.t. \( N \implies w \); it is said \emph{accessible} if there exists \( \alpha, \beta \in \Sigma^* \) s.t. \( N \implies^= \alpha N \beta \). A grammar is said \emph{proper} if all its non-terminals are accessible and productive; it is said \emph{cyclic} if there exists \( N \in N \) s.t. \( N \implies^= N \). We assume w.l.o.g. that \( G \) is proper.

The set of natural integers is denoted \( \mathbb{N} \), that of booleans \( \mathbb{B} \), it is also considered as the semiring \( (\mathbb{B}, \lor, \land, \text{false}, \text{true}) \).

3 Formal Power Series

We give here a minimal recall of basic notions and results inspired from [KS]. A complete description may be found in this reference.

Intuitively a formal power series is a discrete function of \( \Sigma^* \) to a set \( S \): a sequence (represented as a formal sum) of couples (value of \( w \), \( w \)).

\[ A \text{-formal power series} \]

Consider the free monoid \( \Sigma^* \) and a set \( S \), a formal power series \( r \) is a mapping from \( \Sigma^* \) to \( S \). For \( w \in \Sigma^* \), \( r(w) \) is denoted \( (r,w) \) and also called \emph{coefficient} of \( r \) for \( w \). A series \( r \) is denoted as the formal sum \( r = \sum_{w \in \Sigma^*} (r,w) w \).

The collection of all such power series is denoted \( A^\Sigma^* \).

Among these series, we will be concerned by those called \emph{algebraic} power series, related to algebraic languages and context-free grammars. The set \( S \) is then a semiring \( A \) and their collection is denoted \( A^\Sigma^* \).

When the series are finite sums, they are polynomials. Their collection is denoted \( A^{\Sigma^*} \).

The \emph{support} of an algebraic power series \( r \) is

\[ \text{supp}(r) = \{ w \in \Sigma^* \mid (r,w) \neq 0 \} \]

A sum, \( + \), and a product, \( \cdot \), can be defined on algebraic power series: for \( r, s \in A^\Sigma^* \) and all \( w \in \Sigma^* \), \( (r + s, w) = (r, w) + (s, w) \), and \( (r \cdot s, w) = \sum_{w_1, w_2, w_3} (r, w_1) s_2, w_2) \). Their respective neutral element are \( 0 \) and \( 1 \). It follows that \( A^{\Sigma^*} \), \( + \), \( \cdot \), \( 0 \), \( 1 \) is a semiring.

3.1 Algebraic Systems

\[ A \text{-formal power series} \]

Consider \( A \) a CF-grammar \((\Sigma, N, R, N_i)\), a semiring \((A, +, \times, 0, 1)\) and a mapping \( \phi_i \) give rise to the algebraic system with variables in \( N \)

\[ N_i = p_i, 1 \leq i \leq n \] (1)

where each \( p_i \in A^\Sigma^* \), such that

\[
\begin{align*}
(p_i, \alpha) &= \phi_i(N_i \rightarrow \alpha), & \text{if } N_i \rightarrow \alpha &\in R \\
(p_i, \alpha) &= 0, & \text{otherwise}
\end{align*}
\]

which can be represented in matrix notation

\[
N = C, \quad N = \begin{pmatrix} N_1 \\ \vdots \\ N_n \end{pmatrix}, \quad C = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}
\]

This definition differs from [KS]'s because of the introduction of the mapping \( \phi_i : \phi_i(N_i \rightarrow \alpha) = 1 \) for each production in [KS].

3.2 Solutions

A solution of such a system is a vector \( \sigma \) of \( n \) power series in \( A^\Sigma^* \), s.t. replacing \( N_i \) by \( \sigma_i \) in each equation \( N_i = p_i \) gives a valid equality.

So, the existence of a solution depends on the context-free grammar and on the mapping \( \phi_i \).

A solution may be obtained by the approximation sequence

\[ \sigma^0, \sigma^1, ..., \sigma^t, ... \]

where \( \sigma^t \in (A^\Sigma^*)^{n \times 1} \) — implicitly considered as a morphism that replaces each occurrence of the variable \( N_i \) in \( C \) by the polynomial \( \sigma_i^t \) which is the \( t \)th component — as follows

\[ \sigma^0 = 0, \sigma^{t+1} = \sigma^t(C) \]

If the approximation sequence converges, w.r.t. the discrete convergence (see [KS], chap. 3), to \( \sigma \), i.e. if

\[ \lim_{j \to \infty} \sigma^j = \sigma \]

then \( \sigma \) is a solution, called the \emph{strong solution}.

The boolean semiring leads to specific algebraic systems. The choice of a grammar \( \phi_i \): it is a constant function yielding \text{true}, because \text{false} (neutral element of \( \land \)) is not permitted (see section 2). Every \( B^\Sigma^* \) algebraic system has a strong solution and \( L(G) = \text{supp}(\sigma_1) \). Then we see the relation between \( B \) and the recognition.

Stochastic parsing is related to the semiring of positive reals \( \mathbb{R}^+ \), [Ten] performs parsing by explicitly extracting probabilistic algebraic systems.
and solving them. The existence of a solution depends on the mapping \( \phi_{R^n} \) that defines a probability on \( R \): systems are solvable if \( \phi_{R^n} \) induces a probability on \( L(G) \).

The system (1) is in general never solved because languages are generally not finite and the most interesting series, \( \sigma_1 \), is an infinite sum. In fact, such a system has to be stated for each \( x \), but this topic is developed in the parsing section.

4 Dynamic Programming

In our context, dynamic programming can be viewed as a technique, applicable on some recursive statements, that lowers the complexity of computations using memoization to avoid repetitive computation steps.

Consider a recursively defined function \( F \) from a set \( D \) to a set \( E \), and \( D \in D \). The dynamic programming technique is applicable to compute \( F(D) \) if

1. the recursion converges;
2. \((E, \oplus, \otimes, 0, 1)\) is a semiring; and
3. the recursive formulation of \( F \) matches the following pattern: \( \forall i \in N, \exists \{\delta_1, \ldots, \delta_p\} \subseteq D, \exists \{f_1, \ldots, f_p\} \subseteq E \) such that
   - \( \forall i = 1, \ldots, p, F(\delta_i) = f_i \)
   - \( F(D) = \bigoplus_{\delta \in \{\delta_1, \ldots, \delta_p\}} (\bigotimes_{i=1}^{p} F(d_i)) \)

Sedgewick writes in his book [Sed] (p.595): “No one has characterized precisely which problems can be effectively solved with dynamic programming”. In other words, no general pattern, necessary and sufficient to employ dynamic programming, is known.

In our (sufficient and quite general) pattern, the dependence between \( D \) and the \( e_j \) family is not explicit because it would imply a loss of generality. However the convergence of \( F(D) \) implies such a dependence, implicitly.

Computing \( F(D) \) requires the computation of some \( F(d) \); once computed, each \( F(d) \) is stored in a table, (may be) re-used and never re-computed. This tabulation is the memoization contribution to the technique. The order of computation sequence corresponds to an evaluation strategy (see [VdC], section 3.4): e.g. top-down ( \( F(D) \) is solved by decomposing \( D \) down to smaller elements) or bottom-up (starting from smaller elements, the solution is built up to \( F(D) \)).

\( F(D) \) may be described by a pushdown automaton (PDA). Then such an automaton gives both a computation and an evaluation strategy. Under some conditions (see [VdC], chap. 6) it is possible to produce automatically its dynamic programming interpretation: an abstract machine that respects the strategy given by the automaton and performs the computations with tabulations.

Considering the semiring \( A \), the PDA apparently describes a computation in the monoid \( (A, \times, I) \) but it is non-deterministic: the non-determinism is handled by \( \gamma_1 \). That is why the corresponding function \( F \) can be computed by dynamic programming interpretation of the PDA, effectively working with \( (A, \times, \gamma_1, \times, 0, 1) \).

Operationally, a dynamic programming interpretation is based on sharing. This entails sharing of computation structures, i.e. of stacks: naively, two non-deterministic computations create two different and independent stacks, whereas dynamic programming shares their common parts. Two kinds of interpretation, based on a different stack management, are detailed in [VdC], chapter 6.

5 Parsing

Considering the string \( x \) to parse, the grammar \( G \), the semiring \( A \), the mapping \( \phi_1 \) and the associated algebraic system, parsing consists in finding the coefficient \( (\sigma_1, x) \) whenever it exists, which assumes that the algebraic system has a solution.

Then \( \phi_1 \) is extended on \( \Sigma^* \) by association with \( \sigma_1 \):

\[
\phi_1(x) = (\sigma_1, x)
\]

5.1 Coefficient Computation

Practically, no general solution vector is computable and \( (\sigma_1, x) \) must be computed for each given \( x \).

We have for each \( i = 1, \ldots, n \), and each \( w \in \Sigma^* \)

\[
(N_i, w) = (p_i, w)
\]

hence

\[
(N_i, w) = \sum_{N_i \rightarrow \gamma_1 \in R} \phi_1(N_i \rightarrow \gamma_1) x_1(\gamma_1, w)
\]

with

\[
\gamma_1 = y_1 N_i, y_2 N_i, \ldots y_k N_i, y_{k+1}
\]

and

\[
w = y_1 x_1 y_2 x_2 \cdots y_k x_k y_{k+1}
\]
thus
\[(N_i, w) = \sum_{N_i \rightarrow \gamma_i \in \mathcal{R}} \phi_i(N_i \rightarrow \gamma_i) x_i A \prod_{j=1,...,k} (N_{ij}, y_j)\]

This is not a recursive function, but a recursive system, matching the pattern required by dynamic programming:

- the rule set matches the \( \delta_i \) family;
- the application of \( \phi_i \) on a rule matches, for a given \( i \), \( F(\delta_i) \), and
- the set of rules with \( N_i \) as left hand side matches the \( e_j \) family.

Moreover, the value domain of F matches that of coefficients: the semiring \( A \). So, if the system above has a solution, it can be computed using dynamic programming: for each \( N_i \) and each sub-word \( y \) of the string \( x \), the coefficient \( (N_i, y) \) is tabulated.

As for the system (1) (of which variables where algebraic power series), we have a system of the form
\[Y = K\] (2)

which is the transposition into \( A \) of the system (1): \( X \) is a \( A^{m \times 1} \) matrix, \( m \) being the number of coefficients: \( m = n \times (1+\text{Card}\{y \in \Sigma^* | \exists w_1, w_2 \in \Sigma^* \text{ s.t. } x = w_1 y w_2 \}) \).\footnote{"1+..." because of \( \epsilon \).} A solution is a vector \( \zeta \) s.t. replacing each \( \zeta_i \) in \( K \) yields a valid equation, and a solution can be obtained by the same kind of approximation sequence \( \zeta^0 = 0, \zeta^{i+1} = \zeta^i(K) \).

5.2 POLYMORPHIC FUNCTIONS

In the system, a dependence is the transitive closure of the relation defined by: \( \forall i, l \in \mathcal{N}, w, y \in \Sigma^*, (N_i, w) \leadsto (N_i, y) \iff \exists j \text{ s.t. } i_j = l \) in the following equation of the system
\[(N_i, w) = \sum_{N_i \rightarrow \gamma_i \in \mathcal{R}} \phi_i(N_i \rightarrow \gamma_i) x_i A \prod_{j=1,...,k} (N_{ij}, y_j)\]

Simply, we have \( (N_i, w) \leadsto (N_i, y) \) iff \( N_i \Rightarrow (N_i, w) \). Hence, if \( G \) is not cyclic then the system always has a solution: the dependence graph of the relation \( \leadsto \) contains no cycle.

Otherwise, the existence of a solution is not guaranteed and if any, its computation requires a limit: if the fix-points converge on all cycles in the dependence graph, then there is a solution. It consists in an expression in \( A \) with the \( \phi_i(N_i) \) family as only constants, composed by \( +x_i, x_i \) and eventually requiring limits:
\[(\sigma_1, x) = \Phi(+x_i, x_i, (\phi_i(R))_{R \in \mathcal{R}})\]

where \( \Phi \) is a polymorphic function, typed by \((A \times A) \times (A \times A) \times A^\circ \rightarrow A\).

Now, if we want to compute another coefficient of \( x \), keeping the same grammar, but in another semiring, say \( B \), using another mapping \( \phi_b \), we have to rerun the process from scratch. In fact, intuitively, keeping the same grammar gives rise to the same form of algebraic system, because \( \Phi \) form does not depend on \( A \) or \( \phi_i \), but on \( G \). So, if we know "how to go from \( A \) to \( B \)", we know "how to go from \( \phi_i(w) \) to \( \phi_b(w) \)."

6 MORPHISMS AND SEMIRINGS

Consider the semirings \((A, +_A, x_A, 0_A, 1_A)\) and \((B, +_B, x_B, 0_B, 1_B)\) and a semiring morphism \( \mu \) from \( A \) to \( B \) such that
\[\forall R \in \mathcal{R}, \phi_i(R) = \mu \circ \phi_i(R)\]

Then \( \phi_i(w) = \Phi(+_A, x_A, (\phi_i(R))_{R \in \mathcal{R}}) \) implies
\[\mu(\phi_i(w)) = \mu(\Phi(+_A, x_A, (\phi_i(R))_{R \in \mathcal{R}})) = \Phi(+_B, x_B, (\mu(\phi_i(R)))_{R \in \mathcal{R}}) = \Phi(+_B, x_B, (\phi_i(R))_{R \in \mathcal{R}}) = \phi_b(w)\]

under the assumption: existence of limit in \( A \) implies existence of limit in \( B \).

A semiring has a special status: the (parse-) forest semiring. It is special because for a grammar, a string to parse and a mapping \( \phi_k \), there exists a morphism from the forest semiring to \( A \) that computes \( \phi_k(w) \). Therefore we qualify it as canonical.

6.1 THE FOREST SEMIRING

We use the approach of [Lg89] to define forests as grammars, only described by their rules: \((N, P)\) is a derivation forest from \((\Sigma, N, \mathcal{R}, \delta)\).

Definition: A derivation forest is the unit forest \( \delta \) or a couple \((N, P)\), with \( N \) as set of non-terminals and \( P \) as production set (subset of
\(N \times (N \cup \Sigma)^*\), such that there exists a labelling function \(\text{lab}\) from \((\Sigma \cup N)^* \cup P\) to \((\Sigma \cup N)^* \cup R\) verifying

\[
\forall N \in N, \text{lab}(N) \in N, \\
\forall w \in \Sigma^*, \text{lab}(w) = w, \\
\forall X \in \Sigma \cup N^*, \alpha \in (\Sigma \cup N)^*, \\
\text{lab}(X \alpha) = \text{lab}(X) \text{lab}(\alpha), \\
\forall A \rightarrow \alpha \in P, \text{lab}(A \rightarrow \alpha) = \text{lab}(A) \rightarrow \text{lab}(\alpha)
\]

The set of derivation forests is \(T\). Productions are called nodes.

**Definition**: The composition of derivation forests is the function \(\otimes : T \times T \to T\) s.t.

\[
\forall t \in T, t \otimes t_1 = t_1 \otimes t = t.
\]

\[
\forall t_1 = (N_1, P_1) \text{ and } t_2 = (N_2, P_2), t_1 \otimes t_2 = (N_1 \cup N_2, P_1 \cup P_2)
\]

The union being associative, the composition of derivation forests is associative, and has \(t_1\) as neutral element. Thus \((T, \otimes, t_1)\) is a monoid, called derivation-forest monoid. Note that \(\otimes\) is also commutative, then the derivation-forest monoid is commutative.

Consider the set of parts of \(T, \mathcal{P}(T)\), the natural extension of \(\otimes\) to \(\mathcal{P}(T)\) such that \(\otimes\) is distributive with respect to \(\cup\), and the statement \(\forall f \in \mathcal{P}(T), \emptyset \otimes f = f \otimes \emptyset = \emptyset\). Thus \((\mathcal{P}(T), \cup, \otimes, \emptyset, \{t_1\})\) is a semiring: the parse-forest semiring, also simply called forest semiring, noted \(\mathcal{F}\). Furthermore,

1. as the derivation-forest monoid is commutative, so is the parse-forest semiring,

2. \(\{t_1\} \cup \{t_1\} = \{t_1\}\), hence \(\mathcal{F}\) is idempotent, thus partially ordered, and then zerosumfree

3. (consequence) \(\mathcal{F} < \Sigma^* \otimes \mathcal{P}\) is zerosumfree, partially ordered and idempotent (see [KS] chap.5)

4. from \(x \mathcal{F}\) definition, we get \(\forall f \in \mathcal{P}(T), f \mathcal{F} f = f\), i.e. \(x \mathcal{F}\) is idempotent, so, \(\forall i, f^i = f\) and then \(f^* = f\)

**Definition**: canonical parse-forest

A canonical parse-forest \(g\) is a CF-grammar \((\Sigma, N, P, s)\) such that

- it is proper,

- \((N, P)\) is a derivation forest, with a labelling function \(\text{lab}\),

- \(\text{lab}\) verifies \(\text{lab}(s) = N_1,\)

- \(L(g) = \{x\}, x\) the string to parse.

This description of the parse-forest semiring must be completed by the function \(\phi_F\), that must be specified on a grammar to give rise to the associated algebraic system. For this we use a general definition of shared parse forest employed in [Bou].

Given \(G\) and \(x, \phi_F : R \to \mathcal{F}\) such that for all \(R = N_0 \rightarrow x_1 N_1 x_2 N_2 \ldots x_k N_k x_{k+1} \in R\), \(\phi_F(R) = (N, P)\), where

\[
N = \{N_{i,j}^k, j \geq 0, i \geq 0, j \geq 0 \land \\
N_i = N_{i} \rightarrow i = 0, j = |x_i| \land \\
P = \{N_{j, i}^{k, j}, i \geq 0, j \geq 0, k \geq 0, j \geq 0 \land \\
i + |x_i| = i_j \land \\
\text{if } 1 < i \rightarrow j_i - 1 + |x_i| = i_j\}
\]

## 6.2 Morphism from the canonical semiring

Consider a semiring \(A\) and a semiring morphism \(\mu\) from \(\mathcal{F}\) to \(A\), a string \(x\) to parse and its parse forest, represented by \(\Phi(\mathcal{F}, x, (\phi_F(R))_{R \in \mathcal{R}})\).

Then trivially,

\[
\mu(\phi_F(x)) = \mu(\Phi(\mathcal{F}, x, (\phi_F(R))_{R \in \mathcal{R}})) = \Phi(\mathcal{F}, x, (\mu(\phi_F(R))_{R \in \mathcal{R}})) = \Phi(\mathcal{F}, x, (\phi_F(R))_{R \in \mathcal{R}})
\]

If of limit in \(\mathcal{F}\) implies existence of a limit in \(A\).

## 7 Conclusion

We have showed how dynamic programming techniques can be employed to compute decoration in an abstract semiring.

We have described the parse-forest semiring and the corresponding mapping \(\phi_F\), that associates a forest to each grammar rule. We have also introduced a polymorphic function that describes the expression of a coefficient of the input string \(x\) in an abstract semiring.

We believe that this will facilitate the description and the correctness proof of an automaton that would compute the coefficient of \(x\), in an abstract decoration domain — under the assumption that this domain has a semiring structure.

## Acknowledgements
I thank Pierre Boullier for his precious help in many discussions around this paper, for ideas and references.

REFERENCES


Multimodal linguistic inference*

Michael Moortgat
Research Institute for Language and Speech (OTS)
Trans 10, Utrecht
moortgat@let.ruu.nl

Abstract

In this paper we compare grammatical inference in the context of simple and of mixed Lambek systems. Simple Lambek systems are obtained by taking the logic of residuation for a family of multiplicative connectives /, *, \, together with a package of structural postulates characterizing the resource management properties of the * connective. Different choices for Associativity and Commutativity yield the familiar logics NL, L, NLP, LP. Semantically, a simple Lambek system is a unimodal logic: the connectives get a Kripke style interpretation in terms of a single ternary accessibility relation modeling the notion of linguistic composition for each individual system.

The simple systems each have their virtues in linguistic analysis. But none of them in isolation provides a basis for a full theory of grammar. In the second part of the paper, we consider two types of mixed Lambek systems.

The first type is obtained by combining a number of unimodal systems into one multimodal logic. The combined multimodal logic is set up in such a way that the individual resource management properties of the constituting logics are preserved. But the inferential capacity of the mixed logic is greater than the sum of its component parts through the addition of interaction postulates, together with the corresponding interpretive constraints on frames, regulating the communication between the component logics.

The second type of mixed system is obtained by generalizing the residuation scheme for binary connectives to families of *-ary connectives, and by putting together families of different arities in one logic. We focus on residuation for unary connectives, hence on mixed (2,3) frames, as these already represent the complexities in full. We prove a number of elementary logical results for unary families of residuated connectives and their combination with binary families. The existing proposals for unary 'structural modalities' are situated within this general framework, and a number of new linguistic applications is given.

---

1 Linguistic inference: simple Lambek systems

In this paper, we present categorial grammar as a system of linguistic inference — a logic for reasoning about linguistic resources. The logic has a language of type formulae: atomic formulae, or complex ones, constructed in terms of type-forming connectives — our logical constants. We study the categorial language from a model-theoretic and a proof-theoretic point of view. The language of type formulae is used to talk about the linguistic reality that forms the object of grammatical analysis: a reality of structured linguistic expressions. The models for the type language are abstract mathematical structures that capture the relevant aspects of the linguistic reality we are interested in. Moving to the proof-theoretic perspective, we want to know how to perform valid inferences on the basis of our interpreted type language. We are not interested in syntax as the manipulation of meaningless symbols: we want our grammatical proof theory to be sound and complete with respect to the intended models of the linguistic reality. And, from a more computational point of view, we are interested in decidability and tractability as well.

**Binary Multiplicatives.** We start with a quick review of the landscape of binary multiplicative operators. This is extremely well-trodden ground, and the present section contains nothing new. But it sets the agenda for our exploration of more adventurous territory in §2.1 and §2.2.

Consider the language $\mathcal{F}$ of category formulae of a simple Lambek system. $\mathcal{F}$ is obtained by closing a set $\mathcal{A}$ of atomic formulae (or: basic types, prime formulae, e.g. $s, np, n, \ldots$) under binary connectives (or: type forming operators) $/$, $\cdot$, $\backslash$.

$$\mathcal{F} ::= \mathcal{A} \mid \mathcal{F}/\mathcal{F} \mid \mathcal{F} \cdot \mathcal{F} \mid \mathcal{F}\backslash\mathcal{F}$$

Type formulae have a quite general interpretation in the power set algebra of Kripke style relational structures — ternary relational structures in the case of the binary connectives ([10]). A ternary frame is a structure $(W, R^3)$. In the application to formal grammar envisaged here, the domain $W$ is to be thought of as a set of linguistic resources (or: signs, pieces of multidimensional linguistic information). The accessibility relation $R$ can be understood as representing linguistic composition: $Rxyz$ holds in case one can fuse together the information of signs $y$ and $z$ into a sign $x$. We obtain a model by adding a valuation $\nu$ sending prime formulae to subsets of $W$ and satisfying the clauses below for compound formulae.

$$\nu(A \cdot B) = \{x \mid \exists y \exists z[Rxyz \land y \in \nu(A) \land z \in \nu(B)]\}$$
$$\nu(C/B) = \{y \mid \forall x \forall z[Rxyz \land z \in \nu(B) \Rightarrow x \in \nu(C)]\}$$
$$\nu(A\backslash C) = \{x \mid \forall z \forall y[Rxyz \land y \in \nu(A) \Rightarrow x \in \nu(C)]\}$$

We are interested in characterizing a relation of derivability between formulae such that $A \rightarrow B$ is provable iff $\nu(A) \subseteq \nu(B)$. It is not difficult to check that given the above interpretation of compound formulae, the Residuation laws below determine the properties of $\cdot$ vis à vis $\backslash$, $\backslash$ with respect to derivability.

\[(\text{RES}) \quad A \rightarrow C/B \iff A \cdot B \rightarrow C \iff B \rightarrow A\backslash C\]

Putting things together, we see that the anatomy of the most elementary Lambek type logic is given by the basic properties of the derivability relation (Reflexivity, Transitivity) plus the Residuation Laws establishing the relation between $\cdot$ and the two implications $\backslash$, $\backslash$. Below we give the axiomatic presentation of the system known as NL. Following [27], we add combinator proof terms: they will provide a compact way of referring to complete deductions later on. Via a canonical model construction Došen [10] obtains the elementary soundness and completeness result:
in NL provability coincides with semantic inclusion for all ternary frames and all interpretations \( \nu \).

**NL: the Pure Logic of Residuation.** Combinator proof terms. We write \( f: A \to B \) for a proof of the inclusion \( \nu(A) \subseteq \nu(B) \).

\[
\begin{align*}
\text{id}_A &: A \to A \\
\beta(f) &: A \to C/B \\
g &: A \to C/B \\
\gamma(f) &: B \to A/C \\
g &: B \to A/C \\
\end{align*}
\]

**Structural postulates, constraints on frames.** Starting from the pure logic of residuation NL one can unfold a landscape of categorial type logics by gradually relaxing structure sensitivity in a number linguistically relevant dimensions. Below we consider the dimensions of linear precedence (order sensitivity) and immediate dominance (constituent sensitivity). Adding the structural postulates for Associativity or Commutativity (or both) to the pure logic of residuation, one obtains the systems L, NLP, LP. Using Correspondence Theory [5] one computes frame conditions restricting the interpretation of \( R^3 \) for the different structural postulates. Dozen's completeness result for NL is then extended to the stronger logics by restricting the attention to \textsc{Ass} (L), \textsc{Comm} (NLP) or \textsc{Ass+Comm} frames (LP).

**Structural Postulate**

**Frame Condition**

\[
\begin{align*}
\text{ASS} &: A \bullet (B \bullet C) \leftrightarrow (A \bullet B) \bullet C \\
\text{COMM} &: A \bullet B \to B \bullet A \\
& \exists t. Rtxy & \exists v. Ruzv & \exists w. Rvzyx & \exists w. Rzyx
\end{align*}
\]

**Gentzen Calculus.** The axiomatic presentation is the proper vehicle for model-theoretic investigation of the logics we have considered: it closely follows the semantics, thus providing a suitable basis for 'easy' completeness results. But proof-theoretically the axiomatic presentation has a serious drawback: because it is essentially based on Transitivity, it does not offer an appropriate basis for proof search. For proof-theoretic investigation of the categorial type logics one introduces a Gentzen presentation, and proves a Cut Elimination result, with its corollaries of decidability and the subformula property. Of course, one has to establish the equivalence between the axiomatic and the Gentzen presentations of the logic for all this to make sense. For L Lambek [26] has established the essential results. They have been extended to the full landscape of type logics in [22, 11].

In the axiomatic presentation, we consider arrows \( A \to B \) with \( A, B \in \mathcal{F} \). In Gentzen presentation, the derivability relation is stated to hold between a term \( \mathcal{T} \) (the antecedent) and a type formula (the succedent). A Gentzen term is a structured configuration of formulae — a structured database, in the terminology of Gabbay [15]. The term language is defined inductively as \( \mathcal{T} ::=} \mathcal{F} \mid (\mathcal{T}, \mathcal{T}) \). The binary structural connective \( (\cdot, \cdot) \) in the term language tells you how to put together structured databases \( \Delta_1 \) and \( \Delta_2 \) into a structured database \( (\Delta_1, \Delta_2) \). The structural connective mimics the logical connective \( \bullet \) in the type language. A sequent is a pair \( (\Gamma, A) \) with \( \Gamma \in \mathcal{T} \) and \( A \in \mathcal{F} \), written as \( \Gamma \Rightarrow A \).
To compare the two presentations, we define the formula equivalent \( \Delta^0 \) of a structured database \( \Delta \). Let \( (\Delta_1, \Delta_2)^0 = \Delta_1^0 \cdot \Delta_2^0 \) and \( A^0 = A \) for \( A \in \mathcal{F} \). The Gentzen presentation can be shown to be equivalent to the combinator axiomatisation in the sense of the following proposition from [26].

Every combinator \( f : A \to B \) gives a proof of \( A \Rightarrow B \), and every proof of a sequent \( \Gamma \Rightarrow B \) gives a combinator \( f : \Gamma^0 \Rightarrow B \).

As was the case for the combinator presentation, the sequent architecture consists of three components: (i) \([\text{Ax}]\) and \([\text{Out}]\) capture the basic properties of the derivability relation \( \Rightarrow \)': reflexivity and contextualized transitivity for the 'surgical' Out, (ii) each connective comes with two logical rules: a rule of use introducing the connective to the left of \( \Rightarrow \) and a rule of proof introducing it on the right of \( \Rightarrow \), finally (iii) there is a block of structural rules, possibly empty, with different packages of structural rules resulting in systems with different resource management properties.

\textbf{Gentzen presentation: structured databases.} Sequents \( \mathcal{T} \Rightarrow \mathcal{F} \) where \( \mathcal{T} := \mathcal{F} \mid (\mathcal{T}, \mathcal{T}) \). Notation: \( \Gamma[\Delta] \) for an antecedent term \( \Gamma \) containing a distinguished occurrence of the subterm \( \Delta \).

\[
\begin{align*}
[Ax] & : A \Rightarrow A \\
\text{[Ax]} & : \Delta \Rightarrow A \quad \Gamma[\Delta] \Rightarrow C \quad \text{[Cut]} \\
\text{[\text{R}] & : } & (\Gamma, B) \Rightarrow A/B \\
\text{[\text{R}]} & : \Delta \Rightarrow B \quad \Gamma[\Delta] \Rightarrow C \quad \text{[\text{L}]} \\
\text{[\text{L}] & : } & (B, \Gamma) \Rightarrow B\backslash A \\
\text{[\text{L}]} & : \Delta \Rightarrow B \quad \Gamma[\Delta] \Rightarrow C \quad \text{[\text{L}]} \\
\text{[\text{L}] & : } & (A, B) \Rightarrow C \\
\text{[\text{L}]} & : \Gamma[A \cdot B] \Rightarrow C \quad \Gamma \Rightarrow A \quad \Delta \Rightarrow B \quad \text{[\text{L}]}
\end{align*}
\]

\textbf{Structural rules.} Adding the structural rules of Permutation and/or Associativity one obtains coarser notions of linguistic inference, where structural discrimination with respect to the dimensions of precedence and/or dominance is destroyed.

\[
\begin{align*}
\Gamma[\Delta_1, \Delta_2] & \Rightarrow A \quad \text{[P]} \\
\Gamma[\Delta_1, (\Delta_2, \Delta_3)] & \Rightarrow A \quad \text{[P]} \\
\Gamma[\Delta_1, (\Delta_2, \Delta_3)] & \Rightarrow A \quad \text{[P]}
\end{align*}
\]

For the logics \( L \) and \( LP \) where \( \bullet \) is associative, resp. associative and commutative, explicit application of the structural rules is generally compiled away by means of syntactic sugaring of the sequent language. Antecedent terms then take the form of sequences of formulae \( F, \ldots, F \) where the comma is now of variable arity, rather than a binary connective. Reading these antecedents as sequences, one avoids explicit reference to the Associativity rule; reading them as multisets, one also makes Permutation implicit.

\textbf{Characteristic theorems, derived rules of inference.} We close this overview with an inventory of theorems and derived inference rules for the various logics.

1. Application: \( A/B \cdot B \Rightarrow A, B \cdot B\backslash A \Rightarrow A \)
2. Co-application: \( A \Rightarrow (A \bullet B)/B, A \Rightarrow B\backslash (B \bullet A) \)
3. Monotonicity \( \cdot \): if \( A \to B \) and \( C \to D \), then \( A \cdot C \to B \cdot D \)

4. Isotonicity \( /C, C\)\: if \( A \to B \), then \( A/C \to B/C \) and \( C\setminus A \to C\setminus B \)

5. Antitonicity \( /, \setminus C\): if \( A \to B \), then \( C/B \to C\setminus A \) and \( B\setminus C \to A\setminus C \)

6. Lifting: \( A \to B/(A\setminus B), A \to (B/A)\setminus B \)

7. Geach (main functor): \( A/B \to (A/C)/(B/C), B\setminus A \to (C\setminus B)/(C\setminus A) \)

8. Geach (secondary functor): \( B/C \to (A/B)\setminus (A/C), C\setminus B \to (C\setminus A)/(B\setminus A) \)

9. Composition: \( A/B \cdot B/C \to A/C, C\setminus B \cdot B\setminus A \to C\setminus A \)

10. Restructuring: \( (A\setminus B)/C \leftrightarrow A\setminus (B/C) \)

11. (De)Currying: \( A/(B \cdot C) \leftrightarrow (A/C)/B, (A \cdot B)/C \leftrightarrow B/(A\setminus C) \)

12. Permutation: if \( A \to B\setminus C \) then \( B \to A\setminus C \)

13. Exchange: \( A/B \leftrightarrow B\setminus A \)

14. Preposing/Postposing: \( A \to B/(B/A), A \to (A\setminus B)\setminus B \)

15. Mixed Composition: \( A/B \cdot C\setminus B \to C\setminus A, B/C \cdot B\setminus A \to A/C \)

Items (1) to (5) are valid in the weakest logic NL. Together they provide an alternative way of characterizing \((\cdot, /)\) and \((\cdot, \setminus)\) as residuated pairs, i.e. one can replace the RBS inferences by (1)–(5). See [11] and §2.2 below. Lifting is the closest one can get to (2) in ‘product-free’ type languages, i.e. type languages where the role of the product operator (generally left implicit) is restricted to glue together types on the left-hand side of the arrow. Items (7) to (11) mark the transition to L: their derivation involves the structural postulate of associativity for \( \cdot \). Rule (12) is characteristic for systems with a commutative \( \cdot \), NLP and LP. From (12) one immediately derives the collapse of the implications / and \( \setminus \), (13). As a result of this collapse, one gets variants of the earlier theorems obtained by substituting subtypes of the form \( A/B \) by \( B\setminus A \) or vice versa. Examples are (14), an NLP variant of Lifting, or (15), an LP variant of Composition.

**Discussion: rule-based versus logic-based approaches.** The simple Lambeck systems each have their merits and their limitations when it comes to grammatical analysis. As a grammar writer, one would like to exploit the inferential capacities of a combination of different systems. In rule-based frameworks, such as Combinatory Categorial Grammar (CCG, cf [36]), instances of the full scala of type transitions illustrated in (1)–(15) above indeed live together. In the logical setup adopted in this paper, such promiscuity has unpleasant consequences. As we have seen above, the Residuation laws capture the basic properties of the interpretation of the type-forming connectives. In the presence of Residuation, the introduction of theorems from a system with more relaxed resource management into a logic with a higher degree of structural discrimination instantly destroys sensitivity for the relevant structural parameter of the more discriminating logic. For example: NL has a hierarchically structured database which respects constituent structure. For cases of so-called non-constituent coordination, one would like to relax constituent structure. One could try to achieve this by adding Composition (or the Geach laws) to NL. But the addition of such postulates makes NL collapse into L: from Geach one easily obtains the unconditional Associativity postulate for \( \cdot \) via Residuation. We leave this as an exercise for the reader. Similarly, it has been argued that an analysis of Dutch crossed dependencies requires the Mixed Composition laws.
Again, the introduction of this LP theorem within an order-sensitive system such as L causes permutation collapse.

CCG avoids these problems by restricting the attention to a database of rule schemata without facing the semantic consequences of their combination. This route is not open to us if we want to leave intact the idea of a grammar logic, i.e. a semantically interpreted grammar formalism. In the following sections we develop a logical framework supporting mixed styles of categorial inference. Structural collapse is avoided by moving to a multimodal architecture which is better adapted to deal with the fine-structure of linguistic composition.

2 Residuation in mixed logics

2.1 Mixed inference: multimodal systems

Our first generalizing move is from a unimodal setup, where the type-forming connectives are interpreted in terms of a single notion of linguistic composition, to a multimodal architecture. The objective here is to combine the virtues of the distinct logics we have discussed before in one multimodal system, and at the same time to overcome the limitations of the individual systems in isolation. Each of the component logics has its own specific resource management properties: when combining the different logics, we have to take care that these individual characteristics are left intact. We do this by relativizing linguistic composition to specific resource management modes. But also, we want the inferential capacity of the combined logic to be more than the sum of the parts. The extra expressivity comes from interaction postulates that hold when different modes are in construction with one another.

On the syntactic level, the category formulae for the multimodal system are defined inductively on the basis of a set of category atoms $A$ and a set of indices $i \in I$ as shown below. We refer to the $i \in I$ as resource management modes, or modes for short.

$$\mathcal{F} ::= A \mid \mathcal{F}/i \mid \mathcal{F} \circ_i \mathcal{F} \mid \mathcal{F}\setminus_i \mathcal{F}$$

The semantics for the mixed language is a straightforward generalisation of frame semantics for the simple systems. Rather than interpret multiplicative connectives in terms of one privileged notion of linguistic composition, we throw different forms of linguistic composition together and interpret in multimodal frames $(W,\{R_i\}_{i\in I})$.

A valuation on a frame respects the structure of the complex types in the familiar way, interpreting each of the modes $i \in I$ with its own accessibility relation.

$$\begin{align*}
v(A \circ_i B) & = \{ z \mid \exists y. \exists z. R_{i}xyz \land y \in v(A) \land z \in v(B) \} \\
v(C /_i B) & = \{ y \mid \forall z. \forall y. (R_{i}xyz \land y \in v(A)) \Rightarrow x \in v(C) \} \\
v(A \setminus_i C) & = \{ z \mid \forall x. \forall y. (R_{i}xyz \land y \in v(A)) \Rightarrow x \in v(C) \}
\end{align*}$$

We can present the multimodal logic axiomatically or in Gentzen style. In the axiomatic presentation, we have the familiar residuation pattern now relativized to resource management modes:

$$A \rightarrow C /_i B \iff A \circ_i B \rightarrow C \iff B \rightarrow A \setminus_i C$$

In sequent presentation, each residuated family of multiplicatives $\{/_, \circ_i, \setminus_i\}$ has a matching structural connective, again relativized to resource management modes. Antecedent terms are inductively defined as $\mathcal{T} ::= \mathcal{F} \mid (\mathcal{T}, \mathcal{T})^\dagger$. Logical rules insist that use and proof of connectives respect the resource management modes. The explicit construction of the antecedent database in terms of structural connectives derives directly from Belnap’s [6] work on Display Logic, where it serves exactly the
same purpose as it does here, viz. to combine logics with different resource management regimes. \(^1\) In addition, the mode information makes it possible to distinguish distinct forms of linguistic composition with the same resource management properties. For an example, see [28] where the product is split up in a left-headed \(\bullet_1\) and a right-headed \(\bullet_\ast\), introducing a dimension of dependency structure next to the dimensions of precedence and dominance.

The multimodal Gentzen rules for the connectives are presented below. The Axiom sequent and Cut rule remain unchanged — they have no mode restrictions.

\[
\begin{align*}
\text{[R/\|]} & \quad (\Gamma, B)^{i} \Rightarrow A \quad \frac{\Gamma \Rightarrow B \quad \Delta[A] \Rightarrow C}{\Delta[(\Gamma, B, \Gamma)^{i}] \Rightarrow C} & \quad [\text{L/\|}] \\
\text{[R\\]} & \quad (B, \Gamma)^{i} \Rightarrow A \quad \frac{\Gamma \Rightarrow B \quad \Delta[A] \Rightarrow C}{\Delta[(\Gamma, B, \Gamma)^{i}] \Rightarrow C} & \quad [\text{L\\}] \\
\text{[L\bullet_1]} & \quad \Gamma[(A, B)]^{i} \Rightarrow C \quad \frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{(\Gamma, \Delta)^{i} \Rightarrow A \bullet_1 B} & \quad [\text{R\bullet_1}] \\
\end{align*}
\]

In addition to the residuation inferences which are shared by all resource management modes, we now have mode-specific structural options. In axiomatic style, they take the form of structural postulates; in sequent presentation, we have the corresponding structural rules. As an illustration, see the structural postulates/rules for a commutative mode \(\circ\). In the semantics the \(R_c\) interpreting this connective will be constrained to satisfy \((\forall x, y, z \in W)\ R_c x y z \Rightarrow R_c z y x.\)

\[
A \circ_2 B \leftrightarrow B \circ_2 A
\]

**Multimodal Communication.** What we have done so far is simply put together the individual systems discussed before in isolation. This is enough to gain combined access to the inferential capacities of the component logics, and one avoids the unpleasant collapse into the least discriminating logic that results from combining logics without taking into account the mode specifications, cf our discussion of CCC in §1. But as things are, the borders between the constituting logics in our multimodal setting are still hermetically closed. Let us turn then to the question of multimodal communication.

Communication between modes \(i, j\) is obtained semantically by frame conditions linking the interpretation of the relations \(R_i\) and \(R_j\). Two types of constraints can be distinguished. We discuss them in turn in the sections that follow.

- Ordering of the accessibility relations for modes \(i, j\) can be implemented by means of frame conditions of the form \((\forall x, y, z \in W)\ R_c x y z \Rightarrow R_j x y z.\) Corresponding to this form of frame condition, there will be Inclusion Postulates \(A \bullet_1 B \rightarrow A \bullet_2 B\) in the logic.

- Frame conditions ‘mixing’ distinct modes \(i, j\) allow for the statement of distributivity principles regulating the communication between \(R_i, R_j\). This type of constraint is expressed in the logic in terms of Interaction Postulates. They allow for the formulation of constrained multimodal forms of associativity, commutativity, and contraction.

**Inclusion Principles.** One can develop different perspectives on inclusion principles depending on the interpretation one has in mind for the ordering of the \(R_i, R_j\)

\(^1\)See [23, 41] for recent applications in a modal setting. More recently, the same idea has been introduced in Linear Logic in [17].
involved. A natural candidate would be an ordering in terms of the information they provide about the structure of the linguistic resources. From this perspective, one can consider the non-commutative product \(\cdot\) as more informative than the commutative product \(\otimes\), since the former but not the latter is sensitive to the linear order of the resources. In terms of frame conditions, we would impose the constraint \((\forall x, y, z \in W) \ R_{xyz} \Rightarrow R_{\otimes xyz}\).

To illustrate the role of inclusion principles in linguistic inference consider the cut-free Gentzen derivation \((\dagger)\) below. The derivation uses Left and Right logical rules (which depend on the proper sequent punctuation), the inclusion structural rule connecting \(\cdot\)-type and \(\otimes\)-type configurations, and the Permutation structural rule for \(\otimes\)-type configurations. The \((\dagger)\) derivation shows that residuation reverses the order of the derivability arrow for the corresponding implicative theorem, i.e. the non-commutative implication is derivable from the commutative one.

\[
\frac{B \Rightarrow B}{(B, A) \Rightarrow B \otimes A} \quad \frac{A \Rightarrow A}{(A, B) \Rightarrow B \otimes A} \quad \frac{A \otimes B \Rightarrow A}{\cdot \subseteq \otimes} \quad \frac{B \Rightarrow B}{(A \otimes B, B) \Rightarrow A} \quad \frac{A \otimes B \Rightarrow A}{\cdot \subseteq \otimes} \quad \frac{(A \otimes B, B)^* \Rightarrow A}{A \otimes B \Rightarrow A^\circ \otimes} \quad \frac{(A \otimes B)^* \Rightarrow A}{(A \otimes B)^* \Rightarrow A} \quad \frac{A \otimes B \Rightarrow A}{A \otimes B \Rightarrow A^\circ \otimes}
\]

The combinator-style derivations corresponding to these Gentzen proofs depend on the inclusion postulate and transitivity. Below the combinator derivation for \((\dagger)\).

\[
\frac{A \otimes B \Rightarrow A \otimes B}{A \otimes B \cdot B \Rightarrow A \otimes B \otimes B} \quad \frac{A \otimes B \otimes B \Rightarrow A}{A \otimes B \cdot B \Rightarrow A} \quad \frac{A \otimes B \Rightarrow A}{A \otimes B \Rightarrow A / B}
\]

A linguistic illustration of the inclusion principles above is provided by Dutch complement-taking adjectives. Some adjectival heads obligatorily precede their complement, others can either follow or precede them. From our multimodal perspective, the former are typed in terms of the non-commutative implication \(A/B\), the latter in terms of commutative \(A \otimes B\). Now consider a conjunction of commutative \(A \otimes B\) and non-commutative \(A/B\). Given the inclusion postulate \(A \cdot B \Rightarrow A \otimes B\) the conjunction as a whole has to be \(A/B\) — which is as it should be: the commutativity of the \(\otimes\) mode should not spoil the more discriminating resource management of the \(\cdot\) mode.

The reader may be interested in comparing the treatment of inclusion constraints given here with that of Hepple [21] where the derivability arrows are systematically reversed as the result of a different interpretation of the ordering among modes.

**INTERACTION PRINCIPLES.** Among the multimodal interaction principles, we distinguish cases of weak and strong distributivity. The weak distributivity principles do not affect the multiplicity of the linguistic resources. They allow for the realization of mixed associativity or commutativity laws as the multimodal counterparts of the unimodal versions discussed above. Interaction principles of the strong distributivity type duplicate resources, thus giving access to mode-restricted forms of Contraction.

**WEAK DISTRIBUTIVITY.** Consider first interaction of the weak distributivity type. Below one finds principles of mixed associativity and commutativity. Instead of the global associativity and commutativity options characterizing L, NLP, LP, we can now formulate constrained forms of associativity/commutativity, restricted to the situation where modes \(i\) and \(j\) are in construction. (Symmetric duals can be added
with the i mode distributing from the right, and one can split up the two-directional inferences in their one-directional components, if so required.)

\[
\begin{align*}
MP & : A \bullet_i (B \bullet_j C) \quad \leftrightarrow \quad B \bullet_j (A \bullet_i C) \\
MA & : A \bullet_i (B \bullet_j C) \quad \leftrightarrow \quad (A \bullet_i B) \bullet_j C
\end{align*}
\]

The interaction postulates correspond to the frame conditions one finds below \((\forall xyzu \in W)\):

\[
\begin{align*}
MP & : \exists t(R_{xu}zt & \& R_{zt}y) \quad \Leftrightarrow \quad \exists t'(R_{zt}uyt & \& R_{zt}xv) \\
MA & : \exists t(R_{xu}zt & \& R_{zt}y) \quad \Leftrightarrow \quad \exists t'(R_{zt}uyt & \& R_{zt}xv)
\end{align*}
\]

And they manifest themselves in structural rules in Gentsen presentation.

\[
\begin{align*}
\frac{\Gamma[(\Delta_2,(\Delta_1,\Delta_3))^i]}{\Gamma[(\Delta_1,\Delta_2,\Delta_3)^j]} \Rightarrow A} & \quad \text{[MC]} \\
\frac{\Gamma[(\Delta_1,\Delta_2)^i,\Delta_3^j]}{\Gamma[(\Delta_1,\Delta_2,\Delta_3)^j]} \Rightarrow A} & \quad \text{[MA]}
\end{align*}
\]

For linguistic application of these general postulates, we refer to the analysis of Dutch Verb Raising in [30, 28], where it is shown that a multimodal variant of the CCG ‘mixed composition’ law — which in the absence of mode constraints causes collapse of \(L\) into \(LP\), as we saw above — is in fact a theorem in combined logics with the MP/MA interaction principles. An example is given below. The verb cluster \textit{wil lezen} is obtained in terms of a bimodal interaction principle relating, in this particular case, the right-headed dependency mode \(*_r\) and the pre-head Dutch head adjunction mode \(*_w\). The former characterizes the head-final clausal structure of Dutch, and is used in the typing of the verb \textit{lezen} as \(np \_1 iv\). The latter allows the verb-raising trigger \textit{wil}, typed \(vp/\_1 iv\), to form a verb cluster together with the head of its \(iv\) infinitival complement.

\[
\begin{align*}
\text{(dat Marie) boeken (wil lezen)} \\
\text{(that Mary) books (wants read)/that M. wants to read books)} \\
vp/\_1 iv \Rightarrow (np/\_1 vp)/vp(np/\_1 iv)
\end{align*}
\]

Schematically, in ‘Geach’ version, we have the following derivation. Notice that the order sensitivity of the individual modes \(*_r\) and \(*_w\) is respected: the valid forms of mixed composition form a subset of the composition laws derivable within unimodal \(LP\). The principles of Directional Consistency and Directional Inheritance, introduced as theoretical primitives in the rule-based setting of CCG, can be seen here to follow automatically from the individual resource management properties of the modes involved and the distributivity principle governing their communication.

\[
\begin{align*}
& \frac{C \Rightarrow C \quad B \Rightarrow B}{(C, C \_r B)^* \Rightarrow B} \quad \text{\(\_L\)} \\
& \frac{A \Rightarrow A}{(A/\_w B, (C, C \_r B)^\_w) \Rightarrow A} \quad \text{\(\_w L\)} \\
& \frac{(A, A/\_w B, C/\_r B)^\_w \Rightarrow A}{(A/\_w B, C/\_r B) \Rightarrow (C/\_r A)} \quad \text{\(\_R\)} \\
& \frac{(A/\_w B, (C, C \_r B)^\_w) \Rightarrow A}{(A/\_w B) \Rightarrow (C, C \_r B)} \quad \text{\(\_w R\)}
\end{align*}
\]

\textbf{INTERACTION PRINCIPLES: STRONG DISTRIBUTIVITY.} As remarked above, the weak distributivity principles MP, MA keep us within the family of resource neutral logics: they do not affect the multiplicity of the resources in a configuration. Strong distributivity principles are not resource neutral: they duplicate resources. As an example, consider the interaction principle MC below, which strongly distributes mode \(j\) over mode \(i\) thus copying a \(C\) datum. Rather than introducing global
Contraction, this interaction principle allows for a constrained form of copying, restricted to the case where modes $i$ and $j$ are in construction. The computation of the relevant frame condition and structural rule will be familiar by now. (As with the Mixed Associativity/Commutativity principles, a symmetric case for distributivity from the left can be added straightforwardly.)

\[
MC : \quad (A \cdot_i B) \cdot_j C \rightarrow (A \cdot_j C) \cdot_i (B \cdot_j C)
\]

\[(R_tzx & R_t uzx) \Rightarrow \exists t' \exists t'' (R_t t'zx & R_t t''yz & R_l ut' tu'')\]

\[
\Gamma[(\Delta_1, \Delta_3) \cdot_i, (\Delta_2, \Delta_3) \cdot_j] \Rightarrow A
\]

\[
\Gamma[(\Delta_1, \Delta_3) \cdot_i, (\Delta_2, \Delta_3) \cdot_j] \Rightarrow A \quad MC
\]

Grammatical inference requires restricted access to Contraction for the analysis of parasitic gap constructions (†) and coordination of incomplete material (‡). In the former case, one would like the abstractor associated with the *wh* element to bind multiple occurrences of the same variable. Such multiple binding is beyond the scope of resource sensitive inference. In the (‡) example, sentential coordination is generalized to the coordination of sentences missing an object — a process which again requires the copying of resources.

(†) Which books did John (file _ without reading _)?

(‡) John loves but Mary hates beans

In the rule-based framework of CCG, parasitic gaps are handled by means of the combinator $S$ which is introduced as a primitive for this purpose, cf. [37].

\[
S : \quad A/C, (A \backslash B)/C \Rightarrow B/C
\]

In a unimodal setting the $S$ combinator in combination with Residuation causes disaster. In the multimodal framework presented here, a mode-restricted form of the $S$ combinator can be derived from the strong distributivity principle discussed above. In the Gentzen proof below, we give the relevant instance for the derivation of the example sentence (instantiate $A/j C$ as $vp/j np$ for *file*, and $(A \backslash B)/j C$ as $(vp \backslash vp)/j np$ for *without reading*). Mode $j$ here would be the default mode by which the transitive verbs *file* and *read* consume their direct objects; the combination of the $vp$ adjunct *without reading _* with the $vp$ it modifies is given in terms of mode $i$, the 'parasitic' mode which licenses the secondary gap depending on the primary one, the argument of *file*.

\[
S & C
\]

\[
((A/j C, C) \cdot_j (A \backslash B)/j C, C) \Rightarrow B
\]

\[
MC
\]

\[
((A/j C, (A \backslash B)/j C, C) \cdot_j (A \backslash B)/j C, C) \Rightarrow B/j C
\]

\[
\Rightarrow R
\]

Notice that in the case of Right-Node Raising (‡), we can appeal to the same interaction principle to derive the non-constituent coordination from sentential coordination, provided the incomplete conjuncts are put together in the duplicating mode $i$ — structural information that can be projected straightforwardly from the type assignment to the conjunction particle, e.g. $(a \backslash s)/s$. We reserve a fuller treatment of generalized coordination in terms of restricted Contraction for another occasion.

### 2.2 Mixed inference: combining 1-ary and 2-ary families

What we have studied so far is the language of binary connectives — a language well adapted to talk about forms of linguistic composition where two resources are put
together. But sometimes one would like to attribute particular resource management properties to individual resources, rather than to configurations of resources. The required expressivity can be introduced by extending the type language with unary connectives decorating individual formulae. Unary connectives entered the linguistic discussion in 1990 in the work of a number of Edinburgh researchers, cf. [2]. Taking their inspiration from the '!' operator of Linear Logic which licenses Contraction and Weakening for '!' decorated formulae, these authors have introduced structural modalities — unary operators providing controlled access to linguistically relevant structural options, such as Permutation. In the recent literature one finds a panoply of unary operators in addition to the binary multiplicatives. Apart from the structural modalities, we can mention the 'domain modalities' of Morrill and Hepple, identifying semantic intensionality domains ([31]) or purely syntactic domains of locality ([20]), the 'bracket operators' [\textsuperscript{1}[,\textsuperscript{1}] of Morrill [32, 33], implementing locality domains in a different way, or the \textsuperscript{\bigodot} operator of Morrill [32], declaring argument positions as licensing extraction.

Our aim in this section is to develop a general framework that will naturally accommodate the different proposals for unary operators while at the same time providing more fine-grained notions of structural control. The key concept, again, is residuation. We extend the language of binary multiplicatives with a unary pair of residual operators \textsuperscript{\bigodot}, \textsuperscript{\bigcirc} and establish a number of elementary logical results for the extended language. Parallel to our treatment of the binary multiplicatives /, \textbullet, \setminus in the previous section, we start from the most discriminating system, i.e. the pure logic of residuation for \textsuperscript{\bigodot}, \textsuperscript{\bigcirc}. By gradually adding structural postulates, we obtain versions of these unary operators with a coarser resource management regime. And where the linguistic applications require this, we can put together different variants in a multimodal logic (\textlangle i\rangle, \textlangle i\rangle\textsuperscript{\bigodot}, \textlangle i\rangle\textsuperscript{\bigcirc}).

Our agenda for this section is given below. Items (1) and (3), (4) closely follow Lambek's [26] treatment of the binary multiplicatives. We refer to Kurtonina [24] for a thorough investigation of the further logical ramifications of the matters dealt with here.

1. Axiomatic ('combinator') presentations of the pure logic of residuation for \textsuperscript{\bigodot}, \textsuperscript{\bigcirc}.

2. Soundness and completeness via the Do\'sen canonical model construction.

3. Gentzen presentation, equivalence between the axiomatic and the Gentzen presentation.


5. Structural postulates \(T, 4, K\). Items (1)–(4) for the systems with a choice from \(\mathcal{P}(\{T, 4, K\})\).

**Residuation: n-ary generalisation.** The concept of residuation, which as we saw above lies at the heart of categorial type logic, arises in the study of order-preserving mappings. In order to widen our framework from binary to n-ary families of residuated connectives, we first have a brief look at the general algebraic concept. (In [29] the reader can find a more thorough treatment with reference to the source material, such as [14, 7]).

Let \(A = (A, \leq_A)\) and \(B = (B, \leq_B)\) be partially ordered sets. Consider a pair of functions \(f : A \rightarrow B\) and \(g : B \rightarrow A\). The pair \((f, g)\) is called residuated if the inequalities of \((\ast)\) hold. Alternatively, a pair of functions \((f, g)\) is characterized as
residuated by requiring \( f \) and \( g \) to be isotone (\( \dagger \)), and by having the composition of the functions satisfy the inequalities of (\( \ddagger \)).

\[
(\ast) \quad fx \leq_B y \text{ iff } x \leq_A gy
\]

(\( \dagger \)) if \( x \leq_A y \) (\( x \leq_B y \)) then \( fx \leq_B fy \) (\( gx \leq_A gy \))

(\( \ddagger \)) \( fx \leq_B x \), \( x \leq_A g fx \)

Dunn's papers on 'gaggle theory' ([12, 13]) provide an excellent survey of the many guises under which Residuation presents itself in (intuitionistic, modal, relevance, dynamic, temporal, linear, ...) logic, and in Lambek style type logics. Indeed, the pairs of connectives \((\bullet, /)\) and \((\bullet, \backslash)\) are easily recognized as the binary incarnations of the notion of residuation just defined for the case of unary operations \( f, g \). Interpret the partially ordered set \( A (\leq B) \) as the set of type formulae \( F \), ordered by derivability \( \rightarrow \) (i.e. set-theoretic inclusion, semantically). For the right residual pair \((\bullet, /)\) we can read \( f \) as \(- \bullet B\) and \( g \) as \(- / B\), i.e. the product and division operations indexed by some fixed type \( B \). The defining biconditional \( fx \leq y \) iff \( x \leq gy \) then becomes \( A \bullet B \rightarrow C \iff A \rightarrow C / B \). Similarly for the left residual pair \((\bullet, \backslash)\), where we read \( f \) as \( A \bullet - \) and \( g \) as \( A \backslash - \), and obtain \( A \bullet B \rightarrow C \iff B \rightarrow A \backslash C \).

The concept of residuation can be readily generalized to the case of \( n \)-ary connectives, as is shown in [12] in the general logical setting. Discussion of such generalizations for categorial type logics can be found in [9] and [29]. In the context of our Kripke style frame semantics, we now find \( n \)-ary products interpreted via \( n + 1 \)-ary accessibility relations. These products have a residual implication for each of their \( n \) factors. Let us write \( f_\ast(A_1, \ldots, A_n) \) for the product and \( f_{\backslash i}(A_1, \ldots, A_n) \) for the \( i \)-th place residual. And define \( R^{-1}y_1, \ldots, \ldots, y_n \) iff \( Rxy_1, \ldots, y_n \) to facilitate the statement of the interpretation clauses. We require the valuation for the \( n \)-ary families to exhibit the familiar pattern: existential closure of a conjunctive statement for the product, universal closure of disjunctions for the residual implications.

\[
v(f_\ast(A_1, \ldots, A_n)) = \{x | \exists y_1 \ldots \exists y_n (Rxy_1 \ldots y_n \in v(A_1) \& \ldots \& y_n \in v(A_n) \})
\]

\[
v(f_{\backslash i}(A_1, \ldots, A_n)) = \{x | \forall y_1 \ldots \forall y_n ((R^{-1}x)y_1 \ldots y_n \in v(A_1) \& \ldots \& y_j \notin v(A_1)) \Rightarrow y_i \in v(A_i) \})
\]

Given such an interpretation for the compound formulae, the residuation laws are realized in the form shown below.

\[
f_\ast(A_1, \ldots, A_n) \rightarrow B \iff A_i \rightarrow f_{\backslash i}(A_1, \ldots, A_{i-1}, B, A_{i+1}, \ldots, A_n)
\]

**UNARY RESIDUATED PAIRS.** Let us focus now on the case of unary connectives. Consider a residuated pair of connectives \( \Diamond, \square^i \) for which the defining residuation inference \( fx \leq y \) iff \( x \leq gy \) takes the form (\( \dagger \)), given the interpretation clauses shown below.

(\( \dagger \)) \( \Diamond A \rightarrow B \iff A \rightarrow \square^i B \)

\[
v(\Diamond A) = \{x | \exists y (Rxy \& y \in v(A))\}
\]

\[
v(\square^i A) = \{x | \forall y (Ryx \Rightarrow y \in v(A))\}
\]

The valuation for the \( \Diamond, \square^i \) formulae has the required properties for residuation to arise: existential closure of a conjunctive statement for \( \Diamond \), universal closure of a disjunction for the residual \( \square^i \). Note carefully that the interpretation of \( \Diamond \) and \( \square^i \) moves you in opposite directions along the \( R^2 \) accessibility relation. The downnarrow on the universal operator is there to remind you of this fact.

**Kripke Graphs.** A picture may clarify the relation between the unary and the binary residuated pairs of connectives. In the case of \( \bullet \) we make an existential
move along the branching accessibility relation \( R^3 \). In the case of \( \diamond \) we make an existential move in the same direction, this time for a non-branching accessibility relation. In both cases, universal moves in the opposite direction bring you back to the point of origin.

\[
\begin{align*}
A \to C/B \iff \ &A \star B \to C \\
A \star B \to C \iff B \to A \setminus C
\end{align*}
\]

\( \diamond A \to B \iff A \to \square^1 B \)

How shall we interpret \( R^2 \) in this case? In the context of temporal logic, \( \diamond \) and \( \square^1 \) would express future possibility and past necessity. In the context of grammar logic, the accessibility relations model linguistic composition. In the ternary case, \( Rxyz \) holds if we can put together the resources \( y \) and \( z \) into \( x \) — or, from the other perspective, decompose \( x \) into the parts \( y \) and \( z \). In the binary case \( Rxy \), a similar part-whole relation holds between \( y \) and \( x \), but this time \( x \) is obtained not by putting together two independent linguistic resources, but rather by augmenting \( y \) with some fixed piece of information, an information change expressed by the move along the \( R^2 \) relation.\(^2\)

**Axiomatization: Lambek Style.** We now put together the binary and the unary families of connectives and consider the mixed language

\[
\mathcal{F} := \mathcal{A} \mid \mathcal{F} \setminus \mathcal{F} \setminus \mathcal{F} \setminus \mathcal{F} \setminus \diamond \mathcal{F} \setminus \square^1 \mathcal{F}
\]

Axiomatic presentation of the pure logic of resduation for the extended language is given below. As in the previous section, we decorate the arrows \( A \to B \) with combinator proof terms so that we can refer to deductions by means of their combinator.

\[
\begin{align*}
\text{id}_A : A & \to A \\
f : A \to B & \quad g : B \to C \\
\frac{g \circ f : A \to C}{f : \diamond A \to B} \\
\mu(f) : A \to \square^1 B
\end{align*}
\]

\[
\begin{align*}
\beta(f) : A \to C/B \\
g : A \to C/B \\
\frac{\beta^{-1}(g) : A \star B \to C}{f : A \star B \to C}
\end{align*}
\]

\[
\begin{align*}
\gamma(f) : B \to A \setminus C \\
g : B \to A \setminus C \\
\frac{\gamma^{-1}(g) : A \star B \to C}{f : A \star B \to C}
\end{align*}
\]

\(^2\)For an ‘additive’ alternative to our ‘multiplicative’ view on unary operators, see [39, 38].
AXIOMATISATION: DOŠEN STYLE. Below a deductive presentation based on the alternative way of characterizing a pair of residual operations \( f, g \) in terms of Isotonicity (*) and the inequalities (**) for the compositions \( fg, gf \).

(*) \( x \leq y \Rightarrow fx \leq fy, gx \leq gy \)  
(**) \( fgx \leq x, \quad x \leq gfz \)

In this presentation, the unit, co-unit combinators are primitive type transitions, recursively generalized via the Isotonicity rules of inference (Antitonicity for the negative subtype of implications \(/, \cdot, \backslash\)).

\[
\begin{align*}
\text{id}_A &: A \to A & f &: A \to B & g &: B \to C \\
& \quad g \circ f &: A \to C \\
\text{unit}_\Delta &: \Box \Box^\perp A \to A & \text{co-unit}_\Delta &: A \to \Box \Box^\perp A \\
\text{unit}_\cdot &: A/B \cdot B \to A & \text{co-unit}_\cdot &: A \to (A \cdot B)/B \\
\text{unit}_\backslash &: B \cdot (B \backslash A) \to A & \text{co-unit}_\backslash &: A \to B/(B \cdot A) \\
\end{align*}
\]

\[
\begin{align*}
\frac{f &: A \to B}{(f)^\circ &: \Box A \to \Box B} & \quad \frac{f &: A \to B}{(f)^\odot &: \Box \Box^\perp A \to \Box \Box^\perp B} \\
\frac{f &: A \to B \quad g &: C \to D}{f \cdot g &: A \cdot C \to B \cdot D} \\
\frac{f &: A \to B \quad g &: C \to D}{f / g &: A/D \to B/C} & \quad \frac{f &: A \to B \quad g &: C \to D}{g \circ f &: D/A \to C/B} \\
\end{align*}
\]

EQUIVALENCE OF THE DEDUCTIVE PRESENTATIONS. For the \(/, \cdot, \backslash\) fragment, we know the two deductive presentations are equivalent, cf. Lambek [26] for one direction, Došen [11] for the other. We take the Lambek presentation as our starting point here, and show for the extended system how from \( \mu, \mu^{-1} \) we obtain the alternative axiomatisation in terms of Isotonicity and the inequalities for the compositions \( \Box \Box^\perp \) and \( \Box \Box^\perp \) (Term decoration for the right column left to the reader.)

\[
\begin{align*}
\text{id}_{\Box A} &: \Box A \to \Box A & \Box A \to \Box A \\
\mu^{-1}(\text{id}_{\Box A}) &: \Box \Box^\perp A \to A & A \to \Box \Box^\perp A \\
\mu(\text{id}_{\Box B}) &: \Box B \to \Box B & \Box \Box^\perp A \to \Box \Box^\perp A \\
\mu^{-1}(\mu(\text{id}_{\Box B}) \circ f) &: A \to \Box \Box^\perp B & \Box \Box^\perp A \to A \quad A \to B \\
\end{align*}
\]

SOUNDNESS, COMPLETENESS. For ternary frame semantics for the \( \mathcal{F}(/, \cdot, \backslash) \) fragment, Došen [10] proves soundness and completeness on the basis of a canonical model construction. Došen’s results generalize unproblematically to the \( \Box, \Box^\perp \) extended language. We now consider mixed frames \( (W, R^2, R^3) \) with \( W \) the set of linguistic resources as before, and \( R^2, R^3 \) arbitrary binary and ternary relations on \( W \). We have to show that \( \vdash A \to B \) if, for every valuation \( v \) on every frame, \( v(A) \subseteq v(B) \).

(\( \Rightarrow \)) Induction on the length of proofs of \( A \to B \). (\( \Leftarrow \)) Extend Došen’s canonical model construction for the \( R^2 \) relation as follows. For the canonical frame, let \( W \) be the formulae of \( \mathcal{F}(/, \cdot, \backslash, \Box, \Box^\perp) \). In the canonical frame, we define the accessibility relations \( R^2 \) and \( R^3 \) as follows:

\[
R^3(C, A, B) \iff \vdash C \to A \cdot B \quad R^2(A, B) \iff \vdash A \to \Box B
\]
Define the canonical valuation as \( v(A) = \{ B \mid \vdash B \rightarrow A \} \). Now suppose \( v(A) \subseteq v(B) \) but \( \not\vdash A \rightarrow B \). If \( \not\vdash A \rightarrow B \) with the canonical valuation on the canonical frame, \( A \in v(A) \) but \( A \not\in v(B) \) so \( v(A) \not\subseteq v(B) \). Contradiction.

**Canonical model: compound formulae.** We have to check the canonical model construction for the new compound formulae \( \Diamond A, \Box^1 A \). Below the direction that requires a little thinking.

(\( \Diamond \)) Assume \( A \in v(\Diamond B) \). We have to show \( \vdash A \rightarrow \Diamond B \). \( A \in v(\Diamond B) \) implies \( \exists A' \) such that \( R^2 AA' \) and \( A' \in v(B) \). By inductive hypothesis, \( \vdash A' \rightarrow B \). By isotonicity for \( \Diamond \) this implies \( \vdash \Diamond A' \rightarrow B \). We have \( \vdash A \rightarrow \Diamond A' \) by (Def R^2) in the canonical frame. By Transitivity, \( \vdash A \rightarrow \Diamond B \).

(\( \Box^1 \)) Assume \( A \in v(\Box^1 B) \). We have to show \( \vdash A \rightarrow \Box^1 B \). \( A \in v(\Box^1 B) \) implies that \( \forall A' \) such that \( R^2 AA' \) we have \( A' \in v(B) \). Let \( A' \) be \( \Diamond A \). \( R^2 AA' \) holds in the canonical frame since \( \vdash \Diamond A \rightarrow \Diamond A \). By inductive hypothesis we have \( \vdash A' \rightarrow B \), i.e. \( \vdash \Diamond A \rightarrow B \). By Residuation this gives \( \vdash A \rightarrow \Box^1 B \).

**Logical versus structural connectives.** Following the agenda set out in §1 for the binary connectives, we now introduce a Gentzen presentation, and show that it is equivalent to the deductive presentation. For the Gentzen presentation we shall prove Cut Elimination, with its pleasant corollaries of Decidability and the Subformula property.

In order to present Gentzen calculus for the extended type language, we need an \( n \)-ary structural operator for every family of \( n \)-ary logical operators: binary \( (\cdot, \cdot) \) for the family \( /, \cdot \), and unary \( (\cdot) \) for the family \( \Diamond, \Box^1 \). Corresponding to the formula language \( \mathcal{F} \) we have a language of terms \( \mathcal{T} \) (structured configurations of formulæ).

\[
\mathcal{F} ::= A | \mathcal{F}/\mathcal{F} | \mathcal{F} \cdot \mathcal{F} | \mathcal{F} \setminus \mathcal{F} | \Diamond \mathcal{F} | \Box^1 \mathcal{F}
\]

\[
\mathcal{T} ::= \mathcal{F} | (\mathcal{T}, \mathcal{T}) | (\mathcal{T})
\]

**Gentzen presentation.** As before, sequents are pairs \( (\Gamma, A) \), \( \Gamma \in \mathcal{T}, A \in \mathcal{F}, \) written \( \Gamma \Rightarrow A \). We have Belnap-style antecedent punctuation, with for \( \Diamond, \Box^1 \) the unary structural connective \( (\cdot) \) matching the unary logical connectives. Below the rules of use \( [\Diamond L], [\Box^1 L] \) and the rules of proof \( [\Diamond R], [\Box^1 R] \) for the new connectives. As we remarked above, \( \Diamond \) and \( \Box^1 \) can be seen as truncated forms of product and implication. It may be helpful to compare the \( \Diamond \) rules with the rules for \( \cdot \), and the \( \Box^1 \) rules with the rules for an implication, say \( / \).

Compare the \( \Diamond \) rules with the rules for binary product \( \cdot \):

\[
\frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{(\Gamma, \Delta) \Rightarrow A \cdot B} \quad \cdot R
\]

\[
\frac{\Gamma \Rightarrow A \quad \Gamma[A,B] \Rightarrow C}{\Gamma[A \cdot B] \Rightarrow C} \quad \cdot L
\]

Compare again the \( \Box^1 \) rules with the rules for binary implication \( / \):

\[
\frac{\Gamma \Rightarrow A \quad \Gamma[A,B] \Rightarrow B} {[\Gamma,B] \Rightarrow A} \quad / R
\]

\[
\frac{\Delta \Rightarrow B \quad \Gamma[A] \Rightarrow C}{\Gamma[(A/B), \Delta] \Rightarrow C} \quad / L
\]

**Extending the valuation to structured terms.** To obtain direct semantic interpretation for sequents \( \Gamma \Rightarrow A \) we can extend the valuation to antecedent terms.
Compare again the valuation for logical operators $\bullet, \odot$ with that of their structural counterparts $(\cdot, \cdot, \cdot)$

$$v(A \bullet B) = \{ z \mid \exists x \exists y [Rz, xy \land x \in v(A) \land y \in v(B)] \}$$

$$v((\Delta_1, \Delta_2)) = \{ z \mid \exists x \exists y [Rz, xy \land x \in v(\Delta_1) \land y \in v(\Delta_2)] \}$$

$$v(\odot A) = \{ x \mid \exists y [Rxy \land y \in v(A)] \}$$

$$v((\Delta)) = \{ x \mid \exists y [Rxy \land y \in v(\Delta)] \}$$

Truth and validity for sequents are then defined the usual way. We have completeness in case $\Gamma \vdash A$ iff for every valuation $v$ on every frame we have $v(\Gamma) \subseteq v(A)$.

**Equivalence of Gentzen and Combinator Presentations.** To compare the two presentations, we extend the formula representation $\Delta^o$ of a structural configuration $\Delta$ to the language $\mathcal{F}(\cdot, \cdot, \cdot, \odot, \square^l)$ in the obvious way: $(\Delta_1, \Delta_2)^o = \Delta_1^1 \bullet \Delta_2^2$, $(\Delta)^o = \odot \Delta^o$, $A^o = A$. The sequent presentation for the language $\mathcal{F}(\cdot, \cdot, \cdot, \odot, \square^l)$ can then be shown to be equivalent to the combinator axiomatisation in the sense that every combinator $f : A \rightarrow B$ gives a proof of $A \Rightarrow B$, and every proof of a sequent $\Gamma \Rightarrow B$ gives a combinator $f : \Gamma^o \rightarrow B$.

**From Combinators to Sequents.** To obtain the Gentzen rules $[\odot L]$, $[\odot R]$, $[\square^l L]$, $[\square^l R]$ from combinator deductions, we use Isotonicity of $\odot, \square^l$ in addition to the residuation inferences $\mu, \mu^{-1}$. Given the formula equivalent $\Gamma^o$ for sequent terms $\Gamma$, $\mu$ gives $[\square^l R]^o$, $[\odot L]^o$ makes premise and conclusion identical, and Isotonicity for $\odot$ gives $[\odot R]^o$. The only non-trivial case is $[\square^l L]$. Consider first the case where the context $\Gamma$ is empty. The combinator derivation of $[\square^l L]^o$ is given below.

$$\frac{f : A \rightarrow B}{(f)^{\square^l} : \square^l A \rightarrow \square^l B} \quad \mu((f)^{\square^l}) : \odot \square^l A \rightarrow B \quad \frac{A \Rightarrow B}{(\square^l A) \Rightarrow B} \Rightarrow \square^l L$$

Next the case where the context $\Gamma$ in the $[\square^l L]$ premise $\Gamma[A] \Rightarrow B$ is non-empty. Let $g$ be $\Gamma[A]^2 \Rightarrow B$. Let $\pi(g)$ be a sequence of $\mu, \beta, \gamma$ residuation inferences isolating $A$ on the left of the arrow. Then we obtain the formula equivalent of the conclusion of $[\square^l L]$ via the deduction $\pi^{-1}(\mu(\square^l (\pi(g))))$.

**From Sequents to Combinators.** To obtain the combinators $\text{id}, f \circ g$ (Transitivity), $\mu(f), \mu^{-1}(g)$ (Residuation) from sequent derivations, we use the Cut rule. Once we have established the equivalence of the combinator and the sequent presentation, we prove Cut Elimination for the latter. $[\text{Ax}]$ gives $f, \circ g$ is a special case of Cut. The crucial new cases $\mu(f), \mu^{-1}(g)$ follow.

$$\frac{A \Rightarrow A}{(A) \Rightarrow B} \odot R \quad \frac{f : \odot A \Rightarrow B}{(f) : A \Rightarrow B} \Rightarrow (\text{cut})$$

$$\frac{B \Rightarrow B}{(\square^l B) \Rightarrow B} \Rightarrow \square^l L \quad \mu((f)^{\square^l}) : A \Rightarrow \square^l B \Rightarrow \square^l R$$

**Cut Elimination: Principal Cuts.** We now extend the Cut Elimination result to the new connectives $\odot, \square^l$. We proceed by induction on the complexity of the cut formula, and distinguish principal cuts, where the cut formula is active in both cut premises, from permutation conversions, where this is not the case.

Below the new cases of principal cuts, with cut formula $\odot A$ and $\square^l A$. Replacement of a cut on $\odot A$ ($\square^l A$) by a cut on $A$ of smaller degree.
\[
\frac{\Delta \vdash A \quad \diamond R \quad \Gamma[(A)] \vdash B \quad \diamond L}{\Gamma[(\Delta)] \Rightarrow B} \quad \text{(cut)}
\]
\[
\frac{\Delta \Rightarrow \Delta \quad \square^1 R \quad \Gamma[(\square^1 A)] \vdash B \quad \square^1 L}{\Gamma[(\Delta)] \Rightarrow B} \quad \text{(cut)}
\]
\[
\frac{\Delta \Rightarrow A \quad \Gamma[(A)] \Rightarrow B}{\Gamma[(\Delta)] \Rightarrow B} \quad \text{(cut)}
\]

**Cut Elimination: Permutation Conversions.** The new cases where the active formula in the left or right premise is different from the Cut formula allow for the usual elimination strategy: permutation of the Cut rule and the logical rule. The Cut is moved upwards, and becomes of lower degree. Below the left premise antecedent cases for \( \diamond A \) and \( \square^1 A \).

\[
\frac{\Gamma[(A)] \Rightarrow B \quad \Gamma[\diamond A] \Rightarrow B \quad \Delta[B] \Rightarrow C}{\Delta[\Gamma[\diamond A]] \Rightarrow C} \quad \text{(cut)}
\]
\[
\frac{\Delta[A] \Rightarrow B \quad \Gamma[\square^1 A] \Rightarrow B \quad \Gamma[B] \Rightarrow C}{\Gamma[\Delta[\square^1 A]] \Rightarrow C} \quad \text{(cut)}
\]

**Permutation Conversion: Right Premise Antecedent.** Active type \( \diamond A \) or \( \square^1 A \) in the antecedent of the right Cut premise. Notation: \( \Gamma[\Delta_1, \Delta_2] \) for a structure \( \Gamma \) with substructures \( \Delta_1, \Delta_2 \), not necessarily sisters.

\[
\frac{\Delta \Rightarrow A \quad \Gamma[(A, B)] \Rightarrow C \quad \diamond L}{\Gamma[\Delta, \diamond B] \Rightarrow C} \quad \text{(cut)}
\]
\[
\frac{\Delta \Rightarrow A \quad \Gamma[(A, B)] \Rightarrow C \quad \square^1 L}{\Gamma[\Delta, (\square^1 B)] \Rightarrow C} \quad \text{(cut)}
\]

**Permutation Conversion: Right Premise Sucessed.** Active type \( \diamond A \) or \( \square^1 A \) in the successed of the right Cut premise.

\[
\frac{\Gamma \Rightarrow A \quad (\Delta[A]) \Rightarrow B \quad \square^1 R}{\Delta[\Gamma] \Rightarrow \square^1 B} \quad \text{(cut)}
\]
\[
\frac{\Gamma \Rightarrow A \quad (\Delta[A]) \Rightarrow B \quad \diamond R}{\Gamma[\Delta] \Rightarrow \diamond B} \quad \text{(cut)}
\]

**Illustration: Residuation Laws.** As an example, we check the compositions \( \diamond \square^1 \) and \( \square^1 \diamond \) (cf Application, \( \alpha \leq x \), Co-Application, \( x \leq \alpha \)). Below their cut-free Gentzen derivations.
$\frac{A \Rightarrow A}{\Diamond L} \quad \frac{A \Rightarrow A}{\Diamond R}
\frac{(\Diamond A) \Rightarrow A}{\Diamond L} \quad \frac{(A) \Rightarrow \Diamond A}{\Diamond R}
\frac{\Diamond \Diamond A \Rightarrow A}{\Diamond L} \quad \frac{A \Rightarrow \Diamond \Diamond A}{\Diamond R}$

**Structural Postulates.** What we have discussed so far is the pure logic of residuation for the unary family $\Diamond, \Diamond ^1$. By imposing conditions $\text{ASS}, \text{COMM}$ or their combination on ternary frames, we generate the landscape $\text{NL}, \text{L}, \text{NLP}, \text{LP}$ with completeness results for the relevant classes of frames (Došen). Along the same lines, we can develop the substructural landscape for the unary family $\Diamond, \Diamond ^1$ and its binary accessibility relation $R^2$, and for the mixed $R^2, R^3$ system.\(^3\)

The following structural postulates constrain $R^2$ to be transitive (4), or reflexive (T). Communication between $R^2$ and $R^3$ can be established via the strong distributivity postulate $K$, which distributes unary $\Diamond$ over both components of a binary $\star$, or, in a more constrained way, via the weak distributivity postulates $K1, K2$, where $\Diamond$ selects the left or right subtype of a product.

$$
4: \quad \Diamond \Diamond \Diamond A \rightarrow \Diamond A

T: \quad \Diamond (A \star B) \rightarrow \Diamond A \star B

K1: \quad \Diamond (A \star B) \rightarrow \Diamond A \star B

K2: \quad \Diamond (A \star B) \rightarrow A \star \Diamond B

K: \quad \Diamond (A \star B) \rightarrow \Diamond (A \star \Diamond B)
$$

Below the corresponding frame conditions ($\forall x, y, z, w \in W$).

$$
4: \quad (R_{xy} \& R_{yz}) \Rightarrow R_{xz}

T: \quad R_{xz}

K(1,2): \quad (R_{wx} \& R_{xyz}) \Rightarrow \exists z'(R_{y'y} \& R_{wyz'}) \lor \exists z'(R_{z'z} \& R_{wyz'})

K: \quad (R_{wx} \& R_{xyz}) \Rightarrow \exists y' \exists z'(R_{y'y} \& R_{z'z} \& R_{wyz'})
$$

The $K$ condition is the correlation postulate for Relevance logic from Routley and Meyer [35]. See Kurtonina [24] for discussion in the context of logics of linguistic resources. The weak distributivity principles $K1, K2$ play an important role in the applications discussed later in this paper.

**Structural Rules.** The structural rules below translate the postulates $T, 4, K1, K2, K$ from the formula level to the term level. As before, we prove equivalence between the rule and the postulate versions, and show that the Gentzen formulation allows cut-free proof search.

$$
\frac{\Gamma[(\Delta)] \Rightarrow A}{\Gamma[((\Delta))]} \Rightarrow A \quad 4 \quad \frac{\Gamma[(\Delta)] \Rightarrow A}{\Gamma[A]} \Rightarrow A \quad T
$$

$$
\frac{\Gamma[\{(\Delta_1), (\Delta_2)\}] \Rightarrow A}{\Gamma[((\Delta_1, \Delta_2))]} \Rightarrow A \quad K1
\frac{\Gamma[\{(\Delta_1), (\Delta_2)\}] \Rightarrow A}{\Gamma[\{(\Delta_1, \Delta_2)\}] \Rightarrow A} \quad K
\frac{\Gamma[\{(\Delta_1), (\Delta_2)\}] \Rightarrow A}{\Gamma[\{(\Delta_1, \Delta_2)\}] \Rightarrow A} \quad K2
$$

**Structural Rules from Structural Postulates.** We have to extend the equivalence between axiomatic and Gentzen style presentation to the structural postulates and rules. To obtain the sequent rules $T, 4, K$ from combinator deductions, it is enough to consider the case where the context $\Gamma$ is empty, as we have seen

\(^3\)For a related decomposition of the $S4 \star$ modality of Linear Logic, see [6, 18].
above. The following deductions give the formula equivalent of the structural rules $T, 4, K$. We leave $K1, K2$ to the reader.

$$
\begin{align*}
4: \ Diamond \Delta^o & \rightarrow \Diamond \Delta^o \quad f: \Diamond \Delta^o \rightarrow A \\
\therefore \ (\Diamond \Delta^o) & \Rightarrow A
\end{align*}
$$

$$
\begin{align*}
\therefore \ (\Diamond \Delta^o) & \Rightarrow A
\end{align*}
$$

$$
\begin{align*}
T: \ Diamond \Delta^o & \rightarrow \Diamond \Delta^o \quad f: \Diamond \Delta^o \rightarrow A \\
\therefore \ (\Diamond \Delta^o) & \Rightarrow A
\end{align*}
$$

$$
\begin{align*}
\therefore \ (\Diamond \Delta^o) & \Rightarrow A
\end{align*}
$$

$$
\begin{align*}
K: \Diamond (\Delta_1 \bullet \Delta_2) & \rightarrow \Diamond \Delta_1 \bullet \Diamond \Delta_2 \\
f \circ K: \Diamond (\Delta_1 \bullet \Delta_2) & \rightarrow A \\
\therefore \ ((\Delta_1, \Delta_2))^o & \Rightarrow A
\end{align*}
$$

$$
\begin{align*}
\therefore \ ((\Delta_1, \Delta_2))^o & \Rightarrow A
\end{align*}
$$

**Structural postulates from structural rules.** Derivation of the structural postulates via Gentzen proofs is straightforward.

$$
\begin{align*}
A \Rightarrow A & \quad \bullet R \\
(A \Rightarrow \Diamond A) \Rightarrow \Diamond A & \quad \Diamond L
\end{align*}
$$

$$
\begin{align*}
(A) \Rightarrow \Diamond A & \quad \Diamond L
\end{align*}
$$

**Cut elimination: structural rules.** We extend the cut elimination algorithm to logics with a structural rule package from $\mathcal{P}(\{T, 4, K\})$. Recall that in the case of connectives the proof of the Cut Elimination theorem is by induction on the complexity of Cut inferences, measured in terms of the number of connectives in the cut formula. The structural rules do not involve decomposition of formulae, so we need an additional complexity measure here.

Following [11, 4], let the trace of a cut formula $A$ be the sum of the lengths of the paths in the derivations of the cut premises connecting the two occurrences of $A$ with the point of their first introduction in the proof. The cut elimination steps involving structural rules now assimilate to the permutation cases: if a structural rule feeds the cut inference, we can interchange the order of application of the cut and the structural rule, leading to a situation with decreased trace, as the inductive hypothesis requires. Two examples are given below.

$$
\begin{align*}
\Delta[(\Delta')] & \Rightarrow A \\
\Gamma[\Delta\Delta'] & \Rightarrow B \\
\Gamma[\Delta] & \Rightarrow B \\
\therefore \Delta & \Rightarrow A \\
\Gamma[(\Gamma'[\Delta])] & \Rightarrow B \\
\Gamma[(\Gamma'[\Delta])] & \Rightarrow B
\end{align*}
$$

$$
\begin{align*}
\therefore \Delta & \Rightarrow A \\
\Gamma[\Delta] & \Rightarrow B \\
\Gamma[\Delta] & \Rightarrow B \\
\therefore \Delta & \Rightarrow A \\
\Gamma[(\Gamma'[\Delta])] & \Rightarrow B \\
\Gamma[(\Gamma'[\Delta])] & \Rightarrow B
\end{align*}
$$

**Structural postulates: universal variant.** In our discussion of structural postulates for $\bullet$, we have seen that we can express Associativity, Commutativity either via $\bullet$ postulate, or via implicational postulates, if we prefer to keep the language product-free. In a similar vein we could have presented $T, 4, K$ in their $\square^1$ forms:

$$
\begin{align*}
4\square^1: & \quad \square^1 A \rightarrow \square^1 \square^1 A \\
T\square^1: & \quad \square^1 A \rightarrow A \\
K\square^1/ : & \quad \square^1 (A/B) \rightarrow \square^1 A / \square^1 B \\
K\square^1\setminus: & \quad \square^1 (B/A) \rightarrow \square^1 B / \square^1 A
\end{align*}
$$
Below an illustration for the derivation of the universal variant $K\Box^I$.

\[
\frac{B \Rightarrow B \quad A \Rightarrow A}{(A/B, B) \Rightarrow A} \quad \frac{\Box^L A, \Box^L B}{/L}
\]

\[
\frac{((\Box^I A/B), (\Box^I B)) \Rightarrow A}{(\Box^I (A/B), \Box^I B) \Rightarrow A} \quad \frac{\Box^I A}{/R}
\]

\[
\frac{\Box^I (A/B) \Rightarrow \Box^I A/\Box^I B}{K}
\]

**S4: Compilation of structural rules.** We saw above that in the presence of Associativity for $\bullet$, we have a sugared Gentzen presentation where the structural rule is compiled away, and the binary sequent punctuation ($\cdot, \cdot$) omitted. Analogously, for $\Box^I$ with the combination $KT4$ (i.e. $S4$), we have a sugared version of the Gentzen calculus, where the $KT4$ structural rules are compiled away, so that the unary ($\cdot$) punctuation can be omitted. Compare the following. (Notation $\Gamma$ for a term $\Gamma$ of which the (pre)terminal subterms are of the form $\Box^I A$. The 4(cut) step is a series of replacements of terminal $\Box^I A$ by $\Box^I A$ via cuts depending on 4.)

\[
\frac{\Gamma [A] \Rightarrow B}{\Gamma [\Box^I A] \Rightarrow B} \quad \frac{\Box^I L}{/L}
\]

\[
\frac{\Gamma [\Box^I A] \Rightarrow B}{\Gamma [\Box^I A] \Rightarrow B} \quad \sim \quad \frac{\Gamma [A] \Rightarrow B}{\Gamma [\Box^I A] \Rightarrow B} \quad \frac{\Box^I L (S4)}{
}\]

\[
\frac{\Box^I \Gamma \Rightarrow A}{(\Box^I \Box^I) \Gamma \Rightarrow A} \quad \frac{\Box^I L}{/L}
\]

\[
\frac{\Box^I \Gamma \Rightarrow A}{(\Box^I \Box^I) \Gamma \Rightarrow A} \quad \frac{K}{/R}
\]

\[
\frac{\Box^I \Gamma \Rightarrow \Box^I A}{\Box^I \Gamma \Rightarrow \Box^I A} \quad \sim \quad \frac{\Box^I \Gamma \Rightarrow A}{\Box^I \Gamma \Rightarrow \Box^I A} \quad \frac{\Box^I R (S4)}{
}\]

In the sugared version, we recognize the rules of use and proof for the domain modalities of [20, 31].

**Multimodal Generalization.** We have presented the landscape of unary residuated operators from the perspective of one pair of connectives $\Diamond, \Box^I$. The move to a multimodal system where different families of unary operators live together and communicate (with families of the same or of different arity) via inclusion and interaction postulates is entirely straightforward. Below we give the mode restricted forms of the interpretation clauses, the residuation inferences, and the Gentzen rules.

\[
v(\Diamond A) = \{ x \mid \exists y (R_{xy} \land y \in v(A)) \}
\]

\[
v(\Box^I A) = \{ z \mid \forall y (R_{yz} \Rightarrow y \in v(A)) \}
\]

\[
\Diamond A \rightarrow B \quad \text{iff} \quad A \rightarrow \Box^I B
\]

\[
\frac{\Gamma \Rightarrow A}{[R \Diamond]} \quad \frac{\Gamma \Rightarrow \Box^I A}{[R \Box^I]}
\]

\[
\frac{\Gamma (A !) \Rightarrow B}{\Gamma \Rightarrow \Box^I A} \quad \frac{\Gamma [A] \Rightarrow B}{\Gamma [\Box^I A] \Rightarrow B} \quad \frac{[L \Diamond_i]}{
}\]

\[
\frac{\Gamma (\Box^I A !) \Rightarrow B}{\Gamma \Rightarrow \Box^I A} \quad \frac{\Gamma [A] \Rightarrow B}{\Gamma [\Box^I A] \Rightarrow B} \quad \frac{[L \Box^I_i]}{
}\]

**Discussion.** There is a wide range of linguistic applications for the modal operators $\Diamond, \Box^I$. In the next section, we work out one surprising application from
the field of algorithmic proof theory, showing how one can enforce a uniform head-driven search regime for \(L\) on the basis of a modal decoration of \(L\) sequents. But let us first briefly indicate where the various unary operators that have been proposed in the literature can be situated within the general landscape developed here.

At one end of the spectrum, the proposals that come closest to the pure logic of residuation for \(\Diamond, \Box^1\) are Morrill's bracket operators. In one of their incarnations (the version of [32]) these operators are presented with the \([\Diamond L]_1, [\Diamond R]\) and \([\Box^1 R]\) rules we have given above. The \([\Box^1 L]\) rule of [32], however, is inappropriate for the pure residuation system: it derives the non-theorem \(\Box^1 \Diamond A \Rightarrow A\), next to the theorem \(A \Rightarrow \Box^1 \Diamond A\). On the semantic level, Morrill assumes the bracket operators to be interpreted in terms of a functional accessibility relation \(R^2\) — an interpretation which imposes constraints on the allowable models which we have not assumed in our presentation. The linguistic applications of the bracket operators as markers of locality domains can be recast straightforwardly in terms of the more discriminating pure residuation logic for \(\Diamond, \Box^1\) where no functionality constraints are imposed on \(R^2\).

At the other end of the spectrum, we find the S4 domain modality of [20, 31], a universal modality \(\Box\) which assumes the full set of postulates \(K\, T\, 4\). Adding modally controlled structural rules, we obtain the structural modalities of [2] and others. A crucial feature of these operators is the Reflexivity Postulate \(\Box A \rightarrow A\): a modally marked resource \(\Box A\) will at a certain point in the derivation be used as an ordinary datum of type \(A\). Recall that we have presented the binary accessibility relation as a form of linguistic composition: \(R_{xy}\) holds in case \(x\) is the sign one obtains by augmenting \(y\) with the information added in the \(R^2\) move. From this perspective, Reflexivity is an undesirable property, trivializing the \(R^2\) augmentation. In the framework presented here, where we consider a residuated pair of modalities \(\Diamond, \Box^1\) rather than a single modal operator \(\Box\), we can capture the proof-theoretic behaviour of the S4 structural modalities without making Reflexivity (or Transitivity) assumptions about the \(R^2\) accessibility relation. With a translation \(\Box A^j = \Diamond \Box^1 (A)^j\), the \(T\) and \(4\) postulates for \(\Box\) become valid type transitions in the pure residuation system for \(\Diamond, \Box^1\), as the reader can check.

\[
T: \Box A \rightarrow A \quad \Diamond \Box^1 A \rightarrow A
\]

\[
4: \Box A \rightarrow \Box \Box A \quad \Diamond \Box^1 A \rightarrow \Diamond \Box^1 \Box A
\]

One can add (a two-sided version of) the distributivity principle \(K\), if desired, and modally controlled structural rules, e.g. Permutation in the example below, with the S4 rule on the left and the \(\Diamond, \Box^1\) version on the right.

\[
\Gamma[\{\Delta_2, \Delta_1\}] \Rightarrow A \quad \Gamma[\{\Delta_1, \Delta_2\}] \Rightarrow A \quad (\Delta_2 = \Box \Delta') \quad \Gamma[\{(\Delta_2)^*, \Delta_1\}] \Rightarrow A \quad \Gamma[\{(\Delta_1, (\Delta_2)^*)\}] \Rightarrow A
\]

### 3 Modalities as procedural control features

In the literature on automated deduction, it is well known that cut-free Gentzen proof search is still suboptimal from the efficiency perspective: there may be different (cut-free) derivations leading to one and the same proof term. Restricting ourselves to the implicational fragment, the spurious non-determinism in the search space has two causes ([40]): (i) permutability of \([L]\) and \([R]\) inferences, and (ii) permutability of \([L]\) inferences among themselves, i.e. non-determinism in the choice of the active formula in the antecedent. A so-called goal directed (or: uniform) search regime performs the non-branching \([R]\) inferences before the \([L]\) inferences (re (i)), whereas head driven search commits the choice of the antecedent active formula in terms of the goal formula (re (ii)).
In the context of categorial theorem proving, a goal-directed head-driven regime for product-free $L$ has been proposed in Hepple [20] with a proof of the safeness (no proof terms are lost) and non-redundancy (each proof term has a unique derivation). We present the Hepple regime in the format of Hendriks [19] where the reader can find a detailed discussion of the spurious ambiguity problem in the context of Gentzen proof search. (Our labeling for the $^{*\cdot\cdot}$ version of the Axiom sequent, $^{*\cdot\cdot\cdot}L$, is suggestive for the type of modal control we are about to propose.)

$$
\frac{x : p \Rightarrow z : p}{\Gamma, u : B^*, \Gamma' \Rightarrow t : p} \quad \text{[Ax/$^L\cdot$]} \quad \frac{\Gamma, u : B^*, \Gamma' \Rightarrow t : p}{\Gamma, u : B, \Gamma' \Rightarrow t : p^*} \quad \text{[R]}$
$$

$$
\frac{\Delta, x : B \Rightarrow t : A^*}{\Delta \Rightarrow \lambda x.t : A/B^*} \quad \frac{\Delta \Rightarrow u : B^*}{\Delta, s : A/B^*, \Lambda, \Gamma' \Rightarrow t[x/su] : C} \quad \text{[L]}$
$$

$$
\frac{z : B, \Delta \Rightarrow t : A^*}{\Delta \Rightarrow \lambda z.t : B\backslash A^*} \quad \frac{\Delta \Rightarrow u : B^*}{\Gamma, \Delta, s : B\backslash A^*, \Gamma' \Rightarrow t[x/su] : C} \quad \text{[L]}$
$$

COMMENTS. The $L^*$ calculus eliminates the spurious non-determinism of the original presentation $L$ by annotating sequents with a procedural control operator $^{*\cdot\cdot}$. Goal sequents $\Gamma \Rightarrow t : A$ in $L$ are replaced by $L^*$ goal sequents $\Gamma \Rightarrow t : A^*$. With respect to the first cause of spurious ambiguity (permutability of $[L]$ and $[R]$ inferences), the control part of the $[R]$ inferences forces one to remove all connectives from the succedent until one reaches an atomic succedent. At that point, the $^{*\cdot\cdot}$ control is transmitted from succedent to antecedent: the $^{*\cdot\cdot\cdot}[R]$ selects an active antecedent formula the head of which ultimately, by force of the control version of the Axiom sequent $^{*\cdot\cdot\cdot}L$, will have to match the (now atomic) goal type. The $[L]$ implication inferences initiate a $^{*\cdot\cdot}$ control derivation on the minor premise, and transmit the $^{*\cdot\cdot\cdot}$ active declaration from conclusion to major (right) premise. The effect of the flow of control information is to commit the search to the target type selected in the $^{*\cdot\cdot\cdot}[R]$ step. This removes the second source of spurious ambiguity: permutability of $[L]$ inferences. It can be shown that the $L^*$ regime eliminates spurious ambiguity. Syntactically, derivability in $L$ and $L^*$ coincide. Semantically, the set of $L^*$ proof terms forms a subset of the set of $L$ terms. But, modulo logical equivalence, no readings are lost moving from $L$ to $L^*$. Moreover, the $L^*$ system has the desired one-to-one correspondence between readings and proofs.

UNIFORM PROOF SEARCH: MODAL CONTROL. In this section we show how to enforce the Hepple-Hendriks uniform head-driven search regime via a modal translation. The basic idea is to use the logical properties of the connectives $\circ, \Box$ to capture the effects of the $^{*\cdot\cdot}$ procedural control marking in $L^*$. We use the base residuation logic for $\circ, \Box$, plus weak distributivity principles $K1, K2$ for the interaction between the unary and the binary families. For convenience we repeat the frame conditions and the Gentzen transformation of the $K1, K2$ structural postulates from our discussion above.

STRUCTURAL POSTULATE  FRAME CONDITION

$$
K1 : \quad \circ(A \bullet B) \rightarrow \circ A \bullet B \\
K2 : \quad \circ(A \bullet B) \rightarrow A \bullet \circ B \\
(\text{Ruzx} & \text{Ruzx}) \Rightarrow \exists y'(\text{Ruzy} & \text{Ruzy})' \\
(\text{Ruzx} & \text{Ruzx}) \Rightarrow \exists z'(\text{Ruzz} & \text{Ruzx})'
$$

In structural rule format these take the following form.

$$
\frac{\Gamma[((\Delta_1)^*, \Delta_2)] \Rightarrow A}{\Gamma[((\Delta_1, (\Delta_2)^*))^*] \Rightarrow A} \quad \frac{\Gamma[((\Delta_1, \Delta_2))] \Rightarrow A}{\Gamma[((\Delta_1, (\Delta_2)^*))^*] \Rightarrow A} \quad K1 \quad K2
$$
To establish the equivalence with $L^*$ search, we can use the sugared presentation of $L$ where Associativity is compiled away so that binary punctuation $(\cdot, \cdot)$ can be omitted (but not the unary $(\cdot^0)$). This gives the following compiled format for $K1, K2$:

$$\Gamma, (A)^o, \Gamma' \Rightarrow B$$

$$\Gamma, (A, \Gamma')^o \Rightarrow B \quad K'$$

**Translation: Formulae, Sequents.** We define the translation mapping first on the formula level, and then extend it to the level of $L^*$ sequents, where we have to distinguish marked and unmarked formulae. On the formula level, define mappings $(\cdot)^1, (\cdot)^0 : F(\bullet, \diamond) \mapsto F(\bullet, \diamond, \Diamond, \Box, \Box^1)$, for antecedent and succedent formula occurrences respectively.

$$(p)^1 = p \quad (p)^0 = \Box^1 p$$

$$(A/B)^1 = (A)^1 / (B)^0 \quad (A/B)^0 = (A)^0 / \Box^1 (B)^1$$

$$(B \backslash A)^1 = (B)^0 \backslash (A)^1 \quad (B \backslash A)^0 = \Box^1 (B)^1 \backslash (A)^0$$

The formulae of a sequent $\Gamma \Rightarrow A$ in $L^*$ are partitioned by the $^{*'}$ annotation in a set of marked formulae — a singleton, since there is only one $^{*'}$ per sequent — and a set of unmarked formulae. We extend the translation mapping taking this difference into account. The antecedent and succedent translation functions $(\cdot)^1, (\cdot)^0$ below are defined in terms of $(\cdot)^1, (\cdot)^0$, but they act in a different way on marked and on unmarked formulae.

$$(A_1, \ldots, A_n)^1 = \overline{A_1}, \ldots, \overline{A_n} \quad \text{where} \quad \overline{A} = \begin{cases} (A)^1 & \text{if } A \text{ is } ^{*'} \text{ marked} \\ \Box^1 (A)^1 & \text{otherwise} \end{cases}$$

$$(A)^0 = \begin{cases} (A)^0 & \text{if } A \text{ is } ^{*'} \text{ marked} \\ A & \text{otherwise} \end{cases}$$

We now have the following proposition.

$$L^* \vdash \Gamma \Rightarrow A^* \quad \text{iff} \quad L \Diamond K' \vdash (\Gamma)^1 \Rightarrow (A)^0$$

**Equivalence of $L^*$ and $L \Diamond K'$.** The ($\Rightarrow$) direction of the equivalence can be proved by straightforward induction on the length of derivations in $L^*$. For the more delicate ($\Leftarrow$) direction, we have to show that $L \Diamond K'$ does not derive more than $L^*$. We give a case analysis of the choice points in the top-down (backward-chaining) unfolding of the search space, and show that $L \Diamond K'$ can make no moves that would lead the search out of the space defined by the translation mapping.

Below we juxtapose the $L^*$ rules and axiom and their counterpart in $L \Diamond K'$. We treat only one implication. For the $L \Diamond K'$ version, we interleave the proof unfolding with the evaluation of the translation mapping. As an auxiliary notion, we have functions $\text{ACTIVE}$ and $\text{LOCKED}$ which for a sequent return the set of formulae matching the input condition for a logical rule (ACTIVE), and those which no logical rule is applicable to (LOCKED).

Proof search starts with an $L^*$ goal sequent $\Gamma \Rightarrow (A)^*$. The goal type $A$ is either atomic or complex, the antecedent is of length 1 or greater than 1. Consider first the case of a complex goal type and $1 \leq |\Gamma|$. On the left the $L^*[R]$ rule, on the right the corresponding derivation in $L \Diamond K'$. Both $(\downarrow)$ and $(\downarrow)$ stand in a feeding relation with themselves. For the roots of the derivations, we have $\text{ACTIVE}(\downarrow) = \{(A/B)^0\}$, $\text{ACTIVE}(\downarrow) = \{(A/B)^0\}$; for the leaves, $\text{ACTIVE}(\downarrow) = \{A\}$, $\text{ACTIVE}(\downarrow) = \{(A)^0\}$. Note that all antecedent formulae in $(\downarrow)$ have main connective $\Box^1$ as a result of the
translation mapping. The $\Box^1$ connective acts as a lock: embedded connectives in these formulæ will only be accessible after the removal of $\Box^1$.

\[
\begin{array}{c}
\Gamma, B \Rightarrow (A)^* \\
\{1\} \ (\Gamma \Rightarrow (A/B)^*)^R \ \Box^1(\Gamma)^1, \Box^1(B)^1 \Rightarrow (A)^0 \\
\Box^1(\Gamma)^1 \Rightarrow (A)^0/\Box^1(B)^1 \ \{1\}^0 \ (\Gamma)^0 \\
\Box^1(\Gamma)^1 \Rightarrow (A/B)^0
\end{array}
\]

Consider now the case where the recursion on succedent implications bottoms out, i.e. where we reach the `*' marked atomic head of the goal formula. In $L^*$ the only applicable rule in this situation is $[\ast R]$ which transmits the `*' marking from succedent to antecedent. $[\ast R]$ is non-deterministic: any antecedent formula $B$ can receive the `*' marking. In $L \diamond K'$ the active atom is realized as $(p)^0 = \Box^1 p$. The only applicable rule here is $[\Box^1 R]$ which, by resudation, realizes $\Box^1$ as $(\cdot)^0$ on the antecedent. $[\Box^1 R]$ can only be followed by $[K']$, which non-deterministically pushes $(\cdot)^*$ to an arbitrary antecedent formula $\Box^1(\Gamma)^1$. At that point $[\Box^1 L]$ becomes applicable, which through the elimination of $\Box^1$ shifts $(\cdot)^1$ from LOCKED to ACTIVE.

\[
\begin{array}{c}
\Gamma, (B)^*, \Gamma' \Rightarrow p \ \{1\} \ (\Gamma \Rightarrow (p)^*)^R \\
\Box^1(\Gamma)^1, (\Box^1(B)^1)^0, (\Box^1(\Gamma)^1)^0 \Rightarrow p \ \Box^1 L \\
\Box^1(\Gamma)^1, (\Box^1(B)^1)^0, (\Box^1(\Gamma)^1)^0 \Rightarrow p \ \Box^1 K' \\
\Box^1(\Gamma)^1, (\Box^1(B)^1)^0, (\Box^1(\Gamma)^1)^0 \Rightarrow (\cdot)^0 \ \Box^1 P \\
\Box^1(\Gamma)^1, (\Box^1(B)^1)^0, (\Box^1(\Gamma)^1)^0 \Rightarrow (p)^0 \ \{1\}
\end{array}
\]

Next we analyse the possible antecedent configurations for the different choices of active formula. The active formula is either atomic or complex, and the context is either empty or non-empty. Let us consider these in turn, starting with the non-empty context case. If the active formula is atomic, the derivation fails, in $L^*$ and in $L \diamond K'$. If the active formula is complex (i.e. of the form $B/C$ or $C/B$), the only applicable rule, in $L^*$ and in $L \diamond K'$, is $[L] (\{1\} L)$. The derivation branches, initiating uniform head-driven search for the negative subtype of the goal formula in the left premise, and declaring the positive subtype active in the right premise. Roots and leaves of the derivations in $L^*$ and in $L \diamond K'$ agree on the ACTIVE, LOCKED partitioning.

\[
\begin{array}{c}
\Delta \Rightarrow (B)^* \ (\Gamma \Rightarrow (A)^*, \Gamma' \Rightarrow p) \ \{1\} \ (\Delta \Rightarrow (B)^*)^L \\
\Box^1(\Delta)^1 \Rightarrow (B)^0 \ \Box^1(\Gamma)^1, (A)^1, (\Box^1(\Gamma)^1)^0 \Rightarrow p \ \Box^1 L \\
\Box^1(\Gamma)^1, (A)^0/(B)^0, (\Box^1(\Delta)^1, (\Box^1(\Gamma)^1)^0 \Rightarrow p \ \Box^1 P \\
\Box^1(\Gamma)^1, (A/B)^1, (\Box^1(\Delta)^1, (\Box^1(\Gamma)^1)^0 \Rightarrow p
\end{array}
\]

Finally, consider the base cases of the recursion. Below the correspondence when the $L^*$ Axiom sequent, i.e. $[\ast L]$ is reached.

\[
\begin{array}{c}
(p)^* \Rightarrow p \ \ast L \\
(p)^* \Rightarrow p \ \ast L \\
\{1\} p \Rightarrow p \ \{1\}
\end{array}
\]

For the sake of completeness, one should add the case of the trivial initial sequent $p \Rightarrow (p)^*$, though the issue of spurious ambiguity hardly arises here. Below the $L^*$ form and its $L \diamond K'$ counterpart.

\[
\begin{array}{c}
(p)^* \Rightarrow p \ \ast L \\
\{1\} p \Rightarrow (p)^* \ \ast L \\
\{1\} p \Rightarrow (p)^* \ \ast L \\
\{1\} p \Rightarrow (p)^* \ \ast L
\end{array}
\]
ILLUSTRATION: GEACH. Without the constraint on uniform head-driven search, there are two L sequent derivations for the Geach transition. They produce the same proof term.

\[
\begin{align*}
  c & \Rightarrow c & b & \Rightarrow b & /L \\
  b/c, c & \Rightarrow b & a & \Rightarrow a & /L \\
  a/b, b/c, c & \Rightarrow a & /R \\
  a/b, b/c & \Rightarrow a/c & /R \\
  a/b & \Rightarrow (a/c)/(b/c) & /R \\
\end{align*}
\]

GEACH: UNIFORM HEAD-DRIVEN SEARCH. Of these two, only the first survives in the L* regime.

\[
\begin{align*}
  (c)^* & \Rightarrow c & *L \\
  c & \Rightarrow (c)^* & *R \\
  (b/c)^*, c & \Rightarrow b & *L \\
  b/c, c & \Rightarrow (b/c)^* & *R \\
  (a/b)^*, b/c, c & \Rightarrow a & *R \\
  a/b, b/c, c & \Rightarrow (a/b)^* & *R \\
  a/b & \Rightarrow ((a/c)/(b/c))^* & /R \\
\end{align*}
\]

UNIFORM HEAD-DRIVEN SEARCH: MODAL CONTROL. We interleave the proof unfolding and the unpacking of the (\(\Box\))^1, (\(\Box\))^0 translations.

\[
\begin{align*}
  \text{to be cont'd} \\
  (a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow a & (\cdot)^1 & \Box^L \\
  (a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow a & (\Box^1(a/b))^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow a & \Box^L \Box^1 R \\
  (a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow a & (\Box^1(a/b))^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow (\cdot)^0 & \Box^R \Box^1 \Box^0 \\
  (a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow (\cdot)^0 & \Box^1(a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow (\cdot)^0 & \Box^1(a/b)^1, \Box^1(b/c)^1, \Box^1(c)^1 & \Rightarrow (\cdot)^0 & \Box^R \Box^1 \Box^0 \\
  (a/b)^1, \Box^1(b/c)^1 & \Rightarrow (\cdot)^0 & \Box^1(a/b)^1, \Box^1(b/c)^1 & \Rightarrow (\cdot)^0 & \Box^1(a/b)^1, \Box^1(b/c)^1 & \Rightarrow (\cdot)^0 & \Box^R \Box^1 \Box^0 \\
\end{align*}
\]

Consider first the interaction of [\(\Box^1 R\)] rules and selection of the active antecedent type. Antecedent types all have \(\Box^1\) as main connective. The \(\Box^1\) acts as a lock: a \(\Box^1 A\) formula can only become active when it is unlocked by the key \(\Diamond\) (or (\(\Box\))^0 in structural terms). The key becomes available only when the head of the goal formula is reached: through residuation, [\(\Box^1 R\)] transmits \(\Diamond\) to the antecedent, where it selects a formula via [\(\Box^1 R\)].

TRANSMISSION OF THE ACTIVE FORMULA. There is only one key \(\Diamond\) by residuation on the \(\Box^1\) of the goal formula. As soon as it is used to unlock an antecedent formula, that formula has to remain active and connect to the Axiom sequent.
\[
\frac{c \Rightarrow c}{(c)^1 \Rightarrow c} \quad (\cdot)^1
\]

\[
\frac{\Box^1(c)^1 \Rightarrow c}{\Box^1 L}
\]

\[
\frac{\Box^1(c)^1 \Rightarrow \Box^1 c}{\Box^1 R}
\]

\[
\frac{\Box^1(c)^1 \Rightarrow (c)^0}{b \Rightarrow b} \quad (\cdot)^0
\]

\[
\frac{b \Rightarrow b}{(b)^1 \Rightarrow b} \quad (\cdot)^1
\]

\[
\frac{(b)^1 / (b)^0, \Box^1(c)^1 \Rightarrow b}{L}
\]

\[
\frac{(b / c)^1, \Box^1(c)^1 \Rightarrow b}{(\cdot)^1}
\]

\[
\frac{(\Box^1(b / c)^1)^0, \Box^1(c)^1 \Rightarrow b}{K^1}
\]

\[
\frac{\Box^1(b / c)^1, \Box^1(c)^1 \Rightarrow \Box^1 b}{\Box^1 R}
\]

\[
\frac{(b / c)^1, \Box^1(c)^1 \Rightarrow (b)^0}{(a)^1 \Rightarrow a} \quad (\cdot)^0
\]

\[
\frac{a \Rightarrow a}{(a)^1 \Rightarrow a} \quad (\cdot)^1
\]

\[
\frac{(a)^1 / (b)^0, \Box^1(b / c)^1, \Box^1(c)^1 \Rightarrow a}{L}
\]

Below, we show how the wrong identification of the antecedent head leads to failure. The key to unlock $\Box^1(a / b)^1$ has been spent on the wrong formula. As a result, the implication in $(a / b)^1$ cannot become active. Compare with the failure of the corresponding L* derivation above.

\[
\frac{c \Rightarrow c}{(c)^1 \Rightarrow c} \quad (\cdot)^1
\]

\[
\frac{(\Box^1(c)^1)^0 \Rightarrow c}{\Box^1 L}
\]

\[
\frac{\Box^1(c)^1 \Rightarrow c}{\Box^1 c}
\]

\[
\frac{\Box^1(c)^1 \Rightarrow (c)^0}{\Box^1 R}
\]

\[
\frac{\Box^1(c)^1 \Rightarrow (c)^0}{\Box^1(a / b)^1, (b)^1 \Rightarrow a} \quad F A I L S
\]

\[
\frac{\Box^1(a / b)^1, (b)^1 / (c)^0, \Box^1(c)^1 \Rightarrow a}{L}
\]

\[
\frac{\Box^1(a / b)^1, (b / c)^1, \Box^1(c)^1 \Rightarrow a}{(\cdot)^1}
\]

\[
\frac{\Box^1(a / b)^1, (b / c)^1, \Box^1(c)^1 \Rightarrow a}{\Box^1 L}
\]

\[
\frac{(\Box^1(a / b)^1, \Box^1(b / c)^1)^0, \Box^1(c)^1 \Rightarrow a}{K^1}
\]

\[
\frac{\Box^1(a / b)^1, \Box^1(b / c)^1, \Box^1(c)^1 \Rightarrow \Box^1 a}{\Box^1 R}
\]

**LINGUISTIC INTERPRETATION: HEAD FEATURE PERCOLATION.** The above application was purely computational. But, of course, one can also explore a linguistic interpretation of $\otimes$, $\Box^1$ (or multimodal versions) as providing head feature information, with types $\Box^1 A$ projecting headed structures of type $A$. The $K$ distributivity postulates then assume the role of head feature percolation principles. In a multimodal setting, these principles can be relativized to specific mode interactions, cf. the schemata below for interaction between modes $i$ and $j$.

\[
K_{1_{i,j}} : \quad \otimes_i (A \bullet_j B) \rightarrow \otimes_i A \bullet_j B
\]

\[
K_{2_{i,j}} : \quad \otimes_i (A \bullet_j B) \rightarrow A \bullet_j \otimes_i B
\]

\[
K_{i,j} : \quad \otimes_i (A \bullet_j B) \rightarrow \otimes_i A \bullet_j \otimes_i B
\]

For a linguistic application of such relativized $K$ principles consider again the Dutch Verb Raising construction. In §2.1 we analysed verb clusters in terms of interaction postulates communicating between left- and right-headed dependency modes and a head-adjunction mode. Adding head feature information in terms of unary $\otimes, \Box^1$, we would say that the head-selecting $\otimes$ interacts via $K1$ with the left-headed dependency product $\bullet_l$, and via $K2$ with right-headed dependency product $\bullet_r$, and that it strongly distributes over the head-adjunction mode via $K$, identifying the two components of a verb cluster as heads.
4 Conclusion

This paper is a technical investigation of the architecture of mixed categorial type logics. The raison d'être for such an exercise, we hope to have made clear, is the linguistic application of these logics to problems of grammatical analysis — without the linguistic motivation many of the logical issues addressed above would simply not arise. At the same time, we would like to stress that progress on the descriptive level of linguistic analysis is really dependent on a clear understanding of the logical structure of the type systems employed.

The reader who is interested in further results can consult [25]. That paper shows that by means of the $\Diamond, \Box^2$ operators one can systematically recover full control over resource management in the dimensions of linear order, dominance and dependency in logics where such control is lacking at the level of the product/implication fragments.

References


On the Convergence Of ‘Minimalist’ Syntax and Categorial Grammar

Professor Robert C. Berwick, MIT Department of Computer Science
Professor Samuel David Epstein, Harvard Department of Linguistics
Cambridge, MA 02139

ABSTRACT

This paper shows that the so-called “Minimalist Program” of Chomsky (1993, 1995) can be given a natural interpretation as a categorial system in which there is exactly one syntactic (algebraic) operation: namely, “Hierarchically Concatenate” (HC) (what Chomsky calls “Merge”), and also replacing the representations of “D-structure,” “S-structure” and transformations with the derivation lines typical of categorial systems—thus unifying two previously disparate approaches to the analysis of natural language. For example, the general “movement” rule of transformational grammar is easily seen to be a subcase of Hierarchical Concatenation of (alpha, beta), where alpha is a subtree of beta; this automatically derives the usual c-command condition on so-called “empty categories.” The usual semantic interpretation benefits of the categorial approach follow directly.

Further, it demonstrates that by positing a single syntactic concatenation operation one can *derive*—rather than stipulate—the observed grammatical relations in natural languages (viz., Subject-Verb, or Specifier-Head; the notion Head-of; Verb-Complement or Head-Complement; and constituent command or c-command), as well as the primacy of “adjacency” in syntactic constraints.

Finally, this paper shows how the “minimalist program” can be extended to the computational ground of parsing, in that the concatenative system can be naturally interpreted as a generalized, canonical LR parser with a corresponding minimal set of computational operations—suggesting that the (abstract) human parsing system is, like the human language faculty, “perfect” in the sense that the parser delivers to the language faculty exactly those derivational sequences required for the language faculty to “interpret” sentences.

1 AT THE CARTESIAN WELL: THE MINIMALIST PROGRAM & CATEGORIAL GRAMMAR

Imagine the following scene. You are at your favorite beer hall somewhere in Amsterdam—let’s call it the Cartesian Well. Well known meeting place of intelligentsia, you are not surprised when a thin person dressed all in black sidles up to you and whispers in your ear, “Have I got a linguistic theory for you!” You of course yawn, have heard many such fables in your time; besides you have drunk too much. “No, wait,” the figure grabs your shoulder, “I’ve discovered that Chomsky’s latest approach to syntax and categorial grammar are converging.”

Another flat-earth cultist? you think. “Well, hear me out—at least let me buy you another beer.” So you to listen to the tale:

- Chomsky dubs his research program Minimalist Syntax. The idea is that you don’t want to posit any syntactic entities at all beyond what’s absolutely necessary for linguistic description and explanation.
- What machinery is necessary? Lexical items of course—but, minimally, only those, plus elements “composed from” lexical items. (More about composition in a moment!) One begins sentence generation with essentially a multiset, or enumeration of those items. For instance, for the sentence ultimately generated by the syntax as The dog likes it we would have the (unordered) enumeration {the, dog, likes, it}.
- There are no indices, subscripts, bar levels, no phrase names like NP or VP—indeed, no X-bar theory at all. (So much for the intricacies of sub- and supra- indexing in binding.)
- 144 -

- More generally, we dispense with the old transformational generative grammar picture of "levels of representation" such as D-structure, S-structure, LF, and PF, arranged in the familiar inverted Y-diagram with transformational rules mapping between them. Instead, there are only two (natural) representations that stand for the two (natural) interfaces between the language faculty and the rest of the mind/brain and world: one external, namely, the interface to motor systems of speech and perceptual systems of parsing; and the other, the interface to the other cognitive systems of thought, inference, and the like. We may regard this as the longstanding conventional view of language as (sound, meaning) pairs.

- Lexical items are built into more complex objects by a single compositional operation called Merge, more readily thought of as Hierarchical Composition (HC). That is, HC takes two hierarchical representations as input (these will be defined shortly), and produces a new, extended hierarchical representation as output, with one of the two inputs selected as the head or root of the extended representation.\(^1\) For instance, following conventional notation (which we shall dispense with shortly), we would compose the (a Determiner) and dog (a Noun) as follows, projecting the Noun as the Head:

\[
\begin{array}{c}
\text{NP} \\
\text{Det-the N-dog}
\end{array}
\]

In reality, since there can be no phrase labels — these are not lexical items — HC takes as input the two lexical items Det-the and NOUN-dog and composes them, selecting Det-the as the head or functor (the so-called DP hypothesis).\(^2\)

\[
HC(X,Y) \rightarrow \{X\{X,Y\}\}
\]

where the features of the functor Det-the have been projected (i.e., copied) to the Head of the composed item, the first element of the set, or its label (see the figure below). A sentence derivation thus consists of (i) initially selecting a multiset of lexical items; and then (ii) at each step, selecting a pair for input to HC consisting of a selected lexical item and another lexical item or a set resulting from a previous application of HC. In what follows, we shall often identify the composed result of HC simply by its label, or even more simply by an abbreviation for its label, e.g., "Dethe" for the and "D" for the compositional result of HC(Dethe, Ddog), drawing a box around the composed elements. Informally:

\[
\begin{array}{c}
\text{Det-the} \\
\text{Det-the N-dog}
\end{array}
\]

Pursuing the minimalist ideal, we ask why HC takes the form that it does: why does it compose only two items to yield a third? Answer: clearly, HC makes no sense operating on just one item. Its minimal arity is two. Because two arguments evidently suffice (empirically) for natural languages (more deeply: perhaps all grammatical relationships are expressed between adjacent syntactic items, that is, natural grammatical systems are noncounting in the technical sense of McNaughton and Papert, 1967), we do not require (at this point) higher arities for HC. (Binary branching hierarchical structure, independently motivated in current linguistic theory, follows as a result.)

To see how this all fits together so far, let us consider the derivation of the simple sentence the dog likes it that could be described equally well by a context-free phrase structure grammar, or a categorial grammar (example from Epstein (1995:8)). The numbered boxes denote successive applications of HC, and we have omitted much inessential syntactic detail (e.g., Inflectional morphology, etc.). It is to be stressed that this picture is entirely illustrative; the actual representations are simply the derivation lines.

\[
\begin{array}{c}
\text{Det-the N-dog} \\
V_c=S=IP
\end{array}
\]

Derivation lines:\(^3\)

\[^1\text{Hierarchical composition mirrors the linear concatenation of adjacent items in phonology.}\]

\[^2\text{See Alpern, 1987 and discussion below.}\]

\[^3\text{We have put to one side here a minor technical matter and one important issue. The technicality is that the numeration set is not actually a multiset—its duplicate members are distinguished by "some means" that we shall not cover here. The second, more important issue is the question of how a "nonbranching" lexical item like it or John can furnish a function-argument pair as required for input to HC. Roughly, we follow Chomsky's (1993) assumption that these are functors with a single additional empty argument; there are in fact some linguistic arguments for this point. This po-}
}
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Det-the</td>
<td>{ Det-the }</td>
</tr>
<tr>
<td>Select Noun-dogs</td>
<td>{ Noun-dogs }</td>
</tr>
<tr>
<td>HC</td>
<td>D_a</td>
</tr>
<tr>
<td>Select V-likes</td>
<td>{ V-likes }</td>
</tr>
<tr>
<td>Select Noun-it</td>
<td>{ Noun-it }</td>
</tr>
<tr>
<td>HC</td>
<td>D_b</td>
</tr>
</tbody>
</table>

On this view, the derivation of a sentence is simply a line sequence (i.e., proof) of HC operations (lexical selection and HC merger), starting from the lexical multiset. There is no “D-structure” (i.e., representation of lexical items with their thematic roles arrayed in a hierarchical tree) nor any “S-structure”. At any derivation line, the system can decide to “pronounce” — pass to the phonological or motor-articulatory apparatus — its current line derivation. If the motor component can “speak” (spell out) the derivation, then fine; if not, then that proof tree fails (e.g., the system could decide to stop, incorrectly, at the representation of the dog likes, and spell out only those items.

The relationship to classical categorial grammar should be apparent, putting aside for the moment the questions of semantic interpretation and so-called “phrase movement”, to which we return immediately below. Of course, in some cases, the result may not be well-formed (e.g., we could decide to concatenate the dog and V_b first, but this would in effect make the dog the Object (i.e., bear the Patient thematic role); still other possibilities, like the concatenation of Det-the and V-likes yield ill-formed structures; above we show just one of the well-formed derivational sequences). Note that so far we have not yet said how the system “decides” that the output label (i.e. root) of HC is either Det-the or Noun-dog.

As in categorial grammar, the choice of functor is entirely a property of lexical items—in classical arguments, whether we view the as a functor taking an NP argument to its “right” or dog as a functor taking a Determiner argument to its “left”. Whatever choice is made here, the point is that it is an argument-taking property of a now-complex lexical item—again a familiar notion. There is one twist, however: note the deliberate “scare quotes” around the word “right” and “left.” In actuality, we assume that syntactic structure (as opposed to phonological structure) expresses only hierarchical relations and not left-to-right precedence (which is a property of the external, temporal world). In other words, the only choice really made is whether Det-the or Noun-dog is selected as the functor; the directionality at the level of syntax is immaterial—a property of the phonological component, perhaps. Summarizing so far then, we have gained, first, the insight that the “Minimalist Program” is really a version of categorial grammar; and second, that the directionality in classical bi-directional categorial systems is, on this view, an artifact of not separating the “physical interface” level of temporal ordering (phonological order) from the purely syntactic operation of HC. (We shall see below that this separation of precedence from hierarchical or dominance relations has other welcome consequences, namely, it entails the possibility of ambiguous syntactic relations as in Prepositional Phrase attachments, as well as ambiguous quantifier readings.)

Minimally, then, it appears that natural language requires some operation like HC that accounts for the “is-a” (constituent) relations of language. In the (minimally) best of all possible worlds, no other operations need apply. However, it appears at first glance that natural language contains familiar “displacement” operations that move elements around, e.g., the filler-gap relations such as What did John eat or John was arrested described in a variety of grammatical frameworks via “movements” or “slashed categories” or more local operations described in still other accounts such as Bach’s (1983) by "wrap" operations whereby a lexical category such as (a/b)/c is flips its functor around to (a/c)/b, as would happen in, say, auxiliary verb movement or affix hopping.

As is well known, the “long distance” filler-gap dependencies display certain constraints: for example, a “filler” such as Who or John in Who did you see must command its “gap”—the inaudible empty argument position. (E.g., in generalized phrase structure grammar this property is ensured by only introducing slashed categories along with inaudibilia in the same right-hand side of a context-free rules. However, in the current system, we can derive this property “for free”—the first recognizable advantage in adopting the minimalist metaphor. Suppose there is in fact only one syntactic operation, HC. Then in fact we can derive the possibility of “displacement” as a special case of HC with exactly the desired property that “fillers” c-command their “gaps.” Namely, take “displacement” to be that case where we have HC(\alpha, \beta), and \alpha \in \beta (i.e., conventionally, a subtree of \beta). Then this will derive
exactly the cases of wh-questions, topicalizations, etc.—movements generally. Thus in fact, the syntactic components need only one operation.\footnote{This important thesis was discovered first by Kitahara, 1994. We put to one side Bach’s important points about “local” movements, which may well be phonological in nature, since they seem to involve adjacency.}

Turning next to semantic interpretation, we may regard this as the analog of “pronunciation” in the domain of “logical form” or semantics—that is, the interface to the cognitive systems of interpretation, inference, etc. Here too the minimalist program follows categorial grammar rather directly. As each “box” in the third figure above is completed, in the numbered order given—that is, as each HC operation is carried out—the resulting structure is directly (and transparently) interpreted via a “standard” Montagovian approach (or one may substitute one’s own favorite semantic/intensional account here without undue strain). For reasons of space, not much more will be said here about semantic interpretation; the chief point is that the virtually 1-1 correspondence with categorial grammar makes it easy to adopt all the virtues of semantic hygiene that Bach (correctly) advocates.\footnote{We note in passing, though, that the lack of precedence relations in the HC derivation lines also gives us “at no cost” the usual cases of quantifier ambiguity, without having to go through the usual (syntactic) contortions of segments, LF lowering, etc., as is common in so-called government-and-binding accounts: instead, we can simply leave the precedence relations unstated between two quantified NPs, such as everyone and someone in everyone loves someone—precisely because the syntax must leave them unstated. We may picture the NPs as “hanging off” a joint node ahead of the predicate, but this is to be considered as only a graphical aid.}

“If we can take the relations between syntax and semantics as a guide, we would take a homomorphic relation to be the unmarked case, with apparent departures from it providing the most interesting challenges.” (1988:32)

2 Explaining Grammatical Relations

Besides the straightforward connection to categorial grammar, minimalist syntax offers several advantages that do not seem to have been so far widely recognized. Epstein (1995) has remarked that one of the facts we must explain about language is why we observe only certain grammatical relations and not others. For instance, given a sentence like the dog likes it, one would commonly list the following as the significant (perhaps only) grammatical relations:

- the dog stands in the Subject-of relation to the Verb, more generally called the Specifier-Head relation;
- it and likes stand in the Object-of relation, more generally, the Head-Complement relation (or, perhaps, sister-of or govern);
- The VP (old notation) dominates it, etc.;
- The Subject NP the dogs c-commands the VP, the V, and the Object NP, but the Object NP does not c-command the Subject (where c-command is taken to be: \( \alpha \) c-commands \( \beta \) iff the first branching node that dominates \( \alpha \) also dominates \( \beta \)).

This is not meant to be an exhaustive list, but it does illustrate an important point: (1) Why are these relations expressed, but no others out of the infinity of possible relations among two elements in an arbitrary hierarchical structure; (2) Why do the definitions/relations have the form they do—i.e., why is c-command stipulated as “the first branching node...” rather than, say, the seventh?

Epstein’s answer to this question is straightforward: HC provides the “visibility” conditions for all and only the possible grammatical relations. Take for example the relation “c-command.” Given the line-by-line “proof” for sentences in the minimalist framework, we can now derive c-command as follows (Epstein’s example 16):

(1) X c-commands all and only the terms of the category (label) Y with which X was paired by HC in the course of a derivation.

Thus, in our example the dogs like it, the Determiner (Specifier) \( D_a = \text{“the dog”} \) c-commands \( V_b = \text{the VP and all terms of} V_b \), because it was merged with the VP during the course of the derivation. To take an example with “displacement” (in the transformational account), consider a sentence such as She will think he was arrested (Epstein’s example 17). Here, he has by assumption been paired with the inflectional item set associated with was, and therefore c-commands all the terms of that item, namely, arrested and any NP objects of arrested.

Space prevents us from demonstrating how each of the fundamental grammatical relations displayed earlier can be similarly derived from the
basic properties of HC, but the take-home moral is quite strong: (syntactic) grammatical relations are precisely those brought into existence by the syntactic compositional operator.

3 THE COMPUTATIONAL IMPLICATIONS: LR PARSING

So far, we have talked about abstract derivations (proofs) of sentences from lexical multisets, interfacing to the motor/perceptual systems via some linearization process, yielding the left-to-right vocal (alternatively gestural) output that we hear. Turning this problem around, the hearer receives a temporally left-to-right ordered sound signal and, via its perceptual apparatus, recovers (at least conceptually), the hierarchical grammatical relations required in order to interface to the cognitive-meaning component of language—i.e., the view of language as (sound, meaning) pairs. To a first approximation then, the parsing problem becomes: how to recover from the linearized input that contains only precedence relations the hierarchical derivation lines.

Interestingly, a straightforward solution to this interface problem presents itself in a nearly "natural" formulation. Let us define a "natural" solution to the interface problem as follows: the perceptual/parsing system (conversely, the motor/articulatory or production) system should deliver input (alternatively output) to the linguistic faculty that can be easily read—that is, mapped to the same elements that the linguistic faculty employs, with minimal computational effort. In the best case, that mapping will simply be the identity function.

Now consider a canonical (i.e., natural) LR parse of a sentence such as the dog likes the guy. Recall that an LR parse constructs a rightmost derivation in reverse, working left-to-right. If one examines the order in which mergers or completions are built by an LR machine, it is easy to see that they mirror exactly the order of HC operations (in reverse). Moreover, canonical LR operation guarantees that the linguistic condition of strict cyclicity will be obeyed (that is, there can be no "interarboreal" operations that go back and modify to an item already shipped off to semantic interpretation—at least, not in the cases ordinarily considered. In this sense, the perceptual machinery meshes perfectly with the requirements of the language faculty—a surprising condition, if true.⁶

4 CONCLUSIONS FROM THE BEER HALL

To summarize, given the current push towards "minimalism" in the so-called government-and-binding approach seems to have eliminated both government (and binding, not discussed here) in favor of a single hierarchical concatenation operator that meshes perfectly with the classical theory of categorial grammar, as well as providing a natural explanation for the observed grammatical relations and a transparent framework on which to build a model of sentence processing. While this trend surely does not solve all our "religious" problems, it certainly goes a long way towards taking down the "barriers" to a mature, ecumenical framework within which to reach common ground among what has long appeared to be quite disparate accounts of natural language. Perhaps we can all now drink beer together.

REFERENCES


---

⁶Clearly, the language faculty and perceptual apparatus do not mesh in the familiar cases of sentences that are difficult-to-parse or produce—but this is a different sense than the "legibility" and "transparency" requirement intended in the main text.


A Simple Uniform Semantics for Concatenation-based Grammar

Annius V. Groenink
CWI, P.O. Box 94079
1090 GB Amsterdam
avg@cwi.nl

ABSTRACT

We define a more formal version of literal movement grammar (LMG) as outlined in [Gro95c], in such a way that it provides a simple framework that incorporates a large family of grammar formalisms (Head Grammar [Pol84], LCFRS, [Wei88], PMCFG, [KNSK92] and String Attributed Grammars [Eng86]). The semantics is (both in rewriting and least fixed point definitions) simple and elegant, and sheds some new light on shared properties of the mentioned formalisms. We then define a restricted version called simple LMG and show that it generates languages that are not mildly context sensitive, yet preserves the polynomial time recognition property of LCFRS.

INTRODUCTION

This paper consists of three parts. In the first, we propose a more elementary notation of the LMG formalism as introduced the first LMG paper [Gro95c], and call it predicate literal movement grammar. The generalization has a twofold purpose. First, it allows us to give a more elementary semantics, both in rewriting style and as a fixed point operator on sets of terminal strings. Then, we see how it allows us to put a series of tuple-based grammar formalisms of increasing recognising power (LCFRS [Wei88], MCFG, PMCFG [KNSK92], and LMG) in a uniform semantic framework.

In the second part we look at least fixed point interpretations, followed by a discussion on complexity of recognition. We introduce a restricted version of our formalism, simple LMG, and show that it strictly extends LCFRS and PMCFG, yet preserves polynomial time recognition. More precisely, the class of languages described by simple LMG is exactly the class PTIME. The PTIME fragment can be extended to cover input data in the form of a lattice (such as in speech analysis) or arbitrary ordered finite structures (think of pattern recognition in vector based images).

The third part wraps up the story with a classification of the formalisms that have been discussed, and a discussion on mild context-sensitivity and polynomial time. Among other things, we give an example showing that LMG can give accounts of essential structural phenomena in Natural Language known to be beyond the scope of linear context-free rewriting systems.

1 THE PREDICATE LMG FRAMEWORK

Literal movement grammar (LMG, henceforth slash-style LMG) was introduced by the author in [Gro95c], as a formalism which takes a strong left-to-right top-down view on "literal" filler-gap relocation, i.e. passing the terminal words scanned in filler positions down the derivation until they are matched up by a correspondingly typed gap, in the form of what in that paper was called a 'slash item'.

We will redefine LMG here in a version which has more formal appeal.

Definition 1 A predicate literal movement grammar (LMG1) is a tuple \( G = (N, T, V, S, P) \) where \( N, T \) and \( V \) are mutually disjoint sets of nonterminal symbols, terminal symbols and variable symbols, respectively, \( S \in N \), and \( P \) is a set of clauses (which correspond to the notion of a

\(^{1}\)The previous papers [Gro98] and [Gro99b] refer to predicate LMG as CFG (concatenative predicate grammar), but I agree with the readers who thought that giving what is essentially a different representation of the same formalism, a name of its own, might lead to unnecessary confusion.
production in CFG or the slash-style LMG from [Gro95c] of the form
\[
\phi ::= \psi_1, \psi_2, \ldots, \psi_m
\]
where \(\psi_1, \ldots, \psi_m\) are predicates of the form
\[
A(t_1, \ldots, t_n)
\]
where \(t_i \in (T \cup V)^*\).

A predicate LMG clause is instantiated by substituting a string \(w \in T^*\) for each of the variables occurring in the clause.

**Definition 2 (semantics)** Let
\(G = (N, T, V, S, P)\) be a predicate LMG. Then \(G\) is said to recognise a string \(w\) if \(\vdash^G S(w)\) where \(\vdash^G\) is defined inductively as follows: if
\[
A(w_1, \ldots, w_n) ::= B_1(v_{11}, \ldots, v_{1n_1}), \ldots, B_m(v_{m1}, \ldots, v_{mn_m})
\]
is an instantiation of a clause in \(P\), and for each
\[1 \leq k \leq m\]
\[\vdash^G B_k(v_{k1}, \ldots, v_{kn_k})\]
then
\[\vdash^G A(w_1, \ldots, w_n)\]

Note that \(m = 0\) is the base case (zero antecedents).

**Example 1 (Topicalisation/Conjunction)**
An example of an LMG is the following grammar
\[
S(n \ "", \ m) ::= NP(n), VP(v, n)
\]
\[
VP(v_1 \ "\ and" \ v_2, n) ::= VP(v_1, n), VP(v_2, n)
\]
\[
VP(v, n) ::= VT(v), NP(n)
\]
which derives topicaised sentences of the form
"Mary, John saw and admired and kissed."  
An example derivation is shown in figure 1.

**Example 2 (CFGs)** Every CFG can be defined as a predicate LMG; this is analogous to the translation of a CFG into Prolog. Every non-terminal will have one argument which represents its yield. E.g. \(S \rightarrow NP\ VP\) will become \(S(xy) \rightarrow NP(z), VP(y)\).

We can do the same to slash style LMG [Gro95c].

**Example 3 (Slash-style LMG)** The original version of LMG using slash items [Gro95c] [Gro96] can be translated directly into predicate LMG, by introducing an extra argument for each nonterminal. Vice versa, in some predicate LMG, such as the grammar from example 1, one argument that clearly encodes simple left-to-right phrase structure can be removed so as to give a slash-style LMG; the slash-style LMG notation for the grammar in the example is
\[
S \rightarrow n \ "\", \ NP \ VP(n)
\]
\[
VP(n) \rightarrow VP(n) \ "\ and" \ VP(n)
\]
\[
VP(n) \rightarrow VT \ (NP/n)
\]

Note how the slash-style notation is, especially in the case of natural language grammars, more appealing to traditional intuition (but also actually more readable, as it involves less variables per rule).

**Example 4 (Head Grammar)** A head grammar is a CFG that operates on pairs. There are three types of (modified) head grammar rule: a wrapping production:
\[
A \rightarrow \text{wrap}(B, C)
\]
is represented in LMG as
\[
A(b_1, c_1 b_2) ::= B(b_1, b_2), C(c_1, c_2)
\]
and one of the two types of concatenating rule as
\[
A(a_1, a_2 b_1 b_2) ::= B(b_1, b_2), C(c_1, c_2)
\]

**Example 5 (LCFRS and PMCFG)** Linear context-free rewriting systems (LCFRS) and parallel multiple context-free grammars (PMCFG)
are no more than complex notation for restrictions of predicate LMG. Productions in both LCFRS [Wei88] and PMCFG [KNS92] are of the form

$$A \rightarrow f(B_1, \ldots, B_m)$$

where \(f\) is a function over tuples of terminal words defined (symbolically) as

$$f((x_{11}, \ldots, x_{1n_1}), \ldots, (x_{m1}, \ldots, x_{mn_m})) = (t_1, \ldots, t_n)$$

where \(t_i\) are terms over the variables \(x_{ij}\) and terminal symbols. In LCFRS, \(f\) is required to be linear and nonerasing, that is every one of the \(x_{ij}\) should appear exactly once in the \(t_1, \ldots, t_n\). For PMCFG, there is no such restriction.

In the predicate notation of these formalisms, the separate function definition disappears; we simple write a rule as

$$A(t_1, \ldots, t_n) := B_1(x_{11}, \ldots, x_{1n_1}), \ldots, B_m(x_{m1}, \ldots, x_{mn_m})$$

Example 6 (Attribute Grammar) [Eng86]

We have two nonterminals: a start symbol \(Z\) with a designated attribute \(d\); and \(A\) with an inherited attribute \(i\) and a synthesized attribute \(s\). The grammar and the attribute rules are as follows:

- \(Z \rightarrow a\ A\ Z.d := A.s,\ A.i := a\)
- \(Z \rightarrow b\ A\ Z.d := A.s,\ A.i := b\)
- \(A^1 \rightarrow a\ A^2\ A^1.i := A^2.s, A^2.s, A^1.s\)
- \(A^1 \rightarrow b\ A^2\ A^1.s := A^2.s, A^2.s, A^2.i := A^2.i, i\ a\)
- \(A \rightarrow \lambda\ A.s := A.i\ c\ A.i\ c\)

The context-free backbone recognizes arbitrary nonempty strings \(w\) over the alphabet \(T = \{a, b\}\), and the output value for \(w \in T^*\) is \((wc)^{\mid w\mid}\).

The grammar is represented in predicate LMG as

- \(S(z) \rightarrow \) \(Z(x, y)\)
- \(Z(z, az) \rightarrow A(a; z, z)\)
- \(Z(z, bz) \rightarrow A(b; z, z)\)
- \(A(y; zz, az) \rightarrow A(ay; z, z)\)
- \(A(y; zz, bz) \rightarrow A(ay; z, z)\)
- \(A(z; zzxc, \lambda)\).

Note that there is no formal difference between inherited and synthesized attributes—this is in line with the observation that designating an attribute as synthesized or inherited is, once we look at the equations as constraints, semantically irrelevant, and should rather be considered as a hint toward a concrete program or parser that evaluates the "value" of a sentence. The translated SAG is a curious type of LMG, as the S production “throws away" the value \(y\). It will turn out that this type of LMG has less favourable computational properties than the ones defined so far; the example serves mainly to stress that LMG provides a very simple semantics, and a compact notation for attribute grammars.

2 Least Fixed Points and Complexity

The examples have already shown how LMG subsumes the chain CFG \(\subseteq\ BFG\ \subseteq\ LCFRS\ \subseteq\ PMCFG\) of formalisms of increasing generative capacity. It is actually known that these are all strict inclusions; moreover the fixed recognition problems for all these formalisms are in PTIME. We will now answer the question how, in the spirit of these formalisms, we can characterize PTIME itself.

Calculi that describe PTIME have been known for quite some time. The calculus ILFP (integer least fixed point) is introduced in [Rou88]; it applies knowledge about the relationship between bounded arithmetic and complexity to language recognition. The underlying idea is that by talking about positions in the input string, as opposed to about the strings themselves, we can store intermediate steps in the search for a derivation in logspace, which with the Chandra-Kozen-Stockmeyer [CKS81] result on the correspondence between deterministic and alternating Turing machine computations then gives a deterministic PTIME complexity for recognition.

2.1 Fixed Point Interpretations of LMG

Before we redefine LMG to talk about integer positions in strings, let’s present the semantics of LMG in such a way that we interpret the nonterminals as relations over terminal strings. Let \(G = (N, T, V, S, P)\) be an LMG. Let \(\text{NA}\) be the set of assignments to the nonterminals: functions \(\rho\) mapping a nonterminal to a set of arbitrary tuples of strings over \(T\). The set of productions \(P\) can then be viewed as an operator \([G]\) taking an
interpretation function as an argument and producing a new function, defined as follows:

If

\[ A(w_1, \ldots, w_n) := B_1(v_{11}, \ldots, v_{1n_1}), \]
\[ \ldots, \]
\[ B_m(v_{m1}, \ldots, v_{mn_m}) \]

is an instantiation of a clause in \( P \), and for each \( 1 \leq k \leq m, (v_{k1}, \ldots, v_{kn_k}) \in [G](B_k) \), then \( (w_1, \ldots, w_n) \in [G](A) \).

It is easily seen that \([G]\) is a continuous and monotonic operator on the complete partial order \((\mathcal{N}, \subseteq)\) defined by

\[ \rho_1 \subseteq \rho_2 \iff \forall A \in \mathcal{N}, \rho_1(A) \subseteq \rho_2(A) : \]

let \( \rho_1, \rho_2 \) be two assignments, and \( \rho_1 \subseteq \rho_2 \). Let \( a \in ([G], \rho_1)(A) \). Then we have the clause \( R \) and the tuples \( b_1, \ldots, b_m \) from the definition, and \( b_i \in \rho_1(B_i) \). It now follows that for each \( i \), \( b_i \in \rho_2(B_i) \), hence \( a \in ([G], \rho_2)(A) \). So \([G]\) is monotonic. Because the partial order is defined as componentwise set inclusion over a sufficiently general universe, it follows automatically that \([G]\) is also continuous.

We can now validly define the interpretation of a grammar to be the least fixed point of \([G]\):

\[ \mathcal{I}_G = \bigcup_{k=0}^{\infty} [G]^k(\Lambda A. \emptyset) \]

i.e. a function which takes a nonterminal and yields a set of tuples of strings; If \( S \) is the start symbol of \( G \), and its arity throughout the grammar is 1, then \( \mathcal{I}_G(S) \) will be the language recognised by the LMG in the traditional sense.

It is easy to check that the rewriting semantics and the fixed point semantics are equivalent. The fixed point semantics is a useful tool for several purposes. First of all, it gives a more detailed yet mathematically elegant interpretation of grammar. More detailed, because it does not merely characterize the language generated by a single designated start symbol, but characterizes the derivational behaviour of the grammar, without looking at single derivations in particular.

We want to find out how we can restrict the LMG grammars in such a way that recognition can be performed as an alternating search in logspace. For a given string of length \( n \), in log space we can encode a bounded set of numbers ranging from 0 to \( n \) (in binary encoding). This means that we have to encode the arguments of an LMG predicate in a derivation each with a bounded set of numbers. Since in the original interpretation the arguments are strings, the most obvious choice is to encode the arguments as pairs of integers ranging 0 to \( n \) encoding a substring of the input.

Redefine the fixed point semantics as follows; let \( w = a_0 a_1 \cdots a_{n-1} \) be a terminal string of length \( n \); then \( NA_w \) is the set of integer nonterminal assignments \( \rho \) mapping a nonterminal to a set of tuples of pairs of integers. Then \([G]_w \) is defined

If

\[ A(a_{i1} \cdots a_{j1}, \ldots, a_{in} \cdots a_{jn}) := B_1(a_{i1} \cdots a_{j1}, \ldots, a_{in} \cdots a_{jn}), \]
\[ \ldots, \]
\[ B_m(a_{im1} \cdots a_{jm1}, \ldots, a_{inm} \cdots a_{jnm}) \]

is an instantiation of a clause in \( P \), and for each \( 1 \leq k \leq m, ((i_k, j_k), \ldots, (i_{km}, j_{km})) \in [G](w)(B_k) \), then \( (i_1, j_1) \cdots (i_n, j_n) \in [G](w)(A) \).

It is important to see that what is done here, is not the same as taking the string-based LFP interpretation, and intersecting the sets of tuples with the domain of substrings of a given \( w \). If we have an instantiated clause

\[ A(w_1, \ldots, w_n) := B_1(v_{11}, \ldots, v_{1n_1}), \]
\[ \ldots, \]
\[ B_m(v_{m1}, \ldots, v_{mn_m}) \]

such that \( w_1, \ldots, w_n \) are substrings of \( w \), but the \( v_{ij} \) are not, then this instantiation will be ignored in the integer LFP semantics. Hence we want to rule out this type of clause. I.e., we want to make sure that \( w_1, \ldots, w_n \) are substrings of the input, so are the \( v_{ij} \).

Thus simple LMG is defined by disallowing terms other than single variables on the right hand side of the clauses. This way we can uniquely replace each rule by a rule that is talking about integer positions instead of strings.

Definition 3 (simple LMG) An LMG is called simple if its clauses \( R \in P \) are all of the form

\[ A(t_1, \ldots, t_n) := B_1(x_{11}, \ldots, x_{1n_1}), \]
\[ \ldots, \]
\[ B_m(x_{m1}, \ldots, x_{mn_m}) \]

and each of the \( x_{ij} \) appears at precisely once in \( t_1, \ldots, t_n \).
2.2 SIMPLE LML IS IN PTIME

There are two ways to show that the languages generated by simple LMG can be recognised in polynomial time. The first, most formal argument shows that every LMG can be translated into an equivalent formula in the integer string position calculus ILFF [Rou88]; this is quite simple and is sketched in [Gro95a]. The VP rules in the grammar for English topicalization from conjunctive verb phrases

\[
\begin{align*}
\text{VP}(v_1 \ "and" \ v_2, n) & : = \text{VP}(v_1, n), \text{VP}(v_2, n) \\
\text{VP}(v, n) & : = \text{VT}(v), \text{NP}(n)
\end{align*}
\]

for example, are translated into a single formula

\[
\begin{align*}
\text{VP}(i, j, k, l) & \iff \exists i', j'. (i \leq i' < j' \leq j \land \text{and}(i', j')) \\
& \land \text{VP}(i', i', k, l) \\
& \land \text{VP}(j', j', k, l) \\
& \lor (\text{VT}(i, j) \land \text{NP}(k, l))
\end{align*}
\]

The correspondence between the languages defined by ILFF and those recognised by logspace-bounded alternating Turing machines (ATM) shown in [Rou88] then completes the argument.

Here we will sketch a more informal recognition algorithm, which however gives a better indication of what a possible implementation would look like (as it is a deterministic algorithm).

We take the integer representation of LMG grammars, in which every argument is represented as a pair \((l, r)\) of integer indices, as a point of departure (in fact the ILFF translation given above will serve for this purpose).

Given an input string \(w\) of length \(n\), construct memo tables containing a boolean value for each possible predicate \(A(l_1, r_1, \ldots, l_n, r_n)\), where \(l_i, r_i\) are integer values ranging from 0 to \(n\). Reset all the table entries to zero. Now start with the predicate \(S(0, n)\), and recursively check, using the memo table where possible, all possible instantiations of the bound variables \((i' \text{ and } j')\) in the example in all applicable rules.

The procedure for VP rule we just translated is as follows:

\[
\text{VP}(i, j, k, l):
\begin{align*}
\text{if } \text{VP}(i, j, k, l) \text{ memoed then}
\quad & \text{return memoed value}
\text{else}
\quad & \text{memo VP}(i, j, k, l) \text{ as False}
\end{align*}
\]

If \(p\) is the largest number of integer predicate arguments and \(m\) is the largest number of bound variables in each disjunct of the ILFF version of an LMG rule, then the recogniser needs to do \(O(n^m)\) calls or look-ups for each of the predicates. Since there are \(O(n^p)\) predicates, recognition can be performed in deterministic \(O(n^{p+m})\) time and \(O(n^p)\) memoing storage. Constructing a minimally informative parse forest would require \(O(n^{p+m})\) space.\(^2\)

The bound given here seems tight. The rules of a binary, modified head grammar, such as the wrapping rule:

\[
A(a_1 b_1 b_2 a_2) : = B(a_1, a_2), C(b_1, b_2)
\]

are translated into integer based rules with 6 variables \((p = 4, m = 2)\):

\[
A(i, j, l, k) \iff \exists i', l'. (i \leq i' \leq j \land \text{and}(i', l')) \\
\land A(i', i', l', k) \\
\land B(i, i', l', k) \\
\land C(i', j, l', l')
\]

The general recogniser for HG we obtain by applying the sketched algorithm has the well known upper time bound of \(O(n^6)\).

2.3 SIMPLE LML SUBSUMES PTIME

We proceed exactly as in [Rou88]. It is a known result that \(\text{PTIME} = \text{ASP}(\log n)\). Let \(M\)

\(^2\)It should be admitted here that there is a certain amount of handwaving in this argument—the algorithm is recursive, with a maximum recursion depth of \(O(n^p)\)—extra storage and time required to do this recursion is not incorporated into the sketch.
be an alternating Turing machine [CKS81] with a read-only input tape and one binary working tape (the argument can then be extended to cover an arbitrary number of binary working tapes). Let \( M \) be space bounded by \( \log n \), where \( n \) is the length of its input \( w \).

Instantaneous descriptions (ID) of the ATM can be described by a state symbol \( q \) and a tuple \( (h, l, r, ll, rr) \) of integers ranging from 0 to \( n \); \( h \) is the position of the input head, \( l \) and \( r \) describe the contents of the binary work tape left and right of its head, and \( ll \) and \( rr \) represent the amount of work tape space left and right of its head. As Rounds argues, an ID predicate \( q(h, l, r, ll, rr) \) is defined in terms of other ID predicates through a disjunction (existential states) or conjunction (universal states) of other predicates, where the arguments of the predicates are built from \( h, l, r, ll, rr \) through the arithmetical constants and operations 0, 1, \( n - 1 \), \( +1 \), \( -1 \), 2 and \( \sqrt{2} \). The applicability of the moves is checked by equality and nonequality over values derived from \( h, l, r, ll, rr \) by the operators.

We now simulate the ATM in a simple LMG by introducing a 6-ary nonterminal for each state \( q \); its first argument is a copy of the input \( w \); the last five are arbitrary substrings of \( w \), whose length corresponds to the values of \( h, l, r, ll, rr \). The start rule of the grammar is

\[
S(zz) \rightarrow q_0(x, z, z, z, z, z), \text{LengthZero}(x)\]

The informal idea is that the grammar recognises a word \( w \) if and only if \( S(w) \) is derived, hence \( q_0(w, \lambda, \lambda, \lambda, \lambda, \lambda) \) holds,\(^3\) which will correspond precisely to the machine \( M \) halting in an accepting state when given the string \( w \) on the input tape, a blank work tape, and its heads in 0 position. The copy of \( w \) will be passed to each state nonterminal, and will be used both for checking elements of the input and to generate copies of strings for doing arithmetic over 0 \ldots \( n \).

We define a number of auxiliary predicates, such as a schema of clauses defining \( \text{SameLength}(x, y) \) which produces exactly the tuples \( (w_1, w_2) \) where \(|w_1| = |w_2|\), \( \text{EmptyOrLengthOne}(x) \), \( \text{TwiceAsLong}(x, y) \), etc. We can then easily define the arithmetical operations, e.g. if we define

\[
\text{Mult2}(xy, z) \rightarrow \text{TwiceAsLong}(x, z), \text{NextState}(x)
\]

then \( \text{Mult2}(w, v_1) \) is derived by the grammar if and only if it derives \( \text{NextState}(v_2) \), where \( w \) is an auxiliary copy of the input, and \( v_2 \) is any string twice as long as \( v_1 \) (but no longer than \(|w|\)).

Similar constructions define the other arithmetical operations. For each universal state symbol, we introduce a single production that rewrites it to a number of new states. For each existential state, we will have a number of productions which each rewrite it to a single new state. In both cases, a number of extra rules is necessary for evaluation of conditions; the transition itself must be broken up into a series of steps, each step corresponding to the application of one arithmetical operation; each step passes a sufficient number of copies of \( w \) to the next step to preserve the ability of doing modulo \( n \) arithmetic.

Hence we build a grammar that generates \( w \) if and only if the ATM \( M \) accepts \( w \), completing the construction.

### 2.4 Recognition of Nonlinear Finite Structures

We can eliminate the terminals (\( T \)) in the definition of LMG, instead talking about how we can recognise derived relations (the phrases) between positions in a sentence given axioms defining a set of basic relations (the words) between these positions.

**Definition 4** (terminal free LMG) A terminal-free LMG is a tuple \((N, V, S, P)\), where \(N, V\) and \(S\) are as before; productions are as for predicate LMG, but the arguments of nonterminals are now only allowed to be single variables \(x \in V\). Let \(U\) be an arbitrary universe (a set); a terminal free LMG clause is instantiated by substituting an element from \(U\) for each of the variables. The semantics is then as follows; for any instantiated predicate \(\phi\), we have

\[
\phi \vdash_G \phi
\]

and if

\[
A(w_1, \ldots, w_n) \rightarrow B_1(v_{11}, \ldots, v_{1n_1}), \ldots, B_m(v_{m1}, \ldots, v_{mn_m})
\]

\(^3\)Note that this amounts to initializing \( rr \) with the value \( n \) rather than \( \log n \). Although we could initialize it with \( \log n \) (by adding a fairly complicated set of SLMG rules to compute that value), this is not necessary for the construction to succeed.
where $w_i, v_{ij} \in U$, is an instantiation of a clause in $P$, and for each $1 \leq k \leq m$

$$\Gamma_k \vdash^G B_i(u_{k1}, \ldots, u_{kn_k})$$

then

$$\Gamma_1, \ldots, \Gamma_k \vdash^G A(w_1, \ldots, w_n)$$

(where $\Gamma_k$ are sets of predicates).

String-based LMG is an instance of this very general definition; we take $U$ to be the set of nonnegative integers and we encode the string $w = a_0a_1 \cdots a_{n-1}$ by adding $a_0, \ldots, a_{n-1}$ to the set of nonterminals $N$, and postulating the axioms $a_0(0,1), \ldots, a_{n-1}(n-1, n)$. We transform the grammar $G$ to a grammar $G'$ over integer positions instead of strings, as in the formulae in section 2.2; the notion of derivability ($\vdash^G S(w)$) is replaced with

$$a_0(0,1), \ldots, a_{n-1}(n-1, n) \vdash^G S(0,n).$$

Clearly this is not the only interpretation we can imagine. As the form of the axioms allows us to define any finite structure over points in an arbitrary universe $U$, we are now no longer prohibited from defining a string lattice or even any graph; if $U$ (or the part of $U$ addressed in the axioms) is finite, the sketched recognition algorithm will still be polynomial in terms of the number of points. So it seems that this definition extends the scope of tuple-based grammar to the discussion of complexity of more general forms of pattern recognition.

3 Classification & Discussion

We have seen examples of how CFG, HG, LCFRS and PMCFG are represented in the predicate LMG framework. It is known that these are of strictly growing generative capacity; HG can generate the 3-counting language $a^n b^n c^n$ which is not context-free; LCFRS can generate arbitrary counting languages $a_1^n a_2^n \cdots a_k^n$ (for any $k$), but the languages generated by LCFRS satisfy an extended form of pumping lemma, pumping an (even) constant $k$ of substrings.

**Lemma 1** (pumping for LCFRS/MCTAG)

Let the language $L$ be generated by an LCFRS. Then there are constants $n, k$ such that for any $w \in L$ with $|w| > n$, there are strings $u_0, \ldots, u_k$ and $v_1, \ldots, v_k$ such that $w = u_0v_1u_1v_2u_2 \cdots u_{k-1}v_ku_k$, and for any $p \geq 1$, $u_0v_1^pu_1v_2^pu_2 \cdots u_{k-1}v_k^pu_k \in L$.

The PMCFG

$$S(zx) : - S(x)$$

$$S(a).$$

generates the language $a^{\omega}$, which does not grow constantly and hence clearly does not satisfy the pumping lemma.

However, as with context free grammars and LCFRS, subderivations of PMCFG can be freely substituted, hence PMCFG is still closed under arbitrary homomorphism.

To show that simple LMG is of strictly stronger generative capacity, we make two observations. First, simple LMG is closed under intersection: Take two simple LMGs $G_1$ and $G_2$ whose start symbols are $S_1$ and $S_2$ respectively. Then combine the clauses of $G_1$ and $G_2$ (renaming nonterminals where necessary) and add the clause

$$S(x) : - S_1(x), S_2(x)$$

which says "$S(x)$ can be derived if we can derive both $S_1(x)$ and $S_2(x)$." Clearly the resulting grammar generates the intersection of $G_1$ and $G_2$.

The second observation is that we can translate any PMCFG to a simple LMG. Simple LMG does not allow a variable to appear more than once on the LHS of a clause: e.g. the PMCFG clause

$$A(x, yy) : - B(y), C(y)$$

is not a valid simple LMG clause. However (and contrary to PMCFG) simple LMG does allow variables to appear on the right hand side more than once. So we can replace the clause by the fragment

$$A(x, yz) : - B(y), C(y), Eq(y, z)$$

$$Eq(az, ay) : - Eq(z, y) \text{ for every } a \in T$$

$$Eq(\lambda, \lambda).$$

So simple LML subsumes PMCL. Now suppose PMCL and simple LML would be equal, then they would be closed under homomorphism and intersection, which implies that they generate all r.e. languages, and would not be decidable. So we must conclude that PMCL is not closed under intersection, LMG is not closed under homomorphism, and LML strictly includes PMCL.

For a full classification of the different formalisms in their predicate LMG versions, we introduce some terminology.
<table>
<thead>
<tr>
<th>Formalism</th>
<th>Increasing conditions on CPG form</th>
<th>Weakly equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic LMG</td>
<td>—</td>
<td>recursive enumerability</td>
</tr>
<tr>
<td>SAG</td>
<td>First argument of nonterminals does not interact with the others, and is limited to concatenation—i.e. a context free grammar.</td>
<td>—</td>
</tr>
<tr>
<td>Bounded LMG</td>
<td>Length of terminal strings in derivations is polynomially (linearly) bounded in terms of the length of the input string</td>
<td>EXP-POLY time (CLFP, EXPTIME)</td>
</tr>
<tr>
<td>Simple LMG</td>
<td>Bottom-up nonerasing, non-combinatorial</td>
<td>ILFP, PTIME</td>
</tr>
<tr>
<td>Nonerasing PMCFG</td>
<td>Top-down linear, top-down nonerasing</td>
<td>Standard PMCFG</td>
</tr>
<tr>
<td>LCFRS</td>
<td>Bottom-up linear</td>
<td>MCFG, MC-TAG</td>
</tr>
<tr>
<td>HG</td>
<td>Pairs only, restricted operations</td>
<td>TAG, LIG, CCG</td>
</tr>
<tr>
<td>CFG</td>
<td>Singletons</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 2: Hierarchical classification

Definition 5 (properties of LMG) Let $G = (N, T, V, S, P)$ be a LMG, and let $R \in P$ be one of its productions:

$$A(t_1, \ldots, t_n) := B_1(s_{11}, \ldots, s_{1m_1}), \ldots, B_m(s_{m1}, \ldots, s_{mn})$$

then

- $R$ is bottom-up linear if no variable $x$ appears more than once in $t_1, \ldots, t_n$.
- $R$ is top-down linear if no variable $x$ appears more than once in $s_{11}, \ldots, s_{mn}$.
- $R$ is bottom-up nonerasing if each variable $x$ occurring in an $s_{ijk}$ also occurs in at least one of the $t_i$.
- $R$ is top-down nonerasing if each variable $x$ occurring in one of the $t_i$ also appears in one of the $s_{ijk}$.
- $R$ is non-combinatorial if each of the $s_{ijk}$ consists of a single variable.
- $R$ is simple if it is bottom-up nonerasing, bottom-up linear and non-combinatorial.

For all these properties, $G$ has the property if and only if all $R \in P$ have the property.

So an LCFRS is a noncombinatorial, top-down and bottom-up linear, top-down and bottom-up nonerasing LMG. A PMCFG is only top-down nonerasing and top-down linear.

In short, we have the hierarchical classification shown in figure 2.4

3.1 Mild Context-Sensitivity and Polynomial Time

So far we have seen that LCFRS and PMCFG can be extended to simple LMG, which generates a strictly larger class of grammars, but still has polynomial time recognisability; moreover the top-down recognition algorithms for the different types of grammar are not essentially different. Does simple LMG give us an essential increase in expressivity?

Presentations of LCFRS usually go with the definition of mild context-sensitivity (MCS), outlined by Joshi as the class of languages

1. with a limited capacity for describing crossed dependencies
2. recognisable in polynomial time
3. satisfying the constant growth property, that is [Wei88] the language $L$ has associated to it a constant $c_0$ and a finite set of constants $C$ such that for all $w \in L$ where $|w| > c_0$ there is a $w' \in L$ such that $|w| = |w'| + c$ for some $c \in C$.

The constant growth property is in a sense a more general statement of the LCFRS pumping lemma.

4 The figure includes a class (bounded LMG) not treated in this paper, which corresponds to the least fixed point calculus CLFP in [Reu86].
The statement of MCS is motivated by the desire to define classes of grammar which are severely limited in capacity, yet have sufficient strength to describe the basic structure of natural language syntax. It has always been proposed as an attempt to characterize such a class, and there has in particular been a number of arguments that the constant growth property is not satisfactory: Manaster-Ramer [Rad91] pointed out that while \( \{ a^n \mid n \text{ is prime} \} \) does not have the constant growth property, its permuted cousin \( \{ b^n a^n \mid n \text{ is prime} \} \) does.

The following example shows a fragment of Dutch which is constant growth but does not satisfy the LCFRS pumping lemma.

### 3.2 Example

[MR87] gives the following example of a transitive adjoining fragment of Dutch, containing sentences such as:

\[
\ldots \text{dat Jan Piet Marie liet opbellen, that made call}
\text{hoorde uitnodigen,}
\text{heard invite}
\text{hielp ontmoeten en zag omhelzen}
\text{helped meet saw embrace}
\ldots \text{that Jan made Piet call Marie,}
\text{heard [him] invite [her],}
\text{helped [him] meet [her]}
\text{and saw [him] embrace [her]}
\]

The fragment can be characterized as follows:

\[
\ldots \text{dat Jan Piet Marie NP}^k
\text{liet VR}^k \text{opbellen,}
\text{hoorde VR}^k \text{uitnodigen,}
\text{hielp VR}^k \text{ontmoeten}
\text{en zag VR}^k \text{omhelzen.}
\]

The fragment does not satisfy the pumping lemma for TAG and Head Grammar which says that we can create constantly growing subfragments by pumping 4 substrings. Obviously, in the example, 5 strings will need to be pumped.

The pumping lemma for LCFRS states that there is a number \( k \) such that at most \( k \) strings need to be pumped. Increasing the number of conjuncts in Manaster-Ramer's example hence provides, given any LCFRS, an argument that it cannot describe the fragment.

The following simple (slash-style) LMG does generate Manaster-Ramer's fragment [MR87] as sketched in the examples, with an unbounded number of conjuncts.

\[
S \rightarrow \ldots \text{dat NP} \ n \ \text{VP}(n)
\]
\[
\text{VP}(n) \rightarrow \overline{V}(n)
\]
\[
\text{VP}^C(n) \rightarrow \overline{V}(n) \ " \overline{V} \text{VP}^C(n)
\]
\[
\overline{V}(n) \rightarrow \overline{V}(n) \text{ en } \overline{V}(n)
\]
\[
\overline{V}(m) \rightarrow \overline{VT}(\text{NP}/n)
\]
\[
\text{NP} \rightarrow \text{Jan, Marie, Piet}
\]
\[
\text{VT} \rightarrow \text{opbellen, uitnodigen...}
\]
\[
\text{VR} \rightarrow \text{zag, horen, hielp...}
\]

### 3.3 Revising MCS

The ability to describe crossed dependencies in conjunctive VPs is clearly a desirable feature of a grammar formalism in the spirit of LCFRS. This could be seen as an argument that the constant growth property in the definition of LCFRS should in fact not be strengthened to a pumping lemma.

The remaining question is whether there are clearly 'unnatural' languages that are in PTIME but which we want to rule out; one may think of \( a^n \). If we do not rule these out, then the 'limited capacity for describing crossed dependencies' is obvious, and mild context-sensitivity collapses into the single predicate 'recognisable in polynomial time' which is equivalent to 'generated by a simple LMG'.

The pumping lemma for LCFRS is too weak (it doesn't rule out the prefixed prime language \( \{ b^n a^n \mid n \text{ is prime} \} \)), whereas LCFRS does not cater for the example of crossed dependencies and unbounded conjunction.

I believe that the 'flaws' in the definition of constant growth and pumping lemma should be circumvented by claiming that there is a fixed bound to the size of the 'unpumped' part of the string, i.e. there is some form of a finite 'basis'. A revised pumping lemma would be along the lines of

**Lemma 2 (strong finite pumping)** Let \( L \) be a language. Then \( L \) is strongly finitely pumpable if there are constants \( n, k \) such that for any \( w \in L \) with \(|w| > n\), there are strings \( u_0, \ldots, u_k \) and \( v_1, \ldots, v_k \) such that \( \sum u_i + \sum v_i < n \), there is
a p such that \( w = u_0 v_1^p u_1 v_2^p u_2 \cdots u_{k-1} v_k^p u_k \), and for any \( p \geq 1 \), \( u_0 v_1^p u_1 v_2^p u_2 \cdots u_{k-1} v_k^p u_k \in L \).

This does rule out the prefixed prime language because instead of claiming that we can make larger strings from a given string, we are saying that it can be pumped from a string shorter than a constant fixed for the language.

Since this is a stronger pumping lemma than that known for LCFRS/MCTAG, it is not what we are after, since it will again rule out the unbounded conjunctions. However, if we could weaken this version of the pumping lemma into a revised definition of constant growth, it would seem to characterize a valuable property.

### 3.4 Conclusions

We have outlined a formal version of the LMG formalism as presented earlier, and shown that we can define a restriction which models exactly the polynomial time recognisable languages. Moreover, this restriction, the simple LMG, can describe essential fragments of Dutch verb structure which cannot be described by any known smaller classes of grammars within PTIME.

While LCFRS was previously the best known approximation of the ideal class of ‘mildly context-sensitive’ grammars, we believe to have shown by our examples of Dutch, that it is not strong enough; however the alternative presented here is clearly too strong—unnatural languages such as \( a^n \) can now be described. So now the question should be raised how we can exploit the extra power (hidden in the ability to do intersection), without allowing the reduplication given by PMCFG (multiple occurrence of variables on the LHS), which seems to give rise to the ‘unnatural’ languages. It should be noted that LMG grammars which generate these unnatural languages, contain ‘equality’ predicates which consist of one clause for every symbol in the terminal alphabet, which could indeed be seen as an ‘unnatural grammar’.

The proposed general recogniser for LCFRS/LMG which gives the proper bounds for the class of HG (including TAG, LIG and CCG), if informally presented, is as far as we know the first of its kind written on paper. One of the reasons there have not been attempts to define such algorithms before is the claim [KNSK92] that universal recognition of LCFRS is PSPACE-complete and universal recognition of PMCFG is EXP-POLY time complete. However, these results involve constructions which generate grammars whose size is proportional to a given input string, and hence provide only a limited picture of computational reality. Thoughts on possible improvement by reducing top-down prediction based on terminal corners are in progress, and an implementation of a parser based on LMG is to be expected in the near future.

### Acknowledgements

The author is supported by SION grant 612-317-420 of the Netherlands Organization for Scientific Research (NWO). Part of this paper is an elaborated version of the extended abstract [Gro95a] of my talk at the fourth Mathematics of Language workshop in Philadelphia, November 1995. Other papers are available webwise through http://www.cwi.nl/~avg/.

### References


Theory of Texts

G. Rozenberg
Universiteit Leiden
the Netherlands

Abstract

The theory of 2-structures provides a convenient framework for the decomposition and transformation of graph-like structures. In this talk we will discuss, in a tutorial fashion, a subclass of the class of 2-structures, the so-called T-structures. The notion of T-structure is a natural generalization of the notion of linear order. The basic result of the theory of T-structures, which says that a T-structure can be represented by two linear orders, leads to the notion of a text. It generalizes the notion of a word as used in formal language theory. A text may be seen as a word with an additional linear order of its domain. This order determines a "structure" spanned on the word - it may be a tree, but it may be also more general than a tree. We consider context-free grammars for different classes of texts.
A Graph Grammar Approach to Graphical Parsing

J. Rekers
Department of Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
email: rekers@wi.leidenuniv.nl

A. Schürr
Lehrstuhl für Informatik III, RWTH Aachen
Abornstr. 55, D-52074 Aachen, Germany
email: andy@i3.informatik.rwth-aachen.de

ABSTRACT

We present a new graph grammar based approach for defining the syntax of visual languages and for generating visual language parsers. Its main advantage — in comparison to other visual language parsing approaches — is its ability to handle context-sensitive productions which may replace more than one non-terminal at the same time and which may contain very complex context requirements. Its implementation will be part of a forthcoming syntax directed editor toolkit for visual languages and the already existing graph grammar programming environment PROGRES.

1 INTRODUCTION

In reading the proceedings of the past VL workshops or any book on software engineering one cannot help but notice that a large variety of visual languages exists of which only a few are equipped with a proper formal syntax definition. That is regrettable, as such definitions have a number of advantages to offer:

• without a proper syntax definition, new users can only guess the syntax of a graphical language by generalizing from the provided examples,
• a definition could serve as unambiguous specification for syntax directed editors over the language,
• a graphical parser could be generated out of a proper definition, and
• a syntax definition is a necessary precondition for a definition of the semantics of the language.

We think that it would be very beneficiary to the theory of visual languages if a single syntax definition formalism could be agreed on. Such a formalism should be highly expressive, unambiguous, and specifications should be easy to read and develop.

In this paper we will show how graph grammars can be used as syntax definition formalism for graphical languages, and we will develop a graphical parsing algorithm based on these grammars. We will however first argue why graphical parsing would be useful for users of visual languages, and we will show that graphical parsing, if used to its full power, is less trivial than it might seem.

1This paper also has appeared in the proceedings of the IEEE Symposium on Visual Languages (VL'95), pp. 195-202, Darmstadt, Germany, 1995

2 MULTI-STAGE GRAPHICAL ANALYSIS

We propose a multi-stage graphical analysis (parsing) process as depicted in Fig. 1. Usual graphical editing, such as with Idraw or FrameMaker, produces a collection of pictorial elements, which are fed to phase 1 of the analysis process, graphical scanning. It interprets the positions, sizes, and shapes of pictorial elements and produces a spatial relations graph, in which objects are connected by relations like contains, points at, and labels.

Phase 2, low level parsing, maps pictorial elements and their spatial relationships onto more abstract objects and relationships between them. Its output is

0: Graphical Editor
  ↓
Pictorial Elements
  ↓
1: Graphical Scanning
  ↓
Spatial Relations Graph
  ↓
2: Low Level Parsing
  ↓
Abstract Relations Graph
  ↓
3: High Level Parsing
  ↓
Final Output

Figure 1: Our multi-stage graphical analysis approach
an abstract relations graph. For Finite State Automata this phase would recognize a circle that encloses a string as a State, and an arrow with a close-by string as a Transition (cf. productions $p_5, p_6,$ and $p_7$ of [13] or the FSA grammar of Marriott [10]). Fig. 2 shows a FSA diagram and the transformation in its corresponding abstract relations graph. However, low level parsing would also happily accept an unconnected FSA, an automaton with unreachable states, one without a start state, or one with several start states.

In order to catch those kinds of errors, additional more complex grammar rules are needed. Phase 3, high level parsing, is defined by a grammar which states how sentences should be generated, starting from an axiom (cf. productions $p_4, p_5,$ and $p_7$ of [13]). For the FSA case, the axiom would be an Automation, which may be rewritten into a StartState; given a State, one may add an outgoing Transition and a new State; given two States, one may connect them by a Transition. These rules guarantee connected automata, for which all states are reachable from the start state.

It is also possible to perform only some steps of the proposed multi-stage graphical analysis process. For instance, the constraint based graph editor EDGE [9] may be used to create a spatial relations graphs, and phase 1 would not be necessary. Or a syntax directed editor may directly create an abstract relations graph, thereby skipping phases 1 and 2.

2.1 PROCESS FLOW DIAGRAMS AS EXAMPLE

The running example of this paper will be the recognition of well-structured process flow diagrams. An example of the abstract relations graph of a process flow diagram is depicted in Fig. 3. Do note that this graph already suggests work of phases 1 and 2 by the shape of the vertices in this graph, but this is only to make the example less abstract.

Fig. 4 contains the (high-level) grammar for this language. Production 1 of this grammar replaces the axiom PFD with two terminal vertices and one non-terminal vertex, which are connected by means of two n(ext) edges. Production 2 deletes a single Stat vertex and creates a new assign vertex, which inherits an incoming and an outgoing control flow edge from the deleted nonterminal vertex. Two dashed

axiom: PFD

label wildcards:

$B?, C? \in \{ \text{begin, fork, if}\}$

$S?, T? \in \{ \text{end, assign, fork, join, send, receive, if}\}$

$p, q \in \{ n, l, t\}$

productions:

1: $\text{PFD} \quad ::= \quad \begin{array}{c}
\text{begin} \quad \rightarrow \quad \text{Stat} \quad \rightarrow \quad \text{end}
\end{array}$

2: $\begin{array}{c}
B?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
B?\rightarrow\text{assign} \quad T?\quad
\end{array}
\end{array}$

3: $\begin{array}{c}
B?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
B?\rightarrow\text{Stat} \quad \text{Stat} \quad
\end{array}
\end{array}$

4a: $\begin{array}{c}
B?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
B?\rightarrow\text{fork} \quad \text{join} \quad T?\quad
\end{array}
\end{array}$

4b: $\begin{array}{c}
\text{fork} \quad \text{join} \quad ::= \quad \begin{array}{c}
\text{fork} \quad \text{join} \quad \text{Stat} \quad \text{Stat} \quad
\end{array}
\end{array}$

5: $\begin{array}{c}
C?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
C?\rightarrow\text{send} \quad T?\quad
\end{array}
\end{array}$

6: $\begin{array}{c}
B?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
B?\rightarrow\text{Stat} \quad \text{Stat} \quad
\end{array}
\end{array}$

7: $\begin{array}{c}
B?\rightarrow\text{Stat} \quad T? \quad ::= \quad \begin{array}{c}
B?\rightarrow\text{Stat} \quad \text{Stat} \quad
\end{array}
\end{array}$

Figure 3: Sample process flow diagram

Figure 4: Grammar for process flow diagrams
context vertices are used for this purpose. They have to be present when the production is applied, but remain unmodified. The left context vertex is one of \{begin, fork, if\} and therefore source of a n(ext), t(rue) or f(alse) edge. The right context vertex is one of \{end, assign,...\} and therefore always the target of a n(ext) edge. Separately defined label wildcards may be used to construct a single production as an abbreviation for all the above mentioned combinations of possible vertex and edge labels.

The other productions have a similar outline: they extend Stat lists, create new process threads, establish communication channels between them, and produce conditional loops as well as branches. Note that already such a small grammar contains reasonable examples of productions which do not delete any nonterminals (production 3 and 4b) or which replace more than one nonterminal at the same time (production 5).

3 ANALYSIS OF RELATED APPROACHES

When inventing a new syntax definition and parsing approach for graphical languages, the most important thing is to come up with a reasonable solution for the so-called embedding problem. In the case of linear textual languages it is clear how to replace a nonterminal in a sentence by a corresponding sequence of (non-)terminals. But in the case of graphical languages with many possible relationships between language elements we need a far more complicated mechanism for (re-)establishing relationships between old context elements of a replaced nonterminal and its replacing (non-)terminals (see productions of Fig. 4 and Fig. 5).

3.1 EMBEDDING PROBLEMS OF GRAPHICAL LANGUAGES

There are at least three solutions for the embedding problem:

1. Implicit Embedding: formalisms like picture layout grammars [5] or constraint multiset grammars [10] do not distinguish between vertex and relationship objects. As a consequence, all needed relationships between objects are implicitly defined as constraints over their attribute values. Therefore, attribute assignments within productions have the implicit side effect to create new relationships to unknown context elements.

2. Extended Context: all approaches which support the concept of relationships between objects directly, need a special mechanism to embed new (non-)terminal objects in their proper context. A straightforward solution for this problem is to extend left- and right-hand sides of productions with context elements (as we do in Fig. 4). These context elements will not be modified by production applications but may be used as sources or targets for new relationships.

3. Embedding Rules: The last and most powerful solution is an essential part of various forms of graph grammars like [7,14]. These formalisms have separate embedding rules which allow the redirection of arbitrary sets of relationships from a replaced nonterminal to its replacing (non-)terminals.

All three approaches have their specific advantages and disadvantages. The main drawbacks of the first approach are: users are not always aware of the consequences of attribute assignments, and parsers have to spend a lot of time to extract implicitly defined knowledge about relationships from attributes and constraints.

The second approach is in our opinion the most readable one, but the unrestricted use of context elements requires very complex parsing algorithms. Furthermore, it is difficult in this setting to rewrite nonterminals which may participate in a statically unknown number of relationships.

In the latter case, the embedding rule approach is the most convenient one. But embedding rules are difficult to understand and all known parsing algorithms for productions with embedding rules are either hopelessly inefficient or impose very hard restrictions on left- and right-hand sides of productions (see below).

3.2 PROPERTIES OF GRAPHICAL PARSING ALGORITHMS

Summarizing the explanations above, related parsing approaches should be studied and compared by answering the following questions:

- Is the left-hand side of a production restricted to a single nonterminal, which will be replaced by its right-hand side (context-free production)?
- Are there any restrictions for the right-hand sides of productions?
- Does the formalism allow references to additional context elements, which have to be present but remain unmodified during the application of a production?
• Does the proposed type of grammar have more or less complex embedding rules, which establish connections between new elements (created by a production) and the surrounding structure?

• Are there additional restrictions for the set of productions or the form of graphs, which do not fall in the above mentioned categories?

• Is the time and space complexity of the proposed algorithm linear, polynomial, or even exponential with respect to the size of an input graph?

Table 1 provides an overview of our related work studies with respect to these questions.

3.3 Analysis of Graphical Parsing Algorithms

The precedence graph grammar parser of Kaul is an attempt to generalize the idea of operator precedence based parsing and has a linear time and space complexity. The parsing process is a kind of handle rewriting, where graph handles (subgraphs of the input graph) are identified by analysing vertex and edge labels of their direct context. Unfortunately, this approach works only for a very restricted class of graph languages.

The next three entries in the table contain references to Earley-style parsing approaches [2]. The first one by Bunke and Haller [1] uses plex grammars, which are a kind of context-free graph grammars with rather restricted forms of embedding rules. Any nonterminal has only a fixed number of connection points to its context. The second one by Wittenburg [17] uses dotted rules to organize the parsing process for relational grammars, but without presenting any heuristics how to select "good" dotted rules. Furthermore, it is restricted to the case of relational structures, where relationships of the same type define partial orders.

Finally, the approach of Ferucci et al. [4] with so-called INS-RG grammars is a translation of the graph grammar approach of Rozenberg/Weizl [1] into the terminology of relational languages. In this approach right-hand sides of productions may not contain nonterminals as neighbours, thereby guaranteeing local confluence of graph rewriting (parsing) steps. Furthermore, polynomial complexity is guaranteed as long as generated graphs are connected and an upper boundary for the number of adjacent edges at a single vertex is known.

All presented approaches up to now are not adequate for generating process flow graphs. Their embedding rules are not able to rewrite previously unconnected State vertices to pairs of connected Send and Receive vertices (as we do in the rewriting step presented in Fig. 5). And even the remaining two approaches of Marriot and Golin would have their difficulties with process flow diagrams. Their parsing algorithms generalize the bottom-up algorithms of Tomita [16] or Cocks-Younger-Kasami [18,6] for context-free textual grammars. Marriot's constraint multiset grammars [10] offer the concept of context elements and would thereby be able to define the graph rewriting step of Fig. 5. But the accompanying parsing algorithm is not yet able to deal with these context elements.

There remains the picture layout grammars [5] of Golin. His parsing algorithm allows terminal context elements, but has a main focus on productions with one nonterminal on the left-hand side and at most two nonterminals on the right-hand side with predefined spatial relationships between them. A definition of process flow graphs which obeys these restrictions should be possible but would be quite unreadable. Furthermore, the realized parsing algorithm has the following restrictions:

<table>
<thead>
<tr>
<th>Left-hand Side</th>
<th>Right-hand Side</th>
<th>Context Elements</th>
<th>Embedding Rules</th>
<th>Additional Parsing Restrictions</th>
<th>Space/Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaul [7,8]</td>
<td>one nonterminal</td>
<td>directed nonempty graph</td>
<td>no</td>
<td>yes</td>
<td>implicitly def. vertex ordering by edges</td>
</tr>
<tr>
<td>Bunke/Haller [1]</td>
<td>one nonterminal</td>
<td>nonempty plex structure</td>
<td>no</td>
<td>fixed set of connection points</td>
<td>no</td>
</tr>
<tr>
<td>Wittenburg [17]</td>
<td>one nonterminal</td>
<td>nonempty relational structure</td>
<td>no</td>
<td>yes</td>
<td>explicitly def. vertex ordering by relations</td>
</tr>
<tr>
<td>Ferrucci et al. [6]</td>
<td>one nonterminal</td>
<td>rel. structure without nonterminal neighbours</td>
<td>no</td>
<td>yes</td>
<td>(bounded degree, connected structure)</td>
</tr>
<tr>
<td>Rozenberg/Weizl [14]</td>
<td>one nonterminal</td>
<td>dir. graph without nonterminal neighbours</td>
<td>no</td>
<td>yes</td>
<td>(bounded degree, connected graph)</td>
</tr>
<tr>
<td>Marriot [10]</td>
<td>one nonterminal</td>
<td>arbitrary nonempty multiset</td>
<td>(yes)</td>
<td>implicit</td>
<td>acyclic grammar, no context elements</td>
</tr>
<tr>
<td>Golin [5]</td>
<td>one nonterminal</td>
<td>one or two (non-)terminals</td>
<td>one terminal</td>
<td>implicit</td>
<td>finite set of possible attribute values</td>
</tr>
<tr>
<td>Rekers/Schürr [13]</td>
<td>directed graph</td>
<td>directed connected, nonempty graph</td>
<td>directed graph</td>
<td>not yet</td>
<td>global layering condition for grammar</td>
</tr>
</tbody>
</table>

Table 1: Visual language syntax definition and parsing approaches
• Two nonterminals of the same class, which are used for the derivation of a single graphical language sentence, must have different attribute values (otherwise the parser makes the wrong decision to identify them).

• Their must be an upper boundary for the number of possibly used different nonterminal attribute values during parsing (otherwise, the parser tries to create an infinite number of unidentifiable nonterminals).

• Overlapping matches of right-hand sides due to ambiguous derivations may not occur (they cause the parser's immediate termination with a fatal error).

To summarize, all presented parsing approaches are inadequate for defining the language of process flow diagrams in a proper way. There is a strong need for a new syntax definition and parsing approach, where both left- and right-hand sides of productions are arbitrary graphs which share a common context graph. Such a formalism together with its parsing algorithms will be presented within the next section.

4 THE GRAPH PARSING ALGORITHM

Our parsing algorithm for graphical languages is in fact a graph parsing algorithm. In this paper we will only sketch its main characteristics; for a complete presentation of all details including a full proof of correctness and termination, the reader is referred to [13].

Definition 1: A production is of the form \((L, R)\), with \(L\) and \(R\) both graphs. \(L\) and \(R\) may have an intersection.\(\square\)

A production \(p = (L, R)\) can be read in two directions:

• In generation, the host graph \(G\) is searched for a match \(h(L)\) of the left-hand side \(L\). If found, \(h(L)\) is replaced by a copy \(h'(R)\) of the right-hand side, resulting in a graph \(G'\).

• In parsing, the host graph \(G'\) is searched for a match \(h'(R)\) of the right-hand side \(R\). If found, \(h'(R)\) is replaced by a copy \(h(L)\) of the left-hand side \(L\), resulting in a graph \(G\).

Definition 2: The application of a production \(p\) is a production instance \(pi := (p, h, h')\), where graph morphisms \(h: L \rightarrow G\) and \(h': R \rightarrow G'\) relate vertices and edges of \(L\) and \(R\) to vertices and edges of \(G\) and \(G'\). The application of \(p\) to \(G\) with result \(G'\) is denoted as

\[ G \xrightarrow{p} G' \]

Do note that the part in \(G\) that corresponds to \(h(L \cap R)\) and the part in \(G'\) that corresponds to \(h'(L \cap R)\) are equal and thus preserved by the application of \(pi\) in either direction. This part is the context which has to be present, but which itself is not affected by the application of \(pi\).

Definition 3: A derivation from the axiom graph \(A\) to the graph parsed \(G\) is a sequence of production instances \(pi_1, ..., pi_n\) with the following property:

\[ A \xrightarrow{pi_1} G_1 \xrightarrow{pi_2} ... \xrightarrow{pi_n} G_n = G \square \]

One can distinguish two tasks which have to be performed by a graph parsing algorithm:

1. The searching in the host graph for matches of right-hand sides of productions. This is an expensive process which works at graph element level.

2. Each completed match results in a production instance. These have to be combined into a derivation. In the case of ambiguities, it might however happen that more than one derivation exists, or it might happen that a recognized production instance is not useful at all.

During the development of our parsing algorithm it became evident that dealing with these two tasks at the same time results in very complex algorithms. These algorithms would even perform a lot of work which turns out to be useless afterwards. Therefore, we decided to realize a two-phase parsing algorithm as follows:

• The bottom-up phase searches the graph for matches of the right-hand side of productions. On the recognition of such a right-hand side, a production instance \(pi\) is created, and the elements in \(h'(L \setminus R)\) are added to the graph, but nothing is deleted from it. The additions might in turn lead to the recognition of other right-hand sides. The result of the bottom-up phase is the collection \(PI\) of all production instances discovered.

The production instances created have dependency relations among each other, such as above\((pi, pi')\), which means that \(pi\) should occur before \(pi'\) in a derivation, or exclude\((pi, pi')\), which states that \(pi\) and \(pi'\) may not occur in the same derivation. These relations can be computed during the bottom-up phase.

• The dependency relations are used to direct the second phase of the parsing process, the top-down phase. It starts with an empty graph and applies production instances of \(PI\) in such a way that the above and exclude relations are respected. By knowing all possible production instances and their dependency relations in advance, the top-down phase is able to postpone exploration of alternative derivation branches as long as possible. When necessary, these alternative derivations are developed in a pseudo-parallel fashion, with a preference for depth-first development.

4.1 THE BOTTOM-UP PHASE

We will parse the process-flow diagram of Fig. 6a according to the grammar of Fig. 4. In the end this will lead to two possible derivations, as it is ambiguous which of the two assign statements should match
This means that production instances $pi3$, $pi4$, $pi5$, and $pi6$ of Table 2 are created, and the graph is extended to the one of Fig. 6c. In this graph the right-hand side of the production $I$ can be recognized, which means that two production instances are created, and two PFD vertices ($v11$ an $v12$) are added to the graph. That completes the work of the bottom-up phase, and the production instances of Table 2 will be shipped to the top-down phase.

4.2 SEARCH PLANS AND DOTTED RULES

In the description above, we simply stated which matches were possible, but not how these matches are computed. In order to implement searching for matches efficiently, we associate a linear search plan to each production's right-hand side, which predetermines the order in which the match must be constructed. Each search plan starts with the matching of a single vertex, and extends its match at every step by following an edge from the already matched part of the host graph into the still unknown part. For example, a reasonable search plan for production 6 of Fig. 4 would be:

$N1$: $\text{vertex}(\text{ff})$
$E1$: $\text{edge}(N1, \rightarrow, \{f\}, N2$: $\text{vertex}(\text{Stat}))$
$E2$: $\text{edge}(N2, \rightarrow, \{n\}, N1)$
$E3$: $\text{edge}(N1, \rightarrow, \{f\}, N3$: $\text{vertex}(\{\text{end, assign, fork, join, send, receive, if}\})$
$E4$: $\text{edge}(N1, \leftarrow, \{n, t, f\}, N4$: $\text{vertex}(\{\text{begin, fork, if}\})$

In order to be able to construct these search plans, right-hand sides of productions have to be connected.

In constructing a match, the bottom-up phase moves a dot through the search plan: vertices and edges left of the dot are already matched, the ones right of it still have to be matched. Such a dotted rule is attached to the vertex in the host graph where the next to be matched edge starts or ends. It might happen that a searched-for edge is not present in the host graph. In the extended graph, the right-hand side of production $4a$ can be recognized with $\text{Stat}$ vertex $v7$, but it can also be recognized with $\text{Stat}$ vertex $v8$. Similarly, production $4b$ can be recognized in two ways.

**Table 2: The production instances created**

<table>
<thead>
<tr>
<th>pi</th>
<th>prod.</th>
<th>Xlhs</th>
<th>Common</th>
<th>Xrhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pi1$</td>
<td>2</td>
<td>$v7$</td>
<td>$v2$ $v3$</td>
<td>$v3$</td>
</tr>
<tr>
<td>$pi2$</td>
<td>2</td>
<td>$v8$</td>
<td>$v2$ $v5$</td>
<td>$v4$</td>
</tr>
<tr>
<td>$pi3$</td>
<td>$4a$</td>
<td>$v9$</td>
<td>$v1$ $v6$</td>
<td>$v2$ $v7$ $v5$</td>
</tr>
<tr>
<td>$pi4$</td>
<td>$4a$</td>
<td>$v10$</td>
<td>$v1$ $v6$</td>
<td>$v2$ $v8$ $v5$</td>
</tr>
<tr>
<td>$pi5$</td>
<td>$4b$</td>
<td>$v2$ $v8$</td>
<td>$v5$</td>
<td>$v7$</td>
</tr>
<tr>
<td>$pi6$</td>
<td>$4b$</td>
<td>$v2$ $v7$</td>
<td>$v5$</td>
<td>$v8$</td>
</tr>
<tr>
<td>$pi7$</td>
<td>1</td>
<td>$v11$</td>
<td>$v1$ $v9$</td>
<td>$v6$</td>
</tr>
<tr>
<td>$pi8$</td>
<td>1</td>
<td>$v12$</td>
<td>$v1$ $v10$</td>
<td>$v6$</td>
</tr>
</tbody>
</table>

**Figure 7: The matches described by $pi2$**
graph. In that case the dotted rule is suspended, and will be awakened when a promising edge appears.

Say, we have the following subgraph in the host graph and a dotted rule is attached to x with its dot in front of the matching directive:

E4: edge(N1, →, (a), N2: vertex((k)))

In this case both edges match, and y and z both obtain an incremented version of the dotted rule. We also have to leave the original dotted rule (in suspended state) attached to vertex x, as another edge with label a may still appear.

The bottom-up algorithm starts by attaching initial dotted rules to matching vertices in the host graph. Next it repeatedly chooses an active dotted rule to advance. If a dot reaches the end of a search plan, the associated production has been recognized completely. That generates a production instance, and the host graph is extended with the elements in L \ R: vertices may give rise to initial dotted rules, edges may activate suspended dotted rules. This is repeated until there are no remaining active dotted rules.

Our bottom-up phase performs an exhaustive generation of all possible production instances, during which it only extends the graph on every production instance found. Although the algorithm takes care not to perform double work (initial dotted rules are only created for newly created vertices, dotted rules only proceed over newly created edges), this may still lead to non-termination, as the grammar may be cyclic.

We, therefore, have to introduce the following layering restriction on the grammar: every edge and vertex label is assigned a layer number, such that terminal labels belong to layer 0, and non-terminal labels belong to a layer greater than 0. This layer assignment has to be such that every production respects the following order L < R:

\[ L < R \iff \exists k \sum_{i} |L_i| < |R_i| \land \forall j < i \sum_{j} |L_j| = |R_j| \]

with |G|_k the number of elements in G which have a label of layer k. This layering restriction guarantees the termination of the bottom-up phase by disallowing cyclic grammars.

The layering can also be used to process active dotted rules in such a way that productions which generate graph elements of lower layers are given priority. This means that the layers of the elements that are added to the graph will be increasing. This implies again that dotted rules which are waiting for an element of a lower layer can safely be discarded. In practice, this measure avoids almost all suspended dotted rules; see [13] for details.

4.3 The Dependency Relations

A production instance represents the application of a production to some version of the graph, and it indicates the graph elements matched by both sides of the production. By operating on graph elements, production instances depend on each other. In order to reason about these dependencies, we first introduce the abbreviations Lhs, Common, and Rhs for the different sets of elements matched by a production instance, and next introduce the dependency relations above, consequence, excludes, and excludes*.

Definition 4: We define the following abbreviations for a production instance pi := (p, h, , h') with p := (L, R):

- Xlhs(pi) = h(L \ R), the exclusive left-hand side, which are (in generation) the set of deleted graph elements;
- common(pi) = h(L \ R), the set of matched but preserved graph elements;
- Xrhs(pi) = h'(R \ L), the exclusive right-hand side, which are (in generation) the set of created graph elements.

Definition 5: Given these partitions, we can define three cases when a production instance pi should be above a production instance pi' in any derivation in which both occur:

1. Xrhs(pi) \ Xlhs(pi') \not= \emptyset: pi creates an element which is deleted again by pi', or
2. common(pi) \ Xlhs(pi') \not= \emptyset: pi needs a context element which is deleted by pi', or
3. Xrhs(pi) \ common(pi') \not= \emptyset: pi deletes an element which pi' needs as context element.

In case 1 or 2, pi' consumes an element which is produced or deleted by pi. Therefore, pi' must belong to any derivation which contains pi, i.e. pi' is said to be a consequence of pi (and cons* is its transitive closure).

Definition 6: It may also be the case that two production instances exclude each other (directly or indirectly) and may never be part of the same derivation:

- (Xrhs(pi) \ Xrhs(pi')) \not= \emptyset: pi and pi' create the same elements, or
- above(pi, pi') \ above(pi', pi): pi and pi' have a mutual above relation.

The transitive closure excludes* is defined as follows:

\[ \exists p_i, p_i': \mid p_i \text{ cons} p_i \land p_i' \text{ cons} p_i' \land p_i \text{ excludes} p_i' \] .

Based on these definitions, the production instances of Table 2 lead to the relations as depicted in Fig. 8. For example, above(pi8, pi4) holds since:

Xrhs(pi8) \ Xlhs(pi4) = \{v1, v10, v6\} \cap \{v10\} \not= \emptyset.
4.4 THE TOP-DOWN PHASE

The top-down phase composes a subset of the production instances which forms a derivation for the graph parsed. For example, given the production instances of Table 2 and the relations of Fig. 8, correct derivations according to Def. 3 would be: \{p_{17}, p_{13}, p_{16}, p_{11}, p_{12}\} and \{p_{18}, p_{14}, p_{15}, p_{11}, p_{12}\}.

The top-down phase develops such a derivation by keeping a stack of partial derivations, of which only the topmost one is active. The lower ones are starting points for alternative derivations, and will only become active if the ones higher on stack fail.

**Definition 7:** A tuple \((G_c, API_c, EPI_c)\) is a partial derivation for \(G\) in the context of all possible production instances \(PI\). \(G_c\) is the graph built by the applied production instances in \(API_c\); the production instances in \(EPI_c\) are the already excluded ones.\(\square\)

The top-down phase starts with a derivation \((A, \emptyset, \emptyset)\) of length \(0\). Next, it repeats the following steps until a complete derivation has been found, or there are no partial derivations left. It creates at each step a set of candidate production instances \(CPI\), which are those \(pi\) in \(PI \setminus (API_c \cup EPI_c)\) such that all production instances above it are either in \(API_c\) or in \(EPI_c\). It depends on the contents of \(CPI\) what happens next:

- **CPI = \emptyset:** \(API_c\) is a successful derivation if \(G_c = G\); otherwise, the current partial derivation is dropped and the top-down phase continues with the next one on stack.

- There is a \(pi \in CPI\) without an \(excludes^*\) relation to another still applicable production instance: apply \(pi\) by changing the active partial derivation into:
  \((G_c \setminus \text{Lhs}(pi)) \cup \text{Rhs}(pi), API_c \cup \{pi\}, EPI_c)\).

4.5 THE OUTPUT OF THE ANALYSIS

It depends on the application domain what the final result of the analysis process must be and how it should be specified:

1. It might be the case that we are only interested whether a derivation exists or not. A yes/no answer then suffices.

2. Our parsing algorithm represents a derivation as a sequence of production instance applications. This provides all information, but might be too detailed and too abstract to interpret.

3. During the above rewriting process non-terminal elements are created, which are consumed again to create terminal elements. It would also be possible to perform the rewriting by only creating elements. In that case, the final result would be a graph which contains the axiom graph \(A\), the final graph \(G\), and all intermediate graphs as subgraphs. Certain terminals and non-terminals could be filtered out of this graph again.

4. Another solution would be the YACC way: attach an arbitrary action to every production; these actions are then executed in the order of the derivation found. This is a quite low-level solution which can however create any desired data structure.

5. A higher level approach is based on coupling two graph grammars such that parsing a graph with respect to the first grammar is bound to the generation of another graph with respect to the second grammar [15].

Throughout this paper we have only considered method 2; any further processing of derivation results is subject for future research.

5 SUMMARY AND DISCUSSION

Graph grammars are a very powerful mechanism for defining the syntax of graphical languages with well-known theoretical properties [3]. The main drawback of graph grammars until now was the lack of efficiently working parsing algorithms or, more general, the lack of almost any tool support. Within this paper we presented the overall ideas of a new graph grammar parsing algorithm. A complete formal definition of our new approach together with a proof of correctness of the presented parsing algorithm may be found in [13]. Its implementation will be part of a forth-

A flaw in our parsing algorithm is the inability to identify "equivalent" non-terminals, which are the result of local ambiguities. This deficiency is demonstrated in the example of Fig. 6, in which two equivalent Stat nodes (v9 and v110) appear. They are the result of a locally ambiguous interpretation of the fork-join statement. In order to avoid reduplication of parsing efforts, it would be nice if these two Stat nodes could be identified instead of resulting in two completely different derivations.

Parsing algorithms based on cover checks, such as Marriott's [10], are able to identify non-terminals which are the result of local ambiguities: each element knows which input elements it covers; at the moment two elements with the same label and the same cover set occur, the two are identified. A drawback of cover set based parsing algorithms is that they require that the left-hand side of every production is a single non-terminal. They are even not able to deal with non-terminal context elements, since cover sets do not record the history of derivations. This is the reason why Marriott's parsing algorithm works only for grammars without context elements and why Golín's parsing algorithm [5] disallows non-terminal context elements.

However, context elements and more complex left-hand sides are indispensable for producing readable syntax definitions of visual languages. Therefore, our parsing algorithm uses complex history relations instead of simple cover sets and is, thereby, able to handle productions properly, which
• delete more than one vertex at the same time,
• delete nothing at all, i.e. extend a graph only, and
• make use of context elements.
Future research is necessary to find a way to identify non-terminals with equivalent histories during parsing.

REFERENCES


STRONG INTERCHANGEABILITY AND NONLINEARITY
OF PRIMITIVE WORDS

Sándor Horváth

Department of Computer Science
Eötvös Loránd University
Budapest, Múzeum körút. 6-8., II. 2-3.
H-1088, Hungary
e-mail: horvath@cs.elte.hu

ABSTRACT

A primitive word is a nonempty word that cannot be written as a proper power. Such words play an important role in combinatorics on words and in the theory of codes. In this paper we prove four new results (in Theorems 1-4) on the set \( Q \) of all primitive words over a fixed, several-letter alphabet. The first result is that \( Q \) satisfies a strong interchange property. The second one is that the density of nonprimitive words up to length \( n \) tends to 0 at an exponential speed. The third one is that \( Q \) is not a bounded language. The fourth result says that \( Q \) is not a linear context-free language. The conjecture that \( Q \) is not context-free remains unproved.

1 INTRODUCTION

A **primitive word** is a nonempty word not of the form \( x^k \) for any \( k > 1 \). Such words play an important role in combinatorics on words and in the theory of codes (see, e.g., [17], [23], and [2]). This paper has been motivated by the recent papers [16], [3], [4], [5], [20], [6], and studies some combinatorial and related properties of the set of all primitive words (over a fixed, several-letter alphabet).

In general we use the standard notions and notation of formal language theory (as found, e.g., in [21], [9], [10], or [22]). (Within this we write \( \lambda \) for the empty word, and the length of words as well as the cardinality of sets, will be denoted by \(|.|\).) We denote set-theoretic inclusion and proper inclusion by \( \subseteq \) and \( \subsetneq \), respectively. For any two positive integers \( k \) and \( l \), their greatest common divisor will be denoted by \( \gcd(k, l) \). We fix a finite alphabet \( X \) having at least two letters, and denote by \( Q \) the set of all primitive words over \( X \). Below we give some definitions and lemmas to be used in proving our results in Section 2.

The property of primitivity is clearly invariant under reverse (or mirror image) and cyclic permutation of words. The latter invariance is expressed in detail in the following

**Lemma 1.** (Shyr and Thierrin, see e.g. in [23].) Let \( x, y, u, v \in X^+ \) such that \( x = uv \) and \( y = vu \), i.e., \( x \) and \( y \) are nontrivial cyclic permutations of one another. Then for any \( i > 1 \), there exists a \( z \in X^+ \) such that \( x = z^i \) iff there exists a \( z' \in X^+ \) such that \( y = (z')^i \) (\( z \) and \( z' \), if they exist, are - not necessary nontrivial - cyclic permutations of one another). Therefore \( x \in Q \) iff \( y \in Q \).

The next lemma plays a fundamental role in combinatorics on words.

**Lemma 2.** (Fine and Wilf, see e.g. [9] or [17].) Let \( x, y \in X^+ \). If two powers \( x^i \) and
$y^j$ ($i, j > 0$) of $x$ and $y$, respectively, have
a common prefix (or a common suffix) of
length $|x| + |y| - \gcd(|x|, |y|)$, then for some
$z \in X^+$, $x, y \in z^+$.
\end{proof}

This lemma implies the following

\textbf{Lemma 3.} (See, e.g., in [17] or [23].) For
any $x \in X^+$ there is a unique primitive
word $q$ such that $x$ is a power of $q$.
\end{proof}

By the unique existence of $q$ in Lemma
3, the following two definitions make
sense. We introduce Def. 2, based on Def.
1, on a weak analogy of the notion of log-
arithm in the set of positive reals.

\textbf{Definition 1.} (The root of a word, see
[9] or [22].) For any $x \in X^+$ we call the
uniquely existing word $q \in Q$ for which $x$ is
a power of $q$, the root of $x$, and we denote
it by $\text{root}(x)$. For $x = \lambda$ we define the root
of $\lambda$, denoted $\text{root}(\lambda)$, to be $\lambda$. (According
to a generally accepted agreement, for any
$y \in X^+$ $y^0$ is defined to be $\lambda$).

\textbf{Definition 2.} (Logarithm of a word.) For any word $x \in X^+$ we define the loga-
thym of $x$, denoted $\log(x)$, as the uniquely
determined positive integer $r$ such that
$\text{root}(x)^r = x$. For $x = \lambda$ we define $\log(\lambda)$
to be 0.

We remark that in [19] and [22], for the
root of $x$ the notation $\rho(x)$ is used, not
our "root($x$)". Further, for $x \neq \lambda$ we actu-
ally interpret $\log(x)$ as the "base-root($x$)
logarithm of $x". So for any word $x$ the identity
$\text{root}(x)^{\log(x)} = x$ holds, and we have
$x \in Q$ iff $\log(x) = 1$ (and $x \in X^+\setminus Q$
iff $\log(x) \neq 1$).

It is known that every context-free lan-
guage satisfies Ogden, Ross and Winkl-
mann's interchange property (see [19], [18]
or [7]). We shortly call this combinatorial
property of languages the property \textit{ORW}
and we introduce the following strengthening
variant of it.

\textbf{Definition 3.} (\textit{Strengthened interchange
property}, or shortly: the property \textit{ic-
strong}.) We say that a language $L \subseteq X^*$ satisfies the \textit{strengthened interchange
property} or shortly the property \textit{ic-strong},
iff there is $c > 1$ (depending only on $L$)
such that for all $n \geq 2$, $i \geq 0$ and $j \geq 1$
with $j < n$ and $i + j \leq n$, and for all
nonempty subsets $H$ of $L \cap X^*$, there is
$H' \subseteq H$ with the following properties:
$|H'| > |H|/c$ and for any two words $x$ and
$y$ in $H'$, if $x = x_1x_2x_3$ and $y = y_1y_2y_3$
with $|x_1| = |y_1| = i$, $|x_2| = |y_2| = j$, 
we have $x_1y_2x_3, y_1x_2y_3 \in L$, and in this
case we shortly say that $H'$ (and also,
that any pair of elements of $H'$) is \textit{i-j-
interchangeable}.

We remark that the property \textit{ic-strong}
is much stronger than the property \textit{ORW}
since in the former property we have
$|H'| > |H|/c$ instead of $|H'| > |H|/cn^2$,
and also, unlike the original property
\textit{ORW}, in the property \textit{ic-strong} there are
no restrictions on the beginning ($i$) and
length ($j$) of the middle subwords to be
exchanged, other than excluding the triv-
ial cases $j = 0$ and $j = n$.

\textbf{Definition 4.} (\textit{Linear pumping property,
see, e.g., in [1] or [10, Ex. 6.11].}) A lan-
guage $L \subseteq X^*$ satisfies the \textit{linear pump-
ing property} or shortly the property \textit{lin-
pump} iff there is an $m \geq 1$ such that any
$z \in L$ with $|z| > m$ can be factorized into
$z = uvwx$, so that

\begin{enumerate}
\item $|ux| \geq 1,$
\item $|uwx| \leq m,$ and
\item $\forall i \geq 0 : uv^iwx^iy \in L$.
\end{enumerate}

\textbf{Lemma 4.} (\textit{Linear pumping lemma,
see, e.g., in [1] or [10, Ex. 6.11].}) Every linear
context-free language satisfies the prop-
erty \textit{lin-pump} (Def. 4).
\end{proof}
2. The Results

Theorem 1. Q satisfies the property ic-strong (Def. 3 with \(c = 8\), moreover, even \(c = 4\) is enough in the following three cases:

1. \(n\) is of the form \(n = 2^k, k \geq 1\);
2. \(n\) is odd;
3. \(n\) is of the form \(n = 2pk\) where \(p\) is an odd prime, the smallest odd prime divisor of \(n\), \(k \geq 1\), and \(\min\{j, n - j\} \leq n(p - 1)/2p = (p - 1)k\) (the latter condition is simply implied by \(\min\{j, n - j\} \leq n/3\)).

Remarks. 1. As it can be seen from the statement of this theorem, intuitively "with great probability" even \(c = 4\) is enough, and the same is true for having \(|F(x)| \leq 1\) in Lemma 6.

2. We can choose \(c = 8\), and in cases (1) - (3) even \(c = 4\), independently of \(|X|\) (\(\geq 2\)).

We prepare the proof of this theorem with the following two lemmas. The first of them simply follows from the invariance of primitivity under cyclic permutation (see Lemma 1 above) and from the definition of the property ic-strong (Def. 3).

Lemma 5. It is enough to consider only \(r - s\)-interchangeabilities such that \(r + s = n\) and \(r \geq s(\geq 1)\).

Let \(r, s \geq 1\). We define the function \(F\) from \(X^r\) to subsets of \(X^s\) as follows: \(\forall x \in X^r : F(x) := \{y : y \in X^s \text{and} xy \notin Q\}\) (of course, it is possible that \(F(x) = \emptyset\)). For this function \(F\) we have

Lemma 6. For any \(r \geq s \geq 1\) and for any \(x \in X^r\), \(|F(x)| \leq 2\) holds, moreover, putting \(r + s = n\), in the above cases (1) and (2) of Theorem 1 even \(|F(x)| \leq 1\) holds.

Proof sketch. By Lemma 3, \(|F(x)| = \{|y : y \in X^s \text{and} log(xy) > 1\}| = \{|\text{root}(xy) : y \in X^s \text{and} \text{root}(xy) < xy(n)\}| = \{|g : g \in Q, |g| \text{ is a proper divisor of } n, \text{ and } x \text{ is a prefix of } g^{n/|a|}\}|.

We use the inequality \(r \geq s\) (i.e., \(r \geq n/2\)) and Lemma 2. Fixing \(x\), assume indirectly \(|F(x)| \geq 3\) and let \(q_1, q_2, q_3 \in Q\) be three distinct elements of \(\{\text{root}(xy) : y \in F(x)\}\).

Clearly also \(|q_1|, |q_2|, |q_3|\) are distinct, and even for \(i \neq j\) \(|q_i|\) is no multiple of \(|q_j|\), so we can suppose \((n/2 \geq 1)|q_1| > |q_2| > |q_3|\).

Now we show that our assumption leads to a contradiction.

Case (a): \(|q_1| = n/2\). Case (a1): \(|q_2| = n/3, |q_3| = n/5\). Then \(n = 2 \cdot 3 \cdot 5 \cdot k\) for some \(k \geq 1\), and \(|q_2| = 10k, \ |q_3| = 6k, \ gcd(|q_2|, |q_3|) = 2k\). Now \(|q_1| + |q_2| - gcd(|q_2|, |q_3|) = 14k < 15k = n/2\), so by Lemma 2 root \(q_2(q_3) = \text{root}(q_3)\), a contradiction.

Case (a2): \(|q_2| \leq n/3\) and \(|q_3| \leq n/7\). In this case simply \(|q_2| + |q_3| < n/2\), so again by Lemma 2 we get the same contradiction.

Case (b): \(|q_1| < n/2\). Then \(|q_1| \leq n/3, |q_2| < n/4, |q_3| \leq n/5\), and we proceed exactly as in Case (a2).

On the basis of the above, by similar calculations we can see that in cases (1) and (2) of Theorem 1 we even have \(|F(x)| \leq 1\).

Proof of Theorem 1. On the basis of Lemma 5 it is enough to consider \(r - s\)-interchangeabilities such that \(r \geq s \geq 1\) and \(r + s = n\). Let \(\emptyset \neq H \subseteq Q \cap X^n\), and define

\(A := \{x \in X^r : \exists y \in X^s \text{and} xy \in H\}\),

\(B := \{y \in X^s : \exists x \in X^r \text{and} xy \in H\}\).

Since the assertion of the theorem is trivial for \(|B| = 1\), we can suppose \(|B| \geq 2\)

\(\Box\)
(though the argument below is valid for \( |B| = 1 \), too). Let us define for an arbitrary set \( \emptyset \neq Z \subseteq B \), the set

\[
H_Z := \{ x \in A : \forall y \in Z [xy \in Q] \} Z
\]

(here we can have \( H_Z = \emptyset \), too). Now for all \( xy \in H \) there are at least \( 2^{|B|-3} \) sets \( \emptyset \neq Z \subseteq B \) such that \( xy \in H_Z \), because the number of sets \( \emptyset \neq Z \subseteq B \) such that

\[
y \in Z \quad \text{and} \quad Z \cap F(x) = \emptyset,
\]

is \( 2^{|B|-1-|F(x)|} \), and by Lemma 6 \( |F(x)| \leq 2 \). On the other hand the number of nonempty subsets of \( B \) is \( 2^{|B|} - 1 \). So

\[
\sum_{\emptyset \neq Z \subseteq B} |H_Z| \geq 2^{|B|-3}|H|,
\]

and by the pigeonhole principle there is a \( \emptyset \neq Z' \subseteq B \) for which

\[
|H_{Z'}| \geq 2^{|B|-3}|H|/(2^{|B|} - 1) > |H|/8,
\]

therefore the choice \( H' := H_{Z'} \) is suitable. From Lemma 6 it easily follows that in cases (1) and (2) we even have \( |H'| > |H|/4 \), and by some calculation the same can be obtained in case (3), too. \( \Box \)

Using Lemma 6 we can also prove the following theorem. (We shall write \([\cdot]\) for 'integral part'.)

**Theorem 2.** For every \( n \geq 1 \),

\[
|Q \cap X^n| > |X|^n - 2|X|^{[n/2]} =
\]

\[
= |X|^n(1 - 2|X|^{[n/2]} - n) \geq
\]

\[
\geq |X|^n(1 - 2|X|^{-n/2}) =
\]

\[
= |X^n|(1 - 2|X|^{-n/2}).
\]

**Proof sketch.** For \( n \leq 2 \) both inequalities are obvious. For \( n \geq 3 \) (actually even for \( n \geq 2 \)) we obtain the term \( -2|X|^{[n/2]} \) by a closer look at the proof of Lemma 6. On the one hand in applying Lemma 2 (in the proof of Lemma 6) the length of the "common prefix" is always at most \( [n/2] \). On the other hand it is easily seen that for all \( n \geq 2 \), putting \( r = [n/2] \) and \( s = n - [n/2] \) there are \( x \in X^r = X^{[n/2]} \) for which \( |F(x)| \leq 1 \).

From this theorem the next one rather easily follows. (For the definition of a bounded language see, e.g., [8] or [10].)

**Theorem 3.** \( Q \) is not a bounded language.

**Proof sketch.** From Theorem 2 (or even in a simple, direct way) we can see that the function \( f_Q(n) := |\{ w \in Q : |w| \leq n \}| (n \geq 0) \) grows exponentially. (It is easy to see directly that for \( n \geq 1 \),

\[
f_Q(n) - f_Q(n - 1) = |Q \cap X^n| \geq |X|^{[n/2]}.
\]

However, it is not too difficult to show (as is done in [11]) that for a bounded language \( L \) the analogous function \( f_L(n) \) can grow at most polynomially. Therefore \( Q \) cannot be a bounded language. \( \Box \)

Theorem 2 gives also an intuitive explanation for the fact that \( Q \) satisfies various strong, (mostly) context-free-like combinatorial properties (sc-strong, in Theorem 1 above, and various other, strong combinatorial properties, see [4], [5]), namely, since practically "almost all words are primitive" (the relative number of non-primitive words up to a given length \( n \) is exponentially small in \( n \)), so there is "good possibility" to remain inside \( Q \) during combinatorial manipulations on the primitive words. (Of course, the situation is not at all so simple, the proofs of the strong combinatorial properties of \( Q \) are rather complicated.)
Our final result relates Q to the linear context-free languages.

**Theorem 4.** Q is not linear context-free.

**Proof.** By Lemma 4 it is enough to prove that Q does not satisfy the property lin-pump. Suppose the contrary and let \( m \geq 1 \) be a lin-pump constant for Q. Let \( a, b \in X, a \neq b \) and \( z = a^m b a^{2m+ml} b a^m \) (clearly \( z \in Q \)). Now in any lin-pump factorization \( z = uvwx \), necessarily \(|uv|, |xy| \leq m\), so \( uv, xy \in a^* \), and for \( i = 1 + ml/|vx| \) we should have \( z^{(i)} := uv^iwx^iy = a^{m+nl} b a^{2m+ml} b a^{m+n} \in Q \) with some non-negative \( n', n'' \) such that \( n' + n'' = ml \). However, \( z^{(i)} = (a^{m+n'l}ba^{m+n''})^2 \notin Q \), contradiction. \( \square \)

Each of our last two results says that Q does not belong to a certain language class. Combining these two results with an earlier one of this kind (Q is not unambiguous context-free \[20\]), which is stronger than the yet earlier characterization in \[3\], that Q is not deterministic context-free, we can exclude Q from three important, pairwise incomparable language classes. (Here the class of bounded languages is, of course, even "orthogonal" to all Chomsky-classes.)

The following conjecture which was first formulated in \[3\] and has been the main motivation for the works \[3, 4, 5, 20, 6\], and also for the present one, remains unproved.

**Conjecture.** Q is not context-free.

**Acknowledgements**

Both this paper and its preprint variant, \[13\], are based on the work \[12\] done by the author during his three-month visit, in 1993, at the Fachbereich Informatik of the University of Hamburg, which was financially supported by a scholarship from the Commission of the European Communities, Brussels. The author also thanks Matthias Jantzen for confirming his belief that it would be worth trying out Ogden, Ross and Winklmann's interchange condition on Q, and Manfred Kudlek for a similar, interesting talk.

The Hungarian-German Research Project No. OMFB-NPI-102 has also promoted the arrangements for presenting this paper at the workshop AMiLP'95.

**References**


11. Horváth, S.: Néhány egyszerű bizonyítás arra, hogy \( \{a, b\}^* \) nem korlátozott nyelv. (Some easy proofs that \( \{a, b\}^* \) is not a bounded language. In Hungarian.) Budapest, April, 1993. Unpublished draft.


Algebraic Methods in Categorial Grammar

Wojciech Buszkowski
Faculty of Mathematics and Computer Science
Adam Mickiewicz University
Poznań Poland

Abstract

In this paper we discuss different algebraic structures which are natural algebraic frames for categorial grammars. First, absolutely free algebras of functor-argument structures and phrase structures together with powerset algebras of types are used to characterize structure languages of Basic Categorial Grammars and to provide algorithms for equivalence problems and related questions. Second, unification applied to the above frames is employed to develop learning procedures for Basic Categorial Grammars. Third, residuated algebras are used to model language hierarchies of Lambek Categorial Grammars.

1 Algebras of Basic Categorial Grammars

Basic Categorial Grammars (BCG’s) were introduced by Bar-Hillel [3, 4] with origins in Aijdukiewicz [1] who had followed some logical ideas of Frege, Russell and Leśniewski. Linguistic expressions are assigned types; first, lexical assumptions assign types to atomic expressions (symbols, words), and second, Hilbert style deductions based on Modus Ponens are used to derive types of complex expressions.

Types can be represented as purely conditional formulas, applying two conditionals \( \rightarrow \) and \( \leftarrow \). Atomic types are simply (sentential) variables (one may also think of them as constants). The type \( A \rightarrow B \) (resp. \( B \leftarrow A \)) is assigned to functors from type \( A \) to type \( B \) which take their argument on the left (resp. right). Thus, if \( S, PN \) and \( N \) are atomic types of Sentence, Proper Noun and Common Noun, respectively, then \( PN \rightarrow S \) is the type \( VP \), of Verb Phrase, \( S \leftarrow VP \) is the type \( NP \), of full Noun Phrase, and \( NP \leftarrow N \) is the type \( D \), of Determiner. A BCG is defined by a finite set of lexical assumptions \( v : A \) such that \( v \) is an atomic expression and \( A \) is a type. By \( V_G \) we denote the lexicon of the grammar \( G \), i.e. the set of atomic expressions appearing in lexical assumptions. Given a string \( v_1 \ldots v_n \), \( v_i \in V_G \), the grammar \( G \) assigns type \( A \) to this string, if there are assumptions \( v_i : A_i \), \( i = 1, \ldots, n \), such that there is a deduction tree with the root \( A \) and the yield \( A_1 \ldots A_n \). Actually, this definition can be preserved for all kinds of categorial grammar; only, inference rules admissible in deductions are different for different kinds. BCG’s admit two MP-rules:

\[
\begin{align*}
\text{MP}\rightarrow & \quad \frac{A; \ A \rightarrow B}{B}, \\
\text{MP}\leftarrow & \quad \frac{B \leftarrow A; \ A}{B}.
\end{align*}
\]

For example, the grammar defined by lexical
assumptions:

every, the: \( NP \leftarrow N \),

student, test: \( N \),

passed: \( VP \leftarrow NP \),

where \( NP = S \leftarrow VP \), assigns type \( S \) to the expression:

\[
\text{every student passed the test}
\]

on the basis of the following deduction:

\[
\begin{array}{c}
NP \leftarrow N \\
VP \leftarrow NP \\
NP \leftarrow N, N \end{array}
\]

\[
\begin{array}{c}
NP \\
VP \\
S
\end{array}
\]

This deduction can also be represented as the bracketed term:

\([\langle NP \leftarrow N, N \rangle_1, \langle VP \leftarrow NP, [NP \leftarrow N, N \rangle_1, [1])\]

which yields the functor-argument structure of the given expression:

\( [[\text{every, student}]_1, [\text{passed, the test}]_1]_1 \).

Here, the numerical subscript \( i \) means that the \( i \)-th constituent takes the part of the functor. In the example above, always the first constituents are functors, but it is not the case for the structure:

\( [[\text{John}, [\text{likes, Mary}]_2]_3 \)

which is provided by the grammar:

John, Mary: \( PN \),

likes: \( (PN \rightarrow S) \leftarrow PN \).

The set \( FS(V) \), of functor-argument structures over the lexicon \( V \), is defined by the recursive clauses:

(a) \( V \subseteq FS(V) \),

(b) if \( X, Y \in FS(V) \) then \( [X, Y]_i \in FS(V) \),

for \( i = 1, 2 \).

\( X \) is the functor and \( Y \) is the argument in \( [X, Y]_1 \), and the opposite holds for \( [X, Y]_2 \). Clearly, \( FS(V) \) can be treated as an absolutely free algebra with the set \( V \) of free generators and two binary operations \( [\ldots, \ldots]_1 \) and \( [\ldots, \ldots]_2 \). Dropping numerical subscripts changes functor-argument structures into phrase structures. \( PS(V) \) denotes the set of phrase structures over \( V \). Again, \( PS(V) \) is an absolutely free algebra with one binary operation \( [\ldots, \ldots] \).

We assume that each BCG \( G \) distinguishes an atomic type, say \( S \), as the basic type, to be denoted \( S_G \). By \( V_G \) we denote the lexicon of \( G \). We write \( v_1 \ldots v_n \Rightarrow_G A \), if \( G \) assigns type \( A \) to string \( v_1 \ldots v_n \). For \( X \in FS(V_G) \), we write \( X \Rightarrow_G A \), if \( G \) assigns type \( A \) to the yield of \( X \) by a deduction which determines the structure \( X \). We define:

\[
L(G) = \{ a \in V_G^* : a \Rightarrow_G S_G \},
\]

\[
FL(G) = \{ X \in FS(V_G) : X \Rightarrow_G S_G \}.
\]

\( L(G) \) and \( FL(G) \) are called the language and the F-language of \( G \), respectively. \( PL(G) \) denotes the \( P \)-language of \( G \); it arises from \( FL(G) \), after one has dropped all numerical subscripts. Grammars \( G \) and \( G' \) are said to be equivalent (resp. \( P \)-equivalent, \( P \)-equivalent), if \( L(G) = L(G') \) (resp. \( FL(G) = FL(G') \), \( PL(G) = PL(G') \)).

Surely, the notions defined above are quite crucial for the theory of BCG's, as they (or their variants) are for other branches of mathematical linguistics. Below we outline main algebraic methods which are needed to establish
fundamental properties of BCG’s. Actually, one needs universal algebra and combinatorics of trees.

It is useful to see that types can be treated as functor-argument structures over the set $P$, of atomic types; just write $[A, B]_2$ for $A \to B$ and $[B, A]_1$ or $B \leftarrow A$. So, $Tp = FS(P)$ is the set of types. A path (resp. an $F$-path) in structure $X \in FS(V)$ is a sequence $X_0, \ldots, X_n$, of substructures of $X$, such that $X_{i+1}$ is a constituent (resp. the functor) of $X_i$, for all $i < n$; $n$ is the length of this path. The height of $X$ ($h(X)$) is the maximal length of paths in $X$. The $F$-degree of $X$ ($Fd(X)$) is the maximal length of F-paths in $X$. The depth of structure $X$ ($d(X)$) is the minimal length of paths beginning with $X$ and ending with an atom, and the degree of $X$ ($deg(X)$) is the maximal depth of substructures of $X$. Clearly, all $F$-free notions can be defined for phrase structures, as well.

For $L \subseteq FS(V)$, we set:

$$Fd(L) = \sup \{Fd(X) : X \in L\},$$

$$deg(L) = \sup \{deg(X) : X \in L\}.$$ 

The latter notion is also defined for $L \subseteq PS(V)$.

$X[v : Y]$ denotes the substitution of $Y$ for atom $v$ in $X$. Given a lexicon $V$, we set $V_2 = V \cup \{x\}$, where $x$ is a variable not in $V$. Each language $L \subseteq FS(V)$ determines the basic congruence $\sim_L$, in the algebra $FS(V)$, defined as follows: $X \sim_L Y$ iff, for all $Z \in FS(V)$,

$$Z[x : X] \in L \iff Z[x : Y] \in L.$$

The index of $L$ ($\text{ind}(L)$) is the total number of equivalence classes of $\sim_L$. One easily shows that $\sim_L$ is the largest congruence in $FS(V)$ compatible with $L$, that means: for all $X, Y \in FS(V)$,

if $X \sim_L Y$, then $X \in L$ iff $Y \in L$.

The first problem which can be solved by these tools is an algebraic characterization of F-languages of BCG’s. In [11], it has been proven that:

(T1) For $L \subseteq FS(V)$, there is some BCG $G$ such that $L = \text{FL}(G)$ if, and only if, both $\text{ind}(L)$ and $Fd(L)$ are finite.

By $T_P(G)$ we denote the set of all types appearing in lexical assumptions of $G$. The grammar $G$ determines the relation $\sim_G$ on $FS(V_G)$, defined as follows:

$$X \sim_G Y$$

iff, for all $A, X \Rightarrow_G A \iff Y \Rightarrow_G A$.

Clearly, $\sim_G$ is a congruence in $FS(V_G)$ compatible with $FL(G)$. We refer to this relation as the basic congruence determined by $G$. One easily shows the inequalities:

$$\text{ind}(FL(G)) \leq \text{ind}(\sim_G),$$

$$Fd(FL(G)) \leq Fd(T_P(G)),$$

which yield the ‘only if’ part of (T1). For the ‘if’ part, one identifies atomic types with equivalence classes of $\sim_L$ and adds a new atomic type $S$. Languages $L_A$, for $A \in T_P$, are defined by induction on $A$:

(a) $L_p = p$, for atomic $p$; $L_S = L$,

(b) $L_{p \rightarrow B} = \{Y : (3X \in L_p)[X, Y]_2 \in L_B\}$,

(c) $L_{B \rightarrow p} = \{X : (3Y \in L_p)[X, Y]_1 \in L_B\}$.

Actually, we have defined $L_A$ merely for types $A$ of order less than 2, where the order of $A$ ($\text{ord}(A)$) is a nonnegative integer defined as follows:

(a) $\text{ord}(p) = 0$, for atomic types $p$,

(b) $\text{ord}(A \rightarrow B) = \text{ord}(B \leftarrow A) = \max(\text{ord}(B), \text{ord}(A) + 1)$.

One shows that $L_A \neq \emptyset$ only if $Fd(A) \leq Fd(L)$, and consequently, the BCG $G$ defined by the lexical assumptions:

$$v : A \iff v \in L_A$$
is a finite object. This grammar \( G \) satisfies: \( FL(G) = L \). We also have \( \text{ord}(G) \leq 1 \), where \( \text{ord}(G) \) is the maximal order of types in \( Tp(G) \). Consequently:

(T2) Each BCG is \( F \)-equivalent to some BCG \( G' \) with \( \text{ord}(G') \leq 1 \).

To compare BCG’s with Context-Free Grammars (CFG’s), it is useful to characterize \( P \)-languages of BCG’s. Using (T1), one shows [12]:

(T3) For \( L \subseteq PS(V) \), there is some BCG \( G \) such that \( L = PL(G) \) if, and only if, both \( \text{ind}(L) \) and \( \text{deg}(L) \) are finite.

Accordingly, \( P \)-languages of BCG’s is a narrower class than \( P \)-languages of CFG’s: by the result of Thatcher [35], the latter are precisely those \( L \subseteq PS(V) \) for which \( \text{ind}(L) \) is finite. For instance, the CFG defined by production rules \( S \Rightarrow SS \), \( S \Rightarrow a \), \( S \Rightarrow b \) generates the total \( P \)-language \( PS(a, b) \) whose degree is infinite; thus, it is not \( P \)-equivalent to any BCG.

In [4], there is proven the Gaifman theorem: BCG’s are equivalent to CFG’s, that means, both kinds of grammar yield the same class of string languages. This theorem is closely related to the Greibach Normal Form theorem for CFG’s. Original proofs of both theorems use purely combinatorial transformations of one grammar into another one. An algebraic proof of the Gaifman theorem, based on the algebra \( FS(V) \), is given in [17, 16]. First, observe that, for any BCG \( G \), \( \text{deg}(PL(G)) \leq Fd(FL(G)) \) and \( \text{ind}(PL(G)) \) is finite. That yields the first part of the Gaifman theorem: each BCG is \( P \)-equivalent, hence also equivalent, to some CFG. For the second part, fix a CFG \( G \) in the Chomsky Normal Form. Then, \( PL(G) \) is of finite index (by Thatcher’s theorem). By the convention ‘brackets associated to the left’, each phrase structure can uniquely be represented in the form \( vX_1 \ldots X_n \), where \( v \) is an atom and \( n \geq 0 \). The following transformation:

\[
(vX_1 \ldots X_n)^T =
\]

\[
= [v, [X_1^T, [X_2^T, \ldots, [X_{n-1}^T, X_n^T] \ldots]]]
\]

sends each phrase structure \( X \) to a phrase structure \( X^T \) such that \( Fd(X^T) \leq 2 \) and the yield of \( X^T \) equals that of \( X \) (here we identify phrase structures with functor-argument structures whose numerical subscripts always equal 1). We set:

\[
L = PL(G)^T = \{ X^T : X \in PL(G) \}.
\]

The language \( L \subseteq FS(V_G) \) satisfies \( Fd(L) \leq 2 \). It requires a bit of calculation to show that the finiteness of \( PL(G) \) entails the finiteness of \( L \). Then, by (T1), \( L = FL(G) \), for some BCG \( G \). Clearly, \( L(G) = L(G) \), hence \( G \) is equivalent to \( G \).

The global equivalence problem for BCG’s is the question if \( L(G) = L(G') \), for arbitrary BCG’s \( G \) and \( G' \). Since the original proof of the Gaifman theorem yields an effective construction of a BCG \( G \) equivalent to any given CFG \( G \), then the global equivalence problem for CFG’s (which is undecidable [24]) is effectively reducible to that for BCG’s. Consequently, the global equivalence problem for BCG’s is undecidable, and so is the global inclusion problem \( L(G) \subseteq L(G') \). As shown in [13], a further refinement of the methods sketched above provides algorithms for solving global \( F \)-equivalence and \( F \)-inclusion problems for BCG’s and many related problems.

The quotient algebra \( FS(V_G)/ \sim_G \) is denoted \( \text{ALG}(G) \) and called the basic algebra of the BCG \( G \). The quotient operations \( f_1, f_2 \) in the algebra \( \text{ALG}(G) \) are defined in the standard way:

\[
f_1(X/ \sim_G, Y/ \sim_G) = [X, Y]/ \sim_G,
\]
for \( i = 1, 2 \). Here \( X/\sim_G \) denotes the equivalence class of \( X \) with respect to \( \sim_G \). Since \( \sim_G \) is of finite index, then \( \text{ALG}(G) \) is finite. For any BCG \( G \), one can effectively construct a powerset algebra over the set of types which is isomorphic to \( \text{ALG}(G) \).

By \( T(G) \) we denote the set of all subtypes of types from \( Tp(G) \). In the powerset \( P(T(G)) \), we define operations:

\[
F_1(T_1, T_2) = \{ B : (\exists A \in T_1) (B \leftarrow A) \in T_1 \},
\]

\[
F_2(T_1, T_2) = \{ B : (\exists A \in T_1) (A \rightarrow B) \in T_2 \},
\]

for \( T_1, T_2 \subseteq T(G) \). Thus, \( F_1 \) (resp. \( F_2 \)) yields the results of all possible applications of rule \( \text{MP}\leftarrow \) (resp. \( \text{MP}\rightarrow \)) whose left premise is in \( T_1 \) and right premise is in \( T_2 \). The mapping:

\[
h_G(X/\sim_G) = \{ A \in T(G) : X \Rightarrow_G A \}
\]

is well defined, and \( h_G \) is a monomorphism of the algebra \( \text{ALG}(G) \) into the algebra \( (P(T(G)), F_1, F_2) \). The image \( h_G(\text{ALG}(G)) \) is a subalgebra of the latter algebra; this subalgebra will be called the type algebra of \( G \) and denoted \( Tp(G) \). Clearly, \( h_G \) is an isomorphism of \( \text{ALG}(G) \) onto \( Tp(G) \).

For any BCG \( G \), the algebra \( Tp(G) \) can effectively be constructed. For \( Tp(G) \) is the subalgebra of the (finite and effectively given) algebra \( P(T(G)) \) which is generated by the set of all sets:

\[
T_G(u) = \{ A \in T(G) : u \Rightarrow_G A \},
\]

for \( u \in V_G \). Observe that \( u \Rightarrow_G A \) holds if, and only if, \( u : A \) is a lexical assumption of \( G \). Now, many properties of BCG’s can be expressed as properties of their type algebras, and the latter can effectively be verified.

The inclusion \( \sim_G \subseteq \sim_{FL(G)} \) holds for every BCG \( G \). We say that \( G \) is well-formed if \( \sim_G = \sim_{FL(G)} \). For well-formed grammars \( G \), syntactic categories defined as intersubstitutability classes with respect to the F-language of \( G \) are the same as natural equivalence classes determined by the type assignment of \( G \).

(T4) The problem of whether \( G \) is well-formed is decidable.

To prove (T4) we define:

\[
fl(G) = \{ T \in Tp(G) : S_G \subseteq T \}.
\]

One easily shows:

\[
fl(G) = \{ h_G(X/\sim_G) : X \in FL(G) \}.
\]

Thus, \( fl(G) \) represents the F-language \( FL(G) \) in the algebra \( Tp(G) \). Now, \( \sim_G = \sim_{FL(G)} \) holds true if, and only if, the identity is the only congruence in \( Tp(G) \) compatible with \( fl(G) \), and the latter condition admits an effective verification.

In a similar way we prove:

(T5) The global F-equivalence problem for BCG’s is decidable.

Fix two BCG’s \( G_1 \) and \( G_2 \). We assume \( V_{G_1} = V_{G_2} = V \) (otherwise we add new atoms to the lexicons but no new lexical assumptions). Denote \( L_i = FL(G_i) \), for \( i = 1, 2 \). First, observe that \( L_1 = L_2 \) if, and only if, there is an isomorphism \( g \) from the quotient algebra \( FS(V)/\sim_{L_1} \) to the quotient algebra \( FS(V)/\sim_{L_2} \) such that:

\[
g(v/\sim_{L_1}) = v/\sim_{L_2}, \text{ for all } v \in V,
\]

\[
\{ g(X/\sim_{L_1}) : X \in L_1 \} = \\
= \{ X/\sim_{L_2} : X \in L_2 \}.
\]

For the 'only if' direction, take the identity mapping for \( g \). For the 'if' direction, assume there is an isomorphism \( g \) fulfilling the above equalities. By the first equality, we infer:

\[
g(X/\sim_{L_1}) = X/\sim_{L_2},
\]
for all $X \in FS(V)$ ($g$ is a homomorphism). It follows that $\sim_{L_1} \subseteq \sim_{L_2}$, since $g$ is a function, and the converse inclusion is also true, since $g$ is a bijective mapping. Consequently, $\sim_{L_1} = \sim_{L_2}$, and $g$ is the identity mapping. The second equality and the fact that $L_1$ is a union of equivalence classes $X/\sim_{L_1}$, for $X \in L_1$ yield $L_1 = L_2$. Now, the isomorphism condition can be copied in type algebras. Let $\sim_4$ be the largest congruence in $TP(G_i)$ compatible with $fl(G_i)$, for $i = 1, 2$. Then, $FL(G_1) = FL(G_2)$ if, and only if, there is an isomorphism $g'$ from the quotient algebra $TP(G_1)/\sim_1$ to the quotient algebra $TP(G_2)/\sim_2$ such that:

\[
g'(T_{G_1}(v)/\sim_1) = T_{G_2}(v)/\sim_2, \text{ for all } v \in V,
\]

\[
\{g'(T/\sim_1) : T \in fl(G_1)\} = \{T/\sim_2 : T \in fl(G_2)\}.
\]

Clearly, the latter isomorphism condition can effectively be verified.

Analogous methods can be used to find algorithms for many other problems concerning BCG's, for instance, the global P-equivalence problem, the F-inclusion and P-inclusion problems (use an effective construction of a BCG $G$ such that $FL(G) = FL(G_1) \cup FL(G_2)$, and similarly for phrase languages), the problem of rigid stratifiability, a construction of an adequate grammar P-equivalent to a given grammar, a construction of a (restricted) complementation grammar for a given grammar, and so on. The reader is referred to [13, 19] for details.

2 Unification and Learning

The method of unification, extensively used in Logic Programming and Unification Systems in computational linguistics (see PEREIRA and SHIEBER [34], GAZDAR and MELLISH [23]), can be applied to functor-argument structures and types in order to develop quite natural learning procedures for BCG's. The basic learning algorithms of that kind were described in [13, 14, 21], and KANAZAWA [26, 25] elaborated a Gold style learning theory for BCG's, essentially involving these algorithms.

In this section, we briefly outline the algorithms and their applications to the fine algebraic structure of F-languages, generated by BCG's.

Let us recall some basic notions concerning unification. We consider terms (of a first order language) built from constants, variables and function symbols. A substitution is an assignment of terms to variables, and it naturally extends to a mapping from the set of terms to itself. A substitution $\sigma$ is a unifier of a set $T$, of terms, if $\sigma(s) = \sigma(t)$, for all $s, t \in T$. $\sigma$ is a unifier of a family $\{T_1, \ldots, T_n\}$, of sets of terms, if it is a unifier of each $T_i$, for $i = 1, \ldots, n$. A most general unifier (mgu) of a family of sets of terms is a unifier $\sigma$ of this family such that, for every unifier $\alpha$ of this family, there is a substitution $\beta$, such that $\alpha = \beta \sigma$. It is well known that, for any finite family of finite sets of types, one can effectively construct an mgu of this family (if it exists) or prove the nonexistence of any unifier of it (see LLOYD [29], BUSZKOWSKI and PENN [21]). Notice that two mgu's of the same family must be equal up to alphabetic variants.

We assume that atomic types are variables and constants. Thus, the set $T_p$, of all types, can be treated as a set of terms in the above sense. We describe an effective procedure which takes a finite set $L \subseteq FS(V)$ as an input, and returns a 'most general' BCG $G$ such that $L \subseteq FL(G)$.

Fix a nonempty, finite set $L \subseteq FS(V)$. We assign type $S$ to all structures from $L$, and to each occurrence of an argument substructure in a structure from $L$ we assign a different variable. Then, types are assigned to occurrences of functor substructures of structures from $L$.
by the following rules:

\[
\begin{align*}
(F\to) & \quad [X,Y]_2 : B ; X : A , \\
 & \quad Y : A \to B , \\
(F\leftarrow) & \quad [X,Y]_1 : B ; Y : A , \\
 & \quad X : B \leftarrow A .
\end{align*}
\]

Now, we define the so-called general form of \( L(GF(L)) \) as the BCG determined by all assumptions \( v : A \) such that \( v \in V \) and \( A \) has been assigned to \( v \) by the above procedure. We have:

\[
FL(GF(L)) = L.
\]

For any BCG \( G \) and substitution \( \sigma \), let \( \sigma(A) \) denote the BCG determined by the assumptions \( v : \sigma(A) \), for all assumptions \( v : A \) of \( G \). One easily shows:

\[
FL(G) \subseteq FL(\sigma(G)), \text{ for every BCG } G.
\]

We also write \( G_f \subseteq G_g \), if each lexical assumption of \( G_f \) is an assumption of \( G_g \). The basic property of \( GF(L) \) is: for every BCG \( G \), \( L \subseteq FL(G) \) if, and only if, there is a substitution \( \sigma \) such that \( \sigma(GF(L)) \subseteq G \).

For each \( v \in V \), let \( T_L(v) \) be the set of all types \( A \) such that \( v : A \) is a lexical assumption of \( GF(L) \). We set:

\[
T_L = \{ T_L(v) : v \in V \}.
\]

We assume that all sets \( T_L(v) \) are nonempty (otherwise, drop the redundant atoms from \( V \)). Recall that a BCG \( G \) is rigid, if, for any \( v \in V \), there is at most one type \( A \) such that \( v : A \) is an assumption of \( G \). We define \( RG(L) = \sigma(GF(L)) \), where \( \sigma \) is an mgu of \( T_L \). Thus, \( RG(L) \) is a rigid BCG, and \( L \subseteq FL(RG(L)) \). The following theorems, proven in [21], show that \( RG(L) \) is a most general rigid BCG \( G \) such that \( L \subseteq FL(G) \).

(T6) For any nonempty, finite \( L \subseteq FS(V) \), the following conditions are equivalent: (a) \( L \subseteq FL(G) \), for some rigid BCG \( G \), (b) the family \( T_L \) is unifiable.

(T7) For any rigid BCG \( G \), \( L \subseteq FL(G) \) if, and only if, there is a substitution \( \alpha \) such that \( \alpha(RG(L)) \subseteq G \).

To give a simple example, we consider the set \( L \) consisting of two structures:

\[
\begin{align*}
\{ \text{Joan, smiles}, & \}, \\
\{ \text{Joan, smiles, charmingly}, & \}.
\end{align*}
\]

According to the procedure described above, we assign type \( S \) to these two structures, variables \( x, y \) to the first and the second occurrence of 'Joan', respectively, and variable \( z \) to the second occurrence of 'smiles'. By rule \( F\to \), we derive:

\[
\begin{align*}
\text{smiles: } & x \to S, \\
\text{smiles, charmingly: } & y \to S, \\
\text{charmingly: } & z \to (y \to S).
\end{align*}
\]

So, \( GF(L) \) is defined by the following assumptions:

\[
\begin{align*}
\text{Joan: } & x, y, \text{smiles: } x, x \to S, \\
\text{charmingly: } & z \to (y \to S).
\end{align*}
\]

The family \( T_L \) is unifiable; \( \sigma(y) = x, \sigma(z) = x \to S \) is an mgu. Consequently, \( RG(L) \) exists and is given by:

\[
\begin{align*}
\text{Joan: } & x, \text{smiles: } x \to S, \\
\text{charmingly: } & (x \to S) \to (x \to S).
\end{align*}
\]

It is natural to interpret \( x = PN \), hence 'smiles' is of type \( VP \), and 'charmingly' is of type \( VP \to VP \), of Adverb. \( RG(L) \) generates the infinite F-language which consists of structures:

\[
\begin{align*}
\{ \text{Joan, smiles}, & \}, \\
\{ \text{Joan, smiles, charmingly}, & \}, \\
\{ \text{Joan, smiles, charmingly}, & \}, \\
\{ \text{Joan, smiles, charmingly}, & \}, \\
\{ \text{Joan, smiles, charmingly}, & \}, \ldots
\end{align*}
\]
It follows from (T7) that the latter F-language is contained in \( FL(G) \), for every rigid BCG \( G \) such that \( L \subseteq FL(G) \).

An F-language \( L \subseteq FS(V) \) is said to be rigid if \( L = FL(G) \), for some rigid BCG \( G \). KANAZAWA [26, 25] proves that the class of rigid F-languages possesses the finite elasticity, and consequently, there exists a learning function for this class. The same holds for the class of string languages, generated by rigid BCG’s, and the class of string languages, generated by BCG’s which assign at most \( k \) types to each lexical atom, for any \( k \geq 1 \). A computable learning function can be defined by an adaptation of the above algorithm. To prove the finite elasticity of the class of rigid F-languages, Kanazawa establishes the ascending chain condition (ACC) for this class: there is no infinite chain \( L_0 \subset L_1 \subset L_2 \subset \ldots \), of finite F-languages over a finite lexicon.

These results enable us to say more about the fine structure of rigid F-languages \( L \subseteq FS(V) \). Let \( R \) denote the class consisting of the latter F-languages and the total F-language \( FS(V) \) (which is not rigid, by (T1)).

For any set \( L \subseteq FS(V) \), we define:

\[
C(L) = \bigcap \{ L' \in R : L \subseteq L' \}.
\]

We prove:

(T8) For any set \( L \subseteq FS(V) \), \( C(L) \in R \). Further, the operator \( C \) satisfies Tarski’s conditions:

(i) \( L \subseteq C(L) \),
(ii) if \( L_1 \subseteq L_2 \), then \( C(L_1) \subseteq C(L_2) \),
(iii) \( C(C(L)) = C(L) \),
(iv) for any \( L \subseteq FS(V) \), there is a finite set \( L' \subseteq L \) such that \( C(L) = C(L') \).

We prove the first part of (T8). Fix a set \( L \subseteq FS(V) \). If there is no \( L' \in R \) such that \( L \subseteq L' \), then \( C(L) = FS(V) \in R \). Otherwise, fix \( L' \in R \) such that \( L \subseteq L' \). If \( L \neq \emptyset \) is finite, then \( RG(L) \) exists (use (T6)), and \( FL(RG(L)) \) is the least rigid F-language containing \( L \) (use (T7)), hence \( C(L) = FL(RG(L)) \in R \). If \( L = \emptyset \), then \( C(L) = \emptyset \in R \). So, assume \( L \) is infinite. Then, \( L \) is the join of an ascending chain \( L_0 \subset L_1 \subset \ldots \), of finite F-languages. By (T6), \( RG(L_n) \) exists, for all \( n \geq 0 \); we denote \( L'_n = FL(RG(L_n)) \). By (T7), we obtain:

\[
L'_0 \subseteq L'_1 \subseteq L'_2 \subseteq \ldots,
\]

hence, by (ACC), there is an integer \( k \geq 0 \) such that \( L'_n = L'_k \), for all \( k \geq n \). Clearly \( C(L) = L'_k \in R \).

Conditions (i)-(iii) are obvious. To prove (iv), fix \( L \subseteq FS(V) \). We may assume that \( L \) be infinite. If \( L \subseteq L' \), for some rigid \( L' \), then we proceed as above; we have \( C(L) = L'_k = C(L_k) \). So, assume there is no rigid F-language \( L' \) such that \( L \subseteq L' \). Then, \( C(L) = FS(V) \). We must show that \( C(L^\sim) = FS(V) \), for some finite \( L^\sim \subseteq L \). Suppose the contrary. Then, for every finite \( L^\sim \subseteq L \), there is a rigid \( L'' \) such that \( L^\sim \subseteq L'' \). Consequently, \( RG(L^\sim) \) exists. As above, choose an ascending chain \( L_0 \subset L_1 \subset \ldots \), of finite sets, whose join equals \( L \). We define \( L_n \), for all \( n \geq 0 \), as above. By (ACC), there is \( k \geq 0 \) such that \( L'_n = L'_k \), for all \( n \geq k \). Then, \( L \subseteq L'_k \), which contradicts the assumption.

The operator \( C \) is a natural grammatical consequence operator. The deductively closed sets \( L = C(L) \) are precisely the rigid F-languages and the total F-language \( FS(V) \). It would be interesting to find an axiomatization of this operator by means of ‘ inference rules’ defined on functor-argument structures.

Rigid BCG’s and rigid F-languages are typical for the artificial languages of formal logic and mathematics (but they are also useful technical tools for studying general properties of categorial grammars). Syntactic and semantic ambiguities of natural language are reflected by nonrigid grammars. A nonrigid version of the algorithm described above has been elaborated in [21].
The key notion is optimal unification. Let $\mathcal{T}$ be a family of sets of terms and $\sigma$ be a substitution. $\text{Ker}(\sigma)$ is the relation which holds between terms $s$ and $t$ iff $\sigma(s) = \sigma(t)$. We define:

$$T/\sigma = \{[t] \cap T : t \in T \}, \text{ for } T \in \mathcal{T},$$

$$\mathcal{T}/\sigma = \bigcup\{T/\sigma : T \in \mathcal{T}\}.$$ 

$\sigma$ is called an optimal unifier (ou) of the family $\mathcal{T}$, if it satisfies the conditions:

1. (OU.1) $\sigma$ is an mgu of $T/\sigma$,
2. (OU.2) for all $T \in \mathcal{T}$, $s, t \in T$, if $\sigma(s) \neq \sigma(t)$, then the set $\{\sigma(s), \sigma(t)\}$ is not unifiable.

Intuitively, an ou for $\mathcal{T}$ is a most general substitution which unifies the family $\mathcal{T}$ as far as possible. For every nonempty family $\mathcal{T}$, of nonempty, finite sets of terms, one can effectively find finitely many ou’s of $\mathcal{T}$ (they are all ou’s of $\mathcal{T}$ up to alphabetic variants). We write $\sigma \leq_{\mathcal{T}} \sigma'$ if, for every $T \in \mathcal{T}$, the cardinality of $T/\sigma$ is not greater than that of $T/\sigma'$. A minimal unifier (mu) for $\mathcal{T}$ is an ou for $\mathcal{T}$ which is $\leq_{\mathcal{T}}$-minimal in the set of all ou’s of $\mathcal{T}$. By the above, for every nonempty, finite family $\mathcal{T}$, of nonempty, finite sets of terms, one can effectively find all mu’s for $\mathcal{T}$ (up to alphabetic variants).

Let $\mathcal{G}(L)$ be the family of all BCG’s $\sigma(GF)(L)$ such that $\sigma$ is an mu for $\mathcal{T}_L$. The following theorem, proven in [21], shows that $\mathcal{G}(L)$ contains precisely the ‘most general’ minimal BCG’s $G$ such that $L \subseteq FL(G)$. Here, ‘minimality’ is defined with respect to the following relation: $G \leq G'$ iff, for all $v \in V$, the cardinality of $T_G(v)$ is not greater than that of $T_{G'}(v)$.

(T9) Let $L \subseteq FS(V)$ be a finite set. The following conditions are equivalent:
(a) $G$ is minimal in the class of grammars $G'$ such that $L \subseteq FL(G')$, (b) there are $G' \in \mathcal{G}(L)$ and a substitution $\alpha$ such that $G = \alpha(G')$.

Clearly, if $\mathcal{T}_L$ is unifiable, then $RG(L)$ is the only member of $\mathcal{G}(L)$, and minimal grammars $G$ such that $L \subseteq FL(G)$ are rigid. Therefore, the nonrigid procedure is a ‘conservative’ generalization of the rigid one. Unlike the latter, the former always yields an outcome grammar.

Marciniec [30] studies more general versions of the above procedures in which input data can be of the form $X_i : A_i, i = 1, \ldots, m$ (positive data) as well as non-$Y_j : B_j, j = 1, \ldots, n$ (negative data). Roughly, negative data restrict the class of admissible substitutions, and the ‘positive’ procedures, described above, are relativized to this restricted class. Thus, the role of negative data is not merely to sieve out ‘wrong’ outcomes of the positive procedure, but they essentially influence the positive procedure.

### 3 Residuated Algebras

In the last section we pass from absolutely free algebras of structure trees to algebras corresponding to stronger deductive systems applied in categorial grammars. The central notion is residuation.

Types are formed out of atomic types by means of two conditionals $\to$, $\leftarrow$ and product $\circ$. $\Gamma, \Delta$ will denote finite strings of types. Let $\Gamma \vdash A$ mean: there is an MP-deduction of $A$ with yield $\Gamma$. The deduction system of BCG’s can be defined as a sequential system with axioms:

(Id) $\text{A} \vdash \text{A}$, and inference rules:

(E$\rightarrow$) if $\Gamma \vdash A$ and $\Delta \vdash A \to B$, then $\Gamma, \Delta \vdash B$,

(E$\leftarrow$) if $\Gamma \vdash B \leftarrow A$ and $\Delta \vdash A$, then $\Gamma, \Delta \vdash B$. 


The Lambek calculus \( \mathbf{L} \) results from completing the above elimination rules by introduction rules:

(\text{I\text{\rightarrow}}) if \( \Delta \vdash A, \Gamma \vdash B \), then \( \Gamma \vdash A \rightarrow B \),

(\text{I\text{\leftarrow}}) if \( \Gamma, A \vdash B \), then \( \Gamma \vdash B \leftarrow A \),

where \( \Gamma \neq \emptyset \) (dropping this constraint leads to the stronger system \( \mathbf{L}^1 \)). The rules for product are:

(\text{Eo}) if \( \Delta \vdash A \circ B \) and \( \Gamma, A, B, \Gamma' \vdash C \),
then \( \Gamma, \Delta, \Gamma' \vdash C \),

(\text{Io}) if \( \Gamma \vdash A \) and \( \Delta \vdash B \),
then \( \Gamma, \Delta \vdash A \circ B \).

Both \( \mathbf{L} \) and \( \mathbf{L}^1 \) are closed under the cut rule:

(\text{CUT}) if \( \Gamma, A, \Gamma' \vdash B \) and \( \Delta \vdash A \), then \( \Gamma, \Delta, \Gamma' \vdash B \).

Intuitively, rules (\text{I\text{\rightarrow}}), (\text{I\text{\leftarrow}}) enable us to employ hypothetical reasoning in categorical grammars. For instance, given the lexical assumption ‘John: \( PN \)’ and the derivable pattern:

\[ PN, PN \rightarrow S \vdash S, \]

one infers the Montague Type Raising principle:

\[ PN \vdash S \rightarrow (PN \rightarrow S) \equiv NP, \]

which lifts up Proper Nouns to the type of Noun Phrase. As usual for Natural Deduction, the latter derivation can be represented by the lambda term:

\[ \lambda S PN_\rightarrow : (x PN_\rightarrow \rightarrow S) \]

which means that the constant \( j \) (standing for ‘John’) is transformed into the family of predicates holding for it. The reader is referred to van Benthem [5, 6] for a thorough account of Natural Deduction and the lambda calculus in categorial semantics, and to [15] for a formal analysis of different deduction systems connected with linguistically relevant fragments of the lambda calculus (see also Wansing [36] for a version of the lambda calculus appropriate for directional types).

Here we focus on algebraic structures naturally modelling Natural Deduction systems. A residuated semigroup (r. semigroup) is a structure \( \mathcal{M} = (M, \circ, \rightarrow, \leq) \) such that \( (M, \circ) \) is a semigroup, \( \leq \) is a partial ordering on \( M \), and \( \rightarrow, \leq \) are binary operations on \( M \), satisfying the equivalences:

\[ b \leq a \rightarrow c \text{ iff } a \circ b \leq c \text{ iff } a \leq c \leftarrow b, \]

for all \( a, b, c \in M \).

We describe a general powerset construction of residuated semigroups. Let \( \mathcal{A} = (A, \cdot) \) be a semigroup. In the powerset \( P(A) \) we define operations \( \circ, \rightarrow, \leq \) in the following way:

\[ X \circ Y = \{ a \cdot b : a \in X, b \in Y \}, \]

\[ X \rightarrow Y = \{ c \in A : (\forall a \in X) a \cdot c \in Y \}, \]

\[ X \leftarrow Y = \{ c \in A : (\forall b \in Y) c \cdot a \in X \}, \]

for \( X, Y \subseteq A \). \( (P(A), \circ, \rightarrow, \leq) \) is a residuated semigroup; we refer to it as the powerset r. semigroup over the semigroup \( A \). If \( A = V^+ \) is the free semigroup of nonempty finite strings over the lexicon \( V \), then \( P(A) \) is the r. semigroup of languages over \( V \).

More general structures are based on frames \( (U, R) \) such that \( U \) is a nonempty set and \( R \) is a ternary relation on \( U \). In \( P(U) \) one defines operations \( \circ, \rightarrow, \leq \) as follows:

\[ X \circ Y = \{ c : (\exists a \in X)(\exists b \in Y) R(a, b, c) \}, \]

\[ X \rightarrow Y = \{ b : (\forall a, c)(a \in X, R(a, b, c) \text{ imply } c \in Y) \}, \]

\[ X \leftarrow Y = \{ a : (\forall b, c)(b \in Y, R(a, b, c) \text{ imply } c \in X) \}, \]
for $X, Y \subseteq U$. Following Dunn [22], we call the r. semigroup $(P(U), \circ, \Rightarrow, \Leftarrow, \subseteq)$ the concrete r. semigroup over the frame $(U, R)$.

Clearly, the powerset r. semigroup over $(A, \cdot)$ is the concrete r. semigroup over $(A, R)$, where $R(a, b, c)$ holds iff $a \cdot b = c$. While ternary frames are characteristic of Kripke style semantics for relevant logics, powerset models are more in the style of universal algebra and naturally related to the algebra of languages.

An assignment of types in an r. semigroup is defined in the standard way. By a model we mean a pair $(M, \mu)$ such that $M$ is an r. semigroup and $\mu$ is an assignment of types in $M$. The sequent $A_1, \ldots, A_n \vdash A$ is true in this model, if:

$$\mu(A_1) \circ \cdots \circ \mu(A_n) \leq \mu(A),$$

and it is valid in $M$, if it is true in all models $(M, \mu)$. If the underlying semigroup is a monoid $(M, \circ, 1)$, then the sequent $\vdash A$ is true in $(M, \mu)$, if $1 \leq \mu(A)$, and the remaining notions are defined, as above. The powerset r. monoid over the monoid $(A, \cdot, 1)$ is $(P(A), \circ, \Rightarrow, \Leftarrow, \subseteq, \{1\}, \subseteq)$.

Let $\Sigma$ be a set of sequents. $L(\Sigma)$ (resp. $L_1(\Sigma)$) denotes the system $L$ (resp. $L_1$) with (CUT), enlarged with all sequents from $\Sigma$ as new axioms. Basic completeness theorems for the Lambek calculus are the following:

(T10) Sequents derivable in $L(\Sigma)$ are precisely those which are valid in all powerset r. semigroups over arbitrary semigroups.

(T11) Sequents derivable in $L_1(\Sigma)$ are precisely those which are valid in all powerset r. monoids over arbitrary monoids.

(T12) If $\Sigma$ consists of product-free sequents, then product-free sequents derivable in $L(\Sigma)$ are precisely those which are valid in all powerset r. semigroups over free semigroups.

(T13) If $\Sigma$ consists of product-free sequents, then product-free sequents derivable in $L_1(\Sigma)$ are precisely those which are valid in all powerset r. monoids over free monoids.

(T14) Sequents derivable in $L$ (resp. $L_1$) are precisely those which are valid in all powerset r. semigroups (resp. monoids) over free semigroups (resp. monoids).

(T15) Product-free sequents derivable in $L$ (resp. $L_1$) are precisely those which are valid in all powerset r. semigroups (resp. monoids) over finite semigroups (resp. monoids).

For each of these theorems, the 'only if' direction (soundness) is easy: axioms $\text{(Id)}$ are true in all models, and inference rules preserve the truth. Theorems (T12), (T13) were first proven in [7]. One constructs the canonical model $P(\Sigma^+)\downarrow$, where $\Sigma$ is the set of types. The canonical assignment $f$ is defined as follows:

$$f(p) = \{\Gamma : \Gamma \vdash_L p\}.$$  

By induction on type $A$, one proves:

$$f(A) = \{\Gamma : \Gamma \vdash_L A\},$$

for all product-free types $A$. That yields:

$$A_1, \ldots, A_n \vdash_L A \text{ iff } f(A_1) \circ \cdots \circ f(A_n) \subseteq f(A).$$

The 'only if' direction holds by soundness, and the 'if' direction by (Id) and the latter equality. For $L_1$, the argument is similar.

Theorems (T10), (T11) were proven in [10]. They require a more sophisticated construction. Roughly, one affixes to $L$ new rules of the form:

(D) if $\Gamma \vdash A \circ B$, then $(\Gamma, 1, A \circ B) \vdash A$ and $(\Gamma, 2, A \circ B) \vdash B$,  

which enable us to decompose each $\Gamma$ such that $\Gamma \vdash A \circ B$ is derivable into two terms:

$$\Gamma = (\Gamma, 1, A \circ B) \cdot (\Gamma, 2, A \circ B)$$

such that $(\Gamma, 1, A \circ B) \vdash A$ and $(\Gamma, 2, A \circ B) \vdash B$ are derivable. Of course, the language of $\mathcal{L}$ must be extended to admit terms of the above form. For the extended system, one constructs the canonical model, following the lines above, but now the underlying semigroup is not a free semigroup.

Theorem (T14) has been proven by PENTUS [33] by an approximation of a model with partial models; the proof uses many combinatorial tools. Theorem (T15), proven in [8, 18], establishes the finite model property for Lambek systems. One uses generalized powerset models over free semigroups. For a set $K \subseteq \Sigma^+$ such that $K \neq \emptyset$ is closed under nonempty subintervals, one defines the relativized powerset operations $X \circ Y$, $X \Rightarrow Y$, $X \Leftarrow Y$, for $X, Y \subseteq K$:

$$X \circ Y = \{ab \in K : a \in X, b \in Y\},$$

$$X \Rightarrow Y = \{c \in K : (\forall a \in X)(ac \notin K \lor ac \in Y)\},$$

$$X \Leftarrow Y = \{c \in K : (\forall a \in Y)(ca \notin K \lor ca \in X)\}.$$  

Then, $(P(K), \circ, \Rightarrow, \Leftarrow, \subseteq)$ is an r. semigroup which is isomorphic to the powerset r. semigroup $P(K')$, where $K' = K \cup \{x\}$, $x$ being a new element which takes the part of all strings not in $K$. One shows that $\mathcal{L}$ (resp. L1) is complete with respect to all models of that kind with a finite $K$. Details of the proof are somewhat cumbersome.

If powerset models are replaced with concrete models, then analogues of theorems (T10), (T11) can be obtained by a modification of the well known Stone representation theorem for Boolean algebras. One shows:

(T16) Each residuated semigroup is embeddable into a concrete residuated semigroup.

This representation theorem has been proven in DUNN [22] in the following way. Let $\mathcal{M}$ be an r. semigroup. A set $\nabla \subseteq \mathcal{M}$ is called a cone if $a \in \nabla$ and $a \leq b$ entails $b \in \nabla$. Take $U$ equal to the set of cones, and define a ternary relation $R$ on $U$:

$$R(\nabla_1, \nabla_2, \nabla_3) \text{ iff }$$

$$(\forall a, b)(a \in \nabla_1, a \Rightarrow b \in \nabla_2 \text{ imply } b \in \nabla_3).$$

Then, the mapping $h(a) = \{\nabla : a \in \nabla\}$ is a monomorphism of the r. semigroup $\mathcal{M}$ into the concrete r. semigroup $P(U)$.

Interestingly, if $R$ is replaced with the binary operation:

$$\nabla_1 \cdot \nabla_2 = \{b : (\exists a \in \nabla_1)a \Rightarrow b \in \nabla_2\},$$

then $(U, \cdot)$ is a semigroup, and one can consider the powerset r. semigroup $P(U)$ instead of the concrete r. semigroup. Yet, the mapping $h$, defined above, is merely a homomorphism with respect to $\Rightarrow, \Leftarrow$, and $\subseteq$, but not $\circ$. Thus, using the method above, one can show that each $\Rightarrow, \Leftarrow$-reduct of an r. semigroup is embeddable into a powerset r. semigroup. To prove the full representation theorem:

(T17) Each residuated semigroup is embeddable into a powerset residuated semigroup,

one needs decomposition methods of the proof of theorems (T10), (T11).

Other representation and completeness theorems for Lambek systems have been obtained in ANDRÉKA and MIKULÁŠ [2], PANKRATIEV [32] and KURTONINA [28] with respect to algebras of binary relations. These algebras fit an interesting interpretation of Lambek systems as logics of programs (procedures).
Let us pass to categorial grammars. \textit{Lambek Categorial Grammars (LCG's)} result from enriching BCG's with the full strength of the Lambek calculus (here, we restrict ourselves to product-free types). For any type \( A \), the LCG \( G \) determines the language \( L_A(G) \subseteq V_G^+ \) which consists of all strings on \( V_G \), being assigned type \( A \) by \( G \). On the other hand, given languages \( L_G(p) \), for atomic types \( p \), languages \( L^A(G) \) are uniquely determined by powerset operations in the algebra \( P(V_G^+) \):

\[
L^p(G) = L_p(G), \text{ for atomic } p,
\]

\[
L^{A\to B}(G) = L^A(G) \Rightarrow L^B(G),
\]

\[
L^{B\Leftarrow A}(G) = L^B(G) \Leftarrow L^A(G).
\]

There arises a natural problem of compatibility of languages \( L_A(G) \) and \( L^A(G) \). As shown in [7], no Lambek grammar is complete, that means, \( L^A(G) \neq L_A(G) \) holds, for all types \( A \). A weaker condition is correctness: \( L_A(G) \subseteq L^A(G) \), for all types \( A \). It is easy to show that \( G \) is correct, if \( v : A \) in the sense of \( G \) entails \( v \in L^A(G) \), for all lexical atoms \( v \) and types \( A \). Each Lambek grammar can be extended to a correct Lambek grammar by introducing new lexical atoms: for any type \( B \) which appears in lexical assumptions as a subtype, one introduces a new atom \( v_B \) and a new assumption \( v_B : B \). The new grammar \( G' \) is conservative with respect to the initial grammar \( G \):

\[
L_A(G) = L_A(G') \cap V_G^+.
\]

As shown in [20], correct Lambek grammars are precisely those whose language family \( L_p(G) \), for atomic types \( p \), is a minimal solution of the lexical postulates \( v : A \). Thus, for correct Lambek grammars, languages generated by the grammar can be characterized in Algol like style, as the minimal languages satisfying a system of postulates.

Everything we have considered above can be extended toward abstract algebras \( (A,F) \), where \( F \) is a set of operations in the universe \( A \). With each operation \( f \), of arity \( n \geq 1 \), we associate residuation operations \( f/i \), \( i = 1, \ldots, n \), satisfying the equivalence:

\[
f(\ldots,a_i,\ldots) \leq a \iff a_i \leq (f/i)(\ldots,a_i,\ldots),
\]

where \( \leq \) is a partial ordering on \( A \). If the residuation operations exist, for each \( f \in F \), then the structure \( (A,F,\leq) \) is called a residuated algebra. Given an algebra \( (A,F) \), the powerset residuated algebra over this algebra is defined as the set \( P(A) \) with \( \subseteq \) taking the part of partial ordering, and operations \( f \) and \( f/i \) given by:

\[
f(X_1,\ldots,X_n) =
\]

\[
\{ f(a_1,\ldots,a_n) : (\forall i) a_i \in X_i \},
\]

\[
(f/i)(X_1,\ldots,X_n) =
\]

\[
\{ a_i : (\forall j \neq i) (\forall a_j \in X_j) f(a_1,\ldots,a_n) \in X_i \}.
\]

Residuated algebras as general frames for Lambek style categorial grammars have been proposed in [9, 16]. Now, they correspond to multi-modal systems considered in Moortgat [31] and Kurtonina [28] which account for different composition modes in natural language. The \textit{Generalized Lambek Calculus GL} (in the Natural Deduction form) is based on axioms (Id) and rules:

\[(Ef)\text{ if } \Delta \vdash f(A_1,\ldots,A_n), \text{ and } \Gamma[f[A_1,\ldots,A_n]] \vdash C, \text{ then } \Gamma[\Delta] \vdash C,\]

\[(If)\text{ if } \Gamma_i \vdash A_i, i = 1,\ldots,n, \text{ then } f[\Gamma_1,\ldots,\Gamma_n] \vdash f(A_1,\ldots,A_n),\]

\[(Efi)\text{ if } \Gamma_i \vdash (f/i)(A_1,\ldots,A_n) \text{ and } \Gamma_j \vdash A_j, \text{ for all } j \neq i, \text{ then } f[\Gamma_1,\ldots,\Gamma_n] \vdash A_i,\]

\[(If/i)\text{ if } \Gamma[A_1,\ldots,\Gamma_n] \vdash A_i, \text{ then } \Gamma \vdash (f/i)(A_1,\ldots,A_n),\]
(CUT) if \( \Gamma[A] \vdash B \) and \( \Delta \vdash A \),
then \( \Gamma[\Delta] \vdash B \).

Again, the rule (CUT) can be eliminated from
the pure system GL. Now, GL is complete
with respect to residuated algebras (that is an
easy application of Lindenbaum algebras), and
it is also complete with respect to powerset
residuated algebras. The latter theorem has
been proven in Kołowska-Gawiejnowicz
[27] by means of the following representation
theorem: each residuated algebra is embed-
dable into a powerset residuated algebra. The
proof goes by a generalization of the proof of
(T10) in [10].

References

[1] K. Ajdukiewicz, Die syntaktische Kon-
nexität, Studia Philosophica 1, (1935), 1-
27.

[2] H. Andréska and S. Mikulás, Lambek Cal-
culus and Its Relational Semantics, JoLLI

[3] Y. Bar-Hillel, A quasi-arithmetical nota-
tion for syntactic description, Language
29, (1953), 47-58.

[4] Y. Bar-Hillel, C. Gaifman and E. Shamir,
On categorial and phrase structure gram-
155-166.

[5] J. van Bentham, Essays in Logical Se-

Categories, Lambdas and Dynamic Logic

[7] W. Buszkowski, Compatibility of a Cate-
gorical Grammar with an Associated Cate-
gory System, Zeitschrift für mathema-
tische Logik und Grundlagen der Math-
ematik 28, (1982), 229-238.

in the Theory of Syntactic Categories, ibidem
28, (1982), 539-548.

[9] W. Buszkowski, Fregean Grammar and
Residuated Semigroups, in: G. Wechsung
(ed.), Frege Conference 1984, Akademie-

[10] W. Buszkowski, Completeness Results for
Lambek Syntactic Calculus, Zeitschrift für
mathematische Logik und Grundlagen

[11] W. Buszkowski, Typed Functorial Lan-
guages, Bull. Polish Academy of Sciences
Math. 34, (1986), 495-505.

[12] W. Buszkowski, Generative Capacity of
Nonassociative Lambek Calculus, ibidem
34, (1986), 507-516.

Classical Categorial Grammars, ibidem

[14] W. Buszkowski, Discovery Procedures for
Categorial Grammars, in: E. Klein and
J. van Bentheim (eds.), Categories, Poly-
normphism and Unification, University of
Amsterdam, 1987.

J. T. Srzednicki (ed.), Initiatives in Logic,

[16] W. Buszkowski, Generative Power of Cate-
gorical Grammars, in: R. T. Oehrle, E.
Bach and D. Wheeler, Categorial Gram-
mars and Natural Language Structures,

[17] W. Buszkowski, Gaifman’s Theorem on
Categorial Grammars Revisited, Studia


A VARIANT OF A UNIVERSAL METAGRAMMAR OF CONCEPTUAL STRUCTURES. ALGEBRAIC SYSTEMS OF CONCEPTUAL SYNTAX

Vladimir A. Pomichov

Active Member of the New York Academy of Sciences
Department of Discrete Mathematics
Faculty of Mechanics and Mathematics
Moscow State University
Vorobyovy Hills, 119899 Moscow, Russia.
E-mail: vaf@nw.math.msu.su
and Faculty of Applied Mathematics
Moscow State Institute of Electronics and Mathematics (Technical University)
B. Tryokhsvyatitskoye pr. 3/2, 109020 Moscow, Russia
E-mail: root@onti.miem.msk.su; Fax: +7(095) 916 28 07

ABSTRACT

A new logic-algebraic theory is set forth; it is called the theory of K-calculuses, algebraic systems of conceptual syntax, and K-languages (the KCL-theory). The theory describes structured meanings of highly complicated (very likely, arbitrarily complicated) real discourses pertaining to science, technology, medicine, business, law, etc. The KCL-theory is a central constituent of Integral Formal Semantics (IFS) - a powerful approach to mathematical studying semantics and pragmatics of natural language (NL). A part of the KCL-theory may be interpreted as a variant of a Universal Metagrammar of Conceptual Structures.

The KCL-theory enables us to build formal analogues both of sentences and complicated discourses. It should be stressed that it allows us to build formal analogues (of the conceptual level) not only of assertions and narrative texts but also of infinitives with dependent words and of questions.

It is shown that the new theory very considerably enriches the stock of mathematical tools for studying semantics of NL. The conclusion is grounded that the KCL-theory creates the preconditions for developing a mathematical theory of NL use, or a mathematical theory of NL communication. Such a theory is understood as a collection of interrelated mathematical models useful for designing NL-processing systems.

In particular, the KCL-theory opens new large prospects for the theory of full-text data bases, the theory of automated transforming NL-specifications of tasks into formal specifications, and for the theory of hybrid knowledge representation systems (terminological knowledge representation languages).

1. INTRODUCTION

In the nineties, the practice puts forward new demands to the theory of natural-language-processing systems (NLPSs). First of all, these new demands are caused by the tasks of constructing the following kinds of NLPSs: (a) textual (or full-text) data bases (TDBs); (b) computer systems extracting information from natural-language texts (patents, scientific articles, text-books, legal sources, etc.) in order to enrich and up-date knowledge bases of applied intelligent systems; (c) systems capable to transform natural-language specifications of program complexes into their formal specifications; (d) similar systems which are able to transform natural-language specifications of technical objects to be designed into the formal specifications of such objects and to analyse such formal specifications (in order to check the consistency and completeness of these specifications) as the key components of advanced CAD-CAM systems; (e) natural-language interfaces to large knowledge bases or to autonomous intelligent agents. It is known quite well that in many fields of human activity one uses varied mathematical methods and models for effective designing complicated technical systems. For instance, aerodynamics and hydrodynamics help to design airplanes and ships, respectively. But effective mathematical methods are needed also for effective designing complicated program systems.
In the sixties and seventies, the necessity to construct compilers and translators of high-level programming languages lead to the creation of the theory of formal grammars, languages, and translators. There is some similarity between that situation and the situation taking place nowadays in the field of NLPSs. It appears that the practice demands to develop a mathematical theory of natural language (NL) use as a collection of interrelated mathematical means, methods, and models useful for effective designing NLPSs. It should be noted that such a theory must help to overcome difficulties of logical character encountered by the designers of NLPSs. The expediency of creating a mathematical theory of NL use, or a mathematical theory of NL-communication is grounded, in particular, in (Fomichev 1983, 1984; Fomichev 1988a, 1988b, 1992, 1993a; Martin-Vide 1993). In (Fomichev 1993a, 1993b, 1994), the term "Mathematical Linguocybernetics" is introduced for designating a theory of the kind.

The main aim of this paper is to show that we have already today an appropriate mathematical basis for carrying out successful studies on constructing a mathematical theory of NL use, or Mathematical Linguocybernetics. Such a basis is provided at least by Integral Formal Semantics (IFS) of NL. It is a powerful and flexible mathematical approach to studying semantics and pragmatics of NL; this approach is represented in approximately 40 my scientific publications in Russian and English.

2. WIDELY-APPLICABLE METAGRAMMARS OF CONCEPTUAL STRUCTURES FROM THE STANDPOINT OF INTEGRAL FORMAL SEMANTICS OF NATURAL LANGUAGE

The term "Integral Formal Semantics" was introduced in (Fomichev 1993a, 1993b), and basic informal principles of this scientific approach were published in (Fomichev 1993a, 1993b, 1994). However, the first version of IFS was developed in fact in 1978 - 1983. By 1984, a great volume of theoretical results was obtained; its major part is represented in (Fomichev 1981, 1982; Fomichev 1983, 1984). A number of other references may be found in (Fomichev 1994).

The set of basic philosophical ideas of IFS includes, in particular, the following principles.

1. The chief task of researches on the formalization of semantics of natural language (NL) is to be the development of formal models of NLPSs and of such subsystems of NLPSs which belong to their "semantic" components.

The structure of such models of several kinds is suggested, in particular, in (Fomichev 1982, 1992).

2. The studies are to be oriented towards considering not only assertions but also commands and questions which may be inputs of NLPSs.

3. The basis of researches is to be a formal model reflecting many peculiarities of semantic structures of sentences and discourses of arbitrary great length and providing the description of some class Lang of formal languages being convenient for building semantic representations (SRs) of NL-texts in a large spectrum of applications and on different levels of representation. 1992).

Let's call such a model a Widely-Applicable Metagrammar of Conceptual Structures, or a Widely-Applicable Conceptual Metagrammar (CM).

A Widely-Applicable CM should enable us to build formal semantic analogues of sentences and discourses; hence the expressive power of formal languages determined by the model may be very close to the expressive power of NL (if we take into account the surface semantic structure of NL-texts). Besides, a CM is to be convenient for describing various knowledge about the world (Fomichev 1981, 1982; Fomichev 1983, 1984; Fomichev 1988a, 1988b, 1992, 1993b, 1994).

If a model is convenient for describing arbitrary conceptual structures of NL-texts and representing arbitrary knowledge about the world, we'll say about a Universal Metagrammar of Conceptual Structures, or a Universal Conceptual Metagrammar (UCM).
The reason to say about a metagrammar but not about a grammar is as follows. A grammar of conceptual structures is to be a formal model dealing with elements directly corresponding to some basic conceptual items (like "physical object", "space location", etc.). An example of such semi-formal grammar is provided by the known Conceptual Dependency theory of R. Schank. On the contrary, a metagrammar of conceptual structures is to postulate the existence of some classes of conceptual items, to associate in a formal way with arbitrary element from each class certain specific information, and to describe the rules to construct arbitrarily complicated structured conceptual items in a number of steps in accordance with such rules (proceeding from elementary conceptual items and specific information associated with arbitrary elements of considered classes of items).

The most part of the known approaches to the formalization of NL-semantics practically don't give the cues to the construction of a UCM. This applies, in particular, to Montague Grammar, Discourse Representation Theory (DRT), Theory of Generalized Quantifiers, Situation Theory, Dynamic Montague Grammar, Dynamic Predicate Logic. For instance, it is difficult not to agree with the opinion of L. Ahrenberg that "in spite of its name, DRT can basically be described as formal semantics for short sentence sequences rather than as a theory of discourse" (Ahrenberg 1992). This opinion seems to be true also with respect to the content of the monograph (Kamp & Reyle 1993).

All mentioned approaches study only some separate aspects of NL-semantics and don't raise the problem of investigating structured meanings (SMs) of complicated real discourses pertaining to science, technology, medicine, law, business, etc. Besides, these approaches don't study sufficiently deeply the role of knowledge about the world in NL use.

Consider, for example, the texts T1 - T5 determined as follows:

T1 = "An adenine base on one DNA strand links only with a thymine base of the opposing DNA strand. Similarly, a cytosine base links only with a guanine base of the opposite DNA";

T2 = "While an electrical current passes across a solution of the blue vitriol (CuSO4), a pure copper accumulates on the negatively charged electrode. One uses this phenomenon to obtain pure metals";

T3 = "A genotype is a collection of all genes located in chromosomes of an organism";

T4 = "The set of concepts explained both in the Longman Dictionary of Scientific Usage (Moscow, 1989) and in the textbook "The Language of Medicine" by Davi-Ellen Chabner (Moscow, 1981) includes, in particular, the concepts "insulin", "thrombin", "thiamin", "streptococcus", and "pellagra";

T5 = "The pattern associative memory method involves inputting a set of pattern signals to be learned to a group of input elements of an error correcting circuit having one group of input elements for inputting information signals and one group of output elements including the same number of elements as the number of input elements".

We can see in these texts, in particular, the following peculiarities: the references to the meanings of phrases being parts of discourses ("Similarly" in T1, "this phenomenon" in T2); designations of the sets of physical objects (in T3 and T5); the occurrences of the word "concept" (in T4); the designation of a set of concepts (in T4); the occurrences of infinitives with and without dependent words ("to obtain pure metals" in T2 and "to be learned" in T5); the occurrences of homogenous members of sentence; the occurrences of expressions denoting the values of functions determined on sets (e.g., "the number of input elements" in T5); complicated designations of physical objects (e.g., "a solution of the blue vitriol (CuSO4)", in T2).

The expressive power of formal means suggested by the mentioned approaches is insufficient for describing SMs of texts T1 - T5 and of the dominant part of other real texts. That's why we need to invent some effective mathematical ways to represent SMs of real texts, i.e. to develop widely-applicable metagrammars of conceptual structures.
3. ABOUT THE FIRST WIDELY-APPLICABLE FORMAL METAGRAMMARS OF CONCEPTUAL STRUCTURES

The first two widely-applicable formal metagrammars of conceptual structures were suggested in the beginning of the eighties. The first one was outlined in (Fomichev 1981) and completely described in (Fomichev 1982). That grammar was provided by the theory of free S-models, restricted S-calculuses, and restricted S-languages of types 1 - 4.

One year later, in 1983, the theory of formalizing SMs of texts was, first, a little modified and, second, extended in order to have the means for describing dynamic semantics of discourses. As a result, the theory of S-calculuses and S-languages, T-calculuses and T-languages was elaborated (the STCL-theory). Its basic ideas were reported at the First Symposium of the International Federation of Automatic Control (IFAC) on Artificial Intelligence and are represented in two publications of the proceedings of this symposium (Fomichev 1983, 1984). Many examples from (Fomichev 1983) are repeatedly adduced in (Fomichev 1994).

The STCL-theory became the starting point for developing the theory of K-calculuses, algebraic systems of conceptual syntax, and K-languages (the KCL-theory).

4. CHIEF IDEAS OF THE THEORY OF K-CALCULUSES AND STANDARD K-LANGUAGES IN STATIONARY BASES

4.1. GENERAL CHARACTERIZATION

The KCL-theory is represented, in particular, in (Fomichev 1988a, 1988b, 1990, 1992, 1993b, 1994, 1996). This theory may be interpreted as an attempt to suggest a first variant of a UCM.

The central constituent of the KCL-theory is the theory of K-calculuses and standard K-languages in stationary bases (the KCLSB-theory). For explaining its destination, consider an informal notion of a stationary NL-text. Assume that KB is some knowledge base (k.b.). If T is an arbitrary NL-text, then we'll say that T is a stationary text with respect to KB in case T doesn't contain fragments INTRODUCING new concepts and/or relationships (but T may include phrases EXPLAINING such concepts and/or relationships that the considered k.b. KB CONTAINS their designations as informational items).

Then we may qualify the KCLSB-theory as a theory providing formal means which are convenient for describing SMs of such arbitrarily complicated real sentences and discourses which are considered as stationary texts with respect to any k.b.

In other words, it seems that the KCLSB-theory may be interpreted as a Universal Stationary Metagrammar of Conceptual Structures.

Consider (without numerous mathematical details) basic ideas of the KCLSB-theory. We'll proceed from a little simplified version of that theory in comparison with the version represented in (Fomichev 1988a). This new version is partially represented also in (Fomichev 1996).

4.2. SORT SYSTEMS AND TYPES

Each K-calculus and standard K-language (knowledge language) are determined by a system of formal objects called conceptual basis (c.b.). Conceptual bases provide (a) a stock of items for constructing semantic representations (SRs) of NL-texts and fragments of texts, (b) the knowledge enabling us to build the SRs using these items and some special symbols. Sort systems are components of c.b.

A sort system (s.s.) is a tuple of the form

\( (\text{St}, \text{P}, \text{I}, \text{Qu}, \text{Gen}, \text{ToI}) \),
where the components $S_1$, ..., $S_l$ satisfy a number of conditions. $S_i$ is a finite set of symbols called sorts and interpreted as the designations of the most general notions of a considered application domain. The elements $P$, $I$, $Q$ belong to $S_i$ and are called the sorts "sense of proposition", "sense of goal", and "sense of question". These sorts qualify respectively the semantic representations (SRs) of (a) phrases expressing propositions (assertions) and discourses, (b) the meanings of infinitives with dependent words or without dependent words, (c) questions.

Gen is a binary relation on $S_i$ being a partial order on $S_i$. If $s$, $t$ belong to $S_i$, then the designation $s \prec t$ will mean that either $s$ and $t$ are identical or a notion associated with $s$ is a generalization of a notion associated with $t$.

$Tol$ is an antireflexive symmetric relation on $S_i$. If $(s, u)$ belongs to $Tol$, then we suppose that in the regarded domain there is an object which may be characterized from two "orthogonal" points of view by concepts $s$ and $u$.

Each $s_s$ determines a set $T(S)$ of strings called types. For some sort systems $S$, the set $T(S)$ may include, for instance, the strings represented in the left column of the table below. The types of concepts, as distinct from types of objects, have the beginning $\wedge$. The relationship $\wedge$ belongs to $T(S)$ means that (inf, phys) belongs to $Tol$, where inf and phys are interpreted as sorts "informational objects" and "physical object", respectively.

<table>
<thead>
<tr>
<th>Type</th>
<th>Entity qualified by a type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\wedge$ inf * phys</td>
<td>the concept &quot;scientific journal&quot;</td>
</tr>
<tr>
<td>inf * phys</td>
<td>each concrete scientific journal</td>
</tr>
<tr>
<td>$\wedge$ ins</td>
<td>the concept &quot;intelligent system&quot;</td>
</tr>
<tr>
<td>ins * phys</td>
<td>Tom Soyer</td>
</tr>
<tr>
<td>$\wedge$(ins * phys)</td>
<td>the concept &quot;Editorial Board&quot;</td>
</tr>
<tr>
<td>(ins * phys)</td>
<td>Editorial Board of &quot;Informatica&quot; (Slovenia)</td>
</tr>
<tr>
<td>((real, real))</td>
<td>Each binary relation on real numbers</td>
</tr>
<tr>
<td>({{ent}}, {{ent}})</td>
<td>The relation &quot;Belongs to a set&quot;</td>
</tr>
<tr>
<td>({{ent}}, {{ent}})</td>
<td>The relation &quot;Is a subset of a set&quot;</td>
</tr>
<tr>
<td>({{ob}}, [conc])</td>
<td>The relation &quot;An object is qualified by a concept&quot;</td>
</tr>
<tr>
<td>({{ent}}, nat)</td>
<td>The function &quot;Number of elements of a set&quot;</td>
</tr>
</tbody>
</table>

Here the elements [ent], [ob], [conc] are interpreted as types "entity", "object", "concept".

### 4.3. CONCEPT-OBJECT SYSTEMS

As sort systems, concept-object systems are components of conceptual bases. Each concept-object system (c.o.s.) is some five-tuple $Ct$ of the form

$$(X, V, tp, F, Pr)$$

satisfying some special conditions and defined with respect to some s.s. $S$. For arbitrary c.o.s. $Ct$, the components $X$ and $V$ are non-intersecting countable sets of symbols. The elements of $X$ (a primary universe) are interpreted as basic items of texts' SRs designating physical objects, concepts, events, numbers, colours, functions, relations, etc. The elements of $V$ are called variables.

The component $tp$ is a mapping from the union of $X$ and $V$ into $T(S)$. The elements of $F$ are called functional symbols, and the elements of $Pr$ are called predicates; besides, the intersection of $F$ and $Pr$ is empty. The sets $F$ and $Pr$ are subsets of $X$. Predicates designate mainly the relations from $X$ corresponding to the meanings of such verbs that with the help of those the commands and goals are formed ("to press the green button", etc.) while one is acting in a chosen domain. If only relationships are considered, then $Pr$ is to be void.
4.4. CONCEPTUAL BASES

Presume that we define a s.s. S and a c.o.s. Ct for S in order to describe an application domain. Then it is proposed to distinguish in the primary universe X two non-intersecting and finite subsets Int1 and Int2 in the following manner: we distinguish in St two sorts int1 and int2 and suppose that Int1 = {x from X | tp(x) = int1} and Int2 = {x from X | tp(x) = int2}. The elements of Int1 correspond to the meanings of expressions "every", "some", "any", "arbitrary", etc. in situations when these expressions are parts of the word groups in singular. The elements of Int2 are interpreted as semantic items corresponding to the expressions "all", "several", "many", and so on; the minimal requirement is that Int2 contains a semantic item corresponding to the word "all".

Let Int1 contain a distinguished element ref considered as an analogue of the word "some" (when it is associated in a context with singular) in the sense "some quite definite" (but, possibly, unknown). If Ct is a c.o.s., d belongs to X, d denotes a concept, and a SR of a text includes a substring of the form ref d (e.g., the substring some chemist, if ref = some, d = chemist), then we'll suppose that this substring denotes some concrete entity (but not an arbitrary one) which is characterized by the concept d.

Let X contain the elements #→#, #not#, #and#, #or# interpreted as the connectives "is identical with", "not", "and", "or" and the elements #A# and #E# interpreted as universal and existential quantifiers.

Then the systems of quantifiers and logical connectives (s.q.l.c.) are defined (with respect to some s.s. S and c.o.s. Ct) as some systems of the form

( int1, int2, ref, eq, neg, binlog, ext ),

where int1, int2, eq, neg, binlog, ext are different elements from the considered set of sorts St, ref is a distinguished element from X, and some special conditions are satisfied. In particular, it is required that tp(#→#) = eq, tp(#not#) = neg, tp(#and#) = tp(#or#) = binlog, tp(#A#) = tp(#E#) = ext.

Let S be a s.s., Ct be a c.o.s. for S, Ql is a s.q.l.c. for S and Ct, and some special conditions are satisfied. Then the system B of the form

( S, Ct, Ql )

is called a conceptual basis (c.b.).

4.5. THE IDEA OF DETERMINING THREE CLASSES OF FORMULAS

Each c.b. B determines three classes of formulas Ls = Ls(B), Ts = Ts(B), Ys = Ys(B) interpreted as inferred formulas of some calculus of a new kind (called the K-calculus in stationary basis B). The formulas from these three classes are called l-formulas, t-formulas, and y-formulas, respectively.

The set Ls(B) is called the standard K-language (knowledge language) in stationary basis B. The strings of Ls(B) are called also standard K-strings (or K-strings) and are convenient for building SRs of texts which introduce no new notions, relationships, designations of things, sets, events (but such K-strings may explain the notions, relationships, etc.).

Each formula from Ts(B) has the form b & t, where b is a standard K-string, t is a type fromTp(S(B)).

The formulas from Ys(B) have the form a(1) & a(2) &... & a(n) & b, where a(1), a(2),..., a(n), b are standard K-strings, n is not the same for various strings, and b is built from a(1),..., a(n) as from "blocks" (some of these blocks might be slightly transformed) by applying just one time some inference rule.

On technical reasons, we'll use below the designations of logical connectives and quantifiers, the uparrow, and the symbol of identity distinct from the designations of the works (Fomichov 1988a, 1992, 1993b, 1994, 1996). Besides, we'll replace the expression "("*," by the expression "(**".
Example. Consider the question Q1 = "Is it useful to swim and to play tennis or volley-ball?". Let
\[ b_1 = (\text{Swim}(1)** \text{and} \text{ Play}(1)**, (\text{volley-ball} \text{ for} \text{ tennis})) \],
\[ b_2 = \text{Useful}(b_1) \], \[ b_3 = ? \text{Useful}(b_1) \],
and Prop, Goal, Ques be interpreted as the distinguished sorts "sense of proposition", "sense of goal", "sense of question".

Then there is such K-calculus with the subsets of formulas Ls, Ts, Ys that b1, b2, b3 belong to Ls, the strings b1 & Goal, b2 & Prop, b3 & Ques belong to Ts, and the strings Useful & b1 & b2 and b2 & b3 belong to Ys. The K-string b3 may be interpreted as a possible SR of Q1.

In order to determine for arbitrary c.b. B the classes of formulas Ls, Ts, Ys, a group of inference rules P0, P1, ..., P14 is defined (Fomichev 1988a). The rule P0 provides an initial stock of formulas of the first and second kinds in the following way. Let z belongs to the union of X(B) and V(B), t belongs to Tp(S(B)), and t(z) = t. Then, according to the rule P0, z belongs to Ls(B) and the string of the form z & t belongs to Ts(B).

A description (without many details) of the rules P0, P1, ..., P14 may be found in (Fomichev 1992). The paper (Fomichev 1996) contains a modified variant of strict mathematical definitions of the rules P0, P1, ..., P10.

4.6. THE FORM OF SOME INference RULES OF K-CALCULUSES

For each c.b. B, the set Ts(B) is the union of non-intersecting sets T(B,0), T(B,1), ..., T(B,14), and the set Ys(B) is the union of non-intersecting sets Y(B,1), ..., Y(B,14). Let k is one of the numbers 1, ..., 14, and Y(B,k) includes a formula a(1) & a(2) & ... & a(n) & b, where n depends on k, and the strings a(1), ..., a(n), b don't contain the symbol ' & '. Then b belongs to Ls(B), and b is built out of a(1), ..., a(n) as from "blocks" by means of applying just one time the inference rule with the number k. In this case, there is some type t from Tp(S(B)) such that the formula b & t belongs to T(B,k).

Taking this into account and supposing that Ls(B) = L(B), we may exactly define the rule P0 as the assertion

"If z belongs to X(B) and V(B), t is a type from Tp(S(B)), t(z) = t, then z belongs to L(B), and the string of the form z & t belongs to the set T(B,0)".

For arbitrary c.b. B, some partial order ⊥ is determined on Tp(S(B)) called the relation of concretization. If t1, t2 are types then the expression t1 ⊥ t2 means that t2 is a concretization of t1 (in particular, it is possible that t1 ⊥ t2). E.g., if t1 = ^{[\text{ent}]}, t2 = ^{[\text{ins} \ast \text{phys}]}, (t2 is the type of the concept "a group of people" because ins is the sort "intelligent system", phys is the sort "physical object"), then t1 ⊥ t2.

The rule P11 is the assertion

"Let p belong to the set of predicates P, n > 1 or n = 1, t = \text{tp}(p), for n = 1 \text{tp}(p) is a string of the form \{t(0), t(1), ..., t(n-1))\} where t(0), t(1), ..., t(n-1) belong to Tp(S(B)).

If n > 1, then let a(1), ..., a(n-1) belong to L(B), u(1), ..., u(n-1) belong to Tp(S(B)); for i = 1, ..., n-1, a(i) & u(i) belong to the union of T(B,0), T(B,1), ..., T(B,14), t(i) ⊥ u(i) in case a(i) is not a variable from V(B), and t(i) and u(i) are comparable with respect to the relation ⊥ in case a(i) is a variable from V(B).

Let in case n = 1 b = p(**), and in case n > 1 b = p(**, a(1), ..., a(n-1)); I be the sort "sense of goal" of the s.s. S(B).

Then b belongs to L(B), b & I belongs to T(B,11), for n = 1 the string p & b belongs to Y(B,11), for n > 1 the string p & a(1) &... & a(n-1) & b belongs to Y(B,11)".
Example. Considering an appropriate c.b. B, we are able to reflect the structured meaning (SM) of the infinitive group "to call Bill in order to reach Namur" (it may be interpreted as a goal) by the t-formula from the set T(B,11)

(Call1 (**, some person * (First.name, "Bill"), Reach1 (**, some city * (Geogr.name, "Namur"))) & Goal, where the string Goal is the distinguished sort I(B), i.e. "sense of goal".

The rule P14 is destined for describing SMs of questions. It is defined as the assertion

"Let a belong to L(B) and not belong to V(B), t be one of the distinguished sorts P(B) and J(B) (the sorts "sense of proposition" and "sense of goal"), k belong to the set of the numbers 3, 4, 6, 7, 9, 11, 12, the string a & t belong to T(B,k), n = 1 or n = 1, v(1),..., v(n) be the variables from V(B) and be substrings of the string a; the string a doesn't include any substrings of the form h v(j), where h is one of the symbols '>', '#A#', '#B#', '?', j belongs to the set {1,..., n}.

Let b be the string of the form? v(1)?...? v(n) * a, Qu = Qu(B) be the sort "sense of question" of the s.s. S(B).

Then b belongs to L(B), b & Qu belongs to T(B,14), and v(1) &... & v(n) & a & b belongs to Y(B,14)."

Example. Using the rules P0, P4 (for expressing relationships), P7 (for joining the formulas by logical connectives), P14 and choosing some appropriate c.b., we can build a possible semantic representation (SR) of the question Q1 = "What did Professor Jones decide?" in the form of the string

? G1 * #E# t1 (moment) (Decide(x1, t1, G1) #and# Is1(x1, professor) #and# Name(x1, 'Jones') #and# Before(t1, Now)) & Ques, where G1 is to be interpreted as a variable of the sort "sense of goal". The constructed string is a t-formula of some K-calculuses.

### 4.7. EXPRESSIVE POSSIBILITIES OF STANDARD K-LANGUAGES IN STATIONARY BASES

Consider several examples illustrating a small part of the expressive possibilities of standard K-languages in stationary conceptual bases and formulate some hypotheses.

Suppose that E is some NL-expression, Semp is a K-string, and, besides, Semp is a possible SR of E. Then we'll say that Semp is a K-representation (KR) of E.

Example 1. Let T1 = "Somebody hadn't switched off a knife-switch. As a result, the Laboratory No. 23 was burnt down". Then the string

#E# mt1 (mom * (Before, Now)) #not# Switch.off(some person: x1, some knife.switch: x2, mt1): P1 #and# Descri.sit(P1, e1) #and# #E# mt2 (mom * (Before, Now)) Burn.down (some lab * (No, 23): x3, mt2): P2 #and# Descri.sit (P2, e2) #and# Cause (e1, e2)

is a possible KR of T1. Here P1 and P2 are variables of the type being the distinguished sort "sense of proposition"; e1 and e2 are to be interpreted as variables designating, respectively, two situations (events) and having the type being the sort "situation".

Example 2. Let T2 = "Yves said to Mary that he was occupied with rowing and painting. The new for Mary was that Yves was occupied with painting". Then we may consider the formula

#E# mt1 (mom * (Before, Now)) (Say (< Agent, some person * (First name, "Yves"): y1 >, < Addressee, some person * (First name, "Mary"): y2 >, < Moment, mt1 >, < Info, Be.occupied (y1, (rowing #and# painting)): P1 > #and# New (y2, P1, mt1, Be.occupied (y1, painting)))

as a possible KR of T2.
Example 3. Let T3 = "The notion of "molecule" is used in physics, chemistry, and biology". One can determine such a c. b. B that the set of sorts S(B) includes the elements area1 and string; the primary universe X(B) includes the elements area1, string, concept, "molecule", ls.used, Name.conc., physics, some, chemistry, biology, tp(concept) = sconc, tp("molecule") = string, tp(chemistry) = tp(biology) = area1, tp(ls.used) = \{(lsconc, area1)\}, tp(Name.conc.) = \{(lsconc, string)\}, some is the referential quantifier of B, ls.used and Name.conc. are not functional symbols.

Let s1 = Name.conc. (some concept, "molecule"), s2 = concept * (Name.conc., "molecule"), s3 = ls.used (some concept * (Name.conc., "molecule")), (physics #and# chemistry #and# biology)).

Then B(P0, P1, P4) => s1 belongs to Lrs(B); B(P0, P1, P4, P8) => s2 belongs to Lrs(B); B(P0, P1, P4, P8, P1, P7, P4) => s3 is in Lrs(B). The formula s3 is a possible KR of T3.

Example 4. Let T4 = "Teenager is a person having the age from 12 to 19 years". Let u be the string (((teenager #and# person * (Age, x1)) #and# not# Less (x1, 12, year)) #and# #not# Greater (x1, 19, year)).

Then u is a possible KR of T4.

Example 5. We can build complex descriptions of diverse objects and sets of objects. E.g., we can build the following KR of the description of "Informatica" (Slovenia):

some int.sc.journal * (Title, "Informatica") (Country, Slovenia) (City, Ljubljana) (Fields, (artif.intel #and# cogn.science #and# databases)): k225, where k225 is the mark of the knowledge module with the data about "Informatica".

Example 6. The meaning of the question "Is Ghent located in Belgium?" may be represented by the l-formula ? Location(Ghent, Belgium) and t-formula ? Location(Ghent, Belgium) & Ques, where Ques is the sort "sense of question".

Example 7. The phrase "Professor P.Jones advised M.Smith to enter the Stanford University and prepare a Ph.D. thesis on physics" may have a deep KR

(Pose-goal(< Agent1, some person * (Name, "P.Jones") (Qualif, prof): x1 >, < Adressee, some person * (Name, "M.Smith")): x2 >, < Form, advice >, < Goal, (Enter2 (**, <Inst, Stanford.Univ>) #and# Prepare1 (**, <Goal, product, certain ph.d.thesis * (Field1, physics): x3 >)), <Moment, x4 >) #and# Before(x4, Now)).

In a similar manner, one can build KRs of phrases with the verbs "to order", "to request", etc.

Using standard K-languages in stationary bases, it is possible also to describe SMs of the texts T1 - T5 adduced in Section 2. A SR of T2 may be found in (Fomichov 1992) and the SRs of T1 and T3 - in (Fomichov 1993b). The papers (Fomichov 1992, 1993b, 1994, 1996) contain many other examples illustrating high expressive power of these languages.

Thus, K-calculuses and standard K-languages in stationary bases provide many new expressive possibilities and considerably extend the stock of mathematical means for describing SMs of NL-texts.

It seems that with respect to the results represented in the works (Fomichov 1988a, 1990, 1992, 1993b, 1994, 1996), there are sound grounds to formulate the following three hypotheses.

Hypothesis 1.

The expressive possibilities of l-formulas and t-formulas of K-calculuses in stationary bases built with the help of the rules P0, P1, ..., P10 are sufficient and these formulas are convenient for representing on some deep conceptual sublevel the structured meanings (SMs) of such arbitrarily complicated real sentences and discourses which are considered as stationary texts with respect to any k.b. In other words, there are reasons to conjecture that the constructed model restricted by the use of only indicated rules may be interpreted as a variant of a Universal Stationary Metagrammar of Deep Conceptual Syntax (Fomichov 1996). Naturally, it is necessary to carry
out further studies in order to prove or to correct the formulated hypothesis.

Hypothesis 2.

The collection of the rules P0, P1, ..., P10, P11, P14 is convenient for describing SMs of arbitrary stationary NL-texts not including the constructions with the words "respectively", "correspondingly".

Hypothesis 3.

The complete theory of K-calculuses and standard K-languages in stationary bases (dealing with the rules P0, P1, ..., P14) may be interpreted as a variant of a Universal Stationary Metagrammar of Conceptual Structures (or of Conceptual Syntax).

The initial version of the model outlined in this paper was published in (Fomichov 1988a). That monograph was also the basis for preparing the articles (Fomichov 1992, 1993b, 1994). That's why it seems that the work (Fomichov 1988a) provides the first variant of metagrammar mentioned in the formulated third hypothesis.

5. ALGEBRAIC SYSTEMS OF STATIONARY CONCEPTUAL SYNTAX

The theory of K-calculuses and standard K-languages in stationary bases may be extended by means of several algebraic notions.

According to (Mal'tsev 1970, Ershov & Palyutin 1979), partial algebraic systems are the triples of the form

(A, Setop, Setpr), where A is arbitrary non-empty set, Setop is a family of partial algebraic operations on A, and Setpr is a family of predicates on A. In such cases, A is called a carrier of the system.

For arbitrary c.b. B, let Degr(B) be the union of all Cartesian m-degrees of Lx(B), where m = 1, 2, ..., If k is one of the numbers 1, ..., 14, then let Op(B,k) be the collection of all such ordered pairs of the form ((a(1), ..., a(n)), b) that n > 1 or n = 1, a(1), ..., a(n) belong to Lx(B), and the string a(1) & ... & a(n) & b belongs to Y(B,k). Then let Setop(B) = \{Op(B,k) | k = 1, ..., 14\}.

Let B be a c.b. Then for each t from Tp(S(B)) define on Degr(B) the predicate C = C(B,t) in the following way: if d belongs to Degr(B) then C(d) = True iff d belongs to Lx(B) and the string d & t belongs to Ts(B). Let Setpr(B) = \{C(B,t) | t is a type from Tp(S(B))\}. Then let

Sigma(B) = (Degr(B), Setop(B), Setpr(B)).

It is easy to show that for arbitrary c.b. B, Sigma(B) is a partial algebraic system.

The systems of the kind possess, in particular, the following important property: if B is arbitrary c.b., B is arbitrary element of Lx(B) not belonging to X(B) then there are only one number k from the set \{1, 2, ..., 14\} and only one element d from Degr(B) such that the partial operation Op(B,k) maps d into b.

An arbitrary partial algebraic system Delta is called algebraic system of stationary conceptual syntax (a.s.s.c.s.) iff there is such c.b. B that Delta is isomorphic to the system Sigma(B).

A formal language L is called a stationary K-language iff the union of all Cartesian m-degrees of L (m = 1, 2, ...) is a carrier of an a.s.s.c.s.

Algebraic approach enables us to describe formally the common features of the languages and data structures being similar in essence to the standard K-languages but distinct in details of realization from them.

6. BRIEFLY ABOUT THE ELABORATED MEANS TO DESCRIBE DYNAMIC SEMANTICS OF TEXTS

Imagine that a natural-language-processing system (NLPS) is worked out which is able to transform the defini-
tion "A genotype is a collection of all genes located in chromosomes of an organism" into the following semantic representation (SR) E1 being a K-string:

\[(\text{Genotype(\text{arbitrary organism: } x1) }\#\#\#\# \text{ all gene } \#E\# y1 \text{ (chromosome) (Location(#, y1) }\#\text{and# Part (y1, x1)))}.\]

In this situation we suppose that our hypothetical NLPSs has in its knowledge base (k.b.) all concepts occurred in E1.

But really an intelligent system doesn’t know all existing terms. Besides, many new terms are emerging in diverse fields. That’s why it would be useful to have the languages convenient for describing in a formal way the structured meanings (SMs) of texts introducing new notions, functions, relations, designations of things. In other words, the formal languages are necessary which are convenient for reflecting the regularities of dynamic NL-semantics.

It seems that the task of determining such languages was solved in sufficiently general form for the first time in (Fomichev, 1988a).

Every standard K-language in stationary c.b. is a subset of some formal language called standard K-language in dynamic c.b.

The strings of standard K-languages in dynamic c.b. may include the substrings of the forms

\[u \leftarrow t, u : \leftarrow: t, u^* \leftarrow: t.\]

The fragments with the substring \(\leftarrow: t\) are used if a new function is being defined. The fragments with the substring \(*\leftarrow: t\) are necessary for constructing SRs of discourses introducing new predicates (i.e., for introducing semantic analogues of the infinitives). In other cases the substrings of the form \(u \leftarrow t\) are to be used.

For instance, the definition of the notion "a genotype" may be represented by the following string of some K-language in a dynamic c.b.

\[\text{Genotype :\leftarrow: } \{\text{(living-ob, } \{\text{organic-object}\})\}l \{\text{(Genotype(\text{arbitrary organism } x1) }\#\#\#\# \text{ all gene } \#E\# y1 \text{ (chromosome) (Location(#, y1) }\#\text{and# Part(y1, x1)))}.\]

In this string, the substring \(\{\text{(living-ob, } \{\text{organic-object}\})\}\) is interpreted as the type of the introduced function "Genotype".

The notion of an algebraic system of stationary conceptual syntax is extended to the notion of algebraic system of conceptual syntax (a.s.c.s.). A language L is called K-language iff the union of all Cartesian m-degrees of L is the carrier of some a.s.c.s. (Fomichev 1988a; see also Fomichev 1992, 1993b, 1994).

It seems that the theory of standard K-languages in dynamic bases may be interpreted as the first variant of a Universal Metagrammar of Conceptual Structures (or of Conceptual Syntax).

7. RELATED APPROACHES


The main advantages are caused by the fact that the KCL-theory provides powerful and flexible mathematical means for: describing concepts, goals, commands, sets (including sets of concepts, sets of propositions, sets of goals); representing SMs of discourses with references to the meaning of phrases and larger parts of text; build-
ing knowledge modules; considering functions on sets and set-theoretical relationships; describing dynamic semantics of texts; describing much larger spectrum of possibilities to use the logical connectives.

The Theory of Semantic Structures (Jackendoff 1990) is stated in an absolutely non-mathematical form. But in (Zwarts & Verkuyl 1994) a mathematical interpretation of this theory is suggested. In comparison with that mathematical interpretation, the elaborated model possesses many important advantages (see Fomichov 1996).

The KCL-theory is an important component of Integral Formal Semantics (IFS) - a powerful approach to mathematical studying the use of NL; the basic principles and composition of IFS are set forth in (Fomichov 1994). The model enables us to approximate all manners suggested by the mentioned approaches to build SRs of NL-texts and represent knowledge and provides numerous additional expressive possibilities. However, both the model and IFS as a whole are very far from all enumerated approaches to investigating NL-semantics if we take into account the posed goals and used mathematical methods.

Meanwhile, there is one approach to the study of NL-semantics with the help of formal methods which is rather close to IFS as concerns the general goals of researches (but far from the standpoint of technical aspects). It is Episodic Logic (EL) developed by L.K. Schubert and C.H. Hwang (Schubert & Hwang 1989, Hwang 1992, Hwang & Schubert 1993a, 1993b, 1993c). Both approaches advocate the need of highly expressive (natural-language-like) formal systems of semantic and knowledge representations as the ground of a comprehensive framework for developing the theory of NLPSs capable to understand complicated NL-discourses. EL is qualified by its authors as an extended first order, intensional, highly expressive logic; the structure of its formulas called logical forms (LFs) and quasi logical forms (QLFs) reflects many peculiarities of NL-texts.

It is possible to show that the elaborated model allows us to approximate all formulas of EL considered in (Hwang & Schubert 1993b). One of the principal ideas how to do this underlies the example 1 in Subsection 4.7.

It may be pointed out also that the principal idea of EL as a tool for building NL-like semantic and knowledge representations of texts was anticipated as far back as in the works (Fomichov 1981, 1982; Fomitchov 1983, 1984) setting forth the theory of free S-models, restricted S-languages of types 1-4, S-calculuses and S-languages of types 1-5 (see also Fomichov 1994).

Moreover, the expressive power of restricted S-languages of type 4 (Fomichov 1982), S-calculuses and S-languages of types 4 and 5 (Fomitchov 1983, 1984) exceeds the expressive power of EL. In particular, one can show that expressive possibilities of LFs and QLFs considered in all examples in (Hwang & Schubert 1993b) may be modelled by formulas of restricted S-languages of type 4 (Fomichov 1982) or by formulas of S-calculuses and S-languages of type 4 or 5 (Fomitchov 1983, 1984).

It should be added that the model has at least three global distinctive features as concerns its structure and destination in comparison with EL. The first feature is as follows. In fact, the purpose of this paper is to represent in a mathematical form a hypothesis about the general mental mechanisms (or operations) underlying the formation of complicated conceptual structures (or semantic structures, or knowledge structures) out of basic conceptual items. EL doesn’t undertake an attempt of the kind, and 21 Backus-Naur forms used in (Hwang 1992) for defining the basic logical syntax (b.l.s.) rather disguise such mechanisms (operations) in comparison with more general 15 rules described in (Fomichov 1988a, 1992).

The second global distinctive feature is that this paper formulates a hypothesis about a complete collection of operations of conceptual level providing the possibility to build effectively the conceptual structures corresponding to arbitrarily complicated real sentences and discourses pertaining to science, technology, business, medicine, law, etc.

The third global distinction is that the form of describing in EL the b.l.s. is not a strictly mathematical one. E.g., the collection of Backus-Naur forms used for defining b.l.s. in (Hwang 1992) contains the expressions 

- 1-place-pred-const>, = happy | person | certain | probable | \ldots, \ 1-fold-pred-modifier-const >, = plu r l very l former l almost l in-manner | \ldots. The only way to escape the use of three dots in productions is to define some analogue of the notion of a simplified conceptual basis introduced in the present paper.

A large stock of formal expressions for describing surface semantic structure of texts is provided by the Core Language Engine (CLE) - a domain independent system for translating a wide range of English sentences into
formal representations of their literal meanings (Alshawi & van Eijck 1989; Alshawi 1990; Alshawi (Ed.) 1992). The CLE has two representation languages, their expressions are called logical forms (LFs) and quasi logical forms (QLFs). The model described in this paper enables us to approximate all expressive possibilities of LFs and QLFs. The model possesses a number of additional properties. Besides, the model allows us to build much more complicated designations of sets. So the model has essential advantages from the standpoint of practice. As for theoretical aspects, the orientations of the model and CLE are very different, and the model has the same three global distinctive features in comparison with CLE as in comparison with EL.

The model described above has important advantages in comparison with the terminological knowledge representation language (TKRL) L-LILOG developed by a number of German universities and research institutes (Herzog & Rollinger (Eds.) 1991). First, it is possible to show that the model allows us to approximate all expressive possibilities of L-LILOG (Section 5 of Fomichov 1994 may be of help for this). Second, the model enables us to build much more complicated definitions of concepts and designations of objects.

It appears that the most important integral advantage of the constructed model in comparison with all approaches discussed above is that only the model presented in this paper together with the ideas set forth in (Fomichov 1988a, 1992, 1993b, 1994) provides a sound mathematical basis for studying the regularities of conveying information by arbitrarily complicated real NL-texts pertaining to diverse areas of human activity and formalizing arbitrary kinds of NL-dialogue.

8. SOME POSSIBLE APPLICATIONS OF RESULTS

8.1. THEORETICAL APPLICATIONS

In (Fomichov 1992, 1993a; Martin-Vide 1993) it is pointed out at the necessity of mathematical studying NL in a broader context than it is being done traditionally: at the expediency of creating a mathematical theory of NL-communication between active intelligent systems.

It seems that the ideas described above and in (Fomichov 1988a, 1988b, 1992, 1993a, 1993b, 1994, 1996) provide the ground for developing mathematical linguocybernetics, or mathematical theory of natural language use, or mathematical theory of NL-communication - in other terms, mathematical foundations of designing arbitrarily complicated text-based intelligent systems (full-text data bases, etc.) and robust NL-interfaces to autonomous intelligent agents and other applied intelligent systems. In particular, the results may be applied to formalizing abductive inference (see Hobbs, Stickel, Appelt, & Martin 1993) as inference aimed at the construction of the best explanation of conceptual ties between fragments of a discourse.

In comparison with the state of affairs reflected in (Partee, ter Meulen, & Wall 1990, Rosner & Johnson 1992, ICML’93), the KCL-theory considerably extends the limits of mathematical linguistics, opens a door to constructing generative algebraic models of complicated natural-language conceptual structures in addition to analytical algebraic models of language characterized in (Marcus 1993).

The constructed model provides very rich formal means to describe knowledge about the world and, as a consequence, to describe various situations and relations between situations. That's why it seems that the model may give a new strong impulse to the studies in Situation Theory (see Aczel et al. 1993; Devlin 1991).

The results of this paper may be used also for building models of knowledge bases, formalizing common-sense reasoning and heterogeneous reasoning, integrating multiple knowledge sources.

8.2. SIGNIFICANCE FOR PRACTICE

The KCL-theory enables the designers of applied intelligent systems to work up formal languages providing one or several of the following possibilities: to build SRs of complicated real NL-texts pertaining to diverse areas of human activity; to represent the intermediate results of the conceptual analysis of NL-texts; to describe background knowledge; to construct high-level conceptual representations of complicated visual images; to describe
lexical semantics; to construct ontologies of application domains; to develop more powerful and flexible hybrid knowledge representation systems. The model opens a lot of new prospects in all these directions (see Fomichov 1992, 1993a, 1993b, 1994, 1996).

The analysis shows that the KCL-theory provides a unique, powerful and flexible mathematical framework for the studies aimed at realizing such highly complicated scientific-technical projects as the projects of Electronic Library and Librarian, Automatic Knowledge Acquisition, Integrating Multiple Knowledge Sources (Wah et al. 1993), a Knowledge Archives (A Plan 1992; see also Zeleznikar 1993).

9. CONCLUSIONS

The theory shortly outlined in this paper enriched in a leap-like manner the stock of formal means for describing structured meanings (SMs) of NL-texts and representing other conceptual items. Combining the described ideas with the ideas set forth in English in (Fomichov 1992, 1993b, 1994, 1996), we have an effective method of constructing formal models being convenient for describing SMs of practically arbitrary (very likely, arbitrary) real discourses and for representing knowledge about the world. That method was published in full in Russian in the monograph (Fomichov 1988a), which, it seems, provided the first variant of a Universal Conceptual Meta-grammar.

The results stated above bridge a gap between formal semantics of NL and the demands of the theory of text-based intelligent systems and of several other subfields of computer science. They open a way for mathematical studying the regularities of conveying information by arbitrarily complicated real discourses pertaining to science, technology, medicine, business, law, etc. Besides, it may be hoped that obtained results will be effectively used for studying a number of actual theoretical and practical problems in computer and cognitive science.

ACKNOWLEDGEMENTS

I greatly appreciate the kind help of Senior Researcher Alexander Artyomov in transforming the initial file of this paper into the form satisfying the requirements of the Program and Organizing Committees of AMILP'95.

REFERENCES


guage understanding. Minds and Machines, 3, 381-419.


The Method of Rosetta,
Natural Language Translation Using Algebras

Theo M.V. Janssen
Dept. of Math. and Comp. Sc., University of Amsterdam
Plantage Muidergracht 24, 1018TV Amsterdam
theo@fwi.uva.nl

Abstract

Rosetta is a research project (of Philips) for translating natural language. The underlying linguistic theory is an amalgamation of Montague Grammar and transformational grammar. Its basic idea about translating is expressed by the principle of compositionality of translation: two expressions are each other's translation if they are built up from parts which are each other's translation, by means of translation equivalent rules.

This principle will be formalized as follows: translating is a homomorphism from the term algebra for the source language to the term algebra for the target language.

In several other grammatical theories there are proposals for automated translation; we will focus on: Tree Adjoining Grammar, Lexical Functional Grammar, and Functional Grammar. These proposals seem to differ much. However, if one considers the situation from an algebraic perspective, the differences disappear. In all theories the idea emerged that, although natural languages can be very different, the underlying term algebras are much alike. The proposals can be seen as instances of the same algebraic situation as Rosetta.

Given this situation, it can be explained that the proposals claim the same successes and meet the same problems. Idioms are often presented as a success of a proposal, and problems are quantifier scope and long distance dependencies. This situation is however not due to the used linguistic theory, but to the algebraic aspects of the approach. The algebraic perspective gives a unifying approach on the apparently different theories, and makes it possible to compare (and transfer) their methods, problems and solutions.

1 Introduction

This contribution will present a mathematical model that characterizes the essential aspects of a compositional translation system. Starting point of the discussion is the principle of compositionality of translation:

Principle of Compositionality of Translation
Two expressions are each other's translation if they are built up from parts which are each other's translation, by means of rules with the same meaning.

This contribution is an shorted and considerably adapted version of chapter 19 in Rosetta (1994).

2 The Algebraic Model

2.1 Syntax

The compositionality principle speaks about parts, and the mathematical model should have a formal definition of this notion. Since the rules of the syntax determine how expressions are formed, we let the syntax determine what are the parts of an expression. For this purpose, the rules should take inputs and yield an output, and we define the parts of an expression $E$ as those expressions from which $E$ is formed by means of some rule. This means that the rules of syntax can be regarded as operators in an algebra. The parts of an expression can again be expressions, and the principle is intended to hold for these parts as well. So there
can be a chain of 'parts of parts of parts...'. Since the principle is intended to give a constructive approach to translating, this chain should have an end. These final parts can be taken as the generators of the algebra.

Summarizing the above discussion: in a compositional translation system the syntax of source and target language are generated algebras. The parts of an expression E are the expressions that can be used as input for an operator yielding output E. As an example the grammar \( G_E \) will be presented.

The carrier of the algebra \( A_E \) consists of strings like (e.g. \( N(boy) \) or \( S(intelligent \ boy \ cry) \)). Note that the strings start with some symbols that indicate the category of the involved expressions, this is done because we use a single sorted algebra. In case we would use a many sorted algebra, elements would be just the English strings, and the the category information would be provided by the sorts.

The generators of the algebra are generators are the following basic expressions

\[
G_E \text{ Generators:} \\
N(\text{girl}), N(\text{boy}), \text{ADJ}(\text{intelligent}), \text{ADJ}(\text{brave}), \text{IV}(\text{cry}), \text{IV}(\text{laugh})
\]

An example of an operator is \( R_{E1}: N(\alpha) + \text{ADJ}(\beta) \Rightarrow \text{NP}(\beta \ \alpha s) \) can be considered as a partial operator that takes two strings as input; the first one starts with 'N(' and ends with ');' and the second starts with 'ADJ('. The result of application of the operator is a string that starts with 'N(' has next the concatenation of the internal part of the second string, followed by the internal part of the first string, concatenated with 's)'. Again, in a many sorted algebra the labeling would disappear. The operators or rules are

\[
G_E \text{ Operators:} \\
R_{E1}: N(\alpha) + \text{ADJ}(\beta) \Rightarrow \text{NP}(\beta \ \alpha s) \\
R_{E2}: \text{IV}(\alpha) \Rightarrow \text{VP}(\alpha) \\
R_{E3}: \text{IV}(\alpha) \Rightarrow \text{VP}(\text{do not} \ \alpha) \\
R_{E4}: \text{VP}(\alpha) + \text{NP}(\beta) \Rightarrow \text{S}(\beta \ \alpha)
\]

2.2 Translations

The principle of compositionality expresses that the way in which expressions are formed from basic parts, give the complete information needed for determining its translation. In algebraic terminology it means that the terms over an algebra form the domain for the translation relation. The principle also states that the translations of an expression are formed in an analogous way from the translations of parts. So also the range of the translation relation consists of terms. An example is the derivation of \( S(\text{brave girls do not laugh}) \), which is represented by

\[
R_{E4}(R_{E3}(\text{IV}(\text{laugh})), R_{E1}(N(\text{girl}), \text{ADJ}(\text{brave})))
\]

Next, we consider the nature of the translation relation between two languages \( A \) and \( B \). For clarity's sake, we assume for the moment the simplification that in \( A \) and \( B \) each basic expression and each rule has precisely one translation. So there are no synonyms or ambiguities on the level of terms, and the translation relation between \( T_A \) and \( T_B \) is a function. Structural or derivational ambiguities of expressions of \( A \) and \( B \) are possible.

Let us consider a simple example: algebra A, with a two-place operator \( f \), and an algebra B in which the operator \( g \) corresponds with \( f \). Let \( T_{AB} \) denote the translation function from \( T_A \) and \( T_B \). Then the principle of compositionality of translation tells that the translation of the term \( f(a_1, a_2) \) is obtained from \( T_{AB}(a_1) \) and \( T_{AB}(a_2) \) by means of the operation \( g \). So

\[
T_{AB}(f(a_1, a_2)) = g(T_{AB}(a_1), T_{AB}(a_2))
\]

This means that \( T_{AB} \) is a homomorphism.

For the reverse translation the same argumentation holds. So \( T_{BA} \), the reverse translation function, is an homomorphism as well:

\[
T_{BA}(g(b_1, b_2)) = T_{BA}(g)(T_{BA}(b_1), T_{BA}(b_2)).
\]

Note that a translation followed by a reverse translation yields the original term:

\[
T_{BA}(T_{AB}(f(a_1, a_2))) = T_{BA}(T_{AB}(f)(T_{AB}(a_1), T_{AB}(a_2))) = T_{BA}(T_{BA}(f))(T_{BA}(T_{AB}(a_1)), T_{BA}(T_{AB}(a_2))) = f(a_1, a_2)
\]

A homomorphism of which the inverse is a homomorphism as well, by definition is an isomorphism. Hence \( T_{AB} \) is an isomorphism.
We may summarize the formalization obtained so far as follows. For a compositional translation system without synonyms and ambiguities at the level of terms holds: the translation relation is an isomorphism between the term algebras over the source language and the term algebra over the target language.

As an example, we will consider a translation into Dutch of the expressions that are generated by $G_E$. This grammar ($G_D$) is as follows:

$G_D$: Generators:

- $N$(meisje), $N$(jongen), $ADJ$(dapper), $IV$(huilen), $IV$(lachen)

$G_D$: Operators:

- $R_{D1}$: $N(\alpha) + ADJ(\beta) \Rightarrow NP(\beta \epsilon \alpha\sigma)$
- $R_{D2}$: $IV(\alpha) \Rightarrow VP(\alpha)$
- $R_{D3}$: $IV(\alpha) \Rightarrow VP(\alpha \text{ niet})$
- $R_{D4}$: $VP(\alpha) + NP(\beta) \Rightarrow S(\beta \alpha)$

The first generator of $G_D$ corresponds with the first of $G_E$, the second generator with the second, etc., and the same goes for the operators. Hence the translation isomorphism maps the term

$R_{E2}(R_{E3}(IV(laugh)), R_{E1}(N(girl), ADJ(brave)))$

on the term

$R_{D4}(R_{D3}(IV(lachen)), R_{D1}(N(meisje), ADJ(dapper)))$.

In figure 1 the isomorphism between term algebras for $G_E$ and $G_D$ is illustrated by indicating the values for some of the terms in these algebras.

```
  ____________________________
  | brave                      |
  | dapper                     |
  ____________________________
  | R_{E1}(brave, girl)        |
  | R_{D1}(dapper, meisje)     |
  ____________________________
  | girl                       |
  | meisje                     |
  ____________________________
  | R_{E4}(R_{E3}(IV(laugh)), |
  | R_{E1}(N(girl), ADJ(brave))) |
  | R_{D4}(R_{D3}(IV(lachen)), |
  | R_{D1}(N(meisje), ADJ(dapper)) |
```

Figure 1: A translation isomorphism between some terms in $T_E$ and in $T_D$.

2.3 Sets of synonymous translations

Next, we allow that there are synonyms of basic expressions and of rules. Then the translation relation is not a function from $T_A$ to $T_B$ because a term may have several translations. But we can see the relation as a function yielding a set of synonymous terms. In the source language algebra the same can be done, and then the translation relation relates sets of synonymous terms in $T_A$ with such sets in $T_B$. These collections of sets have the structure of an algebra. The result of the application of a set of operators to a set of terms, is defined as the union of application of all operators to all terms. This suggests that the translation function is an isomorphism between the algebras of synonymous terms in $T_A$ and $T_B$. The suggestion can be proven to be correct using a theorem on isomorphisms: since synonymy is a congruence relation, it induces an isomorphism on the corresponding quotient algebra (see Graetzer (1979)). This is shown in the figure below, where an English synonym is added for the adjective brave, and a Dutch one for dapper.
2.4 Ambiguities in translations

Finally, the consequences of ambiguous basic expressions and rules are considered. The introduction of ambiguities on the term level is an essential difference with the traditional situation in Montague grammar (e.g. Montague (1970), Montague (1973)) where the terms uniquely determine the meaning of an expression and form an disambiguated language. However, a more recent development (called ‘flexible Montague grammar’) allows for ‘ambiguous’ terms as well. Then the meanings are sets. See, for example, Partee & Rooth (1983) and Hendriks (1993).

At first glance, ambiguous basic expressions and rules may disturb the symmetry between source language and target language because an ambiguity in the one language needs not to correspond to an ambiguity in the other language. An example in the grammar G_B is cry, which can be translated into Dutch by schreeuwen (meaning shout) and huilen (weep). We consider three ways to conceive ambiguous terms in algebraic perspective:

1. The ambiguities of basic expressions (and of rules) form an exception to the situation that the grammars of source and target algebra are isomorphic algebras. This point of view might be useful if there are few exceptions.

2. Consider the translation relation as a function which yields a set of possible translations (not necessarily synonymous). So the translation of the basic expression cry is the set \{schreeuwen, huilen\}, and the translation of a term is a set of terms. The translation function is a homomorphism from T_A to sets of elements in T_B. This perspective seems attractive if one is interested in only one direction of a translation.

3. The expression cry is not a basic expression of the term algebra, but it is a convenient notation for the set consisting of the basic expressions cry₁ (shout) and cry₂ (weep), and every term containing cry is a notation for a set of terms with cry₁ and cry₂. The translation relation is then an isomorphism between sets of such unambiguous terms. This perspective is the most appropriate for the Rosetta system.

2.5 Summary

The algebraic model for compositional translation is as follows:

A compositional translation system between languages A and B consists of:
1. Generated algebras as syntax for A and B. The parts of an expression \( E \) are the expressions that can be used as input for an operator yielding output \( E \).

2. A translation relation that is defined between \( T_A \) and \( T_B \) and consists of an isomorphism between sets of synonymous terms in \( T_A \) and such sets in \( T_B \).

3 Rosetta and the algebraic model

The present section will show that the grammars of Rosetta form an instance of the algebraic model. Furthermore, it will discuss some issues that are interesting from the algebraic point of view. The attention is restricted to the kernel of the grammar: the syntactic component and to the translation relation, whereas morphology and surface syntax are not considered.

3.1 Lexicon as generating set

The formalization of compositionality requires that some suitable subset of expressions is selected as the set of generators. In mathematics it often is required that a generating set is minimal, or finite. These two properties will be considered below. Furthermore, the definition of lexicon for idioms will be looked at.

A generating set is minimal if no elements of the set can be missed. In some applications one aims at using minimal generating sets, for instance in geometrical applications because the dimension of the object in some cases equals the minimal number of generators. The generating set of the Rosetta algebras is not minimal. The grammars produce, for instance, an S-tree for John kicked the bucket from two different sets of generators. Either from the elements \( \{John, \ bucket, \ kick\} \) in S-LEX, or from \( \{kick \ the \ bucket\} \) in ID-DICT and \( \{John\} \) in S-LEX. If the idiom were not in the lexicon, then still the same S-tree could be formed. This is done because in the translation process idioms are considered as basic expressions.

In applications one often aims at a finitely generated algebra, because such an algebra can be specified by finite means. The generating set of the Rosetta algebra is infinite, because an unlimited number of variables is available. This infinity causes no problem for a finite specification of the translation relation because the translation of these infinitely many elements are completely regular (a variable is translated into a variable with the same index). There are infinitely many different analyses of a sentence which differ only with respect to the choice of a variable. Using some form of normalization, this complication can easily be eliminated.

It is interesting to see that idioms and translation idioms are defined in the lexicons by polynomial symbols. For instance, the translation idiom cocinar is translated into prepare a meal, and this complex expression is defined by giving the keys of the basic words and of the rules which form it, hence by giving the term which describes the derivation of the complex expression. Idioms with an open position give rise to polynomials with variables (e.g. to pull x's leg). Representing idioms by means of polynomials is done for a practical reason: to guarantee that the S-trees that are assigned to them are suitable input for later rules.

3.2 The translation relation as a homomorphism

The translation relation in a system for compositional translation is a homomorphism (in fact a special one: an isomorphism). Due to the properties of homomorphisms, there are several advantages.

1. A homomorphism is fully determined by its values for the generators and operators. This justifies the way in which the Rosetta system translations are defined: by giving the translations of generators and operators. Thus the infinite translation relation is reduced to a finite relation between grammars.

2. The translation is defined for the same operators and generators as used in the syntax to define the language. Thus it is guaranteed that if an expression is generated in the syntax, it is automatically accepted as input for the translation. This situation is in contrast with a situation in which the generated language is defined by one mechanism and the translation by another. Then there is no guarantee that a generated expression is suitable as input for translation.

3. The last, but important, advantage concerns the correctness of the translations. Generators and operators of the Rosetta grammars are designed in such a way that they represent a (basic) meaning or an operation on meanings. So there is an (implicit) algebra of meanings, and the assignment of a meaning to terms is a homomorphism. Hence the meaning of a term is uniquely determined by
the meanings of generators and operators. Furthermore, the generators and operators of the target language have the same meaning as the corresponding generators of the source language. So the algebra of meaning for the source language is identical to the algebra of meanings for target language. Consequently a term of the target language has the same meaning as the corresponding meaning in the source language. So, due to the fact that translation is a homomorphism, the correctness of the translation follows from the correctness of the translations of the generators and operators.

4 Other Examples of Compositional Translation

4.1 Related proposals for compositional translation of natural language

The algebraic model as described above, can be found outside the context of the Rosetta system. In some cases this is not surprising since there is a common background, such as Montague Grammar and compositionality, or other ideas originating from Rosetta publications. Below we mention some examples where the same algebraic model is used.

1. Dowty (1982) noticed the analogy of derivation trees for small fragments of English, Japanese, Breton and Latin, and proposed such derivation trees for translation. For Latin (a language with free word order) the grammar would not generate strings (or structures), but (unordered!) sets of words.

2. Tent (1990) uses Montague grammar to give a method for investigating a typology of languages. She does so by comparing grammatical rules in different languages for the same meaning operation. In fact she designs (very small) isomorphic grammars or English, Japanese, Indonesian and Finnish.

3. The CAT-framework of Eurotra is a (not implemented) proposal for the design of the Eurotra system, the translation project of the EC (e.g. Arnold (1985), Arnold (1986) and Arnold & des Tombes (1987)). This proposal is based upon a variant of the principle of compositionality of translation, viz. The translation of a compound expression is a function of the translations of its parts. This formulation of compositionality is also given by Nagao (1989). The algebraic model of this principle gives rise to a model in which translation is a homomorphism (see Janssen (1989a)). It can be turned into an isomorphism, by introducing quotient algebras. 2.4.

4.2 Proposals for translating natural language that turn out to be compositional

For some proposals it might not be immediately obvious that the compositional approach is followed. Then an algebraic reformulation of a proposal might make the analogy evident. Two examples of this are given below.

A TAG grammar is a tree grammar: its basic expressions are trees, and the operations are operations on trees. Such an operation may for instance replace, under certain conditions, an internal node by a subtree. Abeille, Schabes & Joshi (1990) propose to use synchronized TAG grammars, for translating. A synchronized TAG grammar consists of two TAG grammars. The basic trees of the two grammars are given in pairs: a tree in one grammar with the corresponding tree in the other grammar. The positions in which the expansions may take place are 'synchronized': for each node that may be expanded in one tree, a link with a node in the other tree is given. A derivation starts with two coupled basic expressions, and continues by expanding two linked nodes. This means that the grammars are isomorphic and derivations are made in parallel. So, synchronized TAG grammars are an instance of compositional translation.

Some theories of language do not have constituent structures as a notion in their theory. For instance, Dik's functional grammar ((Dik 1978, Dik 1989)) is inspired by logic: predicate frames constitute the skeleton of the syntax and rules operate on such frames (e.g. attaching a modifier or substitution of an argument). Van der Korst (1989) proposed a translation system based upon functional grammar, and, to a large extent, this can be seen as a compositional translation system. The predicate frames filled with arguments and with modifiers attached, can be considered terms in an algebra, where the frames and modifiers are operations.

That these rather divergent theories fit into the algebraic framework may give rise to the misconception that this holds for all syntactic theories. A counter-example is given by the approach which says that the strings of a language have a structure and that this can be any structure that meets certain well-formedness conditions. So there are no rules, only conditions. Such a system was proposed by MacCawley (1986). Since
there are no rules, there is no algebra with operations, and a compositional translation system in the sense of the principle cannot be designed.

A somewhat related situation arises in the theory of 'Principles and Parameters', because, there too, the conditions are the central part of the theory. Formally, the situation is slightly different because the crucial part of the grammar is a rule (called move-alpha) which can move any constituent to any position, and this movement is controlled by many conditions. An algebraic formulation of this theory is possible, with as most important operator the partial operator corresponding with move-alpha. In this case, the algebraic approach is not interesting because the syntactic algebra hardly has any structure, and it is unlikely that a compositional translation system can be based upon this algebra. Note that in M-grammars many generalizations are used which were proposed within the 'Principles and Parameters' theory, but in a much more structured way than with one movement rule.

4.3 Applications of compositional translation in logic and computer science

Languages that exist next to natural languages are programming languages and logical languages. Translations are made between such languages, as well. We will consider some examples of this, illustrating that the compositional approach, which is innovative for translations between natural languages, is accepted and more or less standard for other kinds of translations.

1. From logic to logic.

There are other situations in which precisely the same happens (compositional translating) but where the motivation is different. Below we discuss translating from logic to logic, and in the next section translating from programming language to programming language.

There are many logical languages, and often translations have been defined among them. The purpose is not to assign meanings, but to investigate the relation between the logics, for instance their relative strength or their relative consistency. We will consider a famous example: the Gödel translation from intuitionistic logic into modal logic. It illustrates the method of using polynomially defined algebras.

In intuitionistic logic the connectives have a special interpretation, a constructive one. For instance \( \phi \lor \psi \) could be read as 'given a proof for \( \phi \), it can be transformed into a proof for \( \psi \). The disjunction \( \phi \lor \psi \) is read as 'a proof for \( \phi \) is available or a proof for \( \psi \) is available'. Since it may be the case that neither a proof for \( \phi \) nor for \( \neg \psi \) is available, it is explained why \( \phi \lor \neg \psi \) is not a tautology in intuitionistic logic. These interpretations have a modal ring, and this is made explicit in the translation into modal logic.

Let us write \( \text{Tr} \) for the translation function. Then clauses of the translation are:

\[(a) \quad \text{Tr}(p) = \Box p, \text{ for } p \text{ an atom}\]
\[(b) \quad \text{Tr}(\phi \lor \psi) = \text{Tr}(\phi) \lor \text{Tr}(\psi)\]
\[(c) \quad \text{Tr}(\phi \land \psi) = \text{Tr}(\phi) \land \text{Tr}(\psi)\]
\[(d) \quad \text{Tr}(\phi \rightarrow \psi) = \Box [\text{Tr}(\phi) \rightarrow \text{Tr}(\psi)].\]

Thus one sees that the disjunction and conjunction operator in intuitionistic logic correspond with the same operator of modal logic, whereas the implication corresponds with a polynomially defined operator. Note that the translation of \( p \lor \neg p \) is \( \Box p \land \Box \neg p \) (which is not a tautology in modal logic).

The above example illustrates that the Gödel translation is an example of the method of compositional translation. A large number of translations between logics is collected in Epstein (1990). Almost all of them are compositional (there they are called there 'grammatical translations'). The few who are not, are also in semantical respects deviant.

2. From natural language to logic.

The aim of Montague grammar is to associate meanings with natural language expressions. This is done by translating natural language into intensional logic. The methodological basis of Montague Grammar is the principle of compositionality of meaning, and, therefore, this translation has to be compositional. In many cases a syntactic operator is translated into a compound expression over the logic. For instance, verb phrase disjunction is translated as \( \lambda x[\alpha(x) \lor \beta(x)] \), where \( \alpha \) and \( \beta \) are the translations of the verb phrases.
3. From programming language to programming language.
Computers have to execute programs written in some programming language, and for this purpose the programming language is translated into machine code. The compiler is the part of the computer software which performs this translation. Often, one instruction from the programming language has to be translated into a compound of machine code instructions. A standard method to perform the translation is the so called 'syntax directed translation'. This method can be seen as a form of compositional translation. The power of the compositional approach is used by (Morris 1973, Thatcher, Wagner & Wright 1979) to design a method which allows to prove the correctness of such a compiler. Following the compositional approach, the problem of proving correctness of an infinite set of possible programs is reduced to proving the correctness of translating the generators and the syntactic constructions.

In all these examples the source algebra and the target algebra were not designed in connection with each other, but with independent motivations. It is, therefore, not surprising that in all cases the source language and the target language have algebraic grammars that are not isomorphic with each other. Nevertheless, the translations are isomorphisms. This is possible because in all cases the translation covers only a subset of the target language. Furthermore in all cases only a subset of the target language arises as output of the translation. For this subset a new algebra is designed using polynomials over the original target language algebra. This derived algebra is then isomorphic to the source language algebra, and a compositional translation is obtained.

5 Power of compositional translation

5.1 Generative Power

In this section, the power of the framework with respect to the generated language is considered. First we will look at compositional grammars in which the rules are unrestricted. The theorem below shows that the unrestricted compositional grammars have the same power as Turing machines. The simulation method used in the proof is interesting because it stimulates the discussion in the next section, in which the role of the reversibility and the measurement condition are investigated.

Theorem 5.1. Any recursively enumerable language can be generated by a compositional grammar.

Proof In order to prove the theorem, we will simulate a Turing machine of the following type. The machine operates on a tape that has a beginning but no end, and it starts on an empty tape with its read/write head placed on the initial blank. The machine acts on the basis of its memory state q and of the symbol read by the head. It may move right (R), left (L) or print a symbol, together with a change of memory state. Two examples of instructions are

\[ I_1 : q_1sq_2R \] (if the Turing machine reads in state \( q_1 \) an \( s \), then its state changes in \( q_2 \) and its head moves to the right)

\[ I_2 : q_1sq_2t \] (if the Turing machine reads in state \( q_1 \) an \( s \), then its state changes in \( q_2 \) and it writes an \( t \))

The machine halts when no instruction is applicable and the string of symbols (neglecting the blanks) is the generated string. We assume that the machine is nondeterministic. The set of all the strings it can generate is the language generated by the machine.

A compositional grammar is of another nature than a Turing Machine. A grammar works not with infinite tapes, and has no memory. These features can be encoded by a finite string in the following way. In any stage of the calculations, the head of the Turing machine has passed only a finite number of positions on the tape. That finite string determines the whole tape, since the remainder is filled with blanks. The current memory state is inserted as an extra symbol in the string on a position to the left of the symbol that is currently scanned by the head. Such strings are elements of the algebra.

Each instruction of the Turing machine will be mimicked by an operation of the algebra. This will be shown below for the two examples mentioned before. Besides this, some additional operations are needed: Operations that add additional blanks to the string if the head stands on the last symbol on the right and has to move to the right, and operations that remove at the end of the calculations the state symbol and the blanks from the string. We will not describe these additional operations in further detail.
$I_1$: The corresponding operator $F_1$ is defined for strings of the form $w_1qs w_2$ where $w_1$ and $w_2$ are strings consisting of symbols from the alphabet and blanks. The effect of $F_1$ is defined by $F_1(w_1qs w_2) = w_1sqtw_2$.

$I_2$: The corresponding operator $F_2$ is defined for strings of the form $F_2(w_1qs w_2) = w_1qtw_2$.

Since the algebra imitates the machine the generated language is the same.

End of Proof

The recursively enumerable languages form the class of languages which can be generated by the most powerful kinds of grammars (unrestricted rewriting systems, transformational grammars, Turing machine languages, etc.), or, more generally, by any kind of algorithm. Theorem 5.1 shows that if a language can be generated by any algorithm, it can be generated by a compositional grammar. So, compositional grammars can generate any language that can be generated by the other grammars. This means that the method of compositionality of translation does not restrict the class of languages that can be dealt with. This conclusion, however, does not apply to M-grammars because they are not arbitrary compositional grammars.

5.2 Parsing

In this section a restriction will be discussed that reduces the generative capacity of compositional grammar to recursive sets. The idea is to use rules that are revertible. If a rule is used to generate an expression, the reverted rule can be used to parse that expression. Let us consider an example.

Suppose that there is a rule specified by $R_1(\alpha, \beta, \gamma) = \alpha \beta s \gamma$. So:

$$R_1(\text{every man, love, a woman}) = \text{every man loves a woman}$$

The idea is to introduce a rule $R_1^{-1}$ such that

$$R_1^{-1}(\text{every man loves a woman}) = \langle \text{every man, love, a woman} \rangle$$

In a next stage other reverse rules might investigate whether the first element of this tuple is a possible noun phrase, whether the second element is a transitive verb, and whether the third element is a noun phrase. Thus, using reverted rules, a parsing algorithm can be designed. A specification of $R_1^{-1}$ might be: find a word ending on an $s$, consider the expression before the verb as the first element, the verb (without the $s$) as the second, and the expression after the verb as the third element.

The following complications may arise with this (or another) rule:

- **Ill-formed input**
  The input of the parsing process might be a string that is not a correct sentence, e.g. \textit{John runs Mary}. Then the given specification of $R_1^{-1}$ is applicable. It is not possible to make the rule so restrictive that it cannot be applied to ill-formed sentence because then rule $R_1^{-1}$ would be as complicated as the whole grammar.

- **Applicable on several positions**
  An application of $R_1^{-1}$ (with the given specification) to \textit{The man who seeks Mary loves Suzy} can be applied both to \textit{seeks}, and to \textit{loves}. The information that the man who is a not a noun phrase can only be available when the rules for noun phrase formation are considered. As in the previous case, it is not attractive to make the formulation of $R_1^{-1}$ that restrictive that is is only applicable to well formed sentence.

- **Infinitely may sources**
  A rule may remove information that is crucial for the reversion. Suppose that a rule may deletes all words after the first word of the sentence. Then for a given output, there is an infinite collection of strings that have to be considered as possible inputs.

The above points illustrate that the reversed rule cannot be an inverse function in the mathematical sense. In order to account for first two points, it is allowed that the reverse rule yields a set of expressions. In order to avoid the last point, it is required that is a finite set.
Requiring that there is a reversed rule, is not sufficient for obtaining a parsing algorithm. For instance, it may be the case that \( y \in R^{-1}(y) \). In order to avoid this, it is required that the all rules form expressions which are more complex (in some sense) than their inputs, and that the reversed rule yields expressions that are less complex than the input. Then there is a guarantee that the process of reverting terminates.

The above considerations lead to two restrictions on compositional grammars which together guarantee recursiveness of the generated language. The restrictions are a generalization of the ones in Landsbergen (1981), and provide the basis of the parsing algorithm of the machine translation system Rosetta (see Rosetta (1994) and of the approach in Janssen (1989b).

1. **Reversibility**
   For each rule \( R \) there is given a reverse rule \( R^{-1} \) such that
   
   (a) for all \( y \) the set \( R^{-1}(y) \) is finite
   
   (b) \( y = R(x_1, x_2, \ldots, x_n) \) if and only if \( (x_1, x_2, \ldots, x_n) \in R^{-1}(y) \)

2. **Measure condition**
   There is a computable function \( \mu \) that assigns to an expression a natural number: its measure. Furthermore
   
   (a) If \( y = R(x_1, x_2, \ldots, x_n) \), then \( \mu(y) \geq \max(\mu(x_1), \mu(x_2), \ldots, \mu(x_n)) \)
   
   (b) If \( (x_1, x_2, \ldots, x_n) \in R^{-1}(y) \) then \( \mu(y) \geq \max(\mu(x_1), \mu(x_2), \ldots, \mu(x_n)) \)

Assume a grammar to be given, together with inverse rules, and with a computable measure condition. A parsing algorithm for M-grammars can be based upon the above two restrictions. Condition 1 makes it possible to find, given the output of a generative rule, potential inputs for the rule. Condition 2 guarantees termination of the recursive application of this search process. So the languages generated by grammars satisfying the requirements are decidable languages. Note that the grammar in the proof of theorem 5.1 does not satisfy the requirements since there is no sense in which the complexity increases if the head moves to the right or the left.

As we have just seen, the grammar used in the proof obeys the reversibility condition but not the measure condition. So, either the combination of the two conditions is responsible for the decrease of generative power, or only the measure condition. The following theorem shows that it is the measure condition.

Theorem 5.2. Let \( G \) be an algebraic grammar with a finite number of generators and rules. Suppose \( G \) satisfies the measure condition. \( G \) then generates a recursive language.

**Sketch of Proof** An algorithm deciding whether an expression belongs to the language is as follows. Determine the complexity of the given expression. Then try all generations that are possible with the given grammar, but stop as soon as the generated (intermediate) results have a complexity greater than the complexity of the given expression. Since the grammar satisfies the measure condition, this process terminates.

**End of Proof**

This theorem can, under certain conditions, be generalized to infinite sets of generators and for rule schemes. One might, for instance, require that only a finite number of generators of a given complexity is available.

In a controlled M-grammar not all the rules need to satisfy the measure condition. However, a variant of the theorem expressing that an expression can be parsed by generation still holds. There is a measure condition on the relation between import and export conditions of subgrammars that restricts the generation on the level of subgrammars. Within the subgrammars the generation is restricted by the control and the measure for iterative rules.

The results presented in this section show that the measure condition is responsible for the decrease in generative power of the compositional grammars. The reversibility condition is only a restriction on the kind of rules used in the grammar, but is not a restriction on the power. It is a restriction that makes an attractive parsing algorithm possible. It would even be incorrect to claim that the reversibility condition guarantees efficient parsing: both with the algorithm sketched in the proof, and with the algorithm based upon reversibility, the running time can be exponential in the length of the input.
5.3 Translation power

Next, we investigate the power of the framework with respect to the relation between source language and target language. The first result shows that by means of unrestricted compositional grammars 'any' language can be translated into 'any' language.

**Theorem 5.3.** Let $L$ be a recursively enumerable language, and $T_r : L \rightarrow M$ a computable translation function of the expressions of $L$ into $M$. We will show that there are isomorphic compositional grammars for $L$ and $M$ such that $T_r$ is an isomorphism.

**Sketch of Proof** In the proof of theorem 5.1 the existence of an algebra as syntax for the source language $L$ is proved. For the target language we take a copy of this algebra and extend it with rules that perform the translation. This is possible for the following reason. Since the function $T_r$ is computable there exists a Turing machine computing $T_r$, and this Turing machine can be simulated by an algebraic grammar. These rules which perform the translation, are considered as transformations of the target language algebra. So, when an expression is generated in the source language algebra, its translation is generated isomorphically in the target language algebra.

**End of Proof**

This result shows that compositionality does not restrict the possibilities of translation. It is, of course, good to know this, but the above theorem does not help to find such a translation, because $T_r$ is assumed to be given. The present book argues that a good method is to design isomorphic grammars. The above theorem does not apply to M-grammars because of the reversibility conditions and measure conditions, but the method might be used to obtain a related result for Turing machines satisfying these two conditions.

The last question concerns the problem of strict isomorphy: given two algebras with partial rules, and a isomorphism between the term algebras, is it decidable whether the term for the source language yields an expression if and only if this is the case for the target language? The answer is negative, as is shown below.

**Theorem 5.4.** It is undecidable, given two algebras $(A, F)$ and $(B, G)$ and a isomorphism $h : A \cup F \rightarrow B \cup G$, whether for all $a \in A, f \in F : f(a)$ is defined if and only if $h(f)(h(a))$ is defined.

**Proof** Let $TM_1$ and $TM_2$ be arbitrary Turing machines. Consider the algebras $A = \langle \{0,1\}^*, \{f\} \rangle$ and $B = \langle \{0,1\}^*, \{g\} \rangle$, where

$$f(w) = \begin{cases} 
1 & \text{if } TM_1(w) \text{ is defined} \\
\text{undefined} & \text{otherwise}
\end{cases}$$

and

$$g(w) = \begin{cases} 
1 & \text{if } TM_2(w) \text{ is defined} \\
\text{undefined} & \text{otherwise}
\end{cases}$$

Let the isomorphism $h : A \rightarrow B$ be defined by $h(f) = g$, $h(0) = 0$ and $h(1) = 1$. Suppose that it is decidable whether $f(w)$ is defined if and only if $g(w)$ is defined. Then it would be decidable whether $TM_1$ and $TM_2$ accept the same language. This is known to be undecidable (the problem can be reduced to the halting problem by taking for one of the machines a Turing machine that accepts all strings). Therefore there cannot be a general method to decide whether for two corresponding algebras the one yields an expression if and only if the other does.

**End of Proof**

Of course M-grammars have no rules of which the applicability conditions depend on a Turing machine that generates a recursively enumerable language. The conditions in the rules of the M-grammars are intended to be decidable, but there is no formal restriction guaranteeing this yet. This means that assuming decidable conditions would not be a good model. Furthermore, also for Turing machines that accept decidable languages, equivalence is not decidable.

Since the notion of strict isomorphy is linked so closely to the equivalence of Turing machines, it is unlikely that strict isomorphy can be proved for two grammars if the conditions are independent in the two languages. Only if applicability conditions for corresponding rules (or rule classes) of different languages are in a well defined connection, such a proof might be possible. For some examples where strict isomorphy is proved, see Landsbergen (1987). All the examples can be seen as many sorted algebras with total rules, which are good candidates for the class of algebras for which total isomorphy is decidable.
References


CONTEXTUAL GRAMMARS WITH DEPTH-FIRST DERIVATION

C. MARTÍN-VIDE*, J. MIQUEL-VERGES*, Gh. PĂUNb,1

a) Research Group in Mathematical Linguistics and Language Engineering
Rovira i Virgili University, Pça. Imperial Tàrraco 1, 43005 Tarragona, Spain

b) Institute of Mathematics of the Romanian Academy
PO Box 1 – 764, 70700 Bucureşti, Romania

ABSTRACT

We introduce and investigate contextual grammars with the derivation performed in the depth-first mode known from Chomsky grammars: the selector used at any step must contain as a subword at least one of the members of the context adjoined at the previous derivation step. Among the interesting features of this type of grammars is the fact that the empty context is very important from generative point of view (this is similar with erasing rules in Chomsky grammars). One also shortly examines the linguistic relevance of these grammars.

1. INTRODUCTION

The contextual grammars, introduced in [8], produce words of arbitrary length by adjoining contexts (pairs of words), iteratively, starting from a given finite set of words. The contexts to be used are selected at each moment depending on the subwords of the currently produced word: each context has associated a set of words to which it may be adjoined.

Many variants were already investigated. The reader is referred to [11], [12] for bibliographical information, as well as to [13] for an account of recent developments. Particularly appealing seem to be, from linguistic point of view, the contextual grammars with maximal use of selectors, introduced in [10], which were proved in [9] to be able to handle all the usual non-context-free features of natural and programming languages.

In the study of Chomsky grammars it is a classical result that context-free grammars are equivalent with push-down automata. These automata have a memory used in the LIFO manner: last-in-first-out. Sometimes it is important to use also the rules of the grammar in this way, rewriting at each step one of the last introduced nonterminals (this is called a depth-first derivation), maybe in combination with a FIFO strategy (first-in-first-out; this is called a breadth-first derivation). The reader is referred to [1], [2] for details and bibliographical information.

This suggests a natural variant of contextual grammars: to use the selectors in such a way to obtain a depth-first derivation: at least one of the members of the context introduced at the preceding step must be a subword of the selector currently used.

Such a derivation has also a good motivation from practical point of view: it is just conceivable to assume that we want to generate a word in an economical way; jumping from a place of the current word to another place can be expensive (imagine that the operation is done by a computer; moving around the word takes time); it is then appealing to look for derivations somehow localized, working at every step somewhere in a neighborhood of the place where we have worked at the previous step.

These grammars have also a posteriori motivation: they are able to handle two of the main non-context-free constructions in natural languages mentioned in [3], [6], [14], [15], [16].

1Research supported by the Academy of Finland, project 11281
[17]: the reduplication and the crossed dependences. Another finding, interesting from theoretical point of view, is the fact that the empty context plays a crucial role in these grammars, because using it we can freely change the working place, without modifying the current word. (This reminds the situation in regulated rewriting area, where the erasing rules are important, too [4].)

2. DEPTH-FIRST CONTEXTUAL GRAMMARS

As usual, given an alphabet $A$, we denote by $A^*$ the set of all words over $A$, including the empty one, which is denoted by $\lambda$. The set of all non-empty words over $A$, hence $A^* - \{\lambda\}$, is denoted by $A^+$. The length of $x \in A^*$ is denoted by $|x|$. The families of finite, regular, context-free, and context-sensitive languages are denoted by $FIN, REG, CF, CS$, respectively. For the elements of formal language theory we use, we refer to [7], [16].

A contextual grammar (with choice) is a construct

$$G = (A, B, (S_1, C_1), \ldots, (S_n, C_n), \ n \geq 1)$$

where $A$ is an alphabet, $B$ is a finite language over $A$, $S_1, \ldots, S_n$ are languages over $A$, and $C_1, \ldots, C_n$ are finite subsets of $A^* \times A^*$.

The elements of $B$ are called axioms (starting words), the sets $S_i$ are called selectors, and the elements of sets $C_i$, written in the form $(u, v)$, are called contexts. The pairs $(S_i, C_i)$ are also called productions. The intuition behind this construction is that the contexts in $C_i$ can be adjoined to words in the associated set $S_i$. Formally, we define the direct derivation relation on $A^*$ as follows:

$$x \Rightarrow_{in} y \text{ iff } x = x_1 x_2 x_3, \ y = x_1 u x_2 v x_3,$$

where $x_2 \in S_i, (u, v) \in C_i$, for some $i, 1 \leq i \leq n$.

Denoting by $\Rightarrow_{in}^*$ the reflexive and transitive closure of the relation $\Rightarrow_{in}$, the language generated by $G$ is

$$L_{in}(G) = \{x \in A^* \mid w \Rightarrow_{in}^* z, \text{ for some } w \in B\}.$$

**Remark.** The above defined relation has been denoted by $\Rightarrow_{in}$ in order to distinguish it from the external derivation defined for $G$, where the context is adjoined at the ends of the derived word: $x \Rightarrow_{ex} y \text{ iff } y = u x v \text{ for } (u, v) \in C_i, x \in S_i \text{ for some } i, 1 \leq i \leq n$. We do not investigate here the external derivation.

A depth-first desrivation in a grammar $G$ as above is a derivation

$$w_1 \Rightarrow_{df} w_2 \Rightarrow_{df} \ldots \Rightarrow_{df} w_m, \ m \geq 1,$$

where

(i) $w_1 \in B, w_1 \Rightarrow_{in} w_2$ in the usual sense,

(ii) for each $i = 2, 3, \ldots, m$, if $w_{i-1} = x_1 x_2 x_3, w_i = x_1 u x_2 v x_3$ ($(u, v)$ is the context adjoined to $w_{i-1}$ in order to get $w_i$), then $w_i = y_1 y_2 y_3, w_{i+1} = y_1 u' y_2 v' y_3$, such that $y_2 \in S_j (u', v') \in C_j$, for some $j, 1 \leq j \leq n$, and $y_2$ contains at least one of the words $u, v$, that is, formally, at least one of the following assertions is true:

- $|y_1| \leq |x_1|, \ |y_1 y_2| \geq |x_1 u|$,
- $|y_3| \leq |x_3|, \ |y_2 y_3| \geq |v x_3|$.
Picturally, the first case is illustrated in Figure 1.

Figure 1

The set of all words \( w_m, m \geq 1 \), generated by depth-first derivations in the grammar \( G \), plus all words of \( B \), constitutes the language generated by \( G \) in the depth-first manner and it is denoted by \( L_{df}(G) \).

If in a grammar \( G \) as above all selectors \( S_1, \ldots, S_n \) are languages in a given family \( F \), then we say that \( G \) is a contextual grammar with \( F \) choice. The families of languages \( L_{\alpha}(G) \), for \( G \) a contextual grammar with \( F \) choice, is denoted by \( CL_{\alpha}(F), \alpha \in \{in, df\} \). As we shall see below, the use of the empty context, \( (\lambda, \lambda) \), is important for depth-first grammars (this context can be removed from usual contextual grammars without changing the language), hence we denote by \( CL_{df}(F) \) the family of languages generated in the depth-first mode by arbitrary grammars, possibly using the empty context, and we use the notation \( CL_{df}(F) \) for the case when this context is not allowed.

Here we consider only grammars with \( FIN \) and \( REG \) selection.

3. EXAMPLES

Consider first a simple grammar:

\[ G_1 = (\{a, b\}, \{ab, aabb\}, (\{abb, aab\}, (\{a, b\})). \]

From \( aabb \) we can perform either one of the following derivations

\[ aabb \rightarrow_{df} aaabb, \quad aabb \rightarrow_{df} gaabbb. \]

The obtained word is the same, \( a^3b^3 \), but the last introduced symbols – those underlined – are different in the two cases, hence the continuation is different from a case to another. More exactly, to the first word only the word-selector \( aab \) can be used, to the second one only the word-selector \( abb \) can be used. We obtain,

\[ aaabb \rightarrow_{df} aaabbbb, \quad aaabbb \rightarrow_{df} aababbb. \]

The process can be iterated, hence we get

\[ L_{df}(G_1) = \{a^n b^n \mid n \geq 1\}. \]
Remark the fact that \( G_1 \) has finite choice and that for the grammar

\[
G'_1 = \{ \{a, b\}, \{ab\}, \{(ab), \{(a, b)\}\} \}
\]

we have \( L_{in}(G'_1) = L_{df}(G_1) \), but \( L_{df}(G'_1) = \{ab, aabb\} \): after \( ab \rightarrow_{df} aabb \) no further derivation step is possible in the depth-first mode.

If, however, we add the production \( \{(b), \{(\lambda, \lambda)\}\} \) to \( G'_1 \), thus obtaining

\[
G''_1 = \{ \{a, b\}, \{ab\}, \{(ab), \{(a, b)\}\}, \{b\}, \{(\lambda, \lambda)\}\}
\]

then again we have \( L_{df}(G''_1) = \{a^n b^n \mid n \geq 1\} \). Here is a depth-first derivation in this grammar:

\[
ab \rightarrow_{df} aabb \rightarrow_{df} aab\lambda b\lambda \rightarrow_{df} aabbb.
\]

Observe how the second step "forgets" the previously introduced symbols and how the third step makes use of the "ghost" subword, identifying \( ab \) with \( ab\lambda \).

Consider also

\[
G_2 = \{ \{a, b, c, d\}, \{abcd\}, \{ab^+ c, \{(a, c)\}, bc^+ d, \{(b, d)\}\} \}
\]

Again two different derivation steps can be performed from the axiom:

\[
abcd \rightarrow_{df} aabcdd \quad (1)
\]

\[
abcd \rightarrow_{df} abbcdd \quad (2)
\]

To the first word we can apply only the second production of \( G_2 \) (the active symbol \( c \) is covered by the selector), to the second word we can apply only the first production. We obtain, respectively,

\[
aabcdd \rightarrow_{df} aabccddd,
\]

\[
abbcdd \rightarrow_{df} aabccddd.
\]

The roles are interchanged: to the first word we can now apply only the first production, and we get a word of the form in (1), to the second word we can apply only the second production, and we get a word of the form in (2). The process can be iterated, at every step the number of \( a \) and \( b \) occurrences (hence of \( c \) and of \( d \), too) being either equal or differing by one. Therefore,

\[
L_{df}(G_2) = \{a^n b^n c^n d^n, a^{n+1} b^n c^{n+1} d^n, a^n b^{n+1} c^n d^{n+1} \mid n \geq 1\}.
\]

We close this section with two examples having linguistic relevance. Up to now, the most convincing non-context-free constructions found in natural languages are those based on reduplication and crossed dependences. The first one appears, for instance, at the level of the vocabulary of the African Bambara language, [3], [6], and even in certain (rather artificial) English constructions, and it leads to the formal language

\[
M_1 = \{x^2 \mid x \in \{a, b\}^*\}.
\]

The second one has been encountered in certain Swiss dialects of German [17], [6], and it leads to the formal language

\[
M_2 = \{a^n b^n c^n d^m \mid n, m \geq 1\}.
\]
Note the similarity between $M_2$ and $L_{df}(G_2)$ (but $L_{df}(G_2)$ is more restrictive, hence "more non-context-free" than $M_2$).

Both languages $M_1$ and $M_2$ can be generated with contextual grammars with regular choice in the depth-first mode, without using empty contexts.

Indeed, consider the grammars:

$$H_1 = \{(a, b, c), (c), \{(c)V^*, \{(a, a), (b, b)\}\}\},$$

$$H_2 = \{(a, b, c, d), \{abcd\}, \{a^+b^+c^+, \{(a, c)\}, \{b^+c^+d^+, \{(b, d)\}\}\}.$$ 

The reader can easily check that

$$L_{df}(H_i) = M_i, \ i = 1, 2.$$ 

Here are some short derivations in each of these grammars:

$$c \quad \Rightarrow_{df} \quad aca \quad \Rightarrow_{df} \quad aqcab \quad \Rightarrow_{df} \quad aabcaab \quad \Rightarrow_{df} \quad aabcaaba,$$

$$abcd \quad \Rightarrow_{df} \quad aabccd \quad \Rightarrow_{df} \quad aabcccd \quad \Rightarrow_{df} \quad aaaaabccddd \quad \Rightarrow_{df} \quad aaaaabccddd.$$ 

Remark in the case of $H_1$ how the depth-first restriction forces the adjoining of the new context in the unique right position, thus ensuring the equality of the two parts of the obtained word.

4. GENERATIVE CAPACITY

Directly from definitions, we have

Lemma 1. (i) $CL_\alpha(FIN) \subseteq CL_\alpha(REG), \ \alpha \in \{in, df\}.$

(ii) $CL_{df}(F) \subseteq CL_\alpha(F), \ F \in \{FIN, REG\}.$

If $B \subseteq A^*$ is a finite language, then $B = L_\alpha(G),$ for $G = (A, B, (\emptyset, \emptyset)),$ hence

Lemma 2. $FIN \subseteq CL_\alpha(F), \ \text{for all} \ \alpha$ and $F.$

The following necessary condition for a language to be in the family $CL_{in}(REG)$ appears in [11], [13]:

Lemma 3. If $L \in CL_{in}(REG), L \subseteq A^*,$ then there are two constants $p, q$ such that every $z \in L, |z| > p,$ can be written in the form $z = uvwxy,$ with $u, v, w, x, y \in A^*, 0 < |vx| \leq q,$ and $uv^ixw^iy \in L$ for all $i \geq 0.$

Corollary. $CL_{df}(REG) - CL_{in}(REG) \neq \emptyset.$

Proof. The language $L_{df}(G_2)$ in the previous section does not have the pumping property in Lemma 3.

In fact, a stronger result is true:

Lemma 4. $CL_{df}(FIN) - CL_{in}(REG) \neq \emptyset.$

Proof. Consider the grammar

$$G = \{(a, b, c), \{a, c\}, (S_1, C_1), (S_2, C_2), (S_3, C_3)\},$$

$$S_1 = \{a\}, \quad C_1 = \{(\lambda, a)\},$$

$$S_2 = \{c\}, \quad C_2 = \{(a^2c^2, c^2b^2)\},$$

$$S_3 = \{a^2c^3, c^5b^2\}, \quad C_3 = \{(a^2, b^2)\}.$$
Starting from the axiom $a$ and using the first production, we can generate all words $a^n$, $n \geq 1$. To a word $a^n$ none of the other two productions can be applied. Starting from $c$ and using $(S_2, C_2)$, we get $a^2c^2c^2b^2$. The second production cannot be applied again. Also after using $(S_2, C_3)$ the second production is not applicable, because no new symbols $c$ are introduced. The first production is not applicable either. Also after using $(S_3, C_3)$, the first production is not applicable: always one introduces two occurrences of $a$ and $S_1$ can cover only one of them. Consequently, only $(S_3, C_3)$ can be used. As in the first example in Section 3, one can see that we can alternately use the word-selectors $a^2c^2$ and $c^5b^2$, in derivations of the form

$$a^2c^2c^2b^2 \Rightarrow_{df} a^2c^2c^2b^2c^5b^2 \Rightarrow_{df} a^4c^2c^5b^2b^2 \Rightarrow_{df} \ldots$$

In conclusion,

$$L_{df}(G) = \{c\} \cup \{a^n \mid n \geq 1\} \cup \{a^{2n}c^2b^{2n} \mid n \geq 1\}.$$

This language is not in the family $CL_{tin}(REG)$: in order to obtain words $a^n$ with arbitrarily large $n$, we need a context $(a^i, a^j), i + j \geq 1$, associated to word-selectors $a^k, k \geq 0$. Such a context can be applied to $a^{2n}c^2b^{2n}$, for $n \geq k$, producing $a^{2n+i+j}c^5b^{2n}$, which is not in $L_{df}(G)$, a contradiction which concludes the proof.

As we have already announced, the use of the empty context is very important:

**Lemma 5.** $CL_{tin}(F) \subseteq CL_{df}(F), \ F \in \{FIN, REG\}$.

**Proof.** Take a contextual grammar $G = (A, B, (S_1, C_1), \ldots, (S_n, C_n))$ and construct the grammar

$$G' = (A, B, (S_1, C_1), \ldots, (S_n, C_n), (S_{n+1}, \{(\lambda, \lambda)\})),$$

where

$$S_{n+1} = A \cup \{u \mid (u, v) \in C_i, \text{ or } (v, u) \in C_i, \text{ for some } 1 \leq i \leq n\}.$$

We obtain $L_{tin}(G) = L_{df}(G')$. As $S_{n+1}$ is finite, the two grammars have selection of the same type.

The inclusion $L_{df}(G') \subseteq L_{tin}(G)$ is obvious: $G'$ has the same productions as $G$, plus that using the empty context, which changes nothing.

However, this last production enables $G'$ to simulate each derivation in $G$ in a depth-first manner. Assume that we have a current word of the form $x_1ux_2uv_3$ (as usual, we have underlined the last introduced context), and assume that we want now to use a selector which does not cover $u$ or $v$. Take an extreme case, and assume that we want to use as a selector the first symbol of $x_1$. Write $x_1 = a_1a_2 \ldots a_k, a_i \in A, 1 \leq i \leq k$, hence the whole word is $a_1a_2 \ldots a_kux_2uv_3$. Using the production $(S_{n+1}, \{(\lambda, \lambda)\})$ we obtain

$$a_1 \ldots a_k \ldots a_{k-1}a_kux_2uv_3 \Rightarrow_{df} a_1 \ldots a_{k-1}a_ku\lambda x_2uv_3 \Rightarrow_{df} a_1 \ldots a_{k-1}a_k\lambda u\lambda x_2uv_3 \Rightarrow_{df} a_1 \ldots a_{k-2}a_{k-1}a_ku\lambda x_2uv_3 \Rightarrow_{df} \ldots$$

and now $a_1$ is activated by the presence of the recently introduced empty word. Consequently, also $L_{tin}(G) \subseteq L_{df}(G')$, hence the proof is complete.

Without using the empty context, the situation is different:

**Lemma 6.** $CL_{tin}(FIN) \not= CL_{df}(REG)$.

**Proof.** For the grammar

$$G = \{(a, b, c), \{(c), \{(a, b)\}\}$$,
we have

\[ L_{in}(G) = \{ a^n b^n a^m c^m \mid n, m \geq 1 \}. \]

However, \( L_{in}(G) \notin CL_{df}(REG) \). Assume the contrary, and take \( G' = (\{a, b, c\}, B, (S_1, C_1), \ldots, (S_p, C_p)) \) such that \( L_{in}(G) = L_{df}(G') \). Because \( B \subseteq L_{df}(G') \), the words in \( B \) must be of the form \( a^i c^j a^k b^l a^j c^j \), \( i, j \geq 0 \). If a context is adjoined to such a word, then either the number of \( a \) and \( b \) occurrences around the first \( c \), or the number of \( a \) and \( b \) occurrences around the second \( c \) are simultaneously increased, but not both. The first occurrences of \( a, b \) and the second occurrences of \( a, b \) are symmetric. Therefore we have

\[ a^i c^j a^k b^l a^j c^j \rightarrow_{df} a^{i_1} a^{i_2} a^{i_3} c^j b^l b^l a^{i_4} a^j c^j, \]

for some \( i_1 + i_2 = i_3 + i_4 = i, k \geq 1 \). The derivation must continue using a selector which covers at least one of the underlined subsequences \( a^k, b^l \). No context \((u, v) \neq (\lambda, \lambda)\) can be used in such a way to contribute to the suffix \( a^i c^j \) without destroying the equality of the numbers of \( a \) and \( b \) occurrences around the second \( c \). Consequently, the language \( L_{df}(G') \) contains words of the form \( a^n b^m a^m c^m \) with at least one of \( n, m \) smaller than \( \max\{s \mid a^s c^s \in B \text{ or } a^s b^s \in B\} \). However, \( L_{in}(G) \) contains such words with arbitrarily large \( n, m \), a contradiction which shows that the equality \( L_{in}(G) = L_{df}(G') \) is impossible. \( \square \)

**Lemma 7.** \( CL_{df}(REG) \subseteq CS \).

**Proof.** Straightforward simulation of a contextual grammar working in the depth-first manner, by a length-increasing grammar. \( \square \)

**Lemma 8.** \( CL_{df}(REG) - CL_{df}(FIN) \neq \lambda \).

**Proof.** The language \( M_2 \) in Section 3 cannot be generated by a grammar with finite selectors, even using the empty context. Indeed, \( M_2 \) contains words \( a^n b^m a^m c^m \) with arbitrarily large \( n \) and \( m \). A context adjoined to such a word must contribute either to both \( a \) and \( c \) subwords, or to both \( b \) and \( d \) subwords, or to all of them. In all cases, selectors of arbitrarily large length are necessary. \( \square \)

Synthesizing the previous lemmas, we have

**Theorem 1.** The relations in Diagram 1 hold, where the arrows indicate proper inclusions. All two families not linked by a path in this diagram are incomparable, with the exception of the pairs involving the family \( REG \), for which we know only that \( CL_{df}(FIN) - REG \neq \emptyset \).
Proof. All inclusions are proved above, excepting \( REG \subseteq CL_{in}(FIN) \), which is from [5].

The corollary of Lemma 3 shows that \( CL_{in}(REG) \subseteq CL_{df}(REG) \) is a proper inclusion. Lemma 6 implies that \( CL_{df}(F) \subseteq CL_{df}(REG) \), \( F \in \{FIN, REG\} \), are proper. Together, Lemma 4 and Lemma 6 imply that both \( CL_{df}(FIN) \) and \( CL_{df}(REG) \) are incomparable with \( CL_{in}(FIN) \) and \( CL_{in}(REG) \). Lemma 4 also implies that \( CL_{in}(FIN) \subseteq CL_{df}(FIN) \) is proper. From Lemma 8 we find that both inclusions \( CL_{df}(FIN) \subseteq CL_{df}(REG) \) and \( CL_{df}(FIN) \subseteq CL_{df}(REG) \) are proper (the properness of \( CL_{in}(FIN) \subseteq CL_{in}(REG) \) is known [11]). Moreover, we obtain in this way the incomparability of \( CL_{df}(REG) \) and \( CL_{df}(FIN) \).

The inclusion \( CL_{df}(REG) \subseteq CS \) is proper because the languages in \( CL_{df}(REG) \) have the bounded length increase property: if \( L \in CL_{df}(REG) \) is an infinite language, then there is a constant \( p \) such that for each \( x \in L \) there is \( y \in L \), different from \( x \), such that \( |y| - |x| \leq p \). The family \( CS \) contains languages which do not have this property (\( \{a^{2n} \mid n \geq 1\} \) is an example).

The family \( CL_{df}(FIN) \) contains non-regular languages (the first example in Section 3 is such a case).

The family \( CL_{df}(REG) \) is incomparable with \( CF \): the language \( L_{in}(G) \) in the proof of Lemma 6 is context-free, the languages \( M_1, M_2 \) in Section 3 are not context-free. Also \( CL_{df}(FIN) \) contains non-context-free languages, because \( CL_{in}(FIN) - CF \neq \emptyset \), [5].

Open problems. Is \( REG \) included in \( CL_{df}(FIN) \) or in \( CL_{df}(REG) \)? Are there context-free languages not in \( CL_{df}(FIN) \) or not in \( CL_{df}(REG) \)?
REFERENCES


Subword Membership Problem for Linear Indexed Languages*

Pál Dömösi
Institute of Mathematics and Informatics, L. Kossuth University
Debrecen, Egyetem tér 1, H-4032, Hungary
domosi@math.klte.hu

and

Jürgen Duske
Institute für Informatik, Universität Hannover
D-3000 Hannover 1, Fed. Rep. Germany
jd@informatik.uni-hannover.de

ABSTRACT

A typical problem in formal language theory is to decide for a word $p$ whether or not it belongs to a given language $L$. This is the so-called membership problem. The subword membership problem is a natural extension of this question.

Given a language $L$ and a word $p$, decide whether or not $p$ belongs to $Sub(L) = \{ p \mid qpr \in L \text{ for some pair } q, r \}$.

We prove that for any linear indexed language $L$, there exists a positive constant $c$ such that if $p \in Sub(L)$ is nonempty, then there exist words $q, r$ with $qpr \in L$ and $|q|, |r| \leq c |p|$. This is a natural extension of an earlier result concerning context-free languages. The decidability of the subword membership problem for indexed languages remains open. (For context-sensitive languages, the problem is known to be undecidable.)

1 INTRODUCTION

For any (formal) language $L$, consider the language $Sub(L)$ of all subwords of elements in $L$ and define the function $f_L : N \to N$ having the possibly minimal complexity such that $p \in Sub(L) \setminus \{ \lambda \}$ implies $qpr \in L$ for some pair $q, r$ of words with $|qr| \leq f_L(|p|)$ (where $\lambda$ is the empty word and $|p|$ denotes the length of $p$).

It is easy to prove that for any regular language $L$, there exists a constant $f_L$. If $L$ is context-free then it can be found a linear function having this property [1]. The existence of this like recursive functions for indexed languages remains open. (There exists an indexed language having only exponential $f_L$ [1].) Moreover, it can be found a context-sensitive language $L$ having only non-recursive $f_L$ [1]. In this paper it is shown that for any linear indexed language there exists a linear function of this type.

If $L$ is context-sensitive then, of course, the following subword membership problem is decidable for $L$ if and only if it can be found a recursive function $f_L$.

Given a language $L$ and a word $p$, decide whether or not $p$ belongs to $Sub(L) = \{ p \mid qpr \in L \text{ for some pair } q, r \}$.

A typical problem in formal language theory is to decide for a word $p$ whether or not it belongs to a given language $L$. This is the so-called membership problem. The subword membership problem is a natural extension of this question.

For all notions and notations not defined here, see [4-7]. A (generative unrestricted, or simply, unrestricted) grammar is an ordered quadruple $G = (V, X, S, P)$ where $V$ and $X$ are disjoint alphabets, $S \in V$, and $P$ is a finite set of ordered pairs $(W, Z)$ such that $Z$ is a word over the alphabet $V \cup X$ and $W$ is a word over $V \cup X$ containing at least one letter of $V$. The elements of $V$ are called variables and those of $X$ terminals. $S$ is called the start symbol. Elements $(W, Z)$ of $P$ are called productions and are written $W \to Z$. If $W \to Z \in P$ implies $W \in V$ then $G$ is called context-free. A word $Q$ over $V \cup X$ derives di-
rectly a word \( R \), in symbols, \( Q \Rightarrow R \), if and only if there are words \( Q_1, Q_2, Q_3, R_1 \) such that \( Q = Q_2 Q_1 Q_3 \), \( R = Q_2 R_1 Q_3 \) and \( Q_1 \Rightarrow R_1 \) belongs to \( P \). The language \( L(G) \) generated by a grammar \( G = (V, X, S, P) \) is the set \( L(G) = \{ w \mid w \in X^* \text{ and } S \Rightarrow w \} \), where \( \Rightarrow \) denotes the reflexive and transitive closure of \( \Rightarrow \). \( L \subseteq X^* \) is a context-free language if we have \( L = L(G) \) for some context-free grammar \( G \).

A linear indexed grammar is a 5-tuple \( G = (V, X, I, S, P) \), where \( V, X, I \) are finite pairwise disjoint sets, the set of variables, terminals, and indices, respectively, \( P \) is a finite set of pairs \( (A_f, a) \), \( A \in V, f \in I \cup \{ \lambda \} \), \( a \in X^* \cup X^*V^*X^* \), the set of productions, and \( S \in V \), the start variable. \( (A_f, a) \in P \) is denoted by \( A_f \rightarrow a \).

Let \( a = uA_1f_1 \ldots f_nv \) with \( u, v \in X^*, A \in V, f_1, \ldots, f_n \in I \cup \{ \lambda \} \). Suppose \( b = uu'B_1f_1' \ldots f'_n \). Then we set \( a \Rightarrow b \). The language \( L(G) \) generated by a linear indexed grammar \( G = (V, T, I, S', P) \) is the set \( L(G) = \{ w \mid w \in X^* \text{ and } S \Rightarrow w \} \), where \( \Rightarrow \) denotes the reflexive and transitive closure of \( \Rightarrow \). A language \( L \) is called linear indexed if \( L = L(G) \) for some linear indexed grammar \( G \).

We shall use the following results.

**Theorem 1.1** (see [3]). If \( L' \subseteq Y^* \) is a linear indexed language, then it can be found an alphabet \( X \), a context-free language \( L \subseteq X^* \) and two homomorphisms \( h_1, h_2 : X^* \rightarrow Y^* \) such that \( L' = \{ h_1(w)h_2(w)^R \mid w \in L \} \) holds.

\((h_2(w)^R \text{ denotes the mirror image of } h_2(w)).\)

**Theorem 1.2** (see [2]). For every context-free grammar \( G \), there exists a pair of effectively computable positive constants \( c, d \) depending only on \( G \) and such that for any \( z \in L(G) \), \( |z| \geq c \max(e, 1) \) \( (e \geq 0) \) and \( e \) positions of \( z \) are excluded, then \( z \) has the form \( wzwy \) where \( |vwx| \leq q^{\max(e, d)} \), and \( v \neq 0, v \neq 0 \), \( v \neq 0 \), \( w \neq 0 \) does not contain excluded positions, and \( uv^iwzw^j \) is in \( L(G) \) for all \( i \geq 0 \).

# 2 Main Result

**Theorem 2.1.** For every linear indexed grammar \( G \), there exists an effectively computable positive constant \( k \) depending only on \( G \) such that for any \( p \in \text{Sub}(L(G)) \setminus \{ \lambda \} \), \( qpr \in L(G) \) for some pair \( q, r \) of words with \( |qr| \leq k |p| \).

**Proof.** Consider a linear grammar \( G \). By Theorem 1.1, it can be found an alphabet \( X \), a context-free language \( L' \) and homomorphisms \( h_1, h_2 : X^* \rightarrow Y^* \) such that

\[ L(G) = \{ h_1(w)h_2(w)^R \mid w \in L' \}. \]

Let \( a = \max([h_i(x) \mid i = 1, 2, x \in X]) \). Consider words \( q, p, r \in X^* \) with \( qpr \in L(G), p \neq \lambda \) and let \( w \in L' \) having \( h_1(w)h_2(w)^R = qpr \). We distinguish the following three cases.

1. \( h_1(w) = qpr_1, h_2(w)^R = r_2, r = r_1r_2 \).

Then we may assume \( w = q'y_0x_1y_1 \ldots x_my_mr' \), where \( x_1, \ldots, x_m \in Y, h_1(x_i) \neq \lambda, h_1(y_j) = \lambda, i = 1, \ldots, m, j = 0, \ldots, m, h_1(x_1 \ldots x_m) = p \). Exclude the positions of \( x_1 \ldots x_m \) in \( w \) and consider the case \( i = 0 \) of Theorem 1.2. There are two possibilities.

2. \( |q'y_0 \ldots y_m r'| \leq c \mid x_1 \ldots x_m \mid \leq c \mid p \mid \).

3. \( |q'y_0 \ldots y_m r'| > c \mid x_1 \ldots x_m \mid \) and (using Theorem 1.2 with \( i = 0 \)) there is a \( w' = q'ya_1y_1 \ldots y_mr'r'' \in L' \) with \( \mid w' \mid < \mid w \mid \), \( h_1(y_j) = \lambda, j = 0, \ldots, m \) and \( h_1(x_1 \ldots x_m) = p \) such that \( h_1(w') \leq h_1(w) \), \( h_2(w') \leq h_2(w) \).

Thus, by an induction, we may assume (ii). But then \( \mid x_1 \ldots x_m \mid \leq a \mid h_1(x_1 \ldots x_m) \mid \leq a \mid p \mid \).

Therefore, there exists a \( w' = q'ya_1y_1 \ldots y_mr'r'' \) such that \( h_1(q') = q, h_1(y_j) = p, h_1(r''r') = r_1, h_2(w')^R = r_2 \) and and \( |qpr| < |h_1(w')^R| \leq a \mid w' \mid \leq a(c+1) \mid p \mid \).

Then \( |qpr | < |h_1(w')^Rh_2(w')^R| \leq 2a \mid w' \mid \leq 2a(c+1) \mid p \mid \).

In these cases we may assume \( k \geq 2a(c+1) \).

(ii) \( h_1(w) = q_1, h_2(w)^R = q_2, q = q_1q_2 \).

We can consider the mirror image of the above case.

(iii) \( h_1(w) = qp_1, h_2(w)^R = p_2r, p = p_1p_2 \).

Then \( w = q'y_0x_1y_1 \ldots x_my_m, \) where \( x_1, \ldots, x_m \in Y, h_1(x_i \ldots x_j) = p_1, h_1(x_i) \neq \lambda, \ell = 1, \ldots, s, h_2(x_j) = x_j \neq \lambda, \ell = 1, \ldots, t \).

The exclude the positions of \( x_1 \ldots x_m \), as before, and apply the case \( i = 0 \) of Theorem 1.2 for \( w = q'y_0x_1y_1 \ldots x_my_m. \) Again, we have (ii) or (ii) (with \( r' = \lambda \)). Moreover, by an induction, we may assume (ii). But then \( \mid x_1 \ldots x_m \mid \leq \mid p \mid \). Therefore, there exist factorizations \( w' = q_1p_1r' \) such that \( h_1(q') = q, h_1(p_1) = p_2, h_2(p_2)^R = p_2, h_2(r')^R = r, \mid qr \mid < |h_1(w')h_2(w')^R| \leq 2a \mid w' \mid < 2a(c+1) \mid p \mid \).
We may assume $k \geq 2a(c + 1)$. Therefore, by $k = 2a(c + 1)$, the proof is complete. □

References

ALGEBRAIC METHODS FOR ANAPHORA RESOLUTION

Celia Rico Pérez
Universidad Alfonso X El Sabio
28691 Villanueva de la Cañada, Madrid
Spain
cerico@uax.es

Jose Simón Granda
Universidad de Alcalá de Henares
C/ Colegios,2
28801 Alcalá de Henares, Madrid
Spain
fsimon@filmo.alcala.es

ABSTRACT

One of the main problems in anaphora resolution is finding antecedents. As shown in a previous study (Rico, 1994a), the type of knowledge required to identify antecedents spring from different sources of linguistic information (morphology, syntax, semantics and pragmatics) and it is only through the coordination of all these sources that we will be able to approach the task from an effective point of view.

The strategy for anaphora resolution we present here allows the integration of the mentioned sources by means of a vector-like model. We consider each anaphoric expression and possible antecedents as vectors and we compare them in such a way that closer vectors indicate anaphoric relations.

This strategy involves the following aspects: (a) defining a basis for representing the selected features; (b) implementing a transducer for transducing qualitative features into quantitative values; (c) defining an operation for comparing vectors and interpreting the results.

The work we present here explores the solutions to the outlined tasks and adopts a democratic approach to anaphora (Carter,1990) as all sources of information play the same role in the selection of the antecedent.

Introduction

Approaches to computational anaphora resolution have recently come to acknowledge the need of integrating different sources of linguistic information if we are to identify antecedents. These studies usually describe an algorithmic pattern that tries to predict anaphoric behaviour so that the link between the anaphora and its antecedent can be established. They normally deal with sentence anaphora and only sometimes face the problem in discourse.

We claim anaphora is a linguistic phenomenon that cannot always be predicted, so that writing rules for the process of antecedent identification is a difficult task. Consider, for instance, the following examples:

1a) Pain shout up Curt’s arm clear to the shoulder, but Jess seemed hardly aware that he had been hit.

1b) Curt drove home a solid left to Jess’ mid section. It was like hitting a sack of salt. Pain shout up Curt’s arm clear to the shoulder, but Jess seemed hardly aware that he had been hit.

As we can see, example (1a) contains an anaphoric relation between he and Curt, whereas example (1b) changes the antecedent to Jess while the sentence containing the anaphoric pronoun remains the same in both examples. This shift from one antecedent to the other is obviously due to the new information introduced in context, which takes us to consider the question of deciding the exact or
real incidence of each source of linguistic and extralinguistic knowledge in the overall process of finding antecedents.

Our hypothesis is based on the observation that each source of knowledge offers only partial data for the identification of antecedents and that it is only through the coordination of all these sources that we will be able to approach the task in an effective way.

In this paper we explore the possibilities of using an algebraic method for anaphora resolution which will provide for the integration of partial data by means of a vector-like model.

Next section briefly describes the different knowledge sources involved in anaphora resolution and advances some of the problems we find in English and Spanish. We then turn to the problem of representing linguistic features as quantitative values, concentrating on morphology and semantics as a further step over our first work. We define here a suitable basis for representing the selected features for both knowledge sources. In the last section of the paper we sketch some aspects for future work.

It is important to mention that this paper is a first report of work in progress, so what we now present are not conclusive results, but our working hypothesis and methodology. In the course of further research we might eventually modify our model or the operations defined in order to adjust them to new findings and results.

**Sources of information for anaphora resolution**

In order to design a coherent structure for antecedent identification, we first conducted an analysis of a set of anaphoric relations in English contained in the SUSANNE corpus (Sampson, 1995) that would help to establish the set of sources of information needed as well as their influence for certain types of anaphora: definite descriptions, demonstrative descriptions, personal pronouns and possessive pronouns. As we will see in the course of this paper, the behaviour of these anaphoric devices is not the same in English and Spanish.

The corpus study is fully described in Rico (1994a) and it showed the influence of different knowledge sources with the aim of deciding on patterns of behaviour for them, a task which proved to be only attainable for the easiest of the anaphoric relations, those in which the antecedent is recoverable from information of the immediate or local situation in terms of a morphological restriction, a clearly prominent position in the sentence or a semantic restriction, as shown in (2):

2) Gray Eyes remained erect. The feathered lance was still above his head.

The problem is to establish these patterns when anaphoric references become more complex. Morphology generally offers a way of discriminating among candidates but consider, for instance, the Spanish possessive pronoun *su*, which means *its, her, his, their* depending on the context. The following example illustrates this ambiguity:

3) Paco no sabía si María estaría en su casa.
(Paco didn’t know whether María would be at his/her house).

The use of the pronoun *su* here introduces an ambiguous reference because we cannot decide between Paco (masculine) or María (feminine) as antecedents. We need some more information since morphology only does not offer a definite answer.

In English we find similar problems when referring to collective nouns— which can take both singular or plural references— or to a group of singular entities which take a plural reference:

4) A weapons carrier took Greg, Todman, Banjo and Walters the two miles from the bivouac area to the strip. It was a rough long ride through the mud and the pot holes. The sky flowered down at them.

When morphology fails to give a solution we need to turn to other knowledge sources such as syntax and semantics.

Our study shows that syntactic information is useful for showing prominent entities in discourse as well as for pointing to parallel structures, which in many cases is enough to locate the antecedent.

5a) There was also a dog. Its ribs showed, it was a yellow nondescript colour, it suffered from a variety of sores, hair had scabbed off its body in patches. It lay with its head on its paws.

But again, the link between dog and it cannot be predicted in terms of syntactic information. As we can see in (5b), a new context forces a new interpretation:
5b) There was also a dog. Its ribs showed, is was a yellow nondescript colour, it suffered from a variety of sores, hair had scabbed off its body in patches. It lay about on the ground in tufts.

We may now think that semantics offers the clue to antecedent identification, but consider the two following examples in Spanish:

6a) Juan pegó a Pedro y le hizo daño.
(Juan hit Pedro and hurt him)

6b) Juan pegó a Pedro que le había hecho daño.
(Juan hit Pedro who had hurt him)

What really matters here is not the fact that "hitting someone will result in hurting this person", but a syntactic restriction on relative pronouns and their antecedents.

As we have seen through the examples shown, we can conclude that it is not possible to predict the influence of knowledge sources for anaphora resolution. Morphology may offer enough information to exclude possible antecedents in most of the cases. Semantics, as we have seen, usually gives conclusive evidence but, still on other instances we need to turn to syntax.

With these findings in mind we think vectors may offer a suitable approach to the problem.

Defining a basis for representing features
According to our initial assumption that the method should be *democratic*, all necessary elements of information would preferably be integrated into a single common model. Many different models for representation of morphological, syntactic, semantic and pragmatic knowledge, have been devised along the last decades, that usually introduce separate formalisms for every linguistic component. To the best of our knowledge, a method for integrating all components into a sole representation has not yet been achieved.

In a first attempt, we considered anaphoric relations as vectors so that values representing linguistic information could be effectively combined (Rico, 1994b). We, then, assumed that distance between vectors indicated anaphoric links. Linguistic features were manually converted into quantitative values so that each anaphora and possible antecedents were represented as vectors in space.

Each linguistic feature (morphologic, syntactic, semantic and pragmatic) was assigned a value according to an attribute-value table devised *ad hoc* (Rico, 1994a: 159-161). The set of all values corresponding to the different attributes of anaphora and each possible antecedent constituted the corresponding vectors for each of them. The dot product was the operation used to calculate the distance between vectors so that closer vectors indicated anaphoric relations.

The dot product then gives us a measure of the relative proximity of the vectors that are multiplied, so that this product reaches its maximum for vectors in the same direction, and the minimum (namely zero) for vectors at right angles. This product can be defined in either of two ways which are mathematically equivalent:

\[
\begin{align*}
(1) & \ a \cdot b = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 \\
(2) & \ a \cdot b = ||a|| \cdot ||b|| \cdot \cos \Theta,
\end{align*}
\]

where \( \Theta \) is the angle defined by the two vectors.

This operation has been profusely used to evaluate the relative 'proximity' of the meanings of words, particularly within the paradigm of connectionism. We have also thought of the possibility of applying a similar operation to morphological and categorial disambiguation, that has proved effective in anaphora resolution.

Another important aspect of vectors is that relationships among vectors are independent of the selected coordinate system. This property is essential, as it allows us to define a basis for our vectors regardless of the elements included in them.

Although vectors provide a sufficient mechanism for the description of many physical entities that require both magnitude and direction or location, there are still certain physical quantities that require a more complex formalism to capture their essence. Such is the case of the pressure at a given point in a fluid, or the inertia of an individual point in a solid in motion. These entities are best defined as correspondences (i.e. functions) of one set of vectors onto another. The space of these vectorial functions is called a tensor. The concept of a tensor then is connected to sets of vectors called vector spaces. Very roughly stated, a tensor is a complex vector that takes other vectors as components. It can be proved that scalars and vectors are special cases of tensors.
Taking these concepts to our field, we can conceive a tensor for representing linguistic knowledge relative to words and expressions. More precisely, we can associate certain words a tensor with several components that specify morphological, syntactic and semantic information.

Strictly speaking, these entities are not exactly real tensors, as these are normally defined imposing the same dimension to all components. Nevertheless, the formalism will be valid for representing knowledge, as far as the operations are consistently defined.

In order to simplify our exposition, we are going to follow our discussion considering two components, corresponding to morphological and semantic information, and we will define the operations we will apply for antecedent assignment.

A basis for morphology
In English, only personal pronouns, possessives and a number of nouns have a mark for gender. In Spanish, determiners, adjectives, nouns and pronouns are marked for one of two genders, at least: masculine and feminine. Some authors even consider the existence of a third gender: neuter, which in practice is restricted to the nominalization of certain adjectives to become 'abstract' nouns.

Likewise, in English, only pronouns and nouns take a mark for number, whilst in Spanish again determiners, adjectives, nouns and pronouns may also take one of two forms for number: singular and plural.

For simplicity, we will represent these two morphological categories using vector notation as \((g, n)\). So for a given word \(W\), the morphological component will be \(M = (g, n)\), where \(g\) belongs to \(\{m, f, i\}\) and \(n\) belongs to \(\{s, p, i\}\) with

\[
\begin{align*}
    m & = \text{masculine}, f = \text{feminine}, i = \text{invariable}, \\
    0 & = \text{null}. \\
    s & = \text{singular}, p = \text{plural}, i = \text{invariable}, \\
    0 & = \text{null}. 
\end{align*}
\]

Using this convention we define an operation composition as follows:

Given two words \(W_1\) and \(W_2\) with morpho-vectors, \(M_1 = (g_1, n_1)\), \(M_2 = (g_2, n_2)\) the composition will be:

\[
M_1 \cdot M_2 = (g_3, n_3) = (g_1 \cdot g_2, n_1 \cdot n_2)
\]

where the compositions of the individual components are defined by the following tables:

<table>
<thead>
<tr>
<th>m</th>
<th>f</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>m</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>f</td>
</tr>
<tr>
<td>i</td>
<td>m</td>
<td>f</td>
</tr>
</tbody>
</table>

Table 1

<table>
<thead>
<tr>
<th>s</th>
<th>p</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>p</td>
</tr>
<tr>
<td>i</td>
<td>s</td>
<td>p</td>
</tr>
</tbody>
</table>

Table 2

It should be remarked that this is simply a conventional notation to simplify the explanation. The operation thus defined is not closed, as the null resulting from the composition \(m \cdot f\) for gender, or \(s \cdot p\) for number does not belong with the initial set of possible values. This simplification is however consistent and reflects the real behaviour of morphological features unification in Spanish.

Formally, the components of these pseudovectors correspond to real vectors of the form

\[
g = (i, j) \\
n = (k, l) 
\]

so that \(i, j, k, l\) are either 0 or 1.

We define a basis with the vectors masculine \((1, 0)\) and feminine \((0, 1)\) which can be easily proved to be orthonormal. On this basis the invariable gender will be represented \((1, 1)\).

If we apply the dot product, as it is usually defined, we get:

\[
\begin{align*}
    m \cdot f & = (1, 0) \cdot (0, 1) = 0 \\
    m \cdot m & = (1, 0) \cdot (1, 0) = 1 \\
    f \cdot f & = (0, 1) \cdot (0, 1) = 1 \\
    m \cdot i & = (1, 0) \cdot (1, 1) = 1 \\
    f \cdot i & = (0, 1) \cdot (1, 1) = 1 \\
    i \cdot i & = (1, 1) \cdot (1, 1) = 2 
\end{align*}
\]
So we know that those couples of words that match for gender must have a dot product greater than or equal to 1.

A similar reflection can be done for number. So technically, when we say a word \( W_n \) is represented morphologically by \( M_n = (g,n) \), this is a simplified notation for

\[
M_n = \begin{bmatrix}
g_1 & n_1 \\
g_2 & n_2
\end{bmatrix}
\]

As pointed out above, this example has been oversimplified. Actually, in order to solve morphological dependencies, several other features must be considered. If we focus in on personal pronouns and possessive ambiguities, a new category person should be added. This category may take either of three values: first, second and third. To build a basis which is consistent with the ones we defined for gender and number, we must consider three vectors:

\[
\begin{align*}
p_1 &= (1, 0, 0) \\
p_2 &= (0, 1, 0) \\
p_3 &= (0, 0, 1)
\end{align*}
\]

that intuitively represent the three persons, on which we can define our dot product again, so that

\[
p_1 \cdot p_1 = p_2 \cdot p_2 = p_3 \cdot p_3 = 1\]

being all other products zero.

Summing up, the morphological representation for a word \( W_n \) in Spanish will be:

\[
M_n = \begin{bmatrix}
g_1 & n_1 & p_1 \\
g_2 & n_2 & p_2 \\
\end{bmatrix}
\]

We will exemplify this with a couple of examples:

7) El PNV dice que el Ejército impide una solución para Euskadi como la del Ulster.

8) Para que la convocatoria sea un éxito mayor que el del 14 D.

In (7) and (8) morphology will solve the anaphora as \( una \) solución is feminine and singular and \( la \) del shares these values, resulting in the dot product being greater than 0. The same applies for \( un \) éxito and \( el \) del, which are masculine and singular entities.

A basis for semantics

The two-level scheme we have adopted for explaining our morphological categorization, can be also useful as a reference for the coding of semantic features.

Attempts at capturing meaning using numeric representations have been a common place in recent years. Distributed representations have been used in connectionist approaches to NLP, in the implementation of neural networks for meaning representation, or in the automatic extraction of meaning from Machine Readable Dictionaries. In these and other fields numeric methods have proved to be both economic and robust.

All these methods rely implicitly on the structural semantic assumption that meaning can be expressed in terms of a restrictive set of semantic primitives. This concept, however, has been understood in quite different ways by researchers in these domains.

Niwa and Nitta (1994), for example, elaborate their word vectors relying on the concept of distance between words. Roughly stated, they take as primitives one thousand middle frequency words from the Collins English Dictionary, which they call origins. Intuitively, these primitive words constitute the basis for their vector space. The components are calculated counting the nodes that separate the word defined from each of the origin words. The word dictionary they use to exemplify is related directly to book. Origin word 2, and indirectly (through word) to unit. Origin 1, and people (through language). The resulting vector is thus (2, 1, 2). As they claim, «in principle, origin words can be freely chosen», since a change of basis can be defined as a linear transformation of components leaving invariant the relationship between couples of vectors. We must point, however, that this is only true if the dimension of the basis is not changed and one to one, linear correspondences can be defined between elements in one and the other basis.

Sutcliffe (1991a, 1991b, 1992), from a different perspective, relates meaning to a set of microfeatures which are more closely connected to the general idea of semantic primitives. These microfeatures were first selected arbitrarily, as he points out, and later extracted from the CED following an objective procedure. This method diverges from Niwa and Nitta’s as the set of features is not established in advance, but inferred from word analysis; those
words that are used more frequently in the definitions constitute the of origins.

We have been compelled to follow a different approach, mainly because of mundane limitations for dealing with Spanish. No dictionary similar to LDOCE or OALDCE is available to researchers in machine format; neither are reference networks or any similar structures. Furthermore, current dictionaries of Spanish do not include statistical information to describe their own contents, so that we can not rely on that information to elaborate our primitives’ list.

In order to overcome these drawbacks we have determined to elaborate our list of primitives out of the information contained in Conceptual Dictionaries, similar to Roget’s Thesaurus or Longman’s Lexicon. For this purpose, we have examined J. Casares (1975) and DILE (1995).

The latter includes in its synoptic hierarchy a list of 1274 head words, close in number to the sample used by Niwa & Nitta, what would allow us to establish comparisons. Unfortunately, we have not been able to trace relationships between the words selected as representative of common concepts, and the vocabulary used in the definitions, what might render our task hard and useless.

The former is organised around a set of primitives selected attending philosophical, rather than lexical criteria. Nevertheless, definitions are fairly consistent, and the system of conceptual primitives much more economical.

We cannot then solve this problem extracting the information directly from any of these dictionaries. Consequently, we have adopted a different strategy, which combines conceptual knowledge as it is structured in these dictionaries, with lexical knowledge as it comes organised in learners dictionaries. We are currently extracting features from DALE following Sutcliffe’s method. Once we have found a set of words using definitions, we will refer these to the concept head words in DILE and Casares in order to check consistency, eliminate redundant origin words and simplify the set of primitives. These tasks, however, are time consuming as the version of the dictionary we have is not structured in a database, but as running text in photocomposition format.

Eventually we hope that making use of the concept of tensor as vector of vectors, we will be able to refer these origin words to any set of semantic features, extracted from conceptual dictionaries in order to refine the network. The basis in principle should be linearly dependent on the second order basis of conceptual primitives. This information may sound redundant for our immediate purposes, but will enable us to make use of it in future applications, such as semantic disambiguation and tagging of corpora.

Conclusions and further work
We have presented along this paper an approach to anaphora resolution based on the assumption that a vector-tensor representation may simplify both the internal representation of grammatical and semantic information, and the process of matching antecedents with their anaphoric references.

We have concentrated on defining a basis for morphology and semantics as a first step in the process of finding a suitable model for anaphora. Syntax is our next goal and we are presently studying the influence of parameters such as function, position and distance with the aim of defining a basis for this type of information. As for pragmatics, it is still difficult to find a set of features which seizes all aspects involved in this knowledge source.

References


A LOGICAL FORMALISM FOR INTERGRAMMATICAL REPRESENTATIONS

Vincenzo Manca
Università di Pisa
Dipartimento di Informatica
Corso Italia, 40 - 56125 Pisa - Italy
e-mail: mancov@di.unipi.it

Keywords: Formal Grammar, Linguistic Analysis, Representation Formalism, Universal Grammar.

1 INTRODUCTION

This paper presents a coherent notational system of fundamental concepts for traditional grammar. Let us consider the usual concepts of linguistic analysis, e.g.: proposition, predicate, substantive, subject, attribute, apposition, complement, genitive, modifier, determiner, coordination, subordination, anaphora, deixis, sentence, quantification, ellipsis. These concepts are used in usual grammars on the basis of their informal linguistic evidence and often with criteria that depend on the particular language they refer to. Nevertheless, their theoretical relevance is based on very deep roots in Western philosophy from Plato, Aristotle and Dionysus Trax, to the Speculative Grammar of Scholastic philosophy, and eventually to the 'Grammar' and 'Logic' of Port Royal (17th century), where the terminological and conceptual apparatus of usual Indo-European grammars were elaborated (see [19]). Moreover, recent studies in the field of linguistic typology and universal grammar (see [9, 5, 20]) confirm the significance of the traditional apparatus.\(^1\)

Therefore, an attempt to systematically symbolize classical linguistic analysis, in terms of mathematical logic notations can be useful for several reasons. The idea of a logical formulation of classical linguistic analysis was already present in the first investigations of modern mathematical logic. Here we can find, even in the context of mathematical theories, some masterpieces of logical analyses of natural language. In this perspective, we can consider: Leibniz's Characteristica Universalis and Grammaticae Cognitiones [15]; Bertrand Russell's theory of propositions [21]; Frege's Begriffsschrift on symbolization of predicative structure of mathematical language [10]; Peano's Formulario Mathematico on notations for general mathematical concepts [16].

The first systematic attempt to use the mathematical logic apparatus in formal representation of (English) texts is Reichenbach's Logic of conversational language [18]. Nevertheless, Reichenbach's proposal is just a collection of interesting examples covering important phenomena of natural language expressiveness, rather than a definition of a rigorous formalism. Several developments emerged around the 1960s. First of these was Lambek's version of earlier categorial grammar [13] where grammatical correctness is considered in terms of deducibility within suitable formal systems. Second, the school of generative semantics which extended Chomsky's paradigm (see for example [4, 12]) claiming that every sentence of natural language has an underlying logical structure which represents its meaning. Third, Montague's approach originated in a few seminal papers [24, 7], where tools from logic are used to give extremely precise analyses of semantics of some fragments of English. While Lambek's approach was developed in a proof theoretic style, Montague's approach was carried on, in its crucial aspect, within a model theoretic setting. However, the above mentioned approaches (generative, model theoretic and proof theoretic) strongly influenced, directly or indirectly, implicitly or explicitly, the successive research on formal analysis of natural language, and in the related fields of knowledge representations, non classical logics, and computational linguistics (see for ex-

\(^1\)Greenberg's seminal paper in [9] develops, in terms of traditional grammatical notions, a structural analysis of 30 languages (European, African, Asiatic, Oceanian, American Indian) which identifies interesting language universals.
A feature, common of many logical approaches to natural language, is their complexity and specificity. On the contrary, one of the main objectives of this paper is to develop a wide range logical formalism we call INTERGRAMMA, comparable with first order logic in the formal description of mathematical theories or with programming languages in the specification of algorithms. This formalism consists of symbolic rules by means of which we can construct formulae built by lexemes of a specific lexicon and symbols expressing logico-grammatical functionalities. In this manner, the denotative content of simple, but non trivial texts can be adequately expressed by a set of intergrammatical formulae. An experimentum crucis for it could be: let \( L \) a lexicon of a language \( L_1 \), a person speaking a language \( L_2 \), and who knows INTERGRAMMA, consulting a bilingual dictionary defining \( L \) into \( L_2 \), should be able to reconstruct, with a good approximation, the meaning represented by an intergrammatical formula over the lexicon \( L \). In this sense, intergrammatical representations can be seen as a sort of formal counterpart of the linguistic projects such as: Volapük, Interlingua, Latino sine flexione, Esperanto, Glossa, Lincos, Loglan [14, 8, 3].

Of course, another important aspect of intergrammatical representations is the lexicon over which these representations are constructed. In order to get a true interlingual character, an intergrammatical formula has to contain lexemes without any semantic ambiguity. This aspect demands for a deep analysis archetypal lexicons, their nature and their structure. A theme for future work would be to find some possible connection between Thom's 16 archetypal logos determined by a topological analysis of natural language [22, 23] and our logical approach.

The style we adopt in presentation of our formalism is deductive in nature, because it is based on inference rules. Two possible extensions of the formalism are: rules for generating surface syntactic structure from intergrammatical representations and rules for expressing specific semantical aspects.

2 GENERAL LOGICO-SYNTACTIC CATEGORIES AND CONSTRUCTIONS

In every natural language there are expressions having the following four fundamental syntactic types: i) sentence (assertive, imperative, interrogative), ii) proposition iii) predicate, iv) substantive.

At a first approximation, we can assume that substantives represent individuals (of some - maybe high order - universe) of fundamental kinds: objects, masses, numbers, qualities, classes, instants, spaces, events, actions, processes, situations. Predicates represent properties over individuals. Propositions represent statements, and sentences represent assertions, questions or commands.

Let us call categoremata any expression having some fundamental syntactic type. The following are general postulates describing basic aspects common to almost all natural languages.

Postulate 1 Any proposition includes a predicate. ◊

Postulate 2 There are elements, usually called determiners, that allow us to transform predicates into substantives. ◊

Postulate 3 Any categoremata \( \varphi \) of (syntactic) type \( \tau \) includes a categoremata \( \ker(\varphi) \) of type \( \tau \) that we call its kernel.◊

We say that a categoremata \( \varphi \) is nuclear if \( \varphi = \ker(\varphi) \). If \( \varphi \) is a nuclear proposition, then its predicate is the biggest (w.r.t. the string inclusion) predicate included in \( \varphi \). For every categoremata \( \varphi \), if it is not nuclear, then the modifier of the categoremata \( \ker(\varphi) \) is the string difference \( \varphi \setminus \ker(\varphi) \).

Postulate 4 If \( \varphi \) is a nuclear proposition and the string difference \( \varphi \setminus \ker(\varphi) \) is not the empty string, then it is a substantive, called the subject of \( \varphi \). ◊

If \( \varphi \) is a non nuclear predicate including a substantive \( \gamma \) we say that \( \gamma \) is a complement of \( \ker(\varphi) \). If \( \gamma \) is nuclear it is a direct complement of \( \varphi \), otherwise it is an indirect complement of \( \varphi \).

Postulate 5 When propositions are related to some extra-linguistic elements, called situations, they determine assertions. ◊

Postulate 6 Linguistic elements, usually called anaphoric pronouns, refer to other categoremata. ◊

Postulate 7 Linguistic elements, usually called deictic pronouns, play a linguistic role depending on the situation they are used. ◊
Anaphora is accomplished by usual pronouns that refer to linguistic entities (in English: he, she, it, ...). Deixis occurs with usual deictic particles, such as personal and demonstrative pronouns that refer to extralinguistic entities.

Postulate 8 Linguistic elements, usually called conjunctive particles, combine categoremata having the same type and produce categoremata of that type. ◇

Postulate 9 Linguistic elements, usually called relative pronouns transform assertions into predicates. ◇

Postulate 10 Linguistic elements, usually called indefinite pronouns, transform assertions into universal assertions. ◇

Postulate 11 Any categorema can be transformed into a substantive. ◇

Postulate 12 There are linguistic elements indicating classes and numbers. ◇

The following schemata of syntactic constructions are derived by the above postulates.

- predication
- determination
- modification
- complementation
- assertion
- anaphora
- deixis
- coordination
- relativization
- quantification
- substantivation
- pluralization

We do not claim that this list includes all the constructions of natural languages. We only think that most texts of any natural language can be represented, in its internal logical structure, by suitable combinations of these syntactical operations.

3 Notations for intergrammatical concepts

In natural languages, predication is usually realized by affix modification on verbal categoremata, by copulative particles, or by juxtaposition. We consider two forms of intergrammatical predication. The first one is realized by the $\epsilon$ operator that is the natural translation of the usual subject-predicate grammatical schema. The symbol $\epsilon$ was introduced by Peano [16] as a formal representation of predication of mathematical language (afterwards, it took on the set-membership meaning that is now standard in mathematical language). The second form of predication is realized by the usual equality relation $\equiv$.

There are two kinds of intergrammatical determination. The first determination operator, given a predicate $Pred$, yields the uniquely determined object $\iota(Pred)$ which satisfies the property represented by predicate $Pred$. The second determination operator, given a predicate $Pred$, yields $\eta(Pred)$, an undetermined element which satisfies the property determined by $Pred$. Operator $\iota$ is Peano-Russell’s descriptive iota operator, while $\eta$ is very similar to Hilbert-Bernays’ eta operator. Reichenbach uses them to describe logically the defined/indivisible difference of natural language [18].

In the following, we assume a set of atomic predicates represented by lexemes of a natural language (e.g. English nouns, verbs and adjectives), that we feel free to specify herein implicitly when they occur in the examples.

Consider the proposition: There is a pen on the table. This means that, in the situation which the statement refers to, there is something that is a pen, something that is the table, and the first thing is placed on the second:

\[ \circ \vdash a \in \text{pen} \]  \[ (1) \]

\[ \circ \vdash b \in \text{table} \]  \[ (2) \]

\[ \circ \vdash a \in \text{on}_b. \]  \[ (3) \]

The $\circ$ symbol denotes the enunciation situation, that is the Hic-et-nunc or Here-and-now state of the affairs with respect to which any extralinguistic element (for spatio-temporal-personal deixis) has to be related. A similar symbol was already used in a analogue way by Reichenbach [18]. Symbols $a, b$, are individual constants, and the symbol $\vdash$ is the assertion symbol, already used by Russell, that connects situations and propositions producing declarative
assertions. When we want obtain commands or questions, then assertion is respectively denoted by ⊢! or ⊢?. (Possibly adding to ? a bound variable).

It is important to note the different roles of descriptive operators, individual constants, and anaphoric constants we are introducing now. In natural language, we can speak of things that do not exist, i.e. we have to distinguish between descriptive entities and existential entities. When, w.r.t. a situation s, we assert a ∈ Pred we assume that something exists in the situation s which verifies the property Pred and we introduce the name a for such an entity. Conversely, given a definite or indefinite description, if we want to refer to the substantive that it identifies, without any existential commitment, then, we put an anaphoric reference such as A := Description, by assuming that B := Description states another anaphoric link, where A and B are two different hypothetical entities satisfying Description.

For example, indicating complementation by ‘dot suffixing’, we can formalize Where is a pen? by:

\[ A := \eta(\text{pen}) \]

\[ \circ \vdash \exists x. (A \in \text{in} \cdot x). \]

and we formalize Take a book! by:

\[ \circ \vdash a = \iota \cdot \text{listen} \]

\[ B := \eta(\text{book}) \]

\[ \circ \vdash ! a \in \text{take} \cdot B . \]

In natural languages, modification and complementation are usually realized by attributive phrases, adverbs, or by complementative particles. A formal representation of these grammatical constructs has to distinguish between expressions such as ‘big man’ and ‘page of the book’. The former leads to the modification big(man), the latter leads to the complementation page.\textit{u} (book). Particles play either modifiative or complementative roles: by changing the meaning of predicates or by extending the number of arguments of the predicates and by indicating complementation roles: \textit{relationship}, causality, finality, location, instrument, quality, quantity, comparison. For example, in English ‘look at’, ‘look after’, ‘look for’, ‘look forward’ express different binary relations where the same basic verb is modified by different prepositions; while in the expression ‘to take a with b’ the predicate ‘take’ is extended to represent a ternary relation, where the role of the argument b is specified by ‘with’. Unfortunately, similar roles are expressed by different particles in different contexts and the same particle can express very different roles in different contexts. For example, in English we have: ‘We agree about most things.’, ‘Let us agree on a date!’, ‘He was waiting for her!’, ‘He left for Paris.’.

In direct complementation, a predicate is extended with a nuclear substantive, while in indirect complementation it is extended with a substantive after a suitable modification. For example, \text{Pred.finality}(a) indicates that a is a substantive that specifies the meaning of Pred with a final role.

Modification, complementation and predication (at different levels) are related to the vague
ess of natural language, inasmuch as they ‘fuse’ meanings. For example, the semantic role of ‘big’ in modifications such as: big\textit{(man)} or big\textit{(success)} is certainly vague. Sometimes it is important to reduce vagueness, and a formal setting is an essential tool for such a task; nevertheless vagueness cannot be completely avoided, if some intrinsic peculiarities of natural language are represented. At a level of logical formalization, it is appropriate to represent modification and complementation differently from predication. To show the importance of this distinction, consider the formalization, considered by Weinreich [25], of the sentence:

\textit{The three bitterly crying children walked home fast.}

Here the constructs of predication, modification and complementation have the same logical representation:

\[ \exists x y t f g. \]

\[ 3\text{child}(x, t) \land f(x) \land cry(f, t) \land \text{bitter}(f, t) \land \]

\[ \text{mention}(x, \circ) \land g(x) \land \text{walk}(g, y, t) \land \]

\[ \text{fast}(g, t) \land \text{home}(y, t) \land \text{have}(x, y, t) \land \]

\[ \text{before}(t, t_0) \land \text{time}(\circ, t_0). \]

One inadequacy of the above formalization is the arbitrariness in the number and in the posi-

\[ \text{The knowledge of many languages is, to a great extent, based on the correct use of particles. Often, the choice of a specific particle confers expressive peculiarities which are very important at a \textit{comnotative} level. Of course, this kind of expressive power of natural language is beyond the limits of the formalism we are considering. It is strictly related to the \textit{denotative} aspect of language.} \]
tion of the arguments of predications. Why not consider a formula \( \text{walk}(g, t) \land \text{to}(g, y, t) \) instead of \( \text{walk}(g, y, t) \), and why not consider the formula
\[
g(x, t) \land \text{walk}(g) \land \text{to}(g, y)
\]
by moving the temporal parameter as an argument of the predicate \( g \)? Furthermore, there is a very subtle implicit assumption: some linguistic information is represented directly by first order predicates: \( \text{have}, \text{home}, \text{mention}, \text{time}, \text{before} \); the others are second order predicates: \( \text{walk}, \text{cry}, \text{bitter}, \text{fast} \). What is the basis for this distinction? It seems that it is dependent upon the context and on the particular language on which is based, and of course the choice is a limit for a formal theory of logical representation of natural language. The same proposition leads to the following intergrammatical formalization where the aforementioned inadequacies are avoided (\( \land \) has the usual meaning of logical conjunction):

\[
\begin{align*}
\Theta & \vdash s \in \text{past}(\text{situation}) \quad (9) \\
a & := \alpha(\text{first}(\text{is} \in \text{act}(\text{child}))) \quad (10) \\
b & := \alpha(\text{house}(a)) \quad (11) \\
P & := (\text{fast}(\text{walk})).\text{destination}(b) \quad (12) \\
Q & := \text{bitter}(\text{cry}) \quad (13) \\
s & \vdash (a \in (P \land Q)) \quad (14)
\end{align*}
\]

Variables are the formal counterparts of indefinite pronouns. They allow the expression of relativization and (universal) quantification (in many languages indefinite and relative pronouns are very similar, e.g., who/whoever, which/whatever, …). Relativization allows us to obtain a predicate from a sentence where a variable occurs. In many languages this construct is expressed by relative clauses. For example, from \((x \in \text{love} . b)\) abstracting with respect to \( x \), we can obtain the predicate \((\lambda x . x \in \text{love} . b)\) or, more concisely, by using the circumflex accent instead of \( \lambda \), \((\hat{x} \in \text{love} . b)\) that identifies the property of any individual such that he loves \( b \).

This kind of relativization is present, but in a simpler form, in Reichenbach [18] and is very similar to the Church’s lambda abstraction, which is the basis of lambda calculus. Relativization is fundamental in the treatment of quantification that overcomes the usual inadequacies of the classic mathematical logic with respect to natural language.

Logical representation of universal quantification becomes problematic in anaphoric contexts. In literature, donkey-sentences and love-sentences are examples of this phenomenon:

\[
\text{Every man loves a woman who loves him.}
\]

In the proposition, the indefinite term ‘Every man’ is dominated by a universal quantification that forbids fixing any anaphoric assignment for ‘him’. The usual logical formalizations in first order logic impose an implicative structure that is not present in the corresponding propositions and are unsatisfactory for several reasons. The following formalization is intrinsically related to the previous treatment of relativization:

\[
\begin{align*}
A & := \eta(\text{situation}) \quad (15) \\
P & := (g \in \text{woman} \land \exists \in \text{loves}(x)) \quad (16) \\
A & \vdash \forall x. A \vdash x \in \text{man} : x \in \text{love} . \eta(P) \quad (17)
\end{align*}
\]

4 Forma\textsuperscript{r}al rules for intergrammatical representations

Consider the following 20 symbols (comma and final dot are not symbols of the alphabet, but punctuation marks). We already introduced most of them informally. The intuitive meaning of the others is the customary one that they have in formal mathematical language. Only the symbol \( \pi \) has the specific sense of a predicate modifier to indicate pluralities, i.e. non empty classes of objects in the same spatio-temporal vicinity.

\[
\epsilon, =, \vdash, \Pi, \vdash ? := \xi, \eta, \ldots, \{, \}, \}
\]

\[
\{, \}, \neg, \land, \lor, \forall, \lambda, \pm \pi, \Theta
\]

Let \( A \) be the alphabet consisting of the previous 20 symbols plus some auxiliary symbols (dot, parentheses and colon)\(^3\), and including also the following alphabets:

\[
\begin{align*}
\mathcal{I} & = \{a, b, \ldots\} \quad \text{(individuals)} \\
\mathcal{V} & = \{x, y, \ldots\} \quad \text{(variables)} \\
\mathcal{S} & = \{s, t, \ldots\} \quad \text{(situations)} \\
\mathcal{P} & = \{A, B, \ldots\} \quad \text{(pronouns)}
\end{align*}
\]

Consider also the following set \( \mathcal{T} \) of types:

\[
\text{Pred, Subst, Prop, Sent, Ind, Sit, Var, Pron, Ref, Class, Num, Cat.}
\]

\(^3\)For uniformity with standard usage, we allow some notational overloading, but this will not introduce any ambiguity.
Let $\mathcal{L}$ be a lexicon, a set of elements called lexemes. We assume that to any lexeme $\alpha$ is associated a linguistic form $\alpha'$ belonging to a set $\mathcal{W}$ of linguistic forms. An intergrammatical representation over $\mathcal{L}$ is a set of formulae $\varphi$ such that, for each of them, the type assignment

$$\varphi : \text{Sent}$$

is derived in the following formal system of type assignments.

### 4.1 Typing Rules

In the sequel, $\mathbb{N}$ is the set of natural numbers. Greek letters different from $\epsilon, \iota, \eta, \lambda$, are used as syntactical variables: $\alpha, \beta, \gamma$ ranging in the sequences of $\mathcal{A} \cup \mathcal{L}$ and $\tau$ ranging in $\mathcal{T}$ (also some latin letter will be, improperly, used as syntactical variables). As usual, $\text{free}(\alpha)$ will indicate the free variables occurring in $\alpha$.

<table>
<thead>
<tr>
<th>$\alpha \in \mathcal{L}$</th>
<th>$\alpha \in \mathcal{P}$</th>
<th>$\alpha \in \mathbb{N}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha : \text{Pred}$</td>
<td>$\alpha : \text{Pron}$</td>
<td>$\alpha : \text{Num}$</td>
</tr>
<tr>
<td>$\alpha \in \mathcal{T}$</td>
<td>$\alpha \in \mathcal{S} \cup {\top}$</td>
<td>$\alpha \in \mathcal{V}$</td>
</tr>
<tr>
<td>$\alpha : \text{Ind}$</td>
<td>$\alpha : \text{Sit}$</td>
<td>$\alpha : \text{Var}$</td>
</tr>
<tr>
<td>$\alpha : \text{Subst}$</td>
<td>$\alpha : \text{Subst}$</td>
<td>$\alpha : \text{Subst}$</td>
</tr>
<tr>
<td>$\alpha : \text{Cat}$</td>
<td>$\alpha : \text{Cat}$</td>
<td>$\alpha : \text{Cat}$</td>
</tr>
</tbody>
</table>

**Basic Rules and Deixis**

<table>
<thead>
<tr>
<th>$\alpha : \text{Subst}, \beta : \text{Pred}$</th>
<th>$\alpha : \text{Subst}, \beta : \text{Subst}$</th>
<th>$\alpha = \beta : \text{Prop}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \in \beta : \text{Prop}$</td>
<td>$\alpha \in \beta : \text{Prop}$</td>
<td>$\alpha = \beta : \text{Prop}$</td>
</tr>
</tbody>
</table>

**Predication**

<table>
<thead>
<tr>
<th>$\alpha : \text{Pred}, \beta : \text{Subst}$</th>
<th>$\alpha : \text{Pred}, \beta : \text{Pred}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(\beta) : \text{Subst}$</td>
<td>$\alpha(\beta) : \text{Subst}$</td>
</tr>
</tbody>
</table>

**Quantification**

<table>
<thead>
<tr>
<th>$\alpha : \text{Pred}, \beta : \text{Subst}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha, \beta : \text{Pred}$</td>
</tr>
</tbody>
</table>

**Modification and Complementation**

<table>
<thead>
<tr>
<th>$\alpha : \text{Pred}, \beta : \text{Subst}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda\alpha, \beta : \text{Pred}$</td>
</tr>
</tbody>
</table>
\[ \frac{\alpha : \text{Pred}}{\{\alpha\} : \text{Class}} \]

\[ \frac{\alpha : \text{Class}}{\alpha : \text{Subst}} \]

\[ \frac{\epsilon(\pi(\alpha)) : \text{Class}}{\eta(\pi(\alpha)) : \text{Class}} \]

\[ \begin{array}{ll}
\alpha : \text{Num}, \beta : \text{Pred} & \\
\alpha(\beta) : \text{Pred} & \\
\pi(\alpha) : \text{Pred} & \\
\alpha \epsilon \beta : \text{Prop} & \\
\alpha \epsilon \beta : \text{Num} & \\
\pm \alpha : \text{Num} & \\
\end{array} \]

Pluralization

4.2 SOME ASSERTION RULES

A complete formal semantics for intergrammatical formulae is beyond the scope of this paper. Nevertheless, we can argue that, in the same style of typing rules, we may add inference rules allowing us to derive assertions from other assertions. In fact, the ability to generate acceptable paraphrases is an essential aspect of comprehension. Therefore, rules which express such a competence add a new level to a formal description of grammar. We give only a partial list of these rules, in order to communicate the basic idea. The rules we will present in the sequel may be more appropriately defined as structural assertion rules, inasmuch as they involve the 20 logical symbols of our formalism.

As usual, \( \alpha[x] \) will indicate an expression where the variable \( x \) occurs, in this case \( \alpha[a] \) is obtained from \( \alpha[x] \), by replacing all occurrences of \( x \) with \( a \) (note that [ ] is already used for substantiation).

\[ \frac{\sigma \vdash \alpha = \epsilon(\beta)}{\sigma \vdash \epsilon \epsilon \beta} \]

\[ \frac{\sigma \vdash \alpha = \eta(\beta)}{\sigma \vdash \epsilon \epsilon \beta} \]

\[ \frac{\sigma \vdash \alpha = \epsilon(\beta), \sigma \vdash \gamma = \epsilon(\beta)}{\sigma \vdash \alpha = \gamma} \]

\[ \begin{array}{ll}
\alpha \wedge \beta : \text{Prop} & \\
\sigma \vdash [\alpha] \epsilon \text{although} & \\
\sigma \vdash a = \iota(\text{assert}) & \\
\sigma \vdash a \epsilon \text{believe.}(\{A \wedge \alpha \wedge \beta\}) & \\
A := \{[\lambda x. x : \alpha \wedge \beta] \epsilon \text{small}\} & \\
\end{array} \]

REFERENCES


Parlevink
The Parlevink project is a language theory and technology project of the University of Twente. Starting-point is a (software) engineering approach to natural language and natural language processing systems. Special attention is paid to possible interactions with theoretical computer science.

Research Topics
In subprojects research is devoted to topics as syntax, semantics and pragmatics. Dialogue modeling is also part of the project, just as connectionist language learning and processing. Support research is done in the areas of formal languages and neural networks. Integration of the different topics takes place in the SCHISMA subproject: design and realization of a prototype natural language accessible theater information and booking system (joint with KPN Research). Other topics that play a role in this integration project are the embedding of such a system in a ‘digital city’ and World Wide Web, speech processing, natural language technology assessment and societal aspects. Another integration project that will start in 1996 is the European funded 21-project on multimedia information retrieval. Other funded projects that will start in 1996 are on neural networks and on societal effects of scientific research.

Ph.D. Research
In 1996 four to five Ph.D. students will perform their research in this project, partly in cooperation with other projects (robust language analysis, pragmatics, dialogue modeling and analysis, design and specification of NLP systems). In the spring of 1995 the Ph.D. thesis 'Little Linguistic Creatures' appeared. In 1996 a Ph.D. thesis on pragmatics in language technology is expected.

Students
It is expected that in the forthcoming years about thirty students will do their M.Sc. research in the Parlevink project. In addition there are many students who take a practical term outside the university (language and neuro engineering).

Workshops
Twice a year a Twente Workshop on Language Technology (TWLT) is organized. Starting as a local event these workshops have now become meetings of international specialists on topics from language engineering. In 1996 two workshops will be organized. The first one, in June, will be on “Dialogue Modeling in Natural Language Systems” (TWLT11); the second on, in September will be on “Computational Humor: Automatic Interpretation and Generation of Verbal Humor” (TWLT12).
Twente Workshops on Language Technology

The TWLT workshops are organised by the PARLEVINK project of the University of Twente. The first workshop was held in Enschede, the Netherlands on March 22, 1991. The workshop was attended by about 40 participants. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 1 (TWLT 1)
Tomita's Algorithm: Extensions and Applications
Eds. R. Heemels, A. Nijholt & K. Sikkel, 103 pages.

Preface and Contents
A. Nijholt (University of Twente, Enschede). (Generalised) LR Parsing: From Knuth to Tomita.
G.J. van der Steen (Vleermuis Software Research, Utrecht). Unrestricted On-Line Parsing and Transduction with Graph Structured Stacks.
T. Vosse (NICI, Nijmegen). Detection and Correction of Morpho-Syntactic Errors in Shift-Reduce Parsing.
R. Heemels (OCé Nederland, Venlo). Tomita's Algorithm in Practical Applications.
M. Lankhorst (University of Twente, Enschede). An Empirical Comparison of Generalised LR Tables.
K. Sikkel (University of Twente, Enschede). Bottom-Up Parallelization of Tomita's Algorithm.

The second workshop in the series (TWLT 2) has been held on November 20, 1991. The workshop was attended by more than 70 researchers from industry and university. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 2 (TWLT 2)
Linguistic Engineering: Tools and Products.

Preface and Contents
A. Nijholt (University of Twente, Enschede). Linguistic Engineering: A Survey.
B. van Bakel (University of Nijmegen, Nijmegen). Semantic Analysis of Chemical Texts.
T. Vosse (NICI, Nijmegen). Detecting and Correcting Morpho-syntactic Errors in Real Texts.
A. van Rijn (CIA/Delft University of Technology, Delft). A Natural Language Interface for a Flexible Assembly Cell.
J. Honig (Delft University of Technology, Delft). Using Delta in Natural Language Front-ends.
D. van den Akker (IBM Research, Amsterdam). Language Technology at IBM Nederland.
The third workshop in the series (TWLT 3) was held on May 12 and 13, 1992. Contrary to the previous workshops it had an international character with eighty participants from the U.S.A., India, Great Britain, Ireland, Italy, Germany, France, Belgium and the Netherlands. The proceedings were available at the workshop. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 3 (TWLT 3)

Connectionism and Natural Language Processing
Eds. M.F.J. Drossaers & A. Nijholt, 142 pages.

Preface and Contents

L.P.J. Veenenturf (University of Twente, Enschede). Representation of Spoken Words in a Self-Organising Neural Net.

P. Wittenburg & U. H. Frauenfelder (Max-Planck Institute, Nijmegen). Modelling the Human Mental Lexicon with Self-Organising Feature Maps.


W. Daelemans & A. van den Bosch (Tilburg University, Tilburg). Generalisation Performance of Back Propagation Learning on a Syllabification Task.

E.-J. van der Linden & W. Kraaij (Tilburg University, Tilburg). Representation of Idioms in Connectionist Models.

J.C. Scholtes (University of Amsterdam, Amsterdam). Neural Data Oriented Parsing.


M.F.J. Drossaers (University of Twente, Enschede). Hopfield Models as Neural-Network Acceptors.


R. Reilly (University College, Dublin). An Exploration of Clause Boundary Effects in SRN Representations.

S.M. Lucas (University of Essex, Colchester). Syntactic Neural Networks for Natural Language Processing.

R. Milkkulainen (University of Texas, Austin). DISCERN: A Distributed Neural Network Model of Script Processing and Memory.

The fourth workshop in the series has been held on September 23, 1992. The theme of this workshop was "Pragmatics in Language Technology". Its aim was to bring together the several approaches to this subject: philosophical, linguistic and logic. The workshop was visited by more than 50 researchers in these fields, together with several computer scientists. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 4 (TWLT 4)

Pragmatics in Language Technology

Preface and Contents

D. Nauta, A. Nijholt & J. Schaake (University of Twente, Enschede). Pragmatics in Language technology: Introduction.

Part 1: Pragmatics and Semiotics

J. van der Lubbe & D. Nauta (Delft University of Technology & University of Twente, Enschede). Semiotics, Pragmatism, and Expert Systems.

F. Vandamme (Ghent). Semiotics, Epistemology, and Human Action.

H. de Jong & W. Werner (University of Twente, Enschede). Separation of Powers and Semiotic Processes.
R. Bod (University of Amsterdam). Data Oriented Parsing as a General Framework for Stochastic Language Processing.
M. Stefanova & W. ter Stal (University of Sofia / University of Twente). A Comparison of ALE and PATR: Practical Experiences.
J.P.M. de Vreught (University of Delft). A Practical Comparison between Parallel Tabular Recognizers.
M. Verlinden (University of Twente). Head-Corner Parsing of Unification Grammars: A Case Study.
Th. Stürmer (University of Saarbrücken). Semantic-Oriented Chart Parsing with Defaults.
G. Satta (University of Venice). The Parsing Problem for Tree-Adjoining Grammars.
F. Barthélémy (University of Lisbon). A Single Formalism for a Wide Range of Parsers for DCGs.
C. Cremers (University of Leiden). Coordination as a Parsing Problem.
M. Wirén (University of Saarbrücken). Bounded Incremental Parsing.
V. Kubon and M. Platek (Charles University, Prague). Robust Parsing and Grammar Checking of Free Word Order Languages.
V. Srinivasan (University of Mainz). Punctuation and Parsing of Real-World Texts.
T.G. Vosse (University of Leiden). Robust GLR Parsing for Grammar-Based Spelling Correction.

The seventh workshop in the series took place on 15 and 16 June 1994. It was devoted to the topic "Computer-Assisted Language Learning" (CALL). The aim was to present both the state of the art in CALL and the new perspectives in the research and development of software that is meant to be used in a language curriculum. By the mix of themes addressed in the papers and demonstrations, we hoped to bring about the exchange of ideas between people of various backgrounds.

Proceedings Twente Workshop on Language Technology 7 (TWLT 7)
Computer-Assisted Language Learning
Eds. L. Appelo, F.M.G. de Jong, 133 pages.

Preface and Contents
L. Appelo, F.M.G. de Jong (IPO / University of Twente). Computer-Assisted Language Learning: Prolegomena
M. van Bodegom (Eurolinguist Language House, Nijmegen, The Netherlands). Eurolinguist test: An adaptive testing system.
B. Cartigny (Escape, Tilburg, The Netherlands). Discatext CD-ROM XA.
H. Altay Guvenir, K. Oflazer (Bilkent University, Ankara). Using a Corpus for Teaching Turkish Morphology.
G. Kempen, A. Dijkstra (University of Leiden, The Netherlands). Towards an integrated system for spelling, grammar and writing instruction.
F. Kronenberg, A. Krueger, P. Ludewig (University of Osnabrueck, Germany). Contextual vocabulary learning with CAVOL.
S. Lobbe (Rotterdam Polytechnic Informatica Centrum, The Netherlands). Teachers, Students and IT: how to get teachers to integrate IT into the (language) curriculum.
C. Schwind (Universite de Marseille, France). Error analysis and explanation in knowledge based language tutoring.

J. Thompson (CTI, Hull, United Kingdom/EUROCALL). TELL into the mainstream curriculum.

M. Zock (Limsi, Paris, France). Language in action, or learning a language by whatching how it works.

Description of systems demonstrated:

APPEAL (Institute of Perception Research, Eindhoven)

Bonacord, Méli-Mélo, et al. (School of European Languages & Cultures, University of Hull)

Computer BBS in language instruction (English Programs for Internationals, University of South Carolina)

Discatext (Escape, Tilburg)

Error analysis and explanation (CNRS, Laboratoire d'Informatique de Marseille)

ItalCultura, RumboHispano and IVANA (Norwegian Computing Centre for the Humanities, Harald)

It's English (Department of Educational Sciences, Utrecht University)

Multimedia course for learning Dutch (SLO, Enschede)

Part of CATT (Department of Computer Engineering and Information Science, Bilkent University, Ankara)

PROMISE (Institut für Semantische Informationsverarbeitung, Universität Osnabrück)

Speech-Melody trainer (Institute of Percepcion Research, Eindhoven)

The Rosetta Stone (Eurolinguist Language House Nijmegen)

Verbarium and Substantarius (SOS Nijmegen)

WOORD (Applied Linguistics Unit, Delft University of Technology)

FLUENT-II (George Mason University, Washington)

The eighth workshop in the series took place on 1 and 2 December 1994. It was devoted to speech, the integration of speech and natural language processing, and the application of this integration in natural language interfaces. The program emphasized research of interest for the themes in the framework of the Dutch NWO programme on Speech and Natural Language that started in 1994.

Proceedings Twente Workshop on Language Technology 8 (TWLT 8)
Speech and Language Engineering
Eds. L. Boeves, A. Nijholt, 176 pages.

Preface and Contents

Chr. Dugast (Philips, Aachen, Germany). The North American Business News Task: Speaker Independent, Unlimited Vocabulary Article Dictation

P. van Alphen, C. In't Veld & W. Schelvis (PTT Research, Leidschendam, The Netherlands). Analysis of the Dutch Polyphone Corpus


J.M. McQueen (Max Planck Institute, Nijmegen, The Netherlands). The Role of Prosody in Human Speech Recognition

L. ten Bosch (IPO, Eindhoven, the Netherlands). The Potential Role of Prosody in Automatic Speech Recognition

P. Baggio, E. Gerbino, E. Giachin & C. Rullent (CSELT, Turin, Italy). Spontaneous Speech Phenomena in Naive-User Interactions

M.F.J. Drossaers & D. Dokter (University of Twente, Enschede, the Netherlands). Simple Speech Recognition with Little Linguistic Creatures

H. Helbig & A. Mertens (FernUniversität Hagen, Germany). Word Agent Based Natural Language Processing

Geunbae Lee et al. (Pohang University, Hyoja-Dong, Pohang, Korea). Phoneme-Level Speech and natural Language Integration for Agglutinative Languages.


G. Veldhuijzen van Zanten & R. op den Akker (University of Twente, Enschede, the Netherlands). More Efficient Head and Left Corner Parsing of Unification-based Formalisms.

G.F. van der Hoeven et al. (University of Twente, Enschede, the Netherlands). SCHISMA: A natural Language Accessible Theatre Information and Booking System.

G. van Noord (University of Groningen, the Netherlands). On the Intersection of Finite Staate Automata and Definite Clause Grammars.

R. Bod & R. Scha (University of Amsterdam, the Netherlands). Prediction and Disambiguation by Means of Data-Oriented Parsing.

The ninth workshop in the series took place on 9 June 1995. It was devoted to empirical methods in the analysis of dialogues, and the use of corpora of dialogues in building dialogue systems. The aim was to discuss the methods of corpus analysis, as well as results of corpus analysis and the application of such results.

Proceedings Twente Workshop on Language Technology 9 (TWLT 9)
Corpus-based Approaches to Dialogue Modelling

Preface and Contents

N. Dahlbäck (NLP Laboratory, Linköping, Sweden). Kinds of agents and types of dialogues.


J. Alexanderson & N. Reithinger (DFKI, Saarbrücken, Germany). Designing the dialogue component in a speech translation system – a corpus-based approach.


M. Rats (ITK, Tilburg, the Netherlands). Referring to topics – a corpus-based study.


N. Fraser (Vocalis Ltd, Cambridge, UK). Messy data, what can we learn from it?

J.A. Andernach (University of Twente, Enschede, the Netherlands). Predicting and interpreting speech acts in a theatre information and booking system.

The tenth workshop in the series took place on 6-8 December 1995. This workshop is organized in the framework provided by the Algebraic Methodology and Software Technology (AMAST) movement. It focussed on algebraic methods in formal languages, programming languages and natural languages. Its aim was to bring together those researchers on formal language theory, programming language theory and natural language description theory, that have a common interest in the use of algebraic methods to describe syntactic, semantic and pragmatic properties of language.
Preface and Contents

Teodor Rus (Iowa City, USA). Algebraic Processing of Programming Languages.

Elco Visser (Amsterdam, NL). Polymorphic Syntax Definition.


Teodor Rus & James S. Jones (Iowa City, USA). Multi-layered Pipeline Parsing from Multi-axiom Grammars.


François Barthélémy (Paris, F).A Generic Tabular Scheme for Parsing.


Michael Moortgat (Utrecht, NL). Multimodal Linguistic Inference.

R.C. Berwick (MIT, USA). Computational Minimalism: The Convergence of the Minimalist Syntactic Program and Categorial Grammar

Annus V. Groenink (Amsterdam, NL). A Simple Uniform Semantics for Concatenation-Based Grammar.

Grzegorz Rozenberg (Leiden, NL) Theory of Texts (abstract only).

Jan Rekers (Leiden, NL) & A. Schürr (Aachen, D). A Graph Grammar Approach to Graphical Parsing.

Sándor Horváth (Debrecen, H)Strong Interchangeability and Nonlinearity of Primitive Words


Vladimir A. Fomichev (Moscow, Russia). A Variant of a Universal Metagrammar of Conceptual Structures.

Theo M.V. Janssen (Amsterdam, NL). The Method of Rosetta, Natural Language Translation Using Algebras.


Pál Dömösi (Kossuth University, Debrecen, Hungary) & Jürgen Duske (University of Hannover, Germany). Subword Membership Problem for Linear Indexed Languages.


Vincenzo Manca (Pisa, I). A Logical Formalism for Intergrammatical Representation.

The proceedings of the workshops can be ordered from Vakgroep SETI, Department of Computer Science, University of Twente, P.O. Box 217, NL-7500 AE Enschede, The Netherlands. E-mail orders are possible: bijron,hoogvlie@cs.utwente.nl.