University of California
Santa Cruz

Analysis and Transformation of Logic Programs

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer and Information Sciences

by

Kjell Erik Post

December 1994

The dissertation of Kjell Erik Post is
approved:

Allen Van Gelder

Charles E. McDowell

Wayne Dai

Dean of Graduate Studies and Research

Contents

Abstract **viii**

1. Introduction **1**

1.1 The Translation Process 3

1.2 Background and Prior Work 3

1.2.1 Parsing 3

1.2.2 Analysis 5

1.2.3 Transformations 8

1.3 Summary of Contributions 9

2. Preliminaries **11**

2.1 Context Free Grammars 11

2.1.1 Derivations and Parse Trees 12

2.1.2 LR Parsing and Parser Generation 13

2.2 Logic Programming 16

2.2.1 Syntax 16

2.2.2 Substitutions and Unification 18

2.2.3 Procedural Semantics 19

2.2.4 Cuts 21

2.2.5 Modes 21

2.2.6 Definite Clause Grammars 22

3. The Parser Generator DDGEN **25**

3.1 Introduction and Background 25

3.1.1 Definitions 27

3.1.2 Background and Prior Work 27

3.1.3 Summary of Contributions 30

3.2 Deferred Decision Parsing 32

3.3 Local Operator Declarations 37

3.4 Ambiguities at Run Time and Induced Grammars 38

3.5 Application to Definite Clause Grammars 43

3.6 Implementation 44

4. Parsing Prolog 47

4.1 Introduction 47

4.2 The Structure of Prolog 48

4.3 Subtleties of Prolog Syntax 49

4.4 Rectifying Prolog 52

4.5 The Prolog Grammar 53

4.6 Implementation and Results 56

5. Bottom-Up Evaluation of Attribute Grammars 57

5.1 Introduction and Background 57

5.1.1 Related Work 58

5.1.2 Summary of Contributions 60

5.2 Definitions 60

5.3 Transformation Methods 66

5.3.1 Method 1: Synthesized Functions 67

5.3.2 Method 2: Coroutines 70

6. Mutual Exclusion Analysis 73

6.1 Introduction 73

6.1.1 Related Work 75

6.1.2 Summary of Contributions 76

6.2 Definitions 77
6.2.1 Rule/Goal Graphs 78
6.3 Deriving Mutual Exclusion 78
6.3.1 Algorithm for Propagating Mutual Exclusion 81
6.3.2 Termination and Complexity 84
6.3.3 Correctness 84
6.3.4 Description of prop 86
6.4 A Larger Example 87
6.5 Limitations 89
6.6 Coding Style and the Use of Cuts 90
7. Epilogue **94**
7.1 Concluding Remarks 94
7.2 Future Work 95
References **98**
Index **104**

List of Figures

1.1 Overview of the translation process 4

2.1 The LR parser 14

2.2 Parse table and execution trace for an LR parser 15

2.3 Structure of logic programs 17

2.4 Algorithm for computing an answer substitution 20

2.5 The effect of a cut 21

2.6 SLD-tree 24

3.1 Standard Parser Generator and Deferred Decision Parser Generator 30

3.2 An example run-time operator table 33

3.3 Subset grammar for Prolog terms 33

3.4 Deferred decision parsing example 37

3.5 Skeleton for induced grammar 40

3.6 Sorted operator table 43

3.7 Grammar subset for ML operator expressions 46

4.1 Prolog syntax for encoding fixity and associativity 49

4.2 Prolog grammar 55

5.1 Attribute grammar definition for binary numbers 61

5.2 Dependency graphs for productions 62

5.3 Parse tree and dependencies for "`10.1`" 63

5.4 Dependencies for grammar symbols 63

5.5 Strong composite graphs. 64

5.6 Definition of procedure $body(a)$ 68

5.7 Attribute grammar using synthesized functions 69

5.8 Attribute grammar using coroutines 71

6.1 Propagation of mutual exclusion 79

6.2 Proof scenario 1 and 2 85

6.3 Parsing program 88

6.4 Part of the rule/goal graph for the parsing program 89

6.5 Execution trace for the parsing program 90

Analysis and Transformation of Logic Programs

Kjell Erik Post

ABSTRACT

Logic programming is based on the idea that inference can be viewed as a computation. The fact that both programs and their specifications can be expressed in the same language makes logic programming very useful for program development.

But it is also known that clarity and efficiency are two rather incompatible demands to put on a programming language. It is thus important to develop techniques for systematically transforming clear but inefficient programs into efficient (although probably rather opaque) final programs. This thesis contains some contributions that hopefully bring us closer to this goal.

Before transformation can take place the input program must be parsed and analyzed to extract properties that are not explicit in the program itself. Parsing is an area of computer science well understood, but with the advent of modern languages, like Prolog and ML, the programmer was able to introduce and change operator symbols. On one hand, operator symbols made the code more readable but it also complicated the parsing job; standard parsing techniques cannot accommodate dynamic grammars. In the first part of this thesis we present an LR parsing methodology, called “deferred decision parsing”, that handles dynamic operator declarations, that is, operators that are declared at run time. It uses a parser generator much like Yacc. Shift/reduce conflicts that involve dynamic operators are resolved at parse time rather than at parser construction time.

As an example of our parser generator we present a grammar for Prolog, a language that has been in use for almost twenty years but still lacks a precise formal syntactic definition.

The parser generator can also serve as a replacement implementation for Definite Clause Grammars, a novel parsing feature of Prolog. However, an LR parser does not normally support inherited attributes. In the next part of the thesis we present two transformation methods for (strong) non-circular attribute grammars that allows them to be evaluated within the environment of an LR parser. Our methods represent a compromise in that attribute evaluation is normally performed on the fly except when, in some evaluation rule, the referenced attributes are unavailable, and the execution of the rule has to be postponed. Suspension and resumption points for these evaluation rules can either be determined statically (method 1) or dynamically (method 2). For both methods we guarantee that resumption takes place as soon as possible.

In the final part of the thesis we present a technique to detect that pairs of rules in a logic program are "mutually exclusive". In contrast to previous work our algorithm derives mutual exclusion by looking not only at built-in, but also user-defined predicates. This technique has applications to optimization of the execution of programs containing these rules. Additionally, the programmer is less dependent on non-logical language features, such as Prolog's "cut", thus creating more opportunities for parallel execution strategies.

Keywords: Logic Programming, Program Transformation, Dataflow Analysis, Parser Generators, Attribute Grammars, Dataflow Analysis, Prolog.

Till min far, Tore Post,
och farfar, Ernst Post.

Acknowledgments

I would first like to express my gratitude towards my advisor Allen Van Gelder, who has supervised and supported this research through all these years. Allen's work is characterized by a great deal of integrity and an unusual strike of balance between theory and practice.

Allen, Charlie McDowell, and Wayne Dai read drafts and gave comments on my thesis. Richard O'Keefe and Roger Scowen has given valuable input on the syntax of Prolog. Manuel Bermudez provided a very nice exposition on LALR(1) parser generation. Lawrence Byrd suggested that LR parsing methodology could be applied to Definite Clause Grammars. Michel Mauny and Pierre Weiss of INRIA supplied production rules for CAML Light. The chapter on mutual exclusion has benefited from discussion with Saumya Debray, Peter Van Roy, and Ola Petersson. I am also indebted to the reviewers on the ILPS'93 and 94 committees for their input on the accepted papers.

I wish to thank the staff, faculty, and grad students at UCSC that I've had the pleasure to meet and work with during these years. In particular I would like to mention Tom Affinito, Søren Søe and his wife Jeanette, Richard Snyder and his wife Kathleen, and our administrative assistant Lynne Sheehan.

The list of my other friends outside UCSC is too long to include here, but Hans Nilsson, Caj Svensson, Torbjörn Näslund, Öjvind Bernander, and Sven Moen have turned out to be very dependable.

Finally a couple of people who have made life-lasting impressions on me: Are Waerland, Joel Robbins, Julius Creel, and Alan Goldhamer. These people have taught me that health can only result from healthful living.

This research was supported in part by NSF grants CCR-8958590, IRI-8902287, and IRI-9102513, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

Santa Cruz, 1994

Kjell Post

1. Introduction

In the last twenty years, we have witnessed a growing popularity in the use of the *logic programming languages*. Originating from such fields as *Artificial Intelligence*, *Automated Theorem-Proving*, and *Formal Language Theory*, logic programming initially received limited attention outside Europe but later gained considerable momentum when the Japanese in 1982 announced that they had chosen logic programming as their vehicle for their Fifth Generation Computer Systems Project *FGCS* [MO81], also known as the "computer science Pearl Harbor". Today, logic programming is part of college curriculums around the world and thousands of "real" applications have been written in Prolog. Although Prolog is the predominant logic programming language, a number of different dialects have evolved for such areas as parallel programming [Tic91], constraint solving [BC93], and process control [AVW93].

As opposed to imperative programming languages such as C, Pascal, and Ada — where the programmer explicitly specifies the flow of the computation — a logic programmer first describes the logical structure of the problem, by declaring facts and rules about objects and their relationships, and then obtains the answers by posing questions to an inference machine. This is generally believed to simplify the programming task since the details of how the answers should be computed are left to the system.

What are some of the other benefits of this "declarative style"? First of all, as the name almost implies, declarative programs can be written more concisely, in a notation closer to mathematics or formal logic. A rule in a logic program is something that can be understood by itself — there are no scope rules and side effects to consider. In addition, programming is usually done on a higher level as these languages typically have facilities such as backtracking, more advanced data structures, automatic memory management, allow the creation and passing of procedures, etc, in effect leading to both shorter programs and an increase in productivity. Additionally, the logical properties of the language makes it easier to reason formally about programs and correctly implement various analysis and transformation tools, such as termination detectors, partial evaluators, etc. Finally, the separation of control from programs and the absence of destructive assignment statements is helpful in the detection of implicit parallelism, thus perhaps making it possible to harness the power of tomorrow's multi-processor machines.

However, on today's single-processor machines, imperative languages are still considered more efficient. This is a consequence of the close correspondence between the underlying hardware, its instruction set, and the statements found in imperative languages: variables are merely abstractions of memory cells, assignment copies data between them, and the control constructs have a natural translation to the test-and-jump instructions.

Logic programming systems, on the other hand, are theorem provers and programs are written in some subset of first order logic. Therefore, the "semantic gap" between a logic programming language and the hardware is much wider than for an imperative language. While waiting for a more suitable architecture, improved execution efficiency for logic programs has been accomplished partly by relying on the programmer for supplying control information, but also increasingly by using optimizing compilers. Today, the majority of implementations for logic programming are for the language Prolog, invented in 1972 by Kowalski and Colmerauer [Kow74, Kow79]. The success of Prolog as a useful programming language is mostly due to D. H. D. Warren and his work on the WAM [War77, War83], an abstract machine for Prolog execution with a relatively easy translation to today's hardware.

Since then, compilers have improved in many ways: they produce better code, generate better diagnostics, and partly relieve the programmer from supplying extra-logical information. Still, the goal of logic programming, namely that the control component of the execution should be under the sole responsibility of the system, is still far away. This thesis makes some contributions towards this goal. In the remainder of this chapter we will examine the translation process of the compiler in more detail, give background on prior work in the various stages, and then summarize the work presented in this thesis.

1.1 The Translation Process

A convenient way to extend, improve, or implement a language is to follow the organization in fig. 1.1. Here we implement a logic programming language by using an existing and efficient Prolog implementation as our target machine. The benefits from this approach should be obvious:

- We avoid the messy details of the underlying machine.
- There is no need to understand or modify the existing compiler.
- All the collected knowledge that has been put into the compiler can be used with impunity.

In fig. 1.1 we mention some typical applications that fit into this framework. As these methods become more understood and widespread they are likely to find their way into commercial implementations. For instance, early Prolog compilers were quite naive in their execution of tail-recursive procedures and depended on the programmer to insert control information, so called “cut” symbols, into the code to prevent the stack from exploding. Today, tail recursion optimization [War86] is part of every serious Prolog system.

In the next section we give an overview of the different stages in the translation process and summarize relevant work in those areas.

1.2 Background and Prior Work

The previous section presented a framework for the translation process which we now examine in more detail by summarizing some important research results. In order to keep the presentation available to a wider audience the discussion in the following two sections will be kept on a fairly high level, saving the technical details for the remaining chapters.

1.2.1 Parsing

The problem of *parsing* the input, that is, recognizing and structuring the input, is a necessary task for any translator. Fortunately, this area is well understood and numerous so

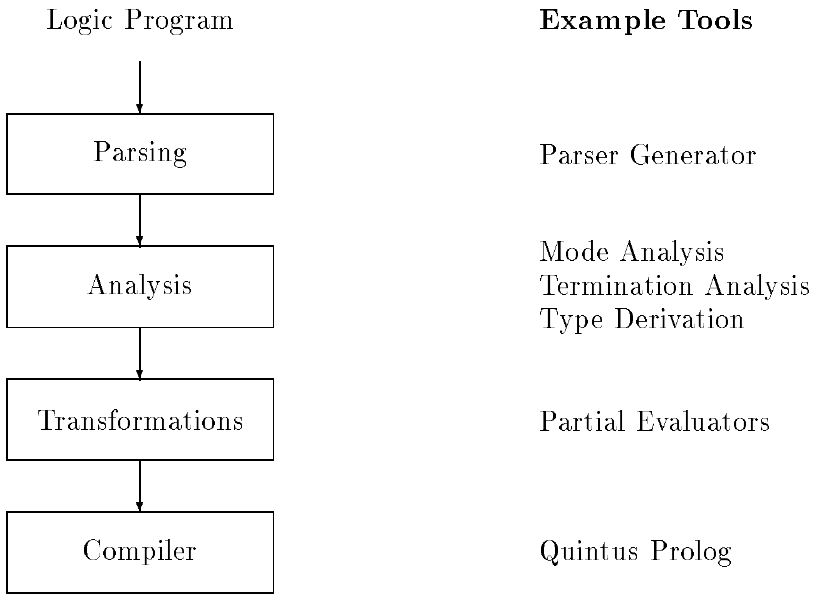


Figure 1.1: Overview of the translation process.

called *parser generators* [Joh75, ASU85, AJ74, FJ88, Udd88, HKR90, Hor90] are available to the language designer, who rarely needs to write a parser by hand. Instead, specifications in the form of production rules, with embedded code or "action" routines, are fed into a parser generator which then generates the parser. The capabilities of parser generators are limited to LR(1) languages, which is usually considered sufficiently large, and a static input grammar, possibly augmented with associativity and precedence declarations for operators such as "+" and "∗".

With the advent of programming languages like Prolog [SS86, CM81], and newer languages such as ML [MTH90] and Haskell [HW90], programmers were allowed to define operators at run time. Although operator expressions are only "syntactic sugar", that is, there is always an equivalent prefix form, Prolog's syntax is generally perceived as being easier to read than LISP's prefix notation. However, user-defined operators seriously complicate the parsing job and to solve these problems, numerous *ad hoc* parsers have been developed. In chapter 3 we examine the Deferred Decision Parser Generator, designed specifically to handle languages with so called dynamic operators while retaining all the benefits of standard parser generators.

Our parser generator also serves as an efficient implementation for *Definite Clause Grammars*, a novel parsing technique in the logic programming community.

1.2.2 Analysis

After the program has been parsed, the translator may enter an analysis stage to extract useful information which may used to optimize the program, provide some form of diagnosis, or compensate for some deficiency in the target language. Because nearly all interesting program properties are undecidable, analyzers either answer “yes, the property holds”, or “the property might hold, but I can not tell”. The following (incomplete) list mentions some analysis methods for logic programming that have been developed throughout the years, primarily for the language Prolog:

Mode analysis Probably the most well-known form of analysis is *Mode analysis* [Deb89, DW88, Mel81]. A *mode* for an argument of a procedure is an adornment, akin to a type declaration, describing how the argument will be bound at the point of invocation. Since procedures compute relations, there is in general no notion of input and output in logic programming. In reality however, the programmer often has an intended direction in mind, something which the compiler can capitalize on when translating the procedure to native code. For instance, if a unification goal $X = Y$ is called with X being a variable, the call to the expensive unification routine can be replaced by a simple assignment; similarly, if both X and Y are *ground* (that is, contain no variables), the unification can be replaced by a test for equality. Mode analysis is often assumed for other analysis tools, in particular the mutual exclusion test described in chapter 6. Modes are also described in chapter 2.

Type derivation Experience with languages such as ML have lead people to incorporate type derivation and type checking into logic programming as well [Pfe92]. As opposed to type systems in older languages, such as Pascal, the types of arguments are *derived* automatically, with minimal assistance from the programmer. Type errors, appearing as inconsistencies in the derivation, can help the programmer find subtle bugs at compile-time, rather than at run-time (by chance). Type information can also guide the compiler in replacing costly operations with cheaper ones, as well as assist in other forms of transformations.

Occur check The so called *occur check* [Pla84, Bee88] in unification, that is, verifying that the substitution that makes two terms equal does not contain a circular binding, is usually omitted in Prolog systems for efficiency reasons. This may lead to infinite loops or even incorrect answers. Therefore, a static analyzer might be used to verify, at compile-time, that circular bindings cannot be created.

Termination Standard Prolog systems employ a depth-first search rule which is efficient but not complete — the system may follow a part of the search tree that is infinitely long, "disappear" in an equally long loop, and fail to deliver some remaining answers. To deal with this "flaw", researchers have designed criteria for which termination is guaranteed [Soh93, Plü90, UVG88, BS89b, SVG91, Ped91]. Again, this illustrates how a deficiency in the language, introduced for efficiency reasons, can be compensated by a static analysis check. The price that is paid, however, is that the analyzer may be overly pessimistic and "reject" a program that does not loop.

Functional computations The concept of "backtracking" is a very powerful concept in logic programming and may be used as a "generate-and-test" construct, to guess a solution to a problem, test it, and, if the test fails, guess again, until no more guesses can be made. In this manner, *all* solutions to the problem can be generated. In many situations the first guess is also the *only* guess and the programmer does not want the system to create the state information for finding the next solution. Traditionally, this is done with the use of a "cut" symbol which cancels certain backtracking activities that would occur in the future. Static analysis can be used to infer these *functional computations* and insert cut symbols automatically [DW89]. The conservative nature of the analyzer guarantees that no solutions are left out because of an inadvertent cut.

Mutual exclusion This form of analysis also deals with restricting backtracking activities, but in a different way: procedures in logic programs, defined by several clauses, sometimes are *mutually exclusive* to each other, meaning that the success of a clause precludes the success of another, thereby making it possible to discard the backtracking information intended for the other clause. Existing algorithms [Mel85, DW89, VR90, ST85] detect mutual exclusion by looking only at built-in predicates, such as "$<$" and "$\geq$". In contrast, the algorithm presented in chapter 6 [Pos94] also examines user-defined predicates.

Parallelizers On multi-processor machines it is of course desirable to keep all processors occupied. Logic programs typically have fewer side-effects than their imperative counterparts, and may therefore have more opportunities for parallelization [Tic91, PN91, CWY91, HB88].

Abstract Interpretation Many analysis methods can be captured in the framework of *abstract interpretation* [CC92, CC77, Jan90, BJCD87, Bru91, Mel87, MS88]. In essence, a property of a program can sometimes be deduced by running an "abstract" form of the program, where operations and data values in the original, "concrete", program have been replaced by corresponding abstract values. Abstract values are chosen so that execution is guaranteed to terminate and so that the abstract value represents some useful information. For instance, integers may be captured by the abstract values $\{neg, 0, pos\}$. In the program, we subsequently replace for instance the concrete multiplication "$*$" with the abstract multiplication "$\hat{*}$" which operates on abstract values, e.g. "$neg \mathbin{\hat{*}} pos = neg$." With this technique it might be possible to detect whether a variable contains a negative value at a certain program point, although, in practically all situations a "don't know"-element has to be part of the abstract set of values, something which the reader may verify by trying to define the abstract addition operation.

1.2.3 Transformations

A translator typically performs two different kinds of transformations. Initially, if the source language represents a superset of the target language, the translator has to "shoehorn" its input by replacing features of the input language by equivalent code in the target language. For imperative languages, this technique has been used in the implementation of for instance RATFOR and C++. In the field of logic programming, the promising new language Goedel is implemented on top of Prolog, and the so called "magic set" transformation [Ram91, BR91, MSU86] has been used to implement a bottom-up search using the top-down search method of Prolog.

A second reason for transformation is optimization, replacing parts of the program by semantically equivalent code that is more efficient, in some respect, usually time or space. Optimization methods in logic programming are normally targeted at the two "expensive" features of the language, unification and backtracking, as can be witnessed by the analysis methods listed in the previous section.

A well-known and very general optimization technique is *partial evaluation* [LS91] whereby a program is in some sense "specialized" with respect to its input. In logic programming, partial evaluation has a particularly easy formulation. For instance, take the procedure *sort*, defined as follows

$$sort(L_1, L_2) \leftarrow permute(L_1, L_2), ordered(L_2).$$

This says that L_2 is a sorted version of L_1 *if* L_2 is a permutation of L_1, *and* L_2 is ordered. Now let's assume that *sort* appears in the body of another procedure that finds the smallest element X in a list L[1]

$$smallest(L, X) \leftarrow sort(L, [X|T]).$$

Then we may replace the "call" to *sort* with its body

$$smallest(L, X) \leftarrow permute(L, [X|T]), ordered([X|T]).$$

[1] Here we use the Prolog notation $[X|T]$ which represents a list whose first element is X and whose tail is T.

Partial evaluation for Prolog is in general more difficult due to the presence of extra-logical features and side effects, although successful partial evaluators have been built, for instance the MIXTUS system [Sah93].

Some other transformation methods in logic programming are: various means of implementing negation [Kun87], the replacement of lists with so called "difference lists" [SS86] (which can be concatenated in constant time), and the automatic insertion of control directives which can be done for instance when information on functional computations and mutual exclusion is available.

1.3 Summary of Contributions

The results presented in this thesis fit directly into the various stages for the framework that we have just presented.

In chapter 3 we present a parsing technique called *deferred decision parsing*, which was developed to solve the problem of parsing languages with dynamic operators, that is, languages where the programmer can change the properties of operator symbols as the program is being parsed. This technique has been built into a parser generator called DDGEN, which generates deferred decision parsers in a manner similar to Yacc.

As a realistic example we examine in chapter 4 the syntax of Prolog, a language riddled by numerous syntactical complications and ambiguities. We suggest reasonable restrictions to make the language deterministic and give a concise and readable grammar to be used with the parser generator.

The parser generator also serves another important purpose, namely as an efficient implementation for Definite Clause Grammars. Conventional implementations based on backtracking parsers can require exponential time. In contrast, our implementation has the advantage that the token stream need not be completely acquired beforehand, and the parsing, being deterministic, is linear time.

On the other hand, syntax definitions usually have attributes and evaluation rules associated with them to convey context-sensitive information. Conventional implementations parse top-down and are thus able to handle some *inherited attributes*, representing “input” to the production that is currently being recognized, something which a bottom-up parser does not normally support. To rectify this situation, we have developed two transformation techniques, presented in chapter 5, that allow a bottom-up parser to emulate the evaluation of arbitrary (non-circular) attribute definitions.

Finally in chapter 6 we present an analysis method for logic programs to find mutually exclusive rules. A very common situation in the definition of a procedure is that its defining rules are mutually exclusive to each other; this may happen for instance when one rule is defined for $X = 0$ and another for $X > 0$. This presents an optimization opportunity: if the system succeeds in proving the goal that excludes the other rule, it does not have to remember to come back to the second clause if the first one fails. This type of analysis has been conducted before but always restricted to looking only at *primitive* test goals, such as arithmetic relational operators. Our method generalizes previous work by propagating information in the call graph and is thus able to derive mutual exclusion between *user-defined* procedures.

2. Preliminaries

In order to set the stage for the following chapters, we first review some notation and concepts on parsing and logic programming. For a more extensive treatment we refer the reader to [ASU85, AJ74, FJ88] for context-free grammars and parsing, and [Llo87, Apt90, CM81, SS86] for logic programming. A unifying presentation of these two fields can also be found in [DM93].

2.1 Context Free Grammars

A *context-free grammar* is a four-tuple $G = \langle V_N, V_T, S, P\rangle$. The finite disjoint sets of *nonterminals* V_N and *terminals* V_T form the *vocabulary* $V = V_N \cup V_T$.

The set $P \subseteq V_N \times V^*$ consists of m *productions* where the p-th production is

$$X_{p0} \rightarrow X_{p1}X_{p2}\ldots X_{pn_p}$$

where $n_p > 0, X_{p0} \in V_N, X_{pj} \in V$ for $1 \leq j \leq n_p$.

$S \in V_N$ is the *start symbol*, which does not appear on the right side of any production. It is normally the left side of the first production.

The word *token* is used synonymously with terminal symbol. As a notational convention, terminal symbols appear in typewriter style, like `id`. Although there is no typographical convention for nonterminal symbols, we will often use upper-case letters such as A, B, C, N, and S, or lower-case italic names, such as *expr*. Either way, nonterminals can always be distinguished because of their appearance in some production's left-hand side. An arbitrary grammar symbol (terminal or nonterminal) is represented by the letter X, lower-case greek letters α, β, γ represent *strings* of grammar symbols, whereas a string of only terminal symbols is denoted w. The notation $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ is shorthand for the productions $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_n$.

The *length* of a string α is written $|\alpha|$ and is simply the number of grammar symbols in the string. There is one special symbol, namely the *empty string* ε, that has zero length.

2.1.1 Derivations and Parse Trees

Given a string of grammar symbols, a production can be seen as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production. A rewriting step can be written abstractly as $\alpha A \beta \Rightarrow \alpha\gamma\beta$, if $A \rightarrow \gamma$ is a production. The transitive closure of this relation is written with the symbol $\overset{*}{\Rightarrow}$. If $\alpha \overset{*}{\Rightarrow} \beta$, we say α *derives* β.

If $S \overset{*}{\Rightarrow} \alpha$, then α is a *sentential form*. A *sentence* is a sentential form without nonterminal symbols. The language *generated* by a grammar G can now be described as the set of all sentences derived from S:

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$$

(where w is a string of terminal symbols).

We assume that all grammars contains no useless productions and that every nonterminal symbol is accessible from the start symbol and can generate a string without nonterminal symbols.

Example 2.1.1: The context-free grammar $\langle\{Z, N, B\}, \{., \mathtt{0}, \mathtt{1}\}, S, P\rangle$, where P is given by the productions below, describes binary numbers and generates sentences such as "`10.1`" ▮

$$\begin{array}{llll}
1. & S & \rightarrow & N\ .\ N \\
2. & N & \rightarrow & N\ B \\
3. & N & \rightarrow & \varepsilon \\
4. & B & \rightarrow & \mathtt{0} \\
5. & B & \rightarrow & \mathtt{1}
\end{array}$$

Example 2.1.2: (*continued*) A derivation from the grammar in example 2.1.1 is

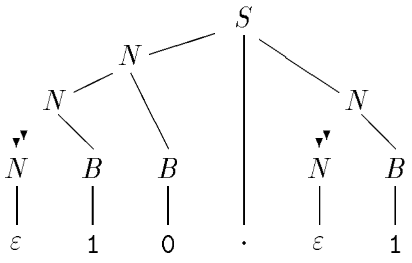
$$S \overset{1}{\Rightarrow} N.N \overset{2}{\Rightarrow} NB.N \overset{2}{\Rightarrow} NBB.N \overset{3}{\Rightarrow} BB.N \overset{5}{\Rightarrow} 1B.N \overset{4}{\Rightarrow} 10.N \overset{2}{\Rightarrow} 10.NB \overset{3}{\Rightarrow} 10.B \overset{5}{\Rightarrow} 10.1$$

The numbers above the arrows indicate the production used in the derivation ▮

If in each derivation step the leftmost (rightmost) nonterminal is rewritten, the derivation is called *leftmost* (*rightmost*). The derivation in example 2.1.2 is a leftmost derivation.

A *parse tree* is a graphical representation of a derivation where the root is labeled by the start symbol and the fringe of the tree corresponds to a sentential form. A grammar that produces two or more parse trees for a sentence is called *ambiguous*.

Example 2.1.3: (*continued*) The (only) parse tree for the sentence "10.1" is shown below ▮



2.1.2 LR Parsing and Parser Generation

A *parser* is a recognizer for a given context-free grammar G. It accepts as its input a string of terminal symbols w and verifies whether $w \in L(G)$ or not. The output can be a *parse tree*, showing the productions that were used in the process of verifying the input string.

The type of parsers described in this thesis are *LR parsers*, also called *shift-reduce* or *bottom-up parsers*, because they recognize the parse tree bottom-up by *reading* (shifting) terminals and — when the complete right-hand side of a production is available — *reducing* the right-hand side to its left-hand side. If the parser is successful in reducing its input to the start symbol, the input is syntactically correct. This process is called a *reverse rightmost derivation* because it traces out a rightmost derivation in reverse.

Example 2.1.4: *(continued)* The *read* and *reduce* steps taken when recognizing "10.1" are as follows: *reduce* $N \rightarrow \varepsilon$; *read* 1; *reduce* $B \rightarrow$ 1; *reduce* $N \rightarrow N\ B$; *read* 0; *reduce* $B \rightarrow$ 0; *reduce* $N \rightarrow N\ B$; *read* .; *reduce* $N \rightarrow \varepsilon$; *read* 1; *reduce* $B \rightarrow$ 1; *reduce* $N \rightarrow N\ B$; *reduce* $S \rightarrow N$. N ▮

In a general setting, knowing when to *read* or *reduce* is not always easy. In addition, the LR parser must also reject erroneous input, and know when to *accept* (announce that the input

```
$push(Stack, S_0)$      (initial state)
$read(X)$
repeat
    $S := top(Stack)$
    case $parse\_table(S, X)$ of
        $shift\ S'$:
            $push(Stack, X)$
            $push(Stack, S')$
            $read(X)$
        $reduce\ A \to \gamma$:
            $pop\ |\gamma|$ state/symbol pairs from $Stack$
            $parse\_table(top(Stack), A)$ now contains $shift\ S'$
            $push(Stack, A)$
            $push(Stack, S')$
        $error$:
            abort
until $parse\_table(S, X) = accept$
```

Figure 2.1: The LR parser.

was correct). Such knowledge is stored in a *parse table* whose construction requires a rather complicated analysis of the production rules (not described here, but see [BL89, AJ74]). The parse table encodes a state machine where a state S may have zero or more outgoing arcs, each labeled by some distinct grammar symbol X. Hence, the parse table can be implemented with an array $parse\_table(S, X)$. Each entry contains one of the four actions:

shift – to a new state and read a new terminal; *reduce* – by a certain production; *accept*, or *error*. With the help of an auxiliary stack of previous states, the parser has enough information to parse the input string without actually storing the symbols in the "next" right-hand side, although for clarity we will put them on the stack as well when we show the steps taken by the parser. The algorithm for the LR parser is shown in fig. 2.1.

An LR *parser generator* is a program that generates the parse table and the parser (which is always the same) for a given grammar. The construction of the parse table imposes certain restrictions on the grammar. For example, there exist grammars for which an input string has more than one parse tree, that is, the string can be parsed in more than one way. Such grammars are ambiguous and give rise to multiple entries, called

Parse Table							
	`0`	`1`	`.`	`eof`	S	N	B
0	r/3	r/3	r/3	r/3	s/1	s/2	
1				acc			
2	s/4	s/5	s/3				s/6
3	r/3	r/3	r/3	r/3		s/7	
4	r/4	r/4	r/4	r/4			
5	r/5	r/5	r/5	r/5			
6	r/2	r/2	r/2	r/2			
7	s/4	s/5		r/1			s/6

Execution Trace		
Stack	*Input*	*Action*
0	`1 0 . 1 eof`	*reduce* $N \rightarrow \varepsilon$
0 N 2	`1 0 . 1 eof`	*shift* 5
0 N 2 `1` 5	`0 . 1 eof`	*reduce* $B \rightarrow$ `1`
0 N 2 B 6	`0 . 1 eof`	*reduce* $N \rightarrow N\ B$
0 N 2	`0 . 1 eof`	*shift* 4
0 N 2 `0` 4	`. 1 eof`	*reduce* $B \rightarrow$ `0`
0 N 2 B 6	`. 1 eof`	*reduce* $N \rightarrow N\ B$
0 N 2	`. 1 eof`	*shift* 3
0 N 2 `.` 3	`1 eof`	*reduce* $N \rightarrow \varepsilon$
0 N 2 `.` 3 N 7	`1 eof`	*shift* 5
0 N 2 `.` 3 N 7 `1` 5	`eof`	*reduce* $B \rightarrow$ `1`
0 N 2 `.` 3 N 7 B 6	`eof`	*reduce* $N \rightarrow N\ B$
0 N 2 `.` 3 N 7	`eof`	*reduce* $S \rightarrow N$ `.` N
0 S 1	`eof`	*accept*

Figure 2.2: Parse table and execution trace for an LR parser. In the parse table, state numbers appear in the left column and grammar symbols along the top row. The symbol `eof` symbolizes the end-of-file marker which terminates the input. The abbreviation "r/n" means *reduce* by production n (cf. example 2.1.1), "s/m" denotes *shift* to state m, and "acc" stands for *accept*; all empty entries are *error* entries.

---

shift/reduce, or *reduce/reduce* conflicts, in the parse table. A parse table with conflicts must be rejected by the parser generator, unless the conflicts can be resolved by some other policy (cf. chapter 3).

Example 2.1.5: (*continued*) A parse table for the binary number grammar is shown in fig. 2.2 along with an execution trace of the LR parser for the input string "`10.1`" ▮

A *semantic action* is simply a piece of code attached to a production. Semantic actions are executed when the corresponding production has been recognized. In an LR parser, this can be arranged simply by calling a routine *action*(n) in the *reduce* case in fig. 2.1, where n is the recognized production.

2.2 Logic Programming

This section provides a short description of the syntax and semantics of logic programs. The syntactical aspects of Prolog are presented in more detail in chapter 3, so at this point we confine ourselves to an informal overview. We also omit the so called *declarative semantics* of logic programs, and instead describe the *procedural semantics* which are needed to understand the material on mutual exclusion in chapter 6.

2.2.1 Syntax

A logic program is a collection of *Horn clauses* (see fig. 2.3). A clause is a sentence of the form

$$A_0 \leftarrow A_1, A_2, \ldots A_n. \qquad (n \geq 0)$$

which is to be understood as the statement

"for all $X_1, \ldots, X_k$, A_0 if A_1 and A_2 and ... and A_n"

where $X_1, \ldots, X_k$ are the variables occurring in the clause. If $n > 0$ we refer to the clause as a *rule*, otherwise, if $n = 0$, we call it a *fact* and write it as

$$A_0.$$

without the implication sign. When actual Prolog code is shown, clauses are usually set in typewriter style, and the implication sign is written "`:-`".

A set of clauses with the same predicate name p and arity n collectively defines the *procedure* p/n .

The *head* A_0 and the *subgoals* $A_1, \ldots, A_n$ are *literals* of the form $p(t_1, \ldots, t_n)$, where p is a *predicate symbol* and the t_i are terms, consisting of either constants, variables or compound terms.

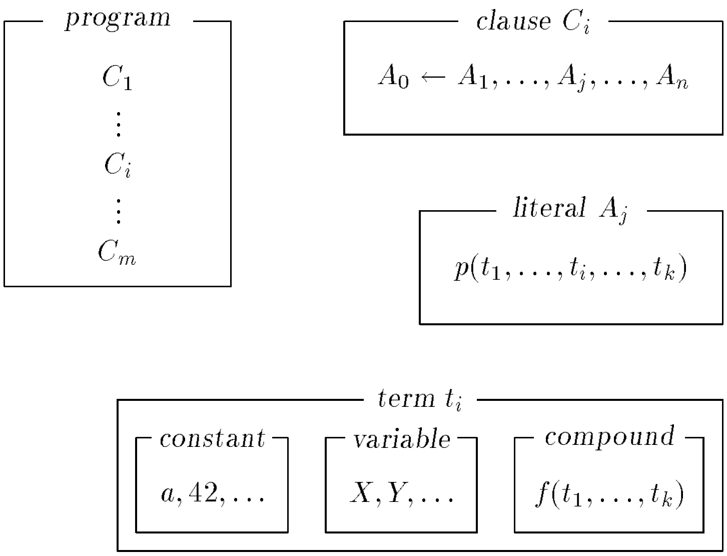


Figure 2.3: Structure of logic programs.

A constant is either a number or an atom. An *atom* is uniquely identified by its name, which is a sequence of characters, either alphanumeric starting with a lower case letter, or some special symbol like "+", "−", "=", "!", "[]", etc, or any sequence of characters delimited by single quotes.

A variable is any sequence of alphanumeric characters (including underscore), starting with either a capital letter, or an underscore.

A compound term has the form $f(t_1, \ldots, t_k)$ and represents a *function symbol* (*functor*) applied to its arguments $t_1, \ldots, t_k$. A functor f of arity k is often written f/k to make its arity apparent.

A common data structure in logic programming is the *list* which is either empty, represented by the constant [], or a compound term with the functor "." ("dot") and two arguments representing the head and the tail of the list. Thus $.(a, .(b, .(c, [])))$ is a list with three elements a, b, and c. The recursive structure of lists can be hidden by using special syntax, writing them more conveniently as $[a, b, c]$. Also, the special notation $[X|Y]$, equivalent to $.(X, Y)$, is useful when the tail of the list is a variable.

Clauses, literals, and terms are called, collectively, *expressions*.

Example 2.2.1: The following clauses collectively define the relation $merge(Xs, Ys, Zs)$ which is true when Zs is an ordered merge of the elements in Xs and Ys.

$merge([], Ys, Ys).$

$merge(Xs, [], Xs).$

$merge([X|Xs], [Y|Ys], [X|Zs]) \leftarrow X < Y, merge(Xs, [Y|Ys], Zs).$

$merge([X|Xs], [Y|Ys], [Y|Zs]) \leftarrow X \geq Y, merge([X|Xs], Ys, Zs).$ ▮

Queries (or *goals*) are given in the form of a conjunction of conditions

$$\leftarrow B_1, B_2, \ldots, B_n. \qquad (n > 0)$$

which is to be understood as the question

"does there there exist $X_1, \ldots, X_k$ such that B_1 and B_2 ...and B_n?"

where $X_1, \ldots, X_k$ are the variables in $B_1, \ldots, B_n$. The resolution engine, presented in section 2.2.3, either constructs a *substitution* $X_1/t_1, \ldots, X_k/t_k$ (see section 2.2.2) or *fails*, depending on whether the query is a theorem that follows from the program or not.

Example 2.2.2: (*continued*) The query

$$\leftarrow merge([2,5,7],[1,3,9], Zs)$$

results in the substitution $Zs/[1,2,3,5,7,9]$ ▮

2.2.2 Substitutions and Unification

Formally, a *substitution* is a finite mapping from variables to terms, and is written as

$$\theta = \{X_1/t_1, \ldots, X_k/t_k\}.$$

Each pair X_i/t_i is called a *binding*. We assume that all variables X_i are distinct and that $X_i \neq t_i, i = 1 \ldots k$. The substitution given by the empty set is called the *identity substitution* and is denoted ϵ[1].

[1] Although this symbol is unfortunate in that it can be confused with the empty string ε, I have followed standard terminology as best as I could, but made them look slightly different. The risk of confusion is minimal however, since they never appear together in the material to come.

A substitution θ can be applied to an expression E. The result, $E\theta$, is called an *instance* of E, and is obtained by simultaneously replacing each variable X_i in E by the term t_i. An instance is *ground* if it contains no variables.

Substitutions can also be *composed*. If

$$\theta = \{X_1/t_1, \ldots, X_k/t_k\}$$

and

$$\sigma = \{Y_1/u_1, \ldots, Y_m/u_m\}$$

the composition $\theta\sigma$ is obtained from the set

$$\{X_1/t_1\sigma, \ldots, X_k/t_k\sigma, Y_1/u_1, \ldots, Y_m/u_m\}$$

by removing all bindings $X_i/t_i\sigma$ for which $X_i = t_i\sigma$ and also all bindings Y_j/u_j for which $Y_j \in \{X_1, \ldots, X_k\}$.

We say that a substitution θ is *more general* than a substitution σ if for some substitution η we have $\sigma = \theta\eta$.

Two expressions E_1 and E_2 are said to be *unifiable* if there exists a substitution θ such that $E_1\theta = E_2\theta$. If so, θ is called a *unifier*. There exists a unification algorithm [Rob65] that for any two expressions produces their most general unifier (*mgu*) if they are unifiable and otherwise reports that the two expressions are not unifiable.

Example 2.2.3: Consider expressions $E_1 = p(X, X), E_2 = p(Y, f(Y))$, and $E_3 = p(a, a)$. Since Y appears in $f(Y)$, E_2 does not unify with E_1; neither does E_2 unify with E_3. However, E_1 and E_3 have a (most general) unifier $\{X/a\}$ ▮

2.2.3 Procedural Semantics

In order to compute an answer substitution for a goal $G : \leftarrow B_1, \ldots, B_n$ the logic programming system tries to prove the negation of G by a process called *SLD-derivation*[2].

[2] Since G itself stands for $\forall X(\neg B_1 \vee \ldots \vee \neg B_n)$, proving the negation amounts to showing $\exists X(B_1 \wedge \ldots \wedge B_n)$.

```
INPUT: A goal $G : \leftarrow B_1, \ldots, B_i, \ldots, B_n$ and a set of program clauses $C_1, \ldots, C_m$.
OUTPUT: An answer substitution $\theta$ with bindings for the variables in $G$.
METHOD:
    $j := 0$
    $G_0 := G$
    $\theta := \epsilon$
    repeat
        select a literal $B_i$ from $G_j$
        $j := j + 1$
        if there is a clause $C :\ B \leftarrow A_1, \ldots, A_k$ such that $B_i$ and $B$ unify then
            rename all variables in $C$ to avoid name conflicts
            $\theta_j := mgu(B_i, B)$
            $G_j := \leftarrow (B_1, \ldots, B_{i-1}, A_1, \ldots, A_k, B_{i+1}, \ldots, B_n)\theta_j$
            $\theta := \theta\theta_j$
        else
            $G_j := fail$
    until $(G_j = \Box \vee G_j = fail)$
```

Figure 2.4: Algorithm for computing an answer substitution.

An SLD-derivation is a sequence of transformation steps where, informally, in each step a literal in the current goal is replaced by the body of a clause whose head unifies with the selected literal. If the derivation produces the *empty goal* $\Box$ in a finite number of steps the derivation is called a *refutation* and the composition of all the unifiers gives the answer substitution. The derivation is said to *fail* if, along the derivation, the selected literal has no matching clause.

The complete algorithm is given in fig. 2.4. From the description it is clear that there is room for some nondeterminism at each derivation step. In Prolog, the selected literal is always the leftmost literal in the goal and clauses with matching heads are tried in textual order. The possibility of several matching heads gives rise to an *SLD-tree* — see fig. 2.6 for an example. In Prolog, the tree is explored depth-first; when a fail-node is discovered, *backtracking* takes place whereby the system restores its state to the closest previous branching point where a new choice for a matching clause can be made.

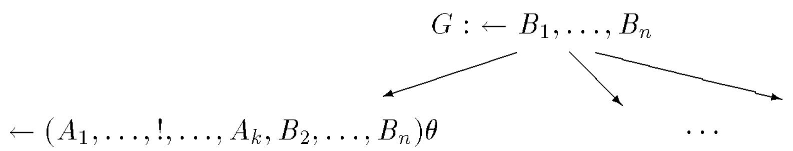

$G : \leftarrow B_1, \ldots, B_n$
$\leftarrow (A_1, \ldots, !, \ldots, A_k, B_2, \ldots, B_n)\theta$
$\cdots$

Figure 2.5: The effect of a cut: the remaining branches are pruned.

2.2.4 Cuts

The *cut* is a control facility found in Prolog. It is written as "!" and may be used as a literal in the body of a clause. The cut always succeeds when selected, but as a side effect it will cancel certain backtracking activities.

In fig. 2.5 we illustrate the effects of a cut. Here we assume that the leftmost literal in the current goal is selected, as in Prolog. The invoked clause contains a cut symbol. If the cut is later selected, all the remaining branches to the right will be "cut off", that is, if the system backtracks to the cut, it will skip all remaining subtrees of G and instead resume the search above G.

A cut that prunes away success nodes from the SLD-tree is said to be *red*. A red cut is harmful because it prevents correct answer substitutions from being derived. If there is no answer in the pruned part, the cut is *green*. Green cuts improve the efficiency of the execution and can be used to sidestep an infinitely large subtree. However, it is not always easy to tell whether a cut is red or green; a small modification to a clause can suddenly make a green cut turn red, change the meaning of the program, and confuse the programmer.

2.2.5 Modes

In a logic program there is no obvious flow of computation — a procedure can for instance be invoked with ground arguments to confirm a solution, or the arguments can be partially bound, so that the procedure will attempt to generate a solution.

It is often the case, however, that the programmer designs a procedure to be invoked in a particular way. He may then opt to give a *mode declaration* for the procedure, specifying how each argument will be bound when the procedure is called. For our purposes we represent mode information for a procedure p/n with an n-tuple over the elements $\{\mathbf{c}, \mathbf{d}\}$, where **c** ("constant") represent all ground terms, and **d** ("don't know") stands for all terms. The mode for a procedure is sometimes given as a superscript, as in $append^{\mathbf{c},\mathbf{c},\mathbf{d}}$. Some researchers use the notation $\{\mathbf{b}, \mathbf{f}\}$ instead of $\{\mathbf{c}, \mathbf{d}\}$ and speak of mode information as "adornments". Mode information may also be inferred statically from a global analysis of the program, but, as with all interesting program properties, a mode inference program may not always give exact information.

2.2.6 Definite Clause Grammars

A *Definite Clause Grammar* (DCG) is a language specification, similar to a context free grammar, that is automatically translated into clauses in a logic programming language. In principle, each production rule is compiled into a clause and the parser inherits the deficiencies of the search mechanism used in the SLD-derivation. In the case of Prolog, this translation method gives rise to a top-down, backtracking parser.

Example 2.2.4: Consider again the productions in example 2.1.1. A simple translation to clauses follows below.

$s(X_0, X_3) \leftarrow n(X_0, X_1), X_1 = ['.'|X_2], n(X_2, X_3).$

$n(X_0, X_0).$

$n(X_0, X_2) \leftarrow n(X_0, X_1), b(X_1, X_2).$

$b(X_0, X_1) \leftarrow X_0 = [0|X_1].$

$b(X_0, X_1) \leftarrow X_0 = [1|X_1].$

Here, the variables X_i hold the input so that the first argument of a clause represents the part of the input that is left to parse upon entering the clause, and the second argument is what is left when leaving the clause.

As a Prolog program, these procedures can be used to refute goals like $\leftarrow s([1,0,'.',1],[])$. It should be noted however, that if the two clauses for n are interchanged, the Prolog system will enter an infinite loop ∎

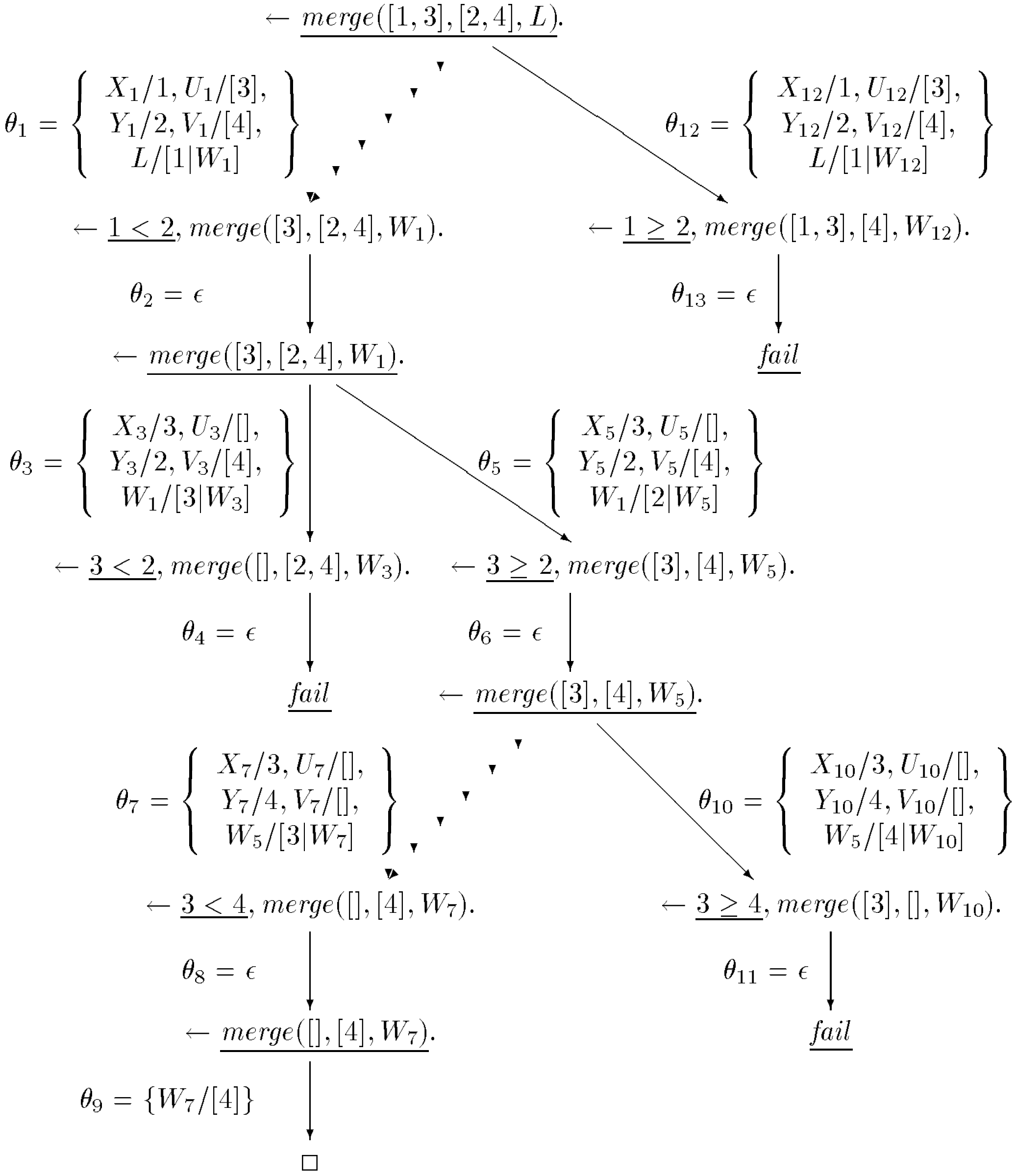

$\leftarrow \underline{merge([1,3],[2,4],L)}.$
$\theta_1 = \left\{ \begin{array}{c} X_1/1, U_1/[3], \\ Y_1/2, V_1/[4], \\ L/[1|W_1] \end{array} \right\}$
$\theta_{12} = \left\{ \begin{array}{c} X_{12}/1, U_{12}/[3], \\ Y_{12}/2, V_{12}/[4], \\ L/[1|W_{12}] \end{array} \right\}$
$\leftarrow \underline{1 < 2}, merge([3],[2,4],W_1).$
$\leftarrow \underline{1 \geq 2}, merge([1,3],[4],W_{12}).$
$\theta_2 = \epsilon$
$\theta_{13} = \epsilon$
$\leftarrow \underline{merge([3],[2,4],W_1)}.$
$\underline{fail}$
$\theta_3 = \left\{ \begin{array}{c} X_3/3, U_3/[], \\ Y_3/2, V_3/[4], \\ W_1/[3|W_3] \end{array} \right\}$
$\theta_5 = \left\{ \begin{array}{c} X_5/3, U_5/[], \\ Y_5/2, V_5/[4], \\ W_1/[2|W_5] \end{array} \right\}$
$\leftarrow \underline{3 < 2}, merge([],[2,4],W_3).$
$\leftarrow \underline{3 \geq 2}, merge([3],[4],W_5).$
$\theta_4 = \epsilon$
$\theta_6 = \epsilon$
$\underline{fail}$
$\leftarrow \underline{merge([3],[4],W_5)}.$
$\theta_7 = \left\{ \begin{array}{c} X_7/3, U_7/[], \\ Y_7/4, V_7/[], \\ W_5/[3|W_7] \end{array} \right\}$
$\theta_{10} = \left\{ \begin{array}{c} X_{10}/3, U_{10}/[], \\ Y_{10}/4, V_{10}/[], \\ W_5/[4|W_{10}] \end{array} \right\}$
$\leftarrow \underline{3 < 4}, merge([],[4],W_7).$
$\leftarrow \underline{3 \geq 4}, merge([3],[],W_{10}).$
$\theta_8 = \epsilon$
$\theta_{11} = \epsilon$
$\leftarrow \underline{merge([],[4],W_7)}.$
$\underline{fail}$
$\theta_9 = \{W_7/[4]\}$
□

Figure 2.6: SLD-tree for the query $\leftarrow merge([1,3],[2,4],L)$. The underlined part in each node represents the selected literal. The answer substitution for a refutation is given by the composition of unifiers along the success branches, in this case $\theta_1\theta_2\theta_5\theta_6\theta_7\theta_8\theta_9 = \{\ldots, L/[1|[2|[3|[4]]]], \ldots\} = \{\ldots, L/[1,2,3,4], \ldots\}$.

3. The Parser Generator DDGEN

Allowing the programmer to define operators in a language makes for more readable code but also complicates the job of parsing; standard parsing techniques cannot accommodate dynamic grammars. In this chapter we present an LR parsing methodology, called *deferred decision parsing*, that handles dynamic operator declarations, that is, operators that are declared at run time, are applicable only within a program or context, and are not in the underlying language or grammar. It uses a parser generator that takes production rules as input, and generates a table-driven LR parser, much like Yacc. *Shift/reduce* conflicts that involve dynamic operators are resolved at parse time rather than at table construction time.

For an operator-rich language, this technique reduces the size of the grammar needed and parse table produced. The added cost to the parser is minimal. Ambiguous operator constructs can either be detected by the parser as input is being read or avoided altogether by enforcing reasonable restrictions on operator declarations. We have been able to describe the syntax of Prolog, a language known for its liberal use of operators, and Standard ML, which supports local declarations of operators.

Definite Clause Grammars (DCGs), a novel parsing feature of Prolog, can be translated into efficient code by our parser generator. The implementation has the advantage that the token stream need not be completely acquired beforehand, and the parsing, being deterministic, is linear time. Conventional implementations based on backtracking parsers can require exponential time.

3.1 Introduction and Background

Syntax often has a profound effect on the usability of a language. Although both Prolog and LISP are conceptually rooted in recursive functions, Prolog's operator-based syntax is generally perceived as being easier to read than LISP's prefix notation. Prolog introduced two innovations in the use of operators:

1. Users could define new operators at run time with great flexibility[1].
2. Expressions using operators were semantically equivalent to prefix form expressions; they were a convenience rather than a necessity.

Notably, new functional programming languages, such as ML [MTH90] and Haskell [HW90], are following Prolog's lead in the use of operators. Proper use of operators can make a program easier to read but also complicates the job of parsing the language.

Example 3.1.1: The following Prolog rules make extensive use of operators to improve readability.

```
X requires Y :- X calls Y.
X requires Y :- X calls Z, Z requires Y.
```

Here, "`:-`" and "`,`" are supplied as standard operators, while "`requires`" and "`calls`" have programmer definitions (not shown) ▮

Languages that support *dynamic operators* have some or all of the following syntactic features:

1. The name of an operator can be any legal identifier or symbol. Perhaps, even "built-in" operators like "$+$", "$-$", and "$*$" can be redefined.
2. The syntactic properties of an operator can be changed by the user during parsing of input.
3. Operators can act as arguments to other operators. For example, if "$\vee$", "$\wedge$", and "$\times$" are infix operators, the sentence "$\vee \times \wedge$" may be proper.
4. Operators can be *overloaded*, in that a single identifier can be used as the name of two or more syntactically different operators. A familiar example is "$-$", which can be prefix or infix.

This chapter addresses problems in parsing such languages, presents a solution based on a generalization of LR parsing and describes a parser generator for this family of languages. After reviewing some definitions, we give some background on the problem, and then summarize the contributions. Later sections discuss several of the issues in detail.

[1] Some earlier languages permitted very limited definitions of new operators; see section 3.1.2.

3.1.1 Definitions

We briefly review some standard definitions concerning operators, and specify particular terminology used in this chapter. An *operator* is normally a unary or binary function whose identifier can appear before, after, or between its arguments, depending on whether its *fixity* is *prefix*, *postfix*, or *infix*. An identifier that has been declared to be operators of different fixities is said to be an *overloaded operator*. We shall also have occasion to consider *nullary* operators, which take no arguments. More general operator notations, some of which take more than two arguments, are not considered here. Besides fixity, operators have two other properties, *precedence* and *associativity*, which govern their use in the language.

An operator's *precedence*, or *scope*, is represented by a positive integer. Here we use the Prolog convention, which is that the larger precedence number means the wider "scope" and the weaker binding strength. This is the reverse of many languages, hence the synonym *scope* serves as a reminder. Thus "+" normally has a larger precedence number than "*" by this convention.

An operator's *associativity* is one of *left*, *right*, or *non*. For our purposes, an *expression* is a term whose principal function is an operator. The precedence of an expression is that of its principal operator. A left (right) associative operator is permitted to have an expression of equal precedence as its left (right) argument. Otherwise, arguments of operators must have lower precedence (remembering Prolog's order). Non-expression terms have precedence 0; this includes parenthesized expressions, if they are defined in the grammar.

3.1.2 Background and Prior Work

Most parsing techniques assume a static grammar and fixed operator priorities. Excellent methods have been developed for generating efficient LL parsers and LR parsers from specifications in the form of production rules, sometimes augmented with associativity and precedence declarations for infix operators [AJ74, ASU85, BL89, FJ88, Joh75]. Parser generation methods enjoy several significant advantages over "hand coding":

1. The language syntax can be presented in a readable, nonprocedural form, similar to production rules.
2. Embedded semantic actions can be triggered by parsing situations.
3. For most programming languages, the tokenizer may be separated from the grammar, both in code and in specification.
4. Parsing runs in linear time and normally uses sublinear stack space.

The price that is paid is that only the class of LR(1) languages can be treated, but this is normally a sufficiently large class in practice.

Earley's algorithm [Ear70] is a general context-free parser. It can handle any grammar but is more expensive than the LR-style techniques because the configuration states of the LR(0) automaton are computed and manipulated as the parse proceeds. Parsing an input string of length n may require $O(n^3)$ time (although LR(k) grammars only take linear time), $O(n^2)$ space, and an input-buffer of size n. Tomita's algorithm [Tom86] improves on Earley's algorithm by precompiling the grammar into a parse table, possibly with multiple entries. Still, the language is fixed during the parse and it would not be possible to introduce or change properties of operators on the fly.

Incremental parser generators [HKR90, Hor90] can be viewed as an application of Tomita's parsing method. They can handle modifications to the input grammar at the expense of recomputing parts of the parse table and having the LR(0) automaton available at run time. Garbage collection also becomes an issue.

To our knowledge these methods have never been applied to parse languages with dynamically declared operators.

Operator precedence parsing is another method with applications to dynamic operators [LdR81] but it can not handle overloaded operators.

Permitting user-defined operators was part of the design of several early procedural languages, such as Algol-68 [vW76] and EL1 [HTSW74], but these designs avoided most of the technical difficulties by placing severe restrictions on the definable operators. First, infix operators were limited to 7 to 10 precedence levels. By comparison, C has 15 built-in precedence levels, and Prolog permits 1200. More significantly, prefix operators always took precedence over infix, preventing certain combinations from being interpreted naturally. (C defines some infix to take precedence over some prefix, and Prolog permits this in user definitions.) For example, no declarations in the EL1 or Algol-68 framework permit `(not X=Y)` to be parsed as `(not (X=Y))`. Scant details of the parsing methods can be found in the literature, but it appears that one implementation of EL1 used LR parsing with a mechanism for changing the token of an identifier that had been dynamically declared as an operator "based on local context" before the token reached the parser [HTSW74]. Our approach generalizes this technique, postponing the decision until after the parser has seen the token, and even until after additional tokens have been read.

Languages have been designed to allow other forms of user-defined syntax besides unary and binary operators. Among them, EL1 included "bracket-fix operators" and other dynamic syntax; in some cases a new parser would be generated [HTSW74]. More recently, Peyton Jones [Jon86] describes a technique for parsing programs that involves user-defined *distfix* operators, for instance `if-then-else-fi`, but without support for precedence and associativity declarations.

With the advent of languages that permit dynamic operators, numerous *ad hoc* parsers have been developed. The tokenizer is often embedded in these parsers with attendant complications. It is frequently difficult to tell precisely what language they parse; the parser code literally defines the language.

Example 3.1.2: Using standard operator precedences, prefix "−" binds less tightly than infix "∗" in popular versions of Prolog. However, `(- 3 * 4)` was found to parse as `((-3) * 4)` whereas `(- X * 4)` was found to parse as `(- (X * 4))`. Numerous other anomalies can be demonstrated ▮

Indeed, written descriptions of the "Edinburgh syntax" for Prolog are acknowledged to be approximations, and the "ultimate definition" seems to be the public domain parser `read.pl`.

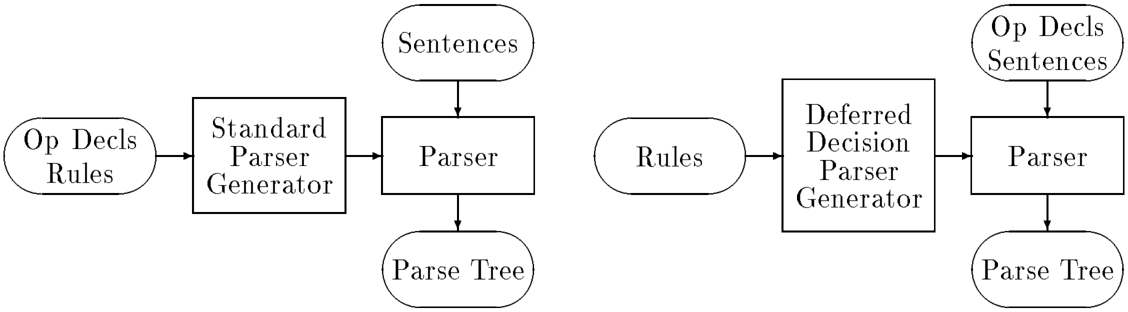


Figure 3.1: Standard Parser Generator and Deferred Decision Parser Generator.

3.1.3 Summary of Contributions

A method called *deferred decision parsing* has been developed with the objective of building upon the techniques of LR parsing and parser generation, and enjoying their advantages mentioned above, while extending the methodology to incorporate dynamic operators. The method is an extension of earlier work by Kerr [Ker89]. It supports all four features that were listed above as being needed by languages with dynamic operators. The resulting parsers are deterministic, and suffer only a small time penalty when compared to LR parsing without dynamic operators. They use substantially less time and space than Earley's algorithm.

In "standard" LR parser generation, as done by Yacc and similar tools, *shift/reduce* conflicts, as evidenced by multiple entries in the initial parse table, are resolved by consulting declarations concerning (static) operator precedence and associativity [Joh75]. If the process is successful, the final parse table becomes deterministic.

The main idea of deferred decision parsing is that *shift/reduce* conflicts involving *dynamic* operators are left unresolved when the parser is generated. At run time the current declarations concerning operator precedence and associativity are consulted to resolve an ambiguity when it actually arises (fig. 3.1). Another important extension is the ability to handle a wider range of fixities, as well as overloading, compared to standard LR parser generators. These extensions are needed to parse several newer languages.

The parser generator, implemented in a standard dialect of Prolog, processes DCG-style input consisting of production rules, normally with embedded semantic actions. The output is Prolog source code for the parser, with appropriate calls to a scanner and an operator module.

The operator module provides a standard interface to the dynamic operator table used by the parser. Thus, the language designer decides what language constructs denote operator declarations; when such constructs are recognized during parsing, the associated semantic actions may interact with the operator module. Procedures are provided to query the state of the operator table and to update it. Interaction with the operator module is shown in the example in fig. 3.7.

We assume that the grammar is designed in such a way that semantic actions that update dynamic operators cannot occur while a dynamic operator is in the parser stack; otherwise a grammatical monstrosity might be created. In other words, it should be impossible for a dynamic operator to appear in a parse tree as an ancestor of a symbol whose semantic action can change the operator table. Such designs are straightforward and natural, so we see no need for a special mechanism to enforce this constraint. For example, if dynamic operator properties can be changed upon reducing a "sentence", then the grammar should not permit dynamic operators between "sentences", for that would place a dynamic operator above "sentence" in some parse trees.

Standard operator-conflict situations are easily handled by a run-time resolution module. However, Prolog offers very general operator-declaration capabilities, which in theory can introduce significant difficulties with overloaded operators. (Actually, these complicated situations rarely arise in practice, as users refrain from defining an operator structure that they cannot understand themselves.) In Edinburgh Prolog [BBP$^+$82] for instance, the user can define a symbol as an operator of all three fixities and then use the symbol as an operator, or as an operand (nullary operator), or as the functor of a compound term. As the Prolog standardization committee recognized in a 1990 report [Sco90],

> "These possibilities provide multiple opportunities for ambiguity and the draft standard has therefore defined restrictions on the syntax so that a) ex-

> pressions can still be parsed from left to right without needing significant backtracking or look-ahead..."

A preliminary report on this work showed that many of the proposed restrictions were unnecessary. The ISO committee has subsequently relaxed most of the restrictions, but at the expense of more complex rules for terms [Sco92].

The modular design of our system permits different conflict resolution strategies and operator-restriction policies to be "plugged in", and thus may serve as a valuable tool for investigating languages that are well-defined, yet have very flexible operator capabilities.

So-called *Definite Clause Grammars* (DCGs) are a syntactic variant of Prolog, in which statements are written in a production-rule style, with embedded semantic actions [BBP$^+$82]. This style permits input-driven programs to be developed quickly and concisely. Our parser generator provides a foundation for DCG compilation that overcomes some of the deficiencies of existing implementations. These deficiencies include the need to have acquired the entire input string before parsing begins, and the fact that backtracking occurs, even in deterministic grammars.

Our point of view is to regard the DCG as a translation scheme in which the arguments of predicates appearing as nonterminals in the DCG are attributes; semantic actions may manipulate these attributes. Essentially, a parser is a DCG in which all attributes are synthesized, and each nonterminal has a parse tree as its only attribute. Synthesis of attributes is accomplished naturally in LR parsing, as semantic actions are executed after the associated production has been reduced. Another research direction has been to correctly handle inherited attributes. This work is discussed in chapter 5.

3.2 Deferred Decision Parsing

The parser generator Yacc disambiguates conflicts in grammars by consulting programmer-supplied information about precedence and associativity of certain tokens, which normally function as infix operators. Deferred decision parsing postpones the resolution of conflicts involving *dynamic operators* until run time.

	Prefix		*Infix*		*Postfix*	
Name	*Prec*	*Assoc*	*Prec*	*Assoc*	*Prec*	*Assoc*
+	300	right	500	left		
−	300	right	500	left		
∗			400	left		
/			400	left		
!					300	left

Figure 3.2: An example run-time operator table.

$$\begin{array}{lcll}
term(T) & \rightarrow & \mathtt{atom}(T) & \\
term(T) & \rightarrow & \mathtt{var}(T) & \\
term(T) & \rightarrow & '(' \; term(T) \; ')' & \\
term(T) & \rightarrow & \mathtt{op}(Name) \; term(T_1) & \{ \; T = .. \; [Name, T_1] \; \} \\
term(T) & \rightarrow & term(T_1) \; \mathtt{op}(Name) \; term(T_2) & \{ \; T = .. \; [Name, T_1, T_2] \; \} \\
term(T) & \rightarrow & term(T_1) \; \mathtt{op}(Name) & \{ \; T = .. \; [Name, T_1] \; \} \\
term(T) & \rightarrow & \mathtt{op}(Name) &
\end{array}$$

Figure 3.3: Subset grammar for Prolog terms.

As a running example we will use a grammar for a subset of Prolog terms with operators of all three fixities. At run time the name of each operator, together with its precedence, fixity, and associativity, is stored in the current operator table (see fig. 3.2 for an example). The parser has access to the current operator table and is responsible for converting tokens from the scanner so that an identifier with an entry in the current operator table is translated to the appropriate dynamic operator token.

The token names for dynamic operators, which should not be confused with the operator names, are declared to the parser generator (cf. example in section 3.3), and appear in the production rules of the grammar. When a production rule contains a dynamic operator token there can be at most one grammar symbol on either side of the dynamic operator, and its position determines the intended fixity. Apart from this, there are no other restrictions on the productions. Normally a single token is sufficient for all dynamic operators. This example uses the single token `op` for prefix, infix, and postfix operators.

Figure 3.3 shows the subset grammar for Prolog terms. The LR(0) collection for this grammar has 11 states of which 4 have *shift/reduce* conflicts.

Rather than trying to resolve each *shift/reduce* conflict at table construction time we will delay the decisions and turn them into a new kind of action, called *resolve*, which takes two arguments: (1) the state to enter if the conflict is resolved in favor of a *shift*, and (2) the rule by which to *reduce* if a reduction is selected. Recall that the user (language designer) declares what tokens constitute dynamic operators (op in this example). Only the conflicts involving two dynamic operators are expected to be resolved at run time. All other conflicts are reported as usual. Conflicts between an operator and a non-operator symbol can always be resolved at table construction time due to the requirement that operators have positive precedence and non-expression terms have precedence 0.

The parser driver for deferred decision parsing is similar to a standard LR parser, except for one difference: instead of directly accessing entries in the parse table, references to parse table entries are mediated through a procedure $\mathit{parse\_action}(S, X)$, where S is the current state of the parser and X is the look-ahead token. It returns an action which may be one of *shift*, *reduce*, *accept*, or *error*. The rule for *parse_action* is

```
If $\mathit{parse\_table}(S, X) = \mathit{resolve}(S', A \rightarrow \alpha\ \mathsf{op}_A\ \beta)$
    then return $\mathit{do\_resolve}(A \rightarrow \alpha\ \mathsf{op}_A\ \beta, X)$
    else return $\mathit{parse\_table}(S, X)$
```

The procedure *do_resolve*, which is called to resolve the *shift/reduce* conflict, has access to the rule that is candidate for reduction, and the look-ahead token. The resolution is done by the policy that is used at table-construction time by Yacc [AJU75, ASU85], with extensions to cover cases that cannot be declared in Yacc (cf. section 3.4). The details, for those conversant with the operation of the LR parser, are as follows. The *shift/reduce* conflict corresponding to the *resolve* action requires a decision when $\alpha\ \mathsf{op}_A\ \beta$ is on top of the stack (top rightmost), and the look-ahead token is op_B, where α and β each consist of zero or one grammar symbols. (Recall that one of the requirements for turning the conflict into a *resolve* action was that op_A and op_B had to be declared as dynamic operators.) The choices are to *reduce*, using production $A \rightarrow \alpha\ \mathsf{op}_A\ \beta$, or *shift* the look-ahead token op_B.

When operators are overloaded, there may be several choices to consider. Even if operators are not overloaded by run-time declarations, there may be the implicit overloading of the declared operator and the nullary operator. The ambiguities that arise from overloading pose serious difficulties in parser design. Our parser treats the nullary operator as having scope equal to the maximum of its declared scopes plus a small “delta”. This treatment guarantees a deterministic grammar if there is no declared overloading (cf. section 3.4) and retains the flexibility of allowing operators to appear as terms.

The overloading of the operators $\mathtt{op}_A$ and $\mathtt{op}_B$ could lead to multiple interpretations of the input string as there are several declarations to consider. In practice one doesn't have to consider all combinations of the declarations; a Prolog grammar, for instance, does not generate sentential forms with two adjacent expressions, so there are only certain fixity combinations worth considering:

Form of $\alpha\ \mathtt{op}_A\ \beta$	*Possible fixity combinations* $(\mathtt{op}_A, \mathtt{op}_B)$
$\alpha \neq \varepsilon, \beta \neq \varepsilon$	(infix, infix), (infix, postfix)
$\alpha \neq \varepsilon, \beta = \varepsilon$	(infix, prefix), (infix, null), (postfix, infix), (postfix, postfix)
$\alpha = \varepsilon, \beta \neq \varepsilon$	(prefix, infix), (prefix, postfix)
$\alpha = \varepsilon, \beta = \varepsilon$	(prefix, prefix), (prefix, null), (null, infix), (null, postfix)

Hence, for a Prolog grammar there are at most four overloading combinations. If the grammar allows adjacent expressions there are at most 16 combinations to consider; this happens when both operators have declarations for all fixities.

The rules below are evaluated for each fixity combination; the resulting actions are collected into a set. The parser will enter an error state if the set of possible actions is either empty — signifying a precedence error — or contains both *shift* and *reduce* actions — indicating ambiguous input, in which case the two possible interpretations of the input string are reported to the user. If there is a unique action, we have successfully resolved the conflict.

1. If $\texttt{op}_A$ and $\texttt{op}_B$ have equal scope, then
    - (a) If $\texttt{op}_A$ is right-associative, *shift*[2].
    - (b) If $\texttt{op}_B$ is left-associative, *reduce*.
2. If $\texttt{op}_A$ is either infix or prefix with wider scope than $\texttt{op}_B$, *shift*.
3. If $\texttt{op}_B$ is either infix or postfix with wider scope than $\texttt{op}_A$, *reduce*.

Example 3.2.1: We examine the deferred decision parser as it reads `-X+Y*Z!`, using the operators in fig. 3.2. Figure 3.4 shows the steps involved. At step 4, state 5 contains the *shift/reduce* conflict $\{term \rightarrow term \bullet \texttt{op(+)}\ term, term \rightarrow \texttt{op(-)}\ term \bullet\}$ for terminal `op`. The operator in the redex, "`-`", and the look-ahead operator "`+`" are overloaded as level 300 prefix right-associative, level 500 infix left-associative and, implicitly, as level $500 + \delta$ nullary. Due to the form of the redex, only the prefix form of "`-`" and infix form of "`+`" are considered. Since "`-`" has narrower scope than "`+`" and $\beta = term \neq \varepsilon$ we satisfy the requirements for the third rule above and *reduce*.

At step 8, state 7 contains the *shift/reduce* conflict $\{term \rightarrow term \bullet \texttt{op(*)}\ term, term \rightarrow term\ \texttt{op(+)}\ term \bullet\}$. The operator in the redex, "`+`", is overloaded as before while the look-ahead operator "`*`" is level 400 infix left-associative and level $400 + \delta$ nullary. This time, the form of the redex tells us to consider the infix versions of both operators, and since infix "`+`" has wider scope than infix "`*`" we have a match for the second rule and *shift*.

At step 11, state 7 contains the *shift/reduce* conflict $\{term \rightarrow term \bullet \texttt{op(!)}, term \rightarrow term\ \texttt{op(+)}\ term \bullet\}$. The operator in the redex, "`*`", is still level 400 infix left-associative and level $400+\delta$ nullary while the look-ahead operator "`!`" is level 300 postfix left-associative and level $300 + \delta$ nullary. The nullary versions are not considered due to the form of the redex. Again the operator in the redex is infix with wider scope than the look-ahead operator, matching the third rule, so we *shift* again ▮

[2] *Shift* is chosen over *reduce* in situations where the input is `1 R 2 L 3`. (`R` and `L` have the same precedence, but are declared right-associative and left-associative, respectively.) This is parsed as `1 R (2 L 3)`, which conform with all Prolog systems that we are aware of, as well as the most recent ISO draft[Sco92].

	Stack	*Input*	*Action*
1	0	op(-) var(X) op(+) var(Y) op(*) var(Z) op(!) eof	*shift* 3
2	0 op(-) 3	var(X) op(+) var(Y) op(*) var(Z) op(!) eof	*shift* 4
3	0 op(-) 3 var(X) 4	op(+) var(Y) op(*) var(Z) op(!) eof	*reduce term* → var(X)
4	0 op(-) 3 *term* 5	op(+) var(Y) op(*) var(Z) op(!) eof	*resolve*(6,*term* → op(-) *term*)
5	0 *term* 10	op(+) var(Y) op(*) var(Z) op(!) eof	*shift* 6
6	0 *term* 10 op(+) 6	var(Y) op(*) var(Z) op(!) eof	*shift* 4
7	0 *term* 10 op(+) 6 var(Y) 4	op(*) var(Z) op(!) eof	*reduce term* → var(Y)
8	0 *term* 10 op(+) 6 *term* 7	op(*) var(Z) op(!) eof	*resolve*(6,*term* → *term* op(+) *term*)
9	0 *term* 10 op(+) 6 *term* 7 op(*) 6	var(Z) op(!) eof	*shift* 4
10	0 *term* 10 op(+) 6 *term* 7 op(*) 6 var(Z) 4	op(!) eof	*reduce term* → var(Z)
11	0 *term* 10 op(+) 6 *term* 7 op(*) 6 *term* 7	op(!) eof	*resolve*(6,*term* → *term* op(*) *term*)
12	0 *term* 10 op(+) 6 *term* 7 op(*) 6 *term* 7 op(!) 6	eof	*reduce term* → *term* op(!)
13	0 *term* 10 op(+) 6 *term* 7 op(*) 6 *term* 7	eof	*reduce term* → *term* op(*) *term*
14	0 *term* 10 op(+) 6 *term* 7	eof	*reduce term* → *term* op(+) *term*
15	0 *term* 10	eof	*accept*

Figure 3.4: Deferred decision parsing example.

3.3 Local Operator Declarations

The advantage of resolving operator conflicts at run time is that the programmer can change the syntactic properties of operators on the fly. When parsing a language with dynamic operators it is the responsibility of the language designer to initialize the operator table with the predefined operators in the language, before parsing commences, and to provide semantic actions to update the table as operator declarations are recognized.

ML is a language with infix operators only, but these can be declared locally in blocks, with accompanying scope rules [MTH90]. Thus in the following example

```
let infix 5 * ; infix 4 + in
    1+2*3  +  let infix 3 * in 1+2*3 end  +  1+2*3
end
```

only the middle use of * would have an unusually low precedence, yielding the answer 23.

To understand how the operator scope rules of ML can be implemented, we study fig. 3.7

which shows an input specification for DDGEN. Here, "::=" is used instead of "→" in the grammar rules.

The line dynop_token(atom(Name), op(Name)) declares op as a dynamic operator; if the scanner returns atom(Name) and Name is present in the current operator table, the token is converted to op(Name).

When the parser encounters a let-expression, the operators whose names are shadowed by local declarations are saved in OldList. Each operator declaration returns information about the operator it shadows, or null if there was no declaration with the same name in an outer scope. The old operators are re-instantiated when we reach the end of the let-expressions.

Calls to the operator table module can be seen in the Prolog rules at the bottom: the first two rules, ml_dcl_op and ml_rem_op, declares and removes an operator in the current operator table, respectively, and returns the old properties. The last rule, ml_rest_op is called to re-instantiate old operators.

3.4 Ambiguities at Run Time and Induced Grammars

Because the language changes dynamically as new operators are being declared, it is important to understand exactly what language is being parsed. The algorithm below constructs the induced grammar, given the input grammar and an operator table. As mentioned earlier, we assume that operator declarations remain constant while a construct involving dynamic operators is being reduced. Thus it makes sense to talk about the induced grammar with which that construct is parsed, even though a later part of the input may be parsed by a different induced grammar.

The induced grammar is obtained by replacing productions containing dynamic operators with a set of productions constituting a precedence grammar. It should be pointed out that the induced grammar is not actually constructed by the parser. The goal of the deferred decision parser is to recognize exactly the strings in the language generated by the induced grammar.

Algorithm 3.4.1: Induced grammar construction.

INPUT: A dynamic operator grammar and the current operator table.

OUTPUT: The induced grammar, defined below.

METHOD: Sort the operator table by precedence so it can be divided into k slots, where all operators in slot $i \in 1 \ldots k$ have the same precedence p_i, and so that slot k holds the operators of widest scope. For simplicity, assume that there is only one dynamic operator, `op`. Now apply the following steps:

1. For each nonterminal A in the dynamic operator grammar:

   (a) Partition the productions defining A into five sets corresponding to the use of an operator in the right hand side (prefix, infix, postfix, operand (nullary), or no operator at all):

   $$\Pi_{pre} = \{A \rightarrow \texttt{op}\ C\} \qquad \Pi_{in} = \{A \rightarrow B\ \texttt{op}\ C\}$$
   $$\Pi_{post} = \{A \rightarrow B\ \texttt{op}\} \qquad \Pi_{null} = \{A \rightarrow \texttt{op}\}$$
   $$\Pi_{rem} = \{A \rightarrow \gamma \ \mid\ \texttt{op} \notin \gamma\}$$

   (b) Build a precedence grammar consisting of $k+1$ layers (fig. 3.5). This will serve as a skeleton for the induced grammar. The i-th layer $(0 < i \leq k)$ holds productions for nonterminal A_i which corresponds to operators with precedence p_i. The 0-th layer defines the sentential forms with 0 precedence, provided that there are any in the dynamic operator grammar.

   (c) For $i \in 1 \ldots k$:

   Take each operator into account by adding productions to either A_i^R, A_i^L or A_i^N, depending on the associativity of the operator being right, left, or none.

      i. First the prefix operators: If $\Pi_{pre} \neq \emptyset$, add productions $A_i^N \rightarrow\ o\ A_{i-1}$, for all prefix, non-associative operators o with precedence p_i, and $A_i^R \rightarrow o\ A_i^R$, for all prefix, right-associative operators o with precedence p_i.

      ii. Next the infix operators: If $\Pi_{in} \neq \emptyset$, add productions $A_i^L \rightarrow A_i^L\ o\ A_{i-1}$, for all infix, left-associative operators o with precedence p_i, and $A_i^N \rightarrow A_{i-1}\ o\ A_{i-1}$, for all infix, non-associative operators o with precedence

	$A \rightarrow A_k$
$k:$	$A_k \rightarrow A_k^R$, $A_k^R \rightarrow A_k^L$, $A_k^L \rightarrow A_k^N$, $A_k^N \rightarrow A_{k-1}$
	$\vdots$
$i:$	$A_i \rightarrow A_i^R$, $A_i^R \rightarrow A_i^L$, $A_i^L \rightarrow A_i^N$, $A_i^N \rightarrow A_{i-1}$
	$\vdots$
$1:$	$A_1 \rightarrow A_1^R$, $A_1^R \rightarrow A_1^L$, $A_1^L \rightarrow A_1^N$, $A_1^N \rightarrow A_0$
$0:$	$A_0 \rightarrow \gamma$, for all $A \rightarrow \gamma$ in Π_{rem}

Figure 3.5: Skeleton for induced grammar. The production $A_1^N \rightarrow A_0$ is included only if $\Pi_{rem} \neq \emptyset$.

p_i, and $A_i^R \rightarrow A_{i-1}\ o\ A_i^R$, for all infix, right-associative operators o with precedence p_i.

iii. Then the postfix: If $\Pi_{post} \neq \emptyset$, add productions $A_i^L \rightarrow A_i^L\ o$, for all postfix, left-associative operators o with precedence p_i, and $A_i^N \rightarrow A_{i-1}\ o$, for all postfix, non-associative operators o with precedence p_i.

iv. Finally, let nullary operators have the maximum of its declared precedences: If $\Pi_{null} \neq \emptyset$, then for all operators o in the operator table, add the production $A_p \rightarrow o$ where p is the highest indexed slot in the operator table containing o.

2. The nonterminals and terminals of the induced grammar are given in the standard way: a grammar symbol which appears on some left hand side is a nonterminal; all other symbols are terminals. Additionally, the dynamic operator grammar and the induced grammar share their start symbols ▮

Example 3.4.1: Consider the Prolog term grammar and the operator table in fig. 3.2. Sorting the table gives us three precedence levels (fig. 3.6). The induced grammar is shown below ($term_i^N$ omitted for brevity). The induced grammar is deterministic (as verified by Yacc).

$$
\begin{array}{lcl}
term_3 & \rightarrow & \texttt{'+'} \mid \texttt{'-'} \mid term_3^L \\
term_3^L & \rightarrow & term_3^L \; \texttt{'+'} \; term_2 \mid term_3^L \; \texttt{'-'} \; term_2 \mid term_2 \\
term_2 & \rightarrow & \texttt{'*'} \mid \texttt{'/'} \mid term_2^L \\
term_2^L & \rightarrow & term_2^L \; \texttt{'*'} \; term_1 \mid term_2^L \; \texttt{'/'} \; term_1 \mid term_1 \\
term_1 & \rightarrow & \texttt{'!'} \mid term_1^R \\
term_1^R & \rightarrow & \texttt{'+'} \; term_1^R \mid \texttt{'-'} \; term_1^R \mid term_1^L \\
term_1^L & \rightarrow & term_1^L \; \texttt{'!'} \mid term_0 \\
term_0 & \rightarrow & \texttt{atom} \mid \texttt{var} \mid \texttt{'('} \; term_3 \; \texttt{')'} \qquad \blacksquare
\end{array}
$$

Although the above example produced a deterministic grammar, unrestricted overloading can produce an ambiguous grammar. However, certain reasonable restrictions on operator declarations and overloading guarantee determinacy (with one look-ahead token) in the deferred decision parser:

1. All declarations for the same symbol have the same precedence.
2. If a symbol's infix associativity is *non*, then (if declared) its prefix and postfix associativities must be *non*.
3. If a symbol's infix associativity is *left*, then (if declared) its prefix associativity must be *non* and its postfix associativity must be *left*.
4. If a symbol's infix associativity is *right*, then (if declared) its prefix associativity must be *right* and there must be no postfix declaration for this symbol.

Theorem 3.4.1: The above restrictions guarantee determinacy.

Proof: The proof is carried out in three steps. We begin by excluding declared overloading altogether and assigning nullary operators the maximum of its declared scopes plus "delta". This leads to deterministic induced grammars, even if operators are used as operands. The reason for this, briefly, is that since overloading is absent, an operator symbol with declared precedence p_i will be confined to only one layer i in the induced grammar. Thus, any possible ambiguity must arise from the implicit overloading of the declared operator and the nullary operator, or possibly the use of a derivation $A_i \overset{*}{\Rightarrow} \alpha A \beta$. Assuming α and β are non-empty strings not containing operator symbols, we end up with a skeleton grammar for layer i as shown below. (Prolog syntax is used to describe the different combinations of fixities and associativities; cf. fig. 4.1.)

$$\begin{array}{lcl}
A_i & \rightarrow & \texttt{fx} \mid \texttt{fy} \mid \texttt{xfx} \mid \texttt{xfy} \mid \texttt{yfx} \mid \texttt{xf} \mid \texttt{yf} \;\; A_i^R \\
A_i^R & \rightarrow & \texttt{fy}\; A_i^R \mid B\; \texttt{xfy}\; A_i^R \mid A_i^L \\
A_i^L & \rightarrow & A_i^L\; \texttt{yfx B} \mid A_i^L\; \texttt{yf} \mid A_i^N \\
A_i^N & \rightarrow & \texttt{fx}\; B \mid B\; \mathrm{xfx}\; B \mid B\; \mathrm{xf} \mid B \\
B & \rightarrow & \texttt{atom} \mid \texttt{var} \mid \texttt{'('}\; A_i\; \texttt{')'}
\end{array}$$

As verified by Yacc, this grammar is deterministic. By induction on the parse tree it then follows that the induced grammar, consisting of all layers, is also deterministic.

Since the previous policy turned out to guarantee deterministic grammars, we now try to relax the requirements on overloading and instead insist on the following two conditions:

1. All declarations for the same symbol must have the same precedence.
2. No symbol must be overloaded to be both left and right associative.

Nullary operators are treated as before. The skeleton layer in the induced grammar now takes the following appearance:

$$\begin{array}{lcl}
A_i & \rightarrow & \texttt{fx} \mid \texttt{fy} \mid \texttt{non} \mid \texttt{right} \mid \texttt{left} \mid \texttt{xf} \mid \texttt{yf} \mid A_i^R \\
A_i^R & \rightarrow & \texttt{fy}\; A_i^R \mid B\; \texttt{right}\; A_i^R \mid \texttt{right}\; A_i^R \mid A_i^L \\
A_i^L & \rightarrow & A_i^L\; \texttt{yf} \mid A_i^L\; \texttt{left}\; B \mid A_i^L\; \texttt{left} \mid A_i^N \\
A_i^N & \rightarrow & \texttt{fx}\; B \mid B\; \mathrm{xf} \mid B\; \mathrm{non}\; B \mid \mathrm{left}\; B \mid B\; \mathrm{right} \mid \mathrm{non}\; B \mid B\; \mathrm{non} \mid B \\
B & \rightarrow & \texttt{atom} \mid \texttt{var} \mid \texttt{'('}\; A_i\; \texttt{')'}
\end{array}$$

In this grammar **non** signifies an overloaded operator that is actually declared to be `fx`, `xfx`, and `xf`. Likewise, `right` is overloaded as `fy`, `xfy`, and `xf`; `left` as `fx`, `yfx`, and `yf`. The induced grammar fragment is ambiguous, as witnessed by the sentential form B `right left` B:

1. $A_i^R \Rightarrow B\; \texttt{right}\; A_i^R \Rightarrow B\; \texttt{right}\; A_i^L \Rightarrow B\; \texttt{right}\; A_i^N \Rightarrow B\; \texttt{right left}\; B$
2. $A_i^R \Rightarrow A_i^L \Rightarrow A_i^L\; \texttt{left}\; B \Rightarrow A_i^N\; \texttt{left}\; B \Rightarrow B\; \texttt{right left}\; B$

Slot	*Prec*	$(o, \mathit{Fix}, \mathit{Assoc})$
1	300	(−, `pre`, `right`), (+, `pre`, `right`), (!, `post`, `left`)
2	400	(∗, `in`, `left`), (/, `in`, `left`)
3	500	(−, `in`, `left`), (+, `in`, `left`)

Figure 3.6: Sorted operator table.

Thus we need something stronger. Working backwards, we can remove the production $A_i^N \rightarrow B$ `right` to make the grammar deterministic again, which leads to the policy just presented ▮

The recent ISO draft proposes a different set of restrictions to avoid ambiguities [Sco92]. Probably neither solution is the final word. We hope our parser generator can be a useful tool to explore design alternatives.

3.5 Application to Definite Clause Grammars

The Definite Clause Grammars found in Prolog were originally designed for parsing highly ambiguous grammars with short sentences, natural languages being the primary example. Since Prolog employs a top-down backtracking execution style, the evaluation of DCGs will resemble the behavior of a top-down parser with backtracking.

In compiler theory, interest is commonly focused on deterministic languages. The benefit of Prolog as a compiler tool has been observed by Cohen and Hickey [CH87]. However, there are several reasons why a top-down backtracking parser is unsuitable for recognizing sentences such as programming language constructs.

- A left-recursive production, such as $E \rightarrow E - T$, will send the parser into an infinite loop, even though the grammar is not ambiguous. There are techniques for eliminating left recursion, but they enlarge the grammar, sometimes significantly [ASU85]. Also, there is no clearcut way to transform the semantic actions correctly.

- Unless the parser is *predictive* [ASU85] it may spend considerable time on backtracking.
- A backtracking parser requires the whole input stream of tokens to be present during parsing.
- Backtracking may be undesirable in a compiler where semantic actions are not idempotent, for example in the generation of object code.

Deterministic bottom-up parsers, on the other hand, run in linear time, require no input buffering, and handle left-recursive productions as well as right-recursive. They do not normally support ambiguous grammars. Nilsson [Nil86] has implemented a nondeterministic bottom-up evaluator for DCGs by letting the parser backtrack over conflicting entries in the parse table.

Our approach is to view Definite Clause Grammars as attribute grammars , which have been studied extensively in connection with deterministic translation schemes [ASU85]. In terms of *argument modes* of DCG goals, *synthesized* attributes correspond to "output" arguments, while *inherited* attributes correspond to "input" arguments. S-attributed definitions, that is, syntax-directed definitions with only synthesized attributes, can be evaluated by a bottom-up parser as the input is being parsed. The L-attributed definitions allow certain inherited attributes as well, thus forming a proper superset of the S-attributed definitions. Implementation techniques for L-attributed definitions based on grammar modifications or post-traversals of the parse tree are known [ASU85]. In chapter 5 we demonstrate how inherited attributes in a bottom-up parser can be handled either by associating a function from inherited to synthesized attributes with nonterminals, or by using coroutine facilities in the language.

3.6 Implementation

We have implemented the deferred decision parser generator in a standard dialect of Prolog. However, there is nothing about the method that prevents it from being incorporated into any standard LR parser generator. There are three issues that need to be considered:

1. The user has to be able to declare dynamic operators, like tokens are declared in Yacc using `%token`.
2. *shift/reduce* conflicts have to be turned into *resolve* actions. This can be done as a post-processing pass on the parse table using the item sets (`y.output` for Yacc). Recall that only conflicts involving dynamic operators are candidates — all other conflicts have to be reported as usual.
3. The parser accesses parse table entries through the procedure *parse_action*.

```
parse(File, Exp) :-
        open(File, read, Stream),
        clear_op, % clear the operator table
        init_scanner(Stream, InitScanState),
        parse(InitScanState, Exp),
        close(Stream).

dynop_token(atom(Name), op(Name)).

exp(E) ::= atexplist(E).
exp(E) ::= exp(E1), op(I), exp(E2), { E = app(I,E1,E2) }.

atexplist(E) ::= atexp(E).
atexplist(E) ::= atexplist(E1), atexp(E2), { E = app(E1,E2) }.

atexp(C) ::= number(C).
atexp(V) ::= atom(V).
atexp(E) ::= let, { begin_block }, dec, in, exp(E), end, { end_block }.

dec ::= dec, ';', dec.
dec ::= infix, number(D), id(I),
                { D9 is 9-D, declare_op(I, infix, left, D9) }.
dec ::= infixr, number(D), id(I),
                { D9 is 9-D, declare_op(I, infix, right, D9) }.
dec ::= nonfix, id(I),
                { remove_op(I) }.

id(N) ::= atom(N).
id(N) ::= op(N).

% Interface to the Operator Module: declare, remove and restore operators.
ml_dcl_op(Name, Fix, Assoc, Prec, Old) :-
        (query_op(Name, F, A, P) -> Old = old(Name, F, A, P) ; Old = null),
        declare_op(Name, Fix, Assoc, Prec).

ml_rem_op(Name, Old) :-
        (query_op(Name, F, A, P) -> Old = old(Name, F, A, P) ; Old = null),
        remove_op(Name, infix).

ml_rest_op([]).
ml_rest_op([null|T]) :- ml_rest_op(T).
ml_rest_op([old(Name,F,A,P)|T]) :- declare_op(Name,F,A,P), ml_rest_op(T).
```

Figure 3.7: Grammar subset for ML operator expressions.

4. Parsing Prolog

In this chapter we study the syntactical properties of Prolog, a programming language that has been in use for more than fifteen years but still lacks a precise and readable formal syntactic definition. The lack of a definition can be attributed to some subtle points in Prolog's syntax that are not immediately obvious to the casual user.

The use of the deferred decision parsing method, presented in chapter 3 avoids the problems of earlier work and the restrictions presented in the more recent ISO proposal. We present a context-free grammar for a variant of Prolog consisting of only 19 productions. Besides being completely deterministic, it has the advantage of being presented in a readable, non-procedural form. The parser itself enjoys all the benefits of a standard LR parser, such as linear running time and complete separation from the tokenizer.

4.1 Introduction

At first, Prolog looks deceptively easy to parse. But study it more closely and you will soon discover small problems, like the significance of whitespace, or more challenging problems, like operator expressions requiring arbitrarily long look-ahead, and eventually some really difficult problems, namely ambiguous operator expressions.

Back in 1984, when the first efforts on a standardization of Prolog began, it was said that syntactical differences between Prolog versions could be trivially overcome, due to the availability of the DEC-10 Prolog parser `read.pl`[O'K84]. Almost ten years later we find that two of the most widely used Prolog systems are based on `read.pl`, but because of changes and bug-fixes disagree on some inputs. Because of the opacity of the parser, a 350-line program with backtracking and its share of cuts, it's very hard for a user to know exactly what the recognized language is.

Reference manuals typically offer a very ambiguous description of the language, acknowledged to be merely an approximation. The truth of the matter is, that we have yet to find a description of Prolog's syntax that is deterministic and readable, both by humans and machines. Let us first make it clear that there is no, and never will be, a deterministic grammar for Prolog, as the language itself is ambiguous. The ISO Prolog standardization committee has recognized the multiple opportunities for ambiguity in operator expressions. Their draft standard has therefore defined restrictions on the syntax so that expressions can be parsed from left to right without needing significant backtracking or look-ahead. But, as we shall demonstrate, it is possible to relax these restrictions, not in the language *per se*, but on operator declarations, and obtain a deterministic grammar without sacrificing the expressive power of Prolog.

4.2 The Structure of Prolog

The part of the Prolog system that handles lexical and syntactical analysis is called the *reader*. There are at least four lexical categories in Prolog; variables, atoms, numbers, and punctuation symbols. A *variable* is any sequence of alphanumeric characters (including underscore), starting with either a capital letter, or an underscore. An *atom* is uniquely identified by its name, which is a sequence of characters, either alphanumeric starting with a lower case letter, or any of +-*/\^<>=‘~:.?@#$&, or one of !, ;, [], {}, or any sequence of characters delimited by single quotes. A *number* can be either an integer or float, and many implementations include facilities for representing numbers in different bases. Lastly, *punctuation symbols* include parenthesis, brackets, and a few others.

In contrast to many other programming languages, there are relatively few syntactic categories in Prolog. The most important of these are constants, terms, and compound terms, as reviewed in chapter 2.

A compound term in the form $f(a_1, \ldots, a_n)$ is said to be written in *standard syntax*. If compound terms were written in standard syntax only, Prolog would be a trivial language to parse, but a difficult one to read. To make Prolog more readable, unary and binary functors can be declared as operators of a given *fixity* — prefix, postfix, or infix — to allow the functor to appear before, after, or between its arguments. The predicate op/3 is used to declare an operator:

	Associativity		
Fixity	left	non	right
prefix		`fx`	`fy`
infix	`yfx`	`xfx`	`xfy`
postfix	`yf`	`xf`	

Figure 4.1: Prolog syntax for encoding fixity and associativity.

:- op(*precedence*, *type*, *name*).

The third argument gives the name of the operator. The first argument is an integer denoting the operator's precedence. If the precedence is 0, the operator is "undefined", that is, removed from the list of declared operators. There is also an upper limit on the precedence, usually set to 1200. The second argument to `op/3`, the *type*, encodes both the fixity and the associativity of the operator (fig. 4.1). In this notation, `f` represents the operator being defined, `x` represents a term with precedence less than `f`, and `y` a term with precedence less than or equal to `f`.

4.3 Subtleties of Prolog Syntax

In this section we discuss some matters relating to Prolog that might not be obvious to the casual user. The rules stated here apply to the family of Prolog implementations that adhere to the so called "Edinburgh syntax" [BBP+82]. Since whitespace is important in some examples, we will use the symbol ⊔ to indicate any sequence of one or more whitespace characters.

1. Compound terms written in standard syntax have precedence 0.

2. Compound terms written in operator notation have the same precedence as the principal functor.

3. If a prefix-declared operator acts as a term, its precedence is unchanged. Any other operator acting as a term is assigned precedence 0.

4. A quoted atom of the form `'`*char-sequence*`'` is interchangable with *char-sequence*, if *char-sequence* itself is a legal atom.

5. Arguments to a functor must have precedence less than 1000.
6. There cannot be a space between a functor and its argument list.
7. If the argument to a prefix operator is a parenthesized term, there must be a space between the operator and the opening parenthesis.

From the first two rules we can for instance conclude that `+(X,Y)` has precedence 0 (rule 1) but `X+Y` has precedence 500 (rule 2), the usual precedence of `+`. As mentioned before, `(X+Y)` has precedence 0.

Rule 3 implies that `+ +` constitutes a precedence-based error because `+` has a prefix, non-associative declaration, and thus the right `+` has precedence 500, which is too much for the left `+`. The motivation for this rule is obscure. It does not appear to make Prolog any easier to use, and it certainly complicates the parsing process. In fact, SICSTUS Prolog, who used to implement this rule, have now discarded it [CWA$^+$91].

Rule 4 simply tells us that for instance `'a+'` and `a+` are not interchangeable because `a+` is not a legal atom. More importantly however is the fact that the comma operator `,` (a standard operator in every Prolog implementation) is not the same as the quoted comma `','`. The reason is simple: the comma is not a legal atom and hence must be quoted to become legal.

Rule 5 is a consequence of the fact that the comma operator is an infix operator of precedence 1000. Thus, `f(X:-Y)` is illegal, since `:-` has precedence 1200. Such a term must be written as `f((X:-Y))` or as `f(:-(X,Y))`. The comma is not only used to separate arguments in compound terms; it is also used as a delimiter in compound goals, tuples, `(...)` and `{...}`, and lists `[...]`. Have you ever tried to change the properties of the comma operator? Your Prolog system will probably let you, but if it is based on the DEC-10 Prolog parser, which has all the uses of comma just mentioned hardcoded into it, your declaration will be ignored.

The last two rules (6 and 7) are related to each other: since a prefix operator applied to a parenthesized term looks a lot like a compound term it was apparently decided that prefix operators must be followed by a space, while functors must not. An example should clarify this. Consider the standard prefix operator `\+`, which represents logical negation. The goal `\+␣(f(X),g(Y))` succeeds only if either `f(X)` or `g(Y)` fails. What would happen if we omitted the space and wrote `\+(f(X),g(Y))`? This would not be flagged as a syntax error. Instead, it would be interpreted as the functor `\+` applied to the two arguments `f(X)` and `g(Y)`. Unless the programmer had defined rules for a predicate `\+` of arity 2 — which is rather unlikely — the goal `\+(f(X),g(Y))` would either generate a warning, or automatically fail, regardless of the success of `f(X)` and `g(Y)`, and the programmer will be very confused.

Let us elaborate on rule 3 a little bit more. As we shall see, allowing an operator to act as a term complicates parsing tremendously.

Example 4.3.1: Consider the following operator declarations:

```
    :- op(200, fx, u).
    :- op(400, yfx, b1).
    :- op(500, yfx, b2).
    :- op(300, yfx, b3).
```

The table below shows some terms involving these operators, and their corresponding parses written in standard syntax. Notice that in all cases, we are using the fact that the infix operators `b1`, `b2`, and `b3` have precedence 0 when used as terms.

Term	*Parse*
`u b1.`	`u(b1)`
`u b1 b2.`	`b1(u,b2)`
`u b1 b2 b3.`	`b2(u(b1),b3)`
`u b1 b2 b3 a.`	`b1(u,b3(b2,a))`

Notice that once the operator `u` has been parsed, any predictive parser would need to know if `u` should be treated as a term, or if the look-ahead symbol `b1` should be its argument. At least four elements of look-ahead are required to determine the correct action for the operator declarations given here. This example can be easily extended to create a situation in which the required look-ahead is arbitrarily large ▮

Example 4.3.2: Declare the following operators:

```
:- op(700, xf, #).
:- op(600, xf, @).
:- op(500, xfy, @).
```

Will Prolog parse `a @ #` as `@(a,#)` or `#(@(a))`? Both parses are acceptable, although the DEC-10 Prolog parser `read.pl` would give you the first answer. It used to be that a well-known Prolog interpreter, whose reader was based on `read.pl`, returned the second answer. More recently they have decided to go back to the first answer again ∎

4.4 Rectifying Prolog

In the previous section we examined a few syntactical curiosities of Prolog. Some of them make the language difficult to parse and tricky to use. We can eliminate the problems by making the following changes:

1. Any quoted symbol is treated as an atom, which prevents it from being used as an operator.
2. If a symbol that is defined as a prefix operator is to act as a functor, it must be quoted.
3. If an operator is parsed as a term, that term is assigned a precedence equal to the maximum of its declared precedences plus a small "delta".

The first condition removes the significance of whitespace, which makes parsing a lot easier, and regularizes syntax for the user. The third condition guarantees some results on deterministic grammars (see chapter 3) and also retains the flexibility of allowing operators to appear as terms.

It should be mentioned that the precedence and/or parsing of certain terms changes under condition 2. In standard Prolog, the term `-(X)` has precedence 0, since it is parsed as a functor applied to an argument. Under our system, `-(X)` has the same precedence as the `-` operator, because it is seen as an operator acting on a parenthesized term. We would have to write this as `'-'(X)` to enforce the functor interpretation. Also, a term like `*(3,4)`, where `*` is defined as an infix operator only, would yield the interpretation `'*'(','(3,4))`. A final example is `-3` vs. `-(3)`. In standard Prolog, these are not equal but in our system they are.

We would like to point out that even though condition 2 simplifies parsing and helps the user by avoiding the tricky problems demonstrated with `\+` previously, it is not hard to implement it in the traditional way, as we will show later.

4.5 The Prolog Grammar

Let us now turn to our proposed grammar. Fig. 4.2 shows the input to our parser generator DDGEN. The scanner we use is `rdtok.pl`[O'K90] with a minor modification: rather than acquiring the whole list of tokens before parsing commences we get the next token each time we perform a *shift*.

The first two lines declares `op` as a dynamic operator; if `atom(Name)` is returned from the scanner and `Name` is present in the current operator table, the token is converted to `op(Name)`. The comma is not considered a legal atom in Prolog, so we must also announce that it too can be an operator (line 2).

A Prolog program consists of a sequence of *sentences* (line 3 and 4). The keyword `empty` denotes an empty right hand side. Each sentence is a Prolog term, usually an operator expression with `:-` as the principal functor, terminated by a full-stop token "`.`". The Prolog system will normally look at each sentence as it has been read, and perform some sort of action, for instance verifying that the head of a procedure is not a number or a variable, or taking care of goals like `:- op(500,yfx,+)`. The call to `action/1` serves that purpose.

The bulk of the grammar describes the term, which can be either a compound term, a prefix, infix, or postfix operator expression, a nullary operator, a variable, name, number, or string, a tuple (either with parenthesis or curly-brackets), a cons expression, or list expression.

Each term carries two synthesized attributes. The first one, T, holds the abstract syntax tree for the term while the second attribute annotates the tree in the following way: if the operator at the root of T is not a comma, the annotation is n, meaning "no comma". If the operator is a comma, however, the annotation is c(C1,C2) where C1 and C2 are annotations for the left and right subtrees of T. There are two good reasons for doing this: The first one is we don't have to describe argument sequences for compound terms, element sequences for lists, element sequences for tuples, and goal sequences for compound goals. The second benefit is that the comma operator is not hardcoded into the grammar and its properties can be changed by the programmer. Now, we don't necessarily advocate changing it, we're merely pointing out the uniform treatment that follows. If the language designer wishes to prevent the user from tampering with the comma operator, it is easy to add such a test before calling the operator module.

To convert a term to a list of arguments we use comma2list(Annotation, Tree, ArgList) where Annotation and Tree are input arguments, and ArgList is the output.

```
comma2list(n, T, [T]).
comma2list(c(C1,C2), (T1,T2), L) :-
        comma2list(C1, T1, L1),
        append(L1, L2, L),
        comma2list(C2, T2, L2).
```

This, in combination with the *univ* operator =.., makes it easy to build the syntax tree.

The semantic action for variables, T = '$VAR'(T1) takes advantage of the fact that the standard procedures write/1 and writeq/1 prints '$VAR'(Var) simply as Var. This comes in handy when the abstract syntax tree needs to be written since variables are represented as var(Name) by the scanner, where Name is a quoted atom containing the variable's name. We use writeq/1 to make sure that what is printed can be re-read by read/1.

If the language designer wishes to implement rules 6 and 7, mentioned in section 4.3, it can done in the following way. The scanner already maintains a state structure containing, among other things, the next character in the input stream. This character can be passed

```
dynop_token(atom(Name), op(Name)).
dynop_token(’,’, op(’,’)).

sentences ::= term(T,_), { action(T) }, ’.’, sentences.
sentences ::= empty.

term(T,n) ::= name(Name), ’(’, term(T1,C), ’)’,
                { comma2list(C,T1,A), T =.. [Name|A] }.
term(T,n) ::= op(Name), term(T1,_), { T =.. [Name,T1] }.
term(T,C) ::= term(T1,C1), op(Name), term(T2,C2),
                { T =.. [Name,T1,T2], (Name = ’,’ -> C = c(C1,C2); C = n) }.
term(T,n) ::= term(T1,_), op(Name), { T =.. [Name,T1] }.
term(T,n) ::= op(T).
term(T,n) ::= var(T1), { T = ’$VAR’(T1) }.
term(T,n) ::= name(T).
term(T,n) ::= number(T).
term(T,n) ::= string(T).
term(T,n) ::= ’(’, term(T,_), ’)’.
term(T,n) ::= ’{’, term(T1,_), ’}’, { T =.. [’{}’,T1] }.
term(T,n) ::= ’[’, term(T1,C), ’|’, term(T2,_), ’]’,
                { comma2list(C,T1,A), append(A,T2,T) }.
term(T,n) ::= ’[’, term(T1,C), ’]’, { comma2list(C,T1,T) }.

name(T)   ::= atom(T).
name(T)   ::= qatom(T).
name([])  ::= ’[’, ’]’.
name({})  ::= ’{’, ’}’.
```

Figure 4.2: Prolog grammar.

back to to the parser as an extra argument to the token just read, for instance `atom(f,’(’)`. It should also be propagated up as a third synthesized attribute for terms so that (1) the production for compound terms can verify that there is a left parenthesis following, and (2) the production for prefix operators can verify that there is a whitespace following. Furthermore, an atom returned from the scanner must not be converted to a dynamic operator if the atom is followed by a left parenthesis.

4.6 Implementation and Results

The grammar was given to our parser generator and had no conflicts, apart from the *shift/reduce* conflicts involving the dynamic operators that are deferred until parse time. We have tested the parser by reading several large programs, among them the parser generator itself and the grammar just described, and writing the terms back out again to another file. By recompiling the output we were able to compare it against the original binary, thus allowing us to quickly test the parser without having to worry about whitespace and renamed variables.

5. Bottom-Up Evaluation of Attribute Grammars

We describe two transformation methods for (strong) non-circular attribute grammars that allows them to be evaluated within the S-attributed environment of an LR parser.

Traditionally the language designer, in order to get the evaluation done in one pass, has been confined to various restrictions of attribute definitions. If a larger class of attribute grammars was required, the entire dependency graph had to be saved for a post-parse traversal.

Our methods represent a compromise in that attribute evaluation is normally performed on the fly except when, in some evaluation rule, the referenced attributes are unavailable, and the execution of the rule has to be postponed. Suspension and resumption points for these evaluation rules can either be determined statically (method 1) or dynamically (method 2). For both methods, we guarantee that resumption takes place as soon as possible.

For the language designer it is now possible to continue using the one-pass model, but with the added option of being able to express "complicated" dependencies seamlessly, only paying for the extra power where it is used. The methods presented here can be used as a preprocessor to any parser generator that supports synthesized attributes, for instance Yacc.

5.1 Introduction and Background

An attribute grammar [Knu68] extends a context-free grammar by attaching attributes to each grammar symbol and by associating evaluation rules to each production, specifying how an attribute value may be computed in terms of other attribute occurrences in the same production. Among other things, attribute grammars have proved themselves useful in the specification of programming languages, thanks to their ability to convey context-sensitive information.

A context-free grammar defines a parse tree for each legal input sentence. In the tree, attributes can be thought of as extra fields attached to each node. Evaluation rules then define the value of an attribute in a node in terms of other attributes in child or parent nodes, a dependency we may depict with an arc to the referenced node. The evaluation problem, that is, the process of assigning a value to each attribute in the tree, can then be completed if the overall dependency graph has a partial order in which each attribute and its defining rule may be visited.

In practice, parsing is done in one pass, either top-down or bottom-up, so that the parse tree does not have to be created explicitly. Likewise, attribute evaluation can also be done "on-the-fly" if a sufficient set of restrictions is imposed on the evaluation rules; for instance, if every attribute depends only on attribute values at the children of the node, then the computed attribute value is called "synthesized" and a bottom-up parser may perform the attribute evaluation as it is parsing the input. Such a subclass is called an S-attributed grammar. Top-down parsers, in addition, are also capable of handling some "inherited" attributes (where an attribute depends on attribute values in ancestor nodes) thus defining the larger subclass called L-attributed grammars. It is not true, however, that a top-down parser, such as an LL(1) parser is a more powerful specification tool than an bottom-up, LR(1) parser [Bro74].

Besides the obvious time and space savings obtained from the one-pass model, attribute values can also be used to solve parsing conflicts, or assist in error recovery.

5.1.1 Related Work

Prior work in this area can be divided into one-pass evaluators, post-parsing evaluators, and compromises thereof.

Top-down parsers have a simple correspondence to L-attributed evaluation [ASU85]. Evaluation of inherited attributes in a bottom-up parser is more difficult but can for instance be arranged by the insertion of so called "copy symbols" [Wat77]. One-pass evaluation methods are surveyed by op den Akker et al [odAMT91]. Inherent in their design is the assumption that dependencies are no more complicated than that the attribute values of the "working frontier" of the parse tree can be kept on a stack so that evaluation can be done in some left-to-right fashion [Boc76]. Therefore, these methods are restricted to the L-attributed grammars.

Post-parsing evaluators save the entire parse tree, or its dependency graph, and are thus capable of handling any non-circular grammar by executing the evaluation rules according to the order of the graph. An interesting variation on this method is described by Mayoh [May81]. He shows that attribute grammars can be reformulated as sets of recursive functions, taking the parse tree as an argument. Related ideas have appeared earlier by Kennedy and Warren [KW76], and later by Courcelle et al [CFZ82]. Our first method differs in that the functions, which are passed as synthesized attributes, are created while parsing, and only when the evaluation rules contains non-S-attributed constructs.

Another related method is given in [JM80] where attribute grammars are analyzed to find out which attributes can be computed during an LR parse. The remaining attributes, called "unknown", are saved for a post-parsing evaluation phase. In contrast, our methods are more "eager" because computation of unknown attributes is not necessarily put off until after the parse.

The functional parser generator FPG [Udd88] generates parsers for Lazy ML, a functional programming language with lazy evaluation. Due to the demand-driven nature of this evaluation method, attribute evaluation comes "for free" and works for all non-circular grammars. Our second method is similar in the respect that resumption of suspended evaluation rules is done automatically by the runtime system, but we use a coroutine construct to explicitly suspend rules. Thus, on the spectrum of labor division, our first method requires explicit suspension and resumption, but assumes nothing of the runtime system; our second method requires explicit suspension, but relies on a coroutine construct to "wake up" procedure calls; and FPG, finally, is completely transparent, but leaves all the work for the lazy evaluator.

5.1.2 Summary of Contributions

In this chapter we present two transformation methods that let a bottom-up parser simulate the evaluation of strongly non-circular grammars and non-circular grammars, respectively. These two subclasses are for all practical purposes large enough and superior to the subclasses developed for traditional one-pass evaluation, such as the commonly used L-attributed grammars.

The common theme for our methods is that evaluation of attributes is conducted bottom-up; when an evaluation rule needs (inherited) attributes, its execution is postponed until the referenced attributes are available. Because of this, the language designer has the full power of (strong) non-circular grammars at hand but does not have to pay the price for "complicated" dependencies in the evaluation rules, except in those parts of the grammar where they are used.

The first method, based on the ideas by Mayoh [May81], and Kennedy and Warren [KW76], postpones evaluation rules by passing them to the ancestor node as synthesized attributes. In addition to the algorithm we give a theorem to guarantee that attribute evaluation is not unnecessarily delayed and that the transformation is safe.

The second method is influenced by ideas in [Udd88] and also by comments from the reviewers of [PVGK93]. While the method is conceptually easier to explain and implement, it assumes that the host language has a coroutine construct to delay the evaluation of a procedure call until a specific condition has been met.

5.2 Definitions

We recall the definition of a context-free grammar from chapter 2 as a four-tuple $G = \langle V_N, V_T, S, P\rangle$ where the set $P \subseteq V_N \times V^*$ consists of m productions and the p-th productions is

$$X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn_p}$$

.

1.	$Z(v_0) \rightarrow N(r_1, l_1, v_1)\ .\ N(r_3, l_3, v_3)$	$\{v_0 = v_1 + v_3, r_1 = 0, r_3 = -l_3\}$
2.	$N(r_0, l_0, v_0) \rightarrow N(r_1, l_1, v_1)\ B(r_2, v_2)$	$\{v_0 = v_1 + v_2, l_0 = l_1 + 1, r_2 = r_0, r_1 = r_0 + 1\}$
3.	$N(r_0, l_0, v_0) \rightarrow \varepsilon$	$\{v_0 = 0, l_0 = 0\}$
4.	$B(r_0, v_0) \rightarrow \mathtt{0}$	$\{v_0 = 0\}$
5.	$B(r_0, v_0) \rightarrow \mathtt{1}$	$\{v_0 = 2^{r_0}\}$

Figure 5.1: Attribute grammar definition for binary numbers.

With each symbol grammar symbol $X \in V$ we associate a finite set $A(X)$ of *attributes*, partitioned into two disjoint sets, the *synthesized attributes* $S(X)$ and the *inherited attributes* $I(X)$. Terminal symbols have no inherited attributes. We define the set of attributes of a production p as $A(p) = \{A(X_{pk}) \mid 0 \leq k \leq n_p\}$. When we wish to be explicit about the occurrence of an attribute $a \in A(X_{pk})$ we sometimes write a_k, or even a_{pk}.

Evaluation rules are statements in some language, attached to productions, specifying how an attribute a_{pk} may be computed in terms of other attributes. We associate therefore a *goal* g_{pka} with each attribute a_{pk}. The goal is usually of the form $a_{pk} = \ldots$ where the right hand side contains references to other attributes, although Prolog specifies the direction of computation through so called modes. A rule is said to be in *normal form* if either $k = 0$ and $a_{pk} \in S(X_{p0})$, or $k > 0$ and $a_{pk} \in I(X_{pk})$.

An *attribute grammar* consists of a context-free grammar, a domain of attribute values and a set of evaluation rules.

Example 5.2.1: Fig. 5.1 shows an attribute grammar definition for binary numbers. When attribute grammars are written in normal form one can always deduce whether an attribute occurrence is synthesized or inherited by looking for their defining rule — only synthesized attributes in the left-hand side and inherited attributes in the right-hand side have rules.

In this example, the synthesized attributes l and v stands for “length” and “value”, respectively. The inherited attribute r denotes the bit radix to which each binary digit is scaled.

To find the value of a binary number then, one may first proceed bottom-up to compute all l’s, then down again, assigning values to the r’s, and finally up again, adding up the v’s ▮

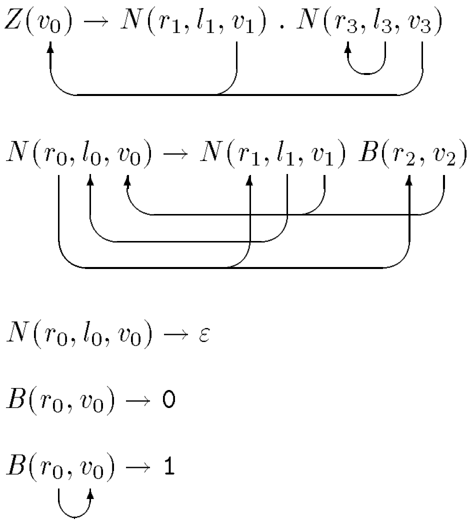

$Z(v_0) \rightarrow N(r_1, l_1, v_1)$. $N(r_3, l_3, v_3)$
$N(r_0, l_0, v_0) \rightarrow N(r_1, l_1, v_1)\ B(r_2, v_2)$
$N(r_0, l_0, v_0) \rightarrow \varepsilon$
$B(r_0, v_0) \rightarrow$ 0
$B(r_0, v_0) \rightarrow$ 1

Figure 5.2: Dependency graphs for productions.

For a production p, a *dependency graph* $D(p)$ contains a set of vertices $A(p)$ and an arc $a' \mapsto a$ only if attribute a depends directly on the value of attribute a' by the evaluation rule of p. We say that a' is a *predecessor* to a and $a' \in pred(a)$.

Example 5.2.2: The evaluation rules in fig. 5.1 induce dependencies in the productions as shown in fig. 5.2 ▮

The dependency graph $D(T)$ for a parse tree T is formed by pasting together smaller graphs corresponding to the dependency graphs for the productions used in the derivation. In general, if T is a parse tree with root node X_{p0}, $D(T)$ is obtained from $D(p)$ and $D(T_1), \ldots, D(T_{n_p})$ by identifying the vertices for attributes of X_{pj} in $D(p)$ with the corresponding vertices for the attributes in $D(T_j)$, $1 \leq j \leq n_p$.

Example 5.2.3: The parse tree for the string "10.1" have the dependencies shown in fig. 5.3 ▮

We will also have the need to consider dependency relations on attributes of grammar symbols. Terminal symbols carry only synthesized attributes, so their relation is by default empty. For a parse tree T and its root $X \in V_N$, the dependencies among the attributes in X

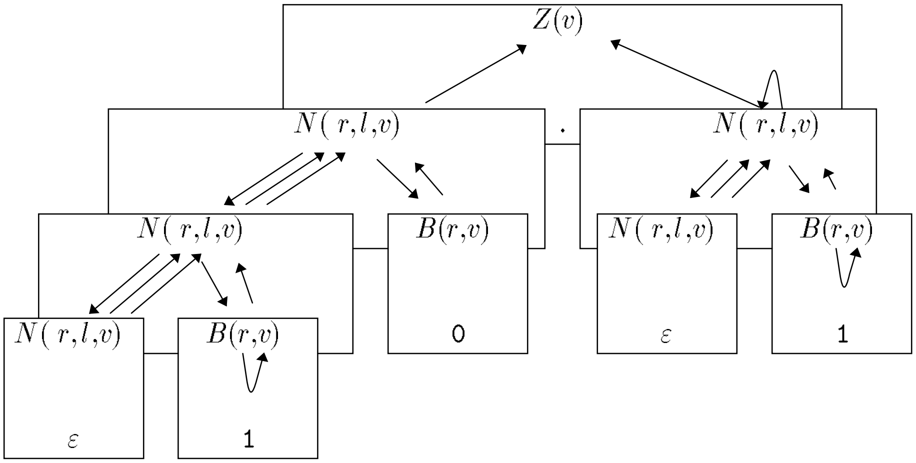


Figure 5.3: Parse tree and dependencies for “`10.1`”.

$$\mathcal{F}(Z) = \{Z(v)\}$$
$$\mathcal{F}(N) = \{N(r,l,v), N(\overset{\curvearrowright}{r,l,v})\}$$
$$\mathcal{F}(B) = \{B(r,v), B(\overset{\curvearrowright}{r,v})\}$$
$$\mathcal{F}(\texttt{0}) = \mathcal{F}(\texttt{1}) = \mathcal{F}(.) = \emptyset$$

Figure 5.4: Dependencies for grammar symbols.

can be specified as the transitive closure of $D(T)$ restricted to the attributes of X, that is, $F(X) = D(T)^{+}|_{A(X)}$. Although there may be an infinite number of parse trees associated with X, the fact that $A(X)$ is finite implies that the set of all $F(X)$, henceforth denoted $\mathcal{F}(X)$, is also finite.

Example 5.2.4: For the attribute grammar in fig. 5.1 the symbols have the dependencies as shown in fig. 5.4 ▮

A *composite graph*

$$D(p)[F(X_{p1}), \ldots, F(X_{pn_p})]$$

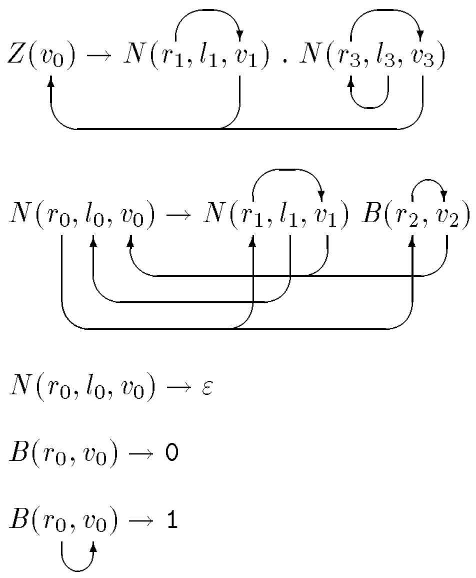

Z(v0) → N(r1, l1, v1) . N(r3, l3, v3)
N(r0, l0, v0) → N(r1, l1, v1) B(r2, v2)
N(r0, l0, v0) → ε
B(r0, v0) → 0
B(r0, v0) → 1

Figure 5.5: Strong composite graphs.

for a production p is obtained from $D(p)$ by adding an arc to a' from a whenever there is an arc to a' from a in $F(X_{pj})$, $1 \leq j \leq n_p$.

A *strong composite graph*

$$D(p)[\mathbf{F}(X_{p1}), \ldots, \mathbf{F}(X_{pn_p})]$$

is formed by adding arcs from $\mathbf{F}(X_{pi})$, defined as the union of the graphs in $\mathcal{F}(X_{pi})$.

Example 5.2.5: The strong composite graphs for the productions in fig. 5.1 are shown in fig. 5.5 ▮

Attribute grammars can be characterized by means of their expressive power. The following is an incomplete list, starting with the strongest form.

1. An attribute grammar is *non-circular* if, for all productions p, there are no cycles in any of the graphs $D(p)[F(X_{p1}), \ldots, F(X_{pn_p})]$ for $F(X_{p1}) \in \mathcal{F}(X_{p1}), \ldots, F(X_{pn_p}) \in \mathcal{F}(X_{pn_p})$.

   An algorithm for verifying that an attribute grammar is non-circular was first given in [Knu71]. The problem has been shown to have exponential complexity [JOR75].

2. An attribute grammar is *strongly non-circular* if, for all productions p, there are no cycles in p's strong composite graph.

   An algorithm for strong non-circularity testing was (accidentally) first given in [Knu68]. Strong non-circularity can be verified in polynomial time [CFZ82].

3. An attribute grammar is *L-attributed* if it is strongly non-circular and the inherited attributes of X_{pj} depend only on the attributes in $I(X_{p0}) \cup A(X_{p1}) \cup \cdots \cup A(X_{pj-1})$.

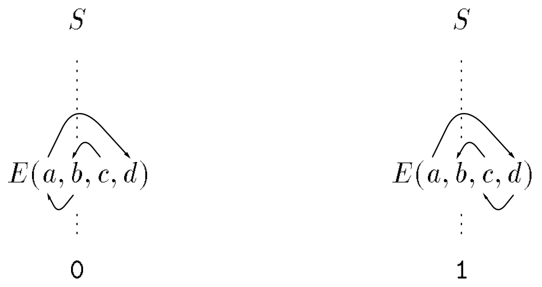
4. An attribute grammar is *S-attributed* if it does not contain inherited attributes at all.

Attribute grammars with circular definitions are usually considered bogus as there is no place to start the evaluation.

Example 5.2.6: The following artificial example demonstrates an attribute grammar that is non-circular, but not strongly non-circular.

$$\begin{array}{lll}
1. & S \rightarrow E(a_1, b_1, c_1, d_1) & \{d_1 = a_1, b_1 = c_1\} \\
2. & E(a_0, b_0, c_0, d_0) \rightarrow \mathtt{0} & \{a_0 = b_0, c_0 = 0\} \\
3. & E(a_0, b_0, c_0, d_0) \rightarrow \mathtt{1} & \{a_0 = 1, c_0 = d_0\}
\end{array}$$

Notice that neither parse tree below has circular dependencies, but together they introduce a loop among the attributes in $\mathbf{F}(E)$, hence the example above is not strongly non-circular.



∎

5.3 Transformation Methods

Evaluators can broadly be divided into two classes: those who perform evaluation on-the-fly — either by imposing restrictions as above or by being "oblivious" to the rules, like Yacc — and those who construct the dependency graph for the parse tree to perform a post-parsing evaluation stage.

By default, LR-parsers that perform on-the-fly evaluation are limited to S-attributed grammars due to the fact that the parse tree is built bottom-up and the "input" to the production, $I(X_{p0})$, is supplied "from above" thus being unavailable when the production is reduced.

We will consider two transformation methods that enable LR-parsers to handle strong non-circular grammars (method 1) and non-circular grammars (method 2), respectively. Our methods represent a compromise between keeping all attribute information on the parse stack and building the dependency graph for the entire parse tree. Whenever possible, our methods will compute an attribute value as soon as all its predecessors become available. Ideally, all attribute values for a production would be computed at the time it is reduced, but the abovementioned problems with the absent inherited attributes sometimes forces us to postpone some of the evaluation, until the inherited attributes are available.

The set of attributes in a production p that *can be* computed at reduction time, denoted $\mathit{Ready}(p)$, is inductively defined as the smallest subset of $A(p)$ that satisfies

- If $a_{pk}, k > 0$, has no predecessor in the strong composite graph for p then a_{pk} is in $\mathit{Ready}(p)$.
- If $a_{pk} \notin I(X_{p0})$ and all its predecessors are in $\mathit{Ready}(p)$, then a_{pk} is also in $\mathit{Ready}(p)$.

For a synthesized attribute a of X let

$$\mathit{Use}(X, a) = \{a' \mid a' \in I(X) \ \wedge \ a' \mapsto a \in \mathbf{F}(X)\}$$

that is, the set of inherited attributes of X that determines a. (Notice that strong non-circularity is used.)

5.3.1 Method 1: Synthesized Functions

In this section we present the first method for evaluating a strongly non-circular attribute grammar within a bottom-up (S-attributed) environment.

Algorithm 5.3.1: INPUT: a set of productions with attributes and rules:

$$X_{p0}(S_0, I_0) \rightarrow X_{p1}(S_1, I_1) \ldots X_{pn_p}(S_{n_p}, I_{n_p}) \qquad \{G_p\}$$

where $S_j = S(X_{pj})$ and $I_j = I(X_{pj})$ respectively, and G_p is a set of goals g_{pka} that computes attributes a_{pk}.

OUTPUT: a set of productions with new attributes and rules:

$$X'_{p0}(S'_0) \rightarrow X'_{p1}(S'_1) \ldots X'_{pn_p}(S'_{n_p}) \qquad \{G'_p\}$$

where $|S_j| = |S'_j|$ for $0 \leq j \leq n_p$, and G'_p is the new rule.

METHOD: The general idea behind the transformation is that, since the the "input" to the production, I_0, is not available when p is reduced, any attribute depending on I_0, in particular the "output" of the production, S_0, can in general not be computed until the ancestor in the parse tree supplies the input. In accordance with this input-output view, we can then return, in place of S_0, a set of functions that returns the synthesized attribute they replace, given the inherited attributes on which they depend.

Productions are processed individually as follows: Let $C(p)$ be the strong composite graph for p. If the graph has a cycle, the algorithm must halt and announce that the attribute grammar is not strongly non-circular.

Otherwise, new attributes are introduced: if X_{pk} is a terminal symbol, the lexical analyzer provides the synthesized attributes S_k, so naturally $S'_k = S_k$. However, for a nonterminal X_{pk}, S_k is replaced by a equal-sized set of of synthesized functions S'_k. Each function $f_{pka} \in S'_k$ corresponds to the synthesized attribute $a \in S_k$ it replaces. The old synthesized attributes $a \in S_k$ in the right side of the production can then be computed by defining new goals $g_{pka} : a = f_{pka}(args)$, where $args$ are the attributes in $Use(X_{pk}, a)$.

```
**procedure** $preds(a)$
**begin**
    **if** $a$ is marked "unknown" **then**
        mark $a$ as "known"
        **for** each $a'$ such that $a' \mapsto a \in C(p)$, $preds(a')$
        $output(g_{pka})$
    **end**
**end**

**procedure** $body(a)$
**begin**
    **for** each $a' \in A(p)$, mark $a'$ as "unknown"
    **for** each $a' \in I_0 \cup Ready(p)$, mark $a'$ as "known"
    $preds$(a)
**end**
```

Figure 5.6: Definition of procedure *body*(a).

It remains to describe how G'_p is built. While G_p had no implicit ordering of the goals but rather served as a specification, G'_p is a *sequence* of goals computing the attributes of $Ready(p)$, followed by a set of function definitions, one for each synthesized attribute in S_0. Initially, G'_p is empty. First visit each attribute a in $Ready(p)$ in some linear order imposed by $C(p)$ and append g_{pka} to G'_p. Then append for each synthesized attribute $a \in S_0$ a definition "$f_{pka}(args) = a$ *where* $body(a)$" of the corresponding $f_{pka} \in S'_0$, where $args$ are the attributes in $Use(X_{pk}, a)$ and $body(a)$ is a (possibly empty) sequence of goals that ultimately computes a (fig. 5.6).

Example 5.3.1: Fig. 5.7 shows the result of applying algorithm 5.3.1 to the attribute grammar in fig. 5.1, based on the strong composite graphs in fig. 5.5.

The reader may notice that the output from the system is no longer a value (namely that of the binary string) but rather a function of no arguments, returning that value. While most authors assume that the start symbol has no inherited attributes we have not felt the need to impose such a restriction; if the returned function is not desirable, one may omit its construction for the start symbol as a special case ▮

$$\begin{array}{lll}
1'. & Z(f_{10v}) \to N(f_{11l}, f_{11v}) \; . \; N(f_{13l}, f_{13v}) & \{\; r_1 = 0, l_1 = f_{11l}(), l_3 = f_{13l}(), v_1 = f_{11v}(r_1), \\
 & & \;\; r_3 = -l_3, v_3 = f_{13v}(v_3), \\
 & & \;\; v_0 = v_1 + v_3, f_{10v}() = v_0 \;\} \\
 & & \\
2'. & N(f_{20l}, f_{20v}) \to N(f_{21l}, f_{21v}) \; B(f_{22v}) & \{\; l_1 = f_{21l}(), l_0 = l_1 + 1, f_{20l}() = l_1, \\
 & & \;\; f_{20v}(r_0) = v_0 \textit{ where } (\; r_1 = r_0 + 1, \\
 & & \qquad\qquad\qquad\qquad\; v_1 = f_{21v}(r_1), \\
 & & \qquad\qquad\qquad\qquad\; r_2 = r_0, \\
 & & \qquad\qquad\qquad\qquad\; v_2 = f_{22v}(r_2), \\
 & & \qquad\qquad\qquad\qquad\; v_0 = v_1 + v_2) \;\} \\
3'. & N(f_{30l}, f_{30v}) \to \varepsilon & \{\; f_{30l}() = l_0 \textit{ where } l_0 = 0, \\
 & & \;\; f_{30v}(r_0) = v_0 \textit{ where } v_0 = 0 \;\} \\
4'. & B(f_{40v}) \to \mathtt{0} & \{\; f_{40v}(r_0) = v_0 \textit{ where } v_0 = 0 \;\} \\
5'. & B(f_{50v}) \to \mathtt{1} & \{\; f_{50v}(r_0) = v_0 \textit{ where } v_0 = 2^{r_0} \;\}
\end{array}$$

Figure 5.7: Attribute grammar for binary numbers using synthesized functions.

Correctness

An important theorem follows immediately from the definitions of $\mathit{body}(a)$ and $\mathit{Ready}(p)$.

Theorem 5.3.1: When all predecessors of an attribute a have been computed, a is also computed as soon as the current production is reduced.

Proof: The theorem has two sides.

- First we must show that no undefined attribute value is referenced in the computation of a. If $a \in \mathit{Ready}(p)$ then this part follows from the fact that all predecessors of a are also in $\mathit{Ready}(p)$ and must have been computed before a, since they are computed in the order of $C(p)$. If a is computed inside the body of a function, this part follows from the definition of $\mathit{body}(a)$ which ensures that the goals for all predecessors of a have been generated *before* a's goal.
- Secondly we must show that the computation of an attribute isn't unnecessarily "frozen" by being enclosed and passed up inside a function. For this part we notice that the only attributes computed inside a function are those who transitively depend on the unavailable $I(X_{p0})$; the other attributes (that can be computed) are captured by the definition of $\mathit{Ready}(p)$ ▮

Implementation

We have implemented algorithm 5.3.1 in Prolog as a front-end to our LR parser generator DDGEN, which in itself generates Prolog parsers [PVGK93]. Together, they serve as a replacement for the traditional implementation of Definite Clause Grammars, namely top-down parsing with backtracking.

In Prolog, synthesized functions can be implemented either by passing goals and using *call* for application, or by creating new predicates with *assert*. In an imperative setting, like C with Yacc, the synthesized functions can be implemented with pointers to "closures", structures with a function pointer and a set of attribute variables holding bindings that have already been made when the function is passed up. Closures have to be allocated dynamically but their space can be reclaimed immediately after the function has been applied.

5.3.2 Method 2: Coroutines

The second method is easier to explain and implement, but requires a coroutine construct in the language. For the purposes of this exposition we will use the rather self-explanatory predicate *when*(*Cond*, *Goal*) from SICSTUS Prolog [CWA$^+$91]. When executed, this predicate blocks the execution of *Goal* until *Cond* is true. The idea of the transformation is that evaluation rules are "frozen" until their referenced attributes are available.

Algorithm 5.3.2: INPUT: a set of productions with attributes and rules:

$$X_{p0}(S_0, I_0) \rightarrow X_{p1}(S_1, I_1) \ldots X_{pn_p}(S_{n_p}, I_{n_p}) \qquad \{G_p\}$$

where the notation is the same as in the previous algorithm.

OUTPUT: The same productions, but with new rules G'_p.

METHOD: For each goal $g_{pka} \in G_p$ add the new goal "$when(C, g_{pka})$" to G'_p where C is the condition that tests whether all referenced attributes in g_{pka} have been computed ▮

Example 5.3.2: The result of applying the algorithm to our running example (fig. 5.1) can be seen in fig. 5.8.

$$
\begin{array}{lll}
1'. & Z(v_0) \rightarrow N(r_1, l_1, v_1)\ .\ N(r_3, l_3, v_3) & \{when((nonvar(v_1), nonvar(v_3)), v_0 = v_1 + v_3), \\
 & & \ r_1 = 0, \\
 & & \ when(nonvar(l_3), r_3 = -l_3)\} \\
2'. & N(r_0, l_0, v_0) \rightarrow N(r_1, l_1, v_1)\ B(r_2, v_2) & \{when((nonvar(v_1), nonvar(v_2)), v_0 = v_1 + v_2), \\
 & & \ when(nonvar(l_1), l_0 = l_1 + 1), \\
 & & \ when(nonvar(r_0), r_2 = r_0), \\
 & & \ when(nonvar(r_0), r_1 = r_0 + 1)\} \\
3'. & N(r_0, l_0, v_0) \rightarrow \varepsilon & \{v_0 = 0, l_0 = 0\} \\
4'. & B(r_0, v_0) \rightarrow \mathtt{0} & \{v_0 = 0\} \\
5'. & B(r_0, v_0) \rightarrow \mathtt{1} & \{when(nonvar(r_0), v_0 = 2^{r_0})\}
\end{array}
$$

Figure 5.8: Attribute Grammar for binary numbers using coroutines.

Although the evaluation rules are supposed to be written in some generic programming language, we have used the Prolog test predicate *nonvar*/1 to block some of the evaluation rules until their referenced attributes become bound ▮

Correctness

We notice that theorem 5.3.1 also holds for this algorithm. In addition to the previous algorithm, this method also handles grammars that are non-circular. One might actually use a circular attribute definition and end up with a set of goals, suspended and waiting for some (external) agent to bind a variable to get the evaluation rolling.

Implementation

The only implementation issue for the second method is the extraction of referenced variables in a goal. For an imperative or functional language, where assignments are used almost exclusively in the evaluation rules, the referenced variables are simply all the variables in the right-hand side of the assignment.

For a relational language however, we depend on mode declarations (cf. section 2.2.5). The referenced attributes in a goal are then simply the variables appearing in **c**-moded argument positions. As an example, suppose the procedure *lookup*(*Name*, *Value*, *Table*) is used in a goal with a mode declaration $(\mathbf{c}, \mathbf{d}, \mathbf{c})$. We can then deduce that *Name* and *Table* are referenced attributes, while *Value* is the computed attribute.

6. Mutual Exclusion Analysis

A technique to detect that pairs of rules are "mutually exclusive" in a logic program is described. In contrast to previous work our algorithm derives mutual exclusion by looking not only on built-in, but also user-defined predicates. This technique has applications to optimization of the execution of programs containing these rules. Additionally, the programmer is less dependent on non-logical language features, such as Prolog's "cut", thus creating more opportunities for parallel execution strategies.

6.1 Introduction

When a relation is defined by several rules in a logic program, there is sometimes the opportunity to realize that if tuples of a certain pattern are derived by one of these rules, no tuples of that pattern can be derived by other rules. This presents an optimization opportunity.

The problem of "mutual exclusion", or disjointness, of rules has certain similarities to constraint inference, but is actually quite a different problem. Both may study order constraints, such as $X > Y$, but constraint inference attempts to use properties of ">", such as transitivity, to infer additional constraints. However, in mutual exclusion detection, the basic fact available is that, for a given pair (X, Y) it is not possible that both $(X > Y)$ and $(Y > X)$ hold: they are *mutually exclusive*. In addition, patterns of mutual exclusion can occur without regard to any ordering concept. For example, we might have $\mathit{tokentype}(X, \mathit{number})$ and $\mathit{tokentype}(X, \mathit{identifier})$ that are known or given as disjoint on their first argument.

Let us mention a few applications of mutual exclusion detection.

1. In a top-down execution strategy with sideways information passing, if a tuple for some subgoal[1] $q^{\mathbf{c}\ldots\mathbf{d}}(X, \ldots, Z)$ is derived using one rule for q and it is known that all

[1] We use the notation that superscripts $\mathbf{c}$ and $\mathbf{d}$ denote "constant" (ground) and "don't know"-arguments at the time the subgoal q is "called."

rules for q are mutually exclusive on their first arguments, then none of the remaining rules need to be tried with this value of X.

Detection of mutual exclusion is particularly useful when a goal matches two or more rules with recursive definitions — if a recursive invocation happens when other rules have yet to be tried, the execution engine must save backtracking information in case the selected rule fails. But if the rules are mutually exclusive to each other, such information can be discarded if the success of all subgoals appearing before the recursive call contradicts the other rules. This creates more opportunities for tail recursion optimization [War86] and will also prevent the execution engine from wasting time on the other rules, in case all solutions are asked for.

2. In a bottom-up execution strategy the union of relations from mutually exclusive rules is a disjoint union; duplicates cannot occur.

3. In a logic language that contains the non-logical "cut" operation, which is explicit in Prolog, and implicit in some others, there is a distinction drawn between "green" cuts and "red" cuts. Recall that "cut" in Prolog is a directive to the execution engine to cancel certain backtracking activities that would normally have occurred in the future. A cut is said to be "green" if the cancelled backtracking could not have produced any additional solution tuples (for reasons known to or believed by the programmer), otherwise it is "red". Red cuts are often considered bad style for much the same reason as "go to"s in a procedural language. Mutual exclusion analysis can provide a *sufficient* condition for cuts to be "green". In a reasonably expressive language (say SQL) the question is undecidable (by essentially the same arguments that show undecidability of "domain independence").

4. The detection of *functionality* [DW89] represents an important space and time saving optimization for Prolog-like languages. Various forms of functionality have been considered; the algorithm by Debray and Warren defines functionality as when "all alternatives produce the same result, which therefore need not be computed repeatedly" [DW89]. With this information at hand, the execution engine does not need

to waste time on, or save state information for finding alternative solutions. Debray and Warren demonstrate that a predicate is functional by showing that each individual clause is functional, and that the clauses are pairwise mutually exclusive. The latter requirement is not a necessary, but sufficient condition. Information produced by the methods outlined in this chapter can therefore improve the precision of their algorithm.

5. In a parallel execution strategy with or-parallelism, a set of processors working on mutually exclusive rules can be relieved of their duties as soon as one of them succeeds. Also, in the Andorra model [HB88], goals with only one matching clause are executed before other goals. Static analysis has been used to detect such properties of goals [CWY91, PN91].

After discussing related work and establishing some required terminology, we illustrate the ideas of our technique by means of a small example before describing the full algorithm. A larger example concludes the chapter.

6.1.1 Related Work

The idea of recognizing mutual exclusion in Prolog programs has been considered by Hickey and Mudambi [HM89], Debray and Warren [DW89], and Van Roy [VR90]. Their methods works on the level of primitives, that is, only built-in predicates such as arithmetic comparisons and unifications are examined. In contrast to earlier work, our algorithm also examines user-defined predicates, even those with recursive definitions.

Several methods for static or dynamic inference of determinacy in the parallel language Andorra have been proposed [PN91, CWY91, KT91]. In the Andorra model, a goal is determinate if it has at most one matching rule; thus the effort has been directed into checking whether two or more rules can satisfy a goal invocation. Whether our method, which is based not only on the information in the head of the rule, but also the presence of mutually exclusive subgoals in the bodies, can be useful for this purpose, remains to be explored.

A related topic, that of detecting *functionality*[2], has been investigated by Debray and Warren [DW89], Sawamura and Takeshima [ST85], and Mellish [Mel85]. A functional predicate is one where all alternatives produce the same result. The notion is related to mutual exclusion, but is not the same, as the following two rules demonstrate.

$$son(X,Y) \leftarrow male(X), father(X,Y).$$
$$son(X,Y) \leftarrow female(X), mother(X,Y).$$

A goal "$\leftarrow son(carl,Y)$" could potentially have several solutions for Y — one for each of Carl's sons — although only the first clause is applicable. Therefore *son* is determinate, but not functional.

Various methods for constraint inference in logical rules have been proposed [UVG88, BS89a, BS89b, KKR90, Las90, SVG91, VG91, BS91], but to our knowledge none have been implemented. The performance of our algorithm could be enhanced in practice if such an inference system were available, by simple rule transformations: whenever a constraint $c(X,Y)$ is inferred to hold for the head of a rule, append it explicitly as an additional subgoal. Then run our algorithm as described on the resulting set of rules.

6.1.2 Summary of Contributions

A new technique for recognizing mutual exclusion among rules in logic programs is presented. While in general the problem is undecidable, a conservative analysis works on many programs, such as databases, natural language parsers, or any type of program where predicates operating on a certain domain divide it into equivalence classes.

Previous researchers have considered what we call primitive mutual exclusion, that is, the analyzer only looks at built-in predicates such as "=" or ">". In contrast, our work facilitates a notation for propagating mutual exclusion information to user-defined predicates, even when they have recursive definitions. (Without recursion, a simple unfolding strategy would reduce the problem back to recognizing primitive mutual exclusion.)

[2] Sometimes called *determinacy* by some authors. We reserve this term for goals that have mutually exclusive rules.

Output from the analysis can be used to optimize various execution strategies. For instance, if all rules of a goal are mutually exclusive to each other, a top-down interpreter may at a certain point be able to commit to the rule it is working on. In languages where recursion is the only looping construct, such information could mean the difference of executing in constant space as opposed to space being linear with time.

6.2 Definitions

Following the standard terminology of [Llo87] some additional terms are defined for this chapter.

The position of a term t in an atom A is given by the relation $A \overset{\psi}{\longmapsto} t$ where ψ is a sequence of numbers that "spells" the argument path to t in A ("Dewey notation"). The head of a sequence ψ is denoted $hd(\psi)$. As an example, if A is the atom $p(X, g(X, Y))$ we have both $A \overset{1}{\longmapsto} X$ and $A \overset{2.1}{\longmapsto} X$. The relation "$\longmapsto$" is transitive: if $A \overset{\psi_1}{\longmapsto} t_1$ and $t_1 \overset{\psi_2}{\longmapsto} t_2$, it follows that $A \overset{\psi_1\psi_2}{\longmapsto} t_2$.

The projection operation "π" from relational algebra is extended in the following manner. If $\Psi = \langle \psi_1, \ldots, \psi_k \rangle$ is a tuple of paths, and A is an atom with a relation R then $\pi_\Psi(R) = \langle t_1, \ldots, t_k \rangle$ where $A \overset{\psi_i}{\longmapsto} t_i$. Sometimes it is convenient to have a relation or a tuple conform to the scheme of another relation; we use the customary notation $\pi_R(S)$ to mean the projection of S onto R's columns, and $\mu[S]$ for the components of μ in the attributes of S.

The notation "$R \| S$" means that the relations R and S are disjoint.

Variables appearing in **c**-moded argument positions of rule heads are of particular interest to us. We restrict the "$\longmapsto$"-relation to such ground variables as follows: if $\langle \delta_1, \ldots, \delta_n \rangle$ is the mode tuple for a predicate p and $p \overset{\psi}{\longmapsto} X$, then $p \overset{\psi}{\longmapsto}_{\mathbf{c}} X$ holds iff $\delta_{hd(\psi)} = \mathbf{c}$.

6.2.1 Rule/Goal Graphs

The input program is stored in a data structure called rule/goal graph. A rule/goal graph is basically a call graph for the program where rules and goals have been made explicit. Starting with the program's goal rule, which we can assume to be of the form "$\leftarrow B$" for simplicity, we create a start goal node for B. If there is a rule $A \leftarrow B_1, \ldots, B_n$ such that $A = B\theta$, then $\theta(A \leftarrow B_1, \ldots, B_n)$ constitutes a rule node and becomes a child of B. The rule node itself becomes the parent of n goal nodes, one for each subgoal in the body of the rule.

Goals are considered equal up to renaming of variables; if a goal already has a node in the graph we don't expand it further but rather create a cyclic edge to the variant subgoal. It is helpful to think of the rule/goal graph as a directed acyclic graph with some occasional backarcs. The "leafs" in the graph then corresponds to goals for built-in procedures such as "=", ">", *true*, or extensional database (EDB) predicates.

6.3 Deriving Mutual Exclusion

Two nodes in the rule/goal graph are said to be mutually exclusive if the relations they represent are disjoint. It is convenient to think of mutual exclusion not only between two rules, but also between a rule and a goal, or between two goals. Thus, mutual exclusion is a symmetric binary relation that can be represented by an edge between two nodes in the rule/goal graph.

In fig. 6.1 we see the rule/goal graph for a program and its goal, along with some mode information. Ultimately we are interested in deriving a mutual exclusion between all rules of *max* (fig. 6.1c) to show that it is determinate.

Underlying the mutual exclusion between two intermediate nodes is always some initial mutual exclusion between two built-in goals. When variables appear in the built-in goals it is essential to show that the variables in the second goal would have been bound in the same way as in the first goal, before we exclude the second goal.

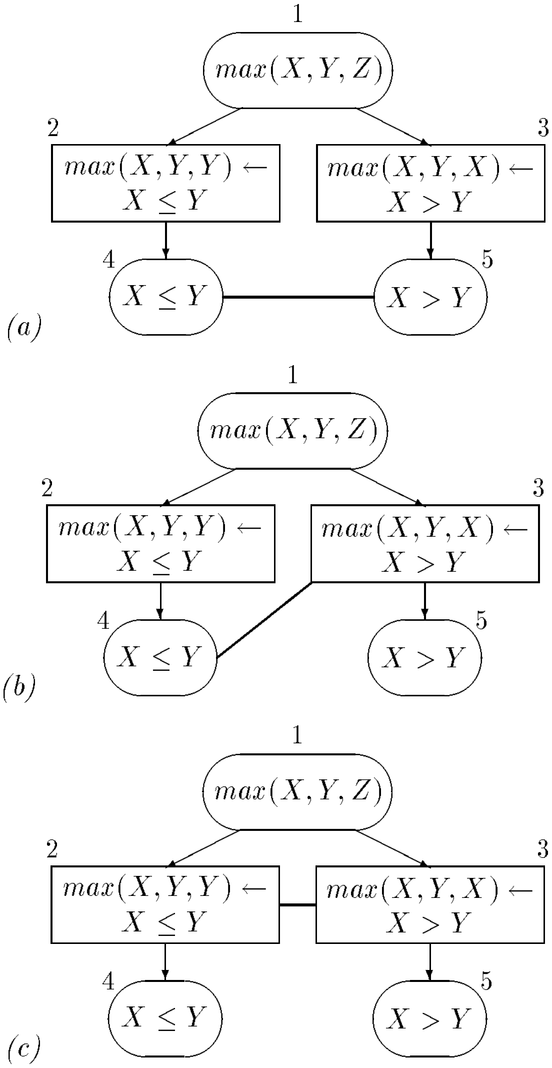

1
$max(X,Y,Z)$
2
$max(X,Y,Y) \leftarrow X \leq Y$
3
$max(X,Y,X) \leftarrow X > Y$
4
$X \leq Y$
5
$X > Y$
(a)
1
$max(X,Y,Z)$
2
$max(X,Y,Y) \leftarrow X \leq Y$
3
$max(X,Y,X) \leftarrow X > Y$
4
$X \leq Y$
5
$X > Y$
(b)
1
$max(X,Y,Z)$
2
$max(X,Y,Y) \leftarrow X \leq Y$
3
$max(X,Y,X) \leftarrow X > Y$
4
$X \leq Y$
5
$X > Y$
(c)

Figure 6.1: Propagation of mutual exclusion (denoted by thick edges) in the rule/goal graph for the program $\{max(X,Y,Y) \leftarrow X \leq Y, max(X,Y,X) \leftarrow X > Y\}$ and the goal "$\leftarrow max^{\mathbf{c,c,d}}(X,Y,Z)$."

In general we do not have to ensure that *all* variables in the built-in goals become bound; only certain variable positions are critical for establishing mutual exclusion. For instance, only X needs to be ground to make the subgoals $(X = [0|T])$ and $(X = [1|T])$ mutually exclusive. If we think of the nodes as representing relations, we can state what the critical positions are by projecting on the interesting columns. For instance, the requirement for the initial edge between node 4 and 5 in fig. 6.1a is $\pi_{1,2}(\leq_4) \| \pi_{1,2}(>_5)$. (Relations are subscripted according to the nodes they appear in.) This states that the first and second column in the respective relations are necessary to demonstrate the mutual exclusion. Another edge, not shown in fig. 6.1 to simplify the exposition, is from node 5 to itself, with the requirement $\pi_{1,2}(>_5)||\pi_{2,1}(>_5)$. This simply states that ">" is anti-symmetric.

We will now illustrate how the mutual exclusion between $\leq_4$ and max_3 can be derived, based on the existing mutual exclusion of $\leq_4$ and $>_5$. It is convenient to think of this step as trying to lift the edge in fig. 6.1a along its right side. The intuition behind this move is that a node n may be mutually exclusive with a *rule* node if n is mutually exclusive with *at least one* of the subgoals of the rule. (The other case, showing mutual exclusion between a node n and a *goal* node requires that n is mutually exclusive with *all rules* for the goal.)

We must now keep track of what happens to the variables X and Y as the goal $>_5$ is "sucked up" into its parent rule. Notice first that only the first and second argument positions of max_3 are specified as having ground mode. If the variables in the goal of max_3 are to be ground, they must pass through **c**-moded argument positions in the head. In this case, both variables do appear in **c**-moded positions of the head: $max_3(X, Y, X) \xmapsto{1}_{\mathbf{c}} X$ and $max_3(X, Y, X) \xmapsto{2}_{\mathbf{c}} Y$ both hold. The condition for mutual exclusion between relations $\leq_4$ and max_3 can be now be stated as follows

$$\pi_{1,2}(\leq_4)||\pi_{1,2}(max_3) \leftarrow \pi_{1,2}(\leq_4)||\pi_{1,2}(>_5).$$

Since the right hand side is already given, the conclusion follows immediately.

In the next step the algorithm would try to lift the right side of the edge in fig. 6.1b and show the mutual exclusion between $\leq_4$ and max_1. This fails however, since $\leq_4$ is not mutually exclusive with all the rules of max_1 (in particular max_2!). Not being able to make any more progress on the right side of the edge, it is time to start working on the left side. The rules max_2 and max_3 are candidates for mutual exclusion because node max_3 is already mutually exclusive with $\leq_4$, one of max_2's subgoals. The variables in the goal appear in **c**-moded positions in the head of the rule, thus

$$\pi_{1,2}(max_2)||\pi_{1,2}(max_3) \leftarrow \pi_{1,2}(\leq_4)||\pi_{1,2}(max_3).$$

Again, the conclusion follows immediately, and, as we shall see in the next section, that is always the case. To find out if *max* is determinate we could now ask the question: is there a fixed set of argument positions that differentiates all rules of *max*, that is, is there a Ψ such that $\pi_\Psi(max_2)||\pi_\Psi(max_3)$? The answer is yes, since $\Psi = \langle 1, 2\rangle$ has been shown to do so.

While we hope that this small example has illustrated the principles of our method, the scenario is generally more complicated because of the ways variables in the built-in goals are aliased as we work our way up the graph. In the next section we state the full algorithm and discuss its complexity.

6.3.1 Algorithm for Propagating Mutual Exclusion

A set NEEDS_PROCESSING is used to hold triples of the form $\langle x, y, u\rangle$, representing mutual exclusion between x and y. Since rules may have several goals serving as mutual exclusion witnesses, we maintain a third element u which is the node used in the process of adding the edge from x to y. Without it, some opportunities may be missed because an incorrect assumption was made about what the distinguishing goal was.

After the rule/goal graph has been built, "leaf" nodes are scanned and used to access a table with requirements for mutual exclusion between built-in goals. When applicable, these requirements are added as initial axioms to a database MUTEX_FACTS. For instance, one entry in the table might read "for two goals $(V < m)$ and $(W > n)$ (where m and n are integers) the axiom $\pi_1(V < m)||\pi_1(W > n)$ can be added if $m \leq n$."

Finally, for each pair (x, y) of leaf nodes that lead to an initial axiom, the triple $\langle x, y, \_\rangle$ is added to the set NEEDS_PROCESSING.

The main algorithm consists of picking an edge and trying to lift it up along one of its sides.

```
while NEEDS_PROCESSING $\neq \emptyset$ do
   let $\langle x, y, v \rangle$ be an element of NEEDS_PROCESSING;
   if not $\mathit{lift\_right}(x, y)$ then $\mathit{lift\_right}(y, x)$;
   NEEDS_PROCESSING := NEEDS_PROCESSING $- \{\langle x, y, v \rangle\}$;
end.
```

In the function *lift_right*(x, y), we will use the function *parents*(y) to mean the set of immediate predecessors of a node. (As the name implies, *parents*(y) does not return v if there is a backarc from v to y.) Since goal nodes can have more than one parent, lifting the right side of an edge can in general be done in more than one way, the effect being that the current edge of NEEDS_PROCESSING is replaced with zero or more new edges. If no mutually exclusive edges are derived between x and any parent of y, *lift_right* returns *false* to indicate that no more progress can be made on this side and that it is time to start lifting the other side.

```
function $\mathit{lift\_right}(x, y)$
   $\mathit{progress} := \mathit{false}$;
   for all $p \in \mathit{parents}(y)$
      if $\langle x, p, y \rangle \notin$ NEEDS_PROCESSING then
         $\mathit{progress} := (\mathit{progress}$ or $\mathit{try}(x, p, y))$;
   end;
   return $\mathit{progress}$;
end;
```

The function *try*(x, p, y) attempts to derive mutual exclusion between nodes x and p, based on the existing mutual exclusion of x and y, where p is a parent of y.

When p is a rule node, only one of its goals needs to be mutually exclusive with x. That goal is y. If the critical argument positions of y can be propagated to the head of p we can safely establish a mutual exclusion between x and p. We use the relation *prop*(Ψ_y, Ψ_p) for propagating the critical argument positions of y (denoted Ψ_y) to the corresponding positions in p (denoted Ψ_p). The definition of the *prop* relation, being rather technical, is given later (cf. definition 6.3.2).

The other case, when p is a goal node, requires that x is mutually exclusive with all rules $p_1, \ldots p_k$ of p. In the relational view, since p is the union of all the p_i relations, the same column numbers used to differentiate x and p must be used to differentiate x from all p_i's. Hence, a fixed set of argument positions Ψ_p is used for this purpose.

```
function try(x, p, y)
   if p is a rule node (with y as one of its goals) then
      FACTS := $\{\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p)$ such that $\pi_{\Psi_x}(x) \| \pi_{\Psi_y}(y)$ & $prop(\Psi_y, \Psi_p)\}$;
   else (p is a goal node with rules $p_1, \ldots, p_k$)
      FACTS := $\{\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p)$ such that $\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p_1)$ & $\cdots$ & $\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p_k)\}$;
   if |FACTS| > 0 then
      NEEDS_PROCESSING := NEEDS_PROCESSING + $\{\langle x, p, y\rangle\}$;
      add FACTS to MUTEX_FACTS;
      return true;
   else return false;
end;
```

When the algorithm has traversed the entire rule/goal graph, queries about mutual exclusion between nodes can be answered using MUTEX_FACTS. As an example, to find out if a goal g is determinate, that is, whether all its rules are pairwise mutually exclusive, we can use the following function.

```
function determinate(g)
   (g is a goal with rules $r_1, \ldots, r_k$)
   for all pairs $(r_i, r_j)$
      if there is no $\Psi_g$ such that $(\pi_{\Psi_g}(r_i) \| \pi_{\Psi_g}(r_j)) \in$ MUTEX_FACTS then
         return false;
   return true;
end;
```

6.3.2 Termination and Complexity

Two properties of the algorithm guarantee termination.

1. The current mutual-exclusion edge, represented by the triple $\langle x, y, v\rangle$, is always replaced by zero or more new edges.
2. New mutual-exclusion edges have at least one node closer to the root of the rule/goal graph, and such edges are not propagated across backarcs.

To derive an upper bound on the time complexity of the algorithm we observe that no triple $\langle x, y, v\rangle$ is processed more than once. With n nodes in the rule/goal graph there can be at most n^3 such triples. However, the time to process one triple depends on how many arguments appear in the relevant nodes, as mutual exclusion is checked for various permutations of those arguments. Assuming the number of arguments in any relation never exceeds some predefined constant, the time per triple can be considered $O(1)$. In conclusion, our algorithm is guaranteed to terminate in so-called semi-polynomial time ($O(n^3)$), *i.e.*, polynomial in the size of the rule/goal graph and exponential in the maximum arity of any relation in the program.

6.3.3 Correctness

As already pointed out, the mutual exclusion detection problem is, in general, unsolvable (cf. [HM89] for an example) and hence no complete algorithm exists. Soundness can be verified by making sure that the function *try* does not derive a mutual exclusion between two nodes when in fact they are not mutually exclusive. In the proof, we study *variable binding relations*. These are relations whose attributes corresponds to the variables appearing in a node, and whose values represent possible bindings for the variables. For a precise definition of variable bindings, see [Ull89, page 748].

Example 6.3.1: If node n contains the goal $p(f(X), g(h(Y), X))$, and is described by the relation

$$\{(f(a), g(h(c), a)), (f(f(a)), g(h(d), f(a)))\},$$

then the variable bindings are $V_n = \{(a, c), (f(a), d)\}$ with the scheme $\{X, Y\}$ ▮

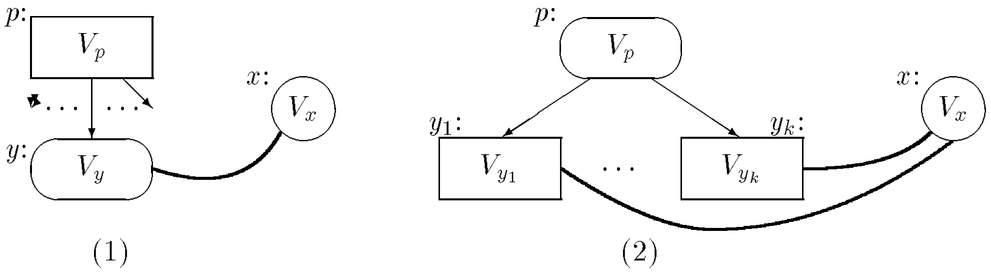


Figure 6.2: Proof scenario 1 and 2.

Definition 6.3.1: In a rule $p' \leftarrow p_1, \ldots, p_m$ the variable bindings V' for p' can be described in terms of a natural join between the variable bindings V_i for the subgoals in the body: $V' = V_1 \bowtie \cdots \bowtie V_m$ (cf. [Ull89, page 751]) ▮

Lemma 6.3.1: If $R = R_1 \bowtie \cdots \bowtie R_m$ then $\pi_{R_i}(R) \subseteq R_i$.

Proof: Assume by contradiction that there is a μ in R such that $\mu[R_i]$ is not in R_i. This implies that μ's attributes didn't agree with R_i. However, this can't be true for otherwise μ would not have been in R in the first place ▮

Theorem 6.3.2: MUTEX_FACTS is sound.

Proof: By induction on the height of the rule/goal-graph, modulo backarcs.

BASIS. The base case is two "leaf" nodes, x and y, representing built-in goals. In this case, it is assumed that the initial axioms describing the mutual exclusion between built-in goals are correct. These axioms are statements of the form $\pi_{\Psi_x}(V_x) || \pi_{\Psi_y}(V_y)$.

INDUCTION. Consider facts added by the function $try(x, p, y)$ where p is a parent of y. The rest of this proof proceeds in two separate cases, fig. 6.2, situations (1) and (2):

1. p is a rule node, with node y as one of its subgoals. The inductive hypothesis is that $\pi_{\Psi_x}(V_x) || \pi_{\Psi_y}(V_y)$, *i.e.*, $\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_y) = \emptyset$. By definition 6.3.1 we have $V_p = \cdots \bowtie V_y \bowtie \cdots$. From lemma 6.3.1 it follows that $\pi_{V_y}(V_p) \subseteq V_y$. Thus, with the

inductive hypothesis, $\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(\pi_{V_y}(V_p)) = \emptyset$. Since the variables in y must appear in the head of the rule we can simplify to $\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_p) = \emptyset$, *i.e.*, $\pi_{\Psi_x}(V_x) \| \pi_{\Psi_y}(V_p)$, which is what we wanted to show.

2. p is a goal node with rule nodes $y_1, \ldots, y_k$. Again, the inductive hypothesis is that $\pi_{\Psi_x}(V_x) \| \pi_{\Psi_y}(V_{y_i})$ is correct for $i = 1 \ldots k$. (Note that Ψ_y is applied to all rules.) As before, the inductive hypothesis may also be stated as $\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_{y_i}) = \emptyset$, $i = 1 \ldots k$. Hence, $\pi_{\Psi_x}(V_x) \cap (\bigcup_i \pi_{\Psi_y}(V_{y_i})) = \emptyset$. Since projection distributes over union, $\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(\bigcup_i V_{y_i}) = \emptyset$. It remains to show that $\pi_{\Psi_y}(\bigcup_i V_{y_i}) = \pi_{\Psi_p}(V_p)$. Intuitively, the left hand side of this conjecture represents a set of values found by reaching inside the V_{y_i}'s and the question is, can these values also be found inside V_p? Observe that, by construction, the rule heads in $y_1 \ldots y_k$ are various *instances* of the goal in node p. Therefore, the same value held by a variable Y in some rule head may also be found in some variable Z in the goal node (although the value might be more "embedded"), provided, of course, that Y is found in the same position in each and every rule head (hence Ψ_y). More formally, let θ_i be the unifier between the goal node and rule head i. If $Z/t_i \in \theta_i$ and $t_i \overset{\rho}{\longmapsto} Y$, $i = 1 \ldots k$, then $\pi_{\psi_y}(V_{y_i}) = \pi_{\rho+\psi_y}(Z)$ ▮

6.3.4 Description of prop

We now turn to the specification of *prop*, used by the function *try* in section 6.3.1. The problem we face is to keep track of certain variable positions for two disjoint built-in goals, and make sure that when they both can be invoked from two different rules, only one of them can succeed since the variables will be bound to the same value (whatever that value is). The relation *prop* is the sole mechanism used for describing how the path to these values change as we move from one node to another in the rule/goal-graph.

In the proof we have already outlined a way to keep track of values within variables. Variable bindings were introduced in the proof simply because it was easier to describe how variable values, rather than full relations, are propagated from subgoals to the head of the rule (definition 6.3.1).

To determine the position of a value inside a literal we only need to append the variable's position in the literal to the values position in the variable. The reader may verify that the following definition of *prop* captures the above ideas.

Definition 6.3.2: Let $H \leftarrow G_1, \ldots, G_k$ be a rule where a the position ψ of some G_i needs to be propagated to a corresponding position ψ' in H.

$$prop(\psi, \psi') \leftarrow \psi = \rho\tau,\ G_i \overset{\rho}{\longmapsto} X, var(X),\ H \overset{\rho'}{\longmapsto}_{\mathbf{c}} X,\ \psi' = \rho'\tau.$$

To simplify the notation we will "lift" *prop* to work on tuples of paths, $prop(\Psi, \Psi')$, with the obvious component-wise meaning $\blacksquare$

Example 6.3.2: Consider the rules

$$q(f(X, g(Y))) \leftarrow Y > 0.$$
$$p(U, V, Z) \leftarrow q(f(X, Z)).$$

The original critical position for "$Y > 0$" is "1" which corresponds to Y. When propagated to the head of q, Y's position becomes "1.2.1", hence $prop(1, 1.2.1)$ holds for q's rule. As we try to follow the path "1.2.1" in the subgoal of p (a more general atom than q's head) we run into Z after "1.2". Mapping "1.2" to the head of p gives us "3" (Z's position in the head) so the critical argument position is now "3" appended with the trailing "1". Hence $prop(1.2.1, 3.1)$ holds for p's rule $\blacksquare$

6.4 A Larger Example

The program in fig. 6.3 implements a parser for a small natural language. All queries are assumed to be ground. In the program, the rules for *np2* specify two ways to form a noun phrase. Being able to follow only one branch at a time, a top-down interpreter would pick the first rule and save backtracking information on the stack so that it can go back and try the second rule, in case the first one fails. Once the interpreter has managed to solve *adj* it is possible to discard the backtracking information just created, since *adj* and *noun* are mutually exclusive in their first argument. No system today, that we are aware of, recognizes this opportunity.

```
s(L) ← np(L, R), vp(R, _).
np(L, R) ← det(L, T), np2(T, R).
np(L, R) ← np2(L, R).
np2(L, R) ← adj(L, T), np2(T, R).
np2(L, R) ← noun(L, R).
vp(L, R) ← verb(L, R).
vp(L, R) ← verb(L, T), np(T, R).

det([the|R], R).
det([a|R], R).
adj([new|R], R).
noun([code|R], R).
noun([bugs|R], R).
verb([has|R], R).
```

Figure 6.3: Parsing program.

Since *np2* is recursive, a top-down interpreter that doesn't recognize the mutual exclusion would generate an amount of backtracking information that is proportional to the number of adjectives in the input string. This information stays around until the user is satisfied with the answer, and quits. If all solutions are asked for, the interpreter would start a series of meaningless attempts on the second rule for *np2*, which we know will fail.

We will now trace the execution of our algorithm for the parsing program, and see how the mutual exclusion between the two rules of *np2* is derived. The rule/goal graph for the interesting part of the program can be seen in fig. 6.4. Initially, all leaf nodes (16, 22, and 20) are mutually exclusive with each other, provided that the variable L is bound. Stated in terms of our axioms, we start with the initial MUTEX_FACTS = $\{\pi_1(16)||\pi_1(22), \pi_1(16)||\pi_1(20), \pi_1(22)||\pi_1(20)\}$ and NEEDS_PROCESSING set to $\{\langle 16, 22, \_\rangle, \langle 16, 20, \_\rangle, \langle 22, 20, \_\rangle\}$.

The table in fig. 6.5 shows the edges picked from the NEEDS_PROCESSING set, along with the new mutual exclusion facts added, and how they were derived. After the execution, we can query the function *determinate* with the goals $noun_{18}$ and $np2_{12}$. In both cases, the answer is *true*. For the entire program, the algorithm would also discover that *np* is

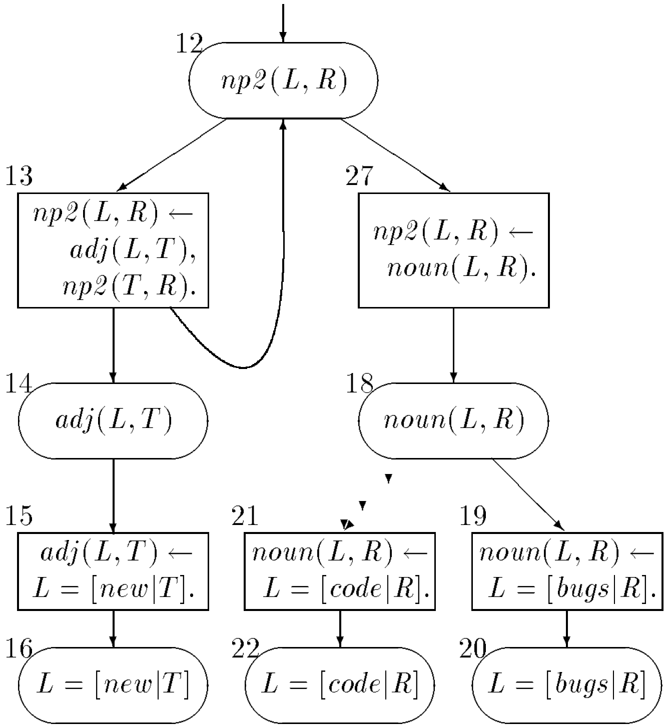

12
$np2(L, R)$
13
$np2(L, R) \leftarrow adj(L, T), np2(T, R).$
27
$np2(L, R) \leftarrow noun(L, R).$
14
$adj(L, T)$
18
$noun(L, R)$
15
$adj(L, T) \leftarrow L = [new|T].$
21
$noun(L, R) \leftarrow L = [code|R].$
19
$noun(L, R) \leftarrow L = [bugs|R].$
16
$L = [new|T]$
22
$L = [code|R]$
20
$L = [bugs|R]$

Figure 6.4: Part of the rule/goal graph for the parsing program.

determinate.

6.5 Limitations

At the moment, we are not even trying to derive mutual exclusion if a local variable of a test goal, which does not occur in a bound argument of the head of the rule, would be involved. The following program illustrates why this is unsound, in general.

$q(b)$. $q(c)$. $r(a, b)$. $s(a, c)$.

$p(X) \leftarrow q(Y), r(X, Y).$

$p(X) \leftarrow q(Y), s(X, Y).$

Assume that p is called with its argument ground. Even though the relations for r and s are disjoint, and are only called with ground arguments, it is not correct to say that the two

$\langle x, y, v\rangle$	New Facts	Derived From
$\langle 16, 22, \_\rangle$	$\pi_1(16) \| \pi_1(21)$	$\pi_1(16) \| \pi_1(22)$
$\langle 16, 20, \_\rangle$	$\pi_1(16) \| \pi_1(19)$	$\pi_1(16) \| \pi_1(20)$
$\langle 22, 20, \_\rangle$	$\pi_1(22) \| \pi_1(19)$	$\pi_1(22) \| \pi_1(20)$
$\langle 16, 21, 22\rangle$	$\pi_1(16) \| \pi_1(18)$	$\pi_1(16) \| \pi_1(21)$
$\langle 16, 19, 20\rangle$	$\pi_1(19) \| \pi_1(15)$	$\pi_1(19) \| \pi_1(16)$
$\langle 22, 19, 20\rangle$	$\pi_1(19) \| \pi_1(21)$	$\pi_1(19) \| \pi_1(22)$
$\langle 16, 18, 21\rangle$	$\pi_1(16) \| \pi_1(27)$	$\pi_1(16) \| \pi_1(18)$
$\langle 19, 15, 16\rangle$	$\pi_1(19) \| \pi_1(14)$	$\pi_1(19) \| \pi_1(15)$
$\langle 16, 27, 18\rangle$	$\pi_1(27) \| \pi_1(15)$	$\pi_1(27) \| \pi_1(16)$
$\langle 19, 14, 15\rangle$	$\pi_1(19) \| \pi_1(13)$	$\pi_1(19) \| \pi_1(14)$
$\langle 27, 15, 16\rangle$	$\pi_1(27) \| \pi_1(14)$	$\pi_1(27) \| \pi_1(15)$
$\langle 27, 14, 15\rangle$	$\pi_1(27) \| \pi_1(13)$	$\pi_1(27) \| \pi_1(14)$

Figure 6.5: Execution trace for the parsing program.

rules for p are mutually exclusive. For instance, in a top-down execution, the local variable Y may get rebound upon backtracking into q.

However, there are other situations where, by detection of functionality (cf. sect 6.1.1), one can safely conclude that a local variable cannot be bound to more than one value.

The rather coarse description of arguments into modes $\{\mathbf{c}, \mathbf{d}\}$ are for practical purposes too rigid to be useful. As an example, for the two goals $(X = [0|T])$ and $(X = [1|T])$ it is too restrictive to demand that X should be $\mathbf{c}$-moded considering the number of programs that deal with partially instantiated data structures.

6.6 Coding Style and the Use of Cuts

The lack of proper analysis and optimization tools in Prolog compilers have forced programmers to explicitly insert "cuts" in their clauses to prevent the search engine from saving backtracking information.

In this section we study some typical uses of cuts that has been gathered from textbooks and software libraries. Traditionally, one can identify three uses of cuts [CM81].

1. "If you get this far, you have picked the correct rule for this goal."
2. "If you get to here, you should stop trying to satisfy this goal."

3. "If you get to here, you have found the only solution to this problem, and there is no point in ever looking for alternatives."

The second use of the cut is often in combination with *fail* and is employed when the entire predicate needs to fail. The third use of the cut is used with the previously mentioned functional computations [DW89]; one will often find this type of cut at the very end of a clause. In this chapter, we are naturally interested with the first usage, where the cut is usually found at the beginning of the clause, possibly after some test goals.

Example 6.6.1: A very common task for Prolog predicates is to iterate over the elements in a list, as seen in the typical `append` predicate.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Because of a feature called *clause indexing* (which most Prolog systems implement, see for instance [CWA+91]) it is not necessary to insert a cut into the first clause. Clause indexing works by letting the interpreter examine the principal functor of the first argument to select the right clause. Clause indexing does not work for other argument positions, or when the first argument of two clauses do not unify, but their principal functor are the same ▮

Example 6.6.2: Another common idiom in Prolog is to iterate over the integers, such as in computing $n!$.

```
fact(0, 1) :- !.
fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.
```

Here, clause indexing is not enough to figure out that the two cases `N=0` and `N>0` are mutually exclusive. The cut symbol in the first clause is therefore common ▮

Example 6.6.3: In a few instances, user-defined predicates are used as tests in clauses. The following example, taken from a scanner, is an example of where our algorithm is indispensable to avoid the insertion of cuts.

```
tokenize([C|Cs], int(Num)) :-
    digit(C), !,
    Acc is C-48,
    tokenize_int(Cs, Acc, Num).


tokenize([C|Cs], id(Name)) :-
    letter(C), !,
    tokenize_id(Cs, [C], Name).
```

Other examples requiring our algorithm are databases, parsers, and the problem of the Dutch national flag in [O'K90], where colors can be red, white, or blue ▮

For the type of mutual exclusion detection discussed here, where the cut is placed early in the clause body, after some initial tests, the author has estimated that about 95% of all programs can be handled by previous algorithms [HM89, DW89, VR90]. For the remaining 5% it seems that our algorithm does the job — the limitations mentioned in section 6.5 is not an issue in practice.

Rather, the author has found another problem, namely that many programmers find it convenient to define two or more clauses with overlapping cases, so that the clauses provide the same answer when backtracking takes place.

Example 6.6.4: Consider the problem of finding the largest number in a list. Naively, one might define this predicate as follows.

```
    max([X], X).
    max([X|T], X) :- max(T, Y), X > Y.
    max([X|T], Y) :- max(T, Y), X =< Y.
```

The clauses are not pairwise mutually exclusive, since `[X]` unifies with `[X|T]`. Also, this predicate does not lend itself to tail recursion optimization. If the predicate had been written in the following way, we would have been able to discover that the clauses for $max/3$ are mutually exclusive. In addition, all recursive calls are now tail recursive.

```
    max([H|T], M) :- max(T, H, M).
    max([H|T], A, M) :- H > A, max(T, H, M).
    max([H|T], A, M) :- H =< A, max(T, A, M).
    max([], M, M).
```

Example 6.6.5: Another example of “bad” coding style is the translation of switch-statements from imperative programming languages.

```
eval(X+Y, Env, Val) :- !, ...
eval(X-Y, Env, Val) :- !, ...
eval(X*Y, Env, Val) :- !, ...
eval(X/Y, Env, Val) :- !, ...
eval(Var, Env, Val) :- lookup(Var, Val, Env).
```

With clause indexing, it is not possible to specify that the last pattern should be anything *except* a term whose principal functor is one of +, −, ∗, or /. Hence programmers tend to put cuts in all the preceding clauses. Rewriting the code to make it possible for our algorithm to discover the mutual exclusion is clumsy at best:

```
eval(X+Y, Env, Val) :- ...
eval(X-Y, Env, Val) :- ...
eval(X*Y, Env, Val) :- ...
eval(X/Y, Env, Val) :- ...
eval(Var, Env, Val) :-
    Var \== _+_, Var \== _-_, Var \== _*_, Var \== _/_,
    lookup(Var, Val, Env).
```

A better solution, in the author's opinion, is to change the representation so that variables are not represented as just Prolog variables.

```
eval(X+Y, Env, Val) :- ...
eval(X-Y, Env, Val) :- ...
eval(X*Y, Env, Val) :- ...
eval(X/Y, Env, Val) :- ...
eval(var(Var), Env, Val) :- lookup(Var, Val, Env).
```

Now, even clause indexing suffices to avoid the cuts, plus each clause can be understood in isolation.

7. Epilogue

In this thesis we have covered some analysis and transformation methods that have been developed with an eye towards the new high-level programming languages and tools, specifically in the logic programming field. Here we summarize the contributions, discuss some limitations, and give research ideas for future work.

7.1 Concluding Remarks

In chapter 3 we addressed some of the problems in parsing languages with dynamic operators, identified the shortcomings of the current parsing methods, and finally proposed a new parsing technique, deferred decision parsing, that postpones resolving *shift/reduce* conflicts involving operators to run time.

This technique has been built into an LR style parser-generator that produces deterministic, efficient, and table-driven parsers. Prototype parsers for Prolog (cf. chapter 4) and Standard ML have successfully been generated. Reasonably liberal operator overloading is supported.

We have also pointed out some of the drawbacks of using a top-down parser for traditional parsing tasks, such as in compiler construction, and argued that a bottom-up parser is a much better replacement.

More work needs to be done to categorize exactly what types of languages can be parsed with the deferred decision method. Another area that hasn't been addressed is error handling and recovery from errors.

In chapter 4 we covered the syntactical problems with Prolog, suggested some minor changes to make the language easier to parse and use, and also gave a grammar that we think is superior in maintenance, readability and size compared to previous methods.

In chapter 5 we presented two transformation methods for (strong) non-circular attribute grammars that allows them to be evaluated in the S-attributed environment of an LR-parser.

They represent a compromise between the one-pass and post-parse evaluation methods in that evaluation of certain "complicated" rules are sometimes postponed. Possible research directions include a similar setting for LL parsers, and perhaps a way to control the execution ordering of certain rules.

Finally, in chapter 6 a new conservative approximation technique for the undecidable problem of recognizing mutual exclusion among rules in logic programs was presented. The information is derived statically (at compile-time), and may aid in both time and space optimizations during execution. Additionally, the programmer is less dependent on non-logical language features, such as Prolog's "cut", thus creating more opportunities for parallel execution strategies.

7.2 Future Work

Here we mention some possible research directions that have arisen during the course of the author's work on this thesis.

Covered clauses A helpful diagnostic tool to detect the opposite of mutually exclusive rules, namely *covering clauses*, would reassure the programmer that he has handled all "input cases". For instance, given the procedure `merge`:

```
merge(Xs, [], Xs).
merge([], Ys, Ys).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X >= Y, merge([X|Xs], Ys, Zs).
```

we would like to be informed that we have handled all four combinations of the two input arguments being either the empty list or non-empty. Sometimes the programmer omits a rule on purpose, as in the following definition of `member(X, Xs)`, which is true when `X` is a member of the list `Xs`.

```
member(X, [X|_]).
member(X, [_|Xs]) :- member(X, Xs).
```

Obviously no X can be member of the empty list, but we think it would be better if the programmer explicitly said so with a fail clause:

```
member(X, []) :- fail.
member(X, [X|_]).
member(X, [_|Xs]) :- member(X, Xs).
```

Type information for arguments seems to be an essential key in detecting whether clauses in a rule covers all input, although it is not clear what the requirements for this type system should be.

Better Mode Information Many analysis programs rely on mode information, either supplied by the programmer or inferred by a mode analyzer. However, the mode systems currently in use are too crude. At best, an argument can be described as ground, partially instantiated, or as a variable. In many applications, this is not enough. For instance, in the mutual exclusion analyzer presented in chapter 6, it is too restrictive to demand that X should be **c**-moded for the two goals ($X = [0|T]$) and ($X = [1|T]$). What is needed is some sort of type system that describes how a data structure is (partially) instantiated.

Mutual Exclusion The algorithm in chapter 6 can be improved by analyzing propagation of bindings from head variables to local variables. As mentioned in section 6.1.1 and 6.5, constraint inference methods and detection of functionality can be applied.

In this thesis I have not been specific on how mutual exclusion information can be used to optimize programs. In [DW89] there is a section on choice points that should be studied by anyone who undertakes an implementation. Specifically, the *savecp/cutto*-primitives must be used rather than ordinary cuts to eliminate other clauses, for otherwise choice points created by subgoals leading up to the cut gets killed too.

Even though not all clauses in a procedure are pairwise mutually exclusive, it is sometimes possible to prune away *some* clauses with a given test. Whether this optimization has any value is an open question. Remember that a driving force behind our research was to guarantee proper tail recursion by not leaving choice points behind for eliminated clauses. If not all remaining clauses can be eliminated, it doesn't really matter whether there are two or fifty left to try — most Prolog interpreters create only one choice point anyway.

Further analysis and improvement of the mutual exclusion algorithm may lead to a description in terms of abstract interpretation [CC92, CC77, Jan90, BJCD87, Bru91]. Since our method mimics a bottom-up execution, the integration with [MS88] seems most probable.

References

[AJ74] A. V. Aho and S. C. Johnson. LR parsing. *Computing Surveys*, June 1974.

[AJU75] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. *Communications of the ACM*, 18(8):441–52, 1975.

[Apt90] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 495–574. MIT Press/Elsevier, 1990.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, ISBN 0-201-10088-6, 1985.

[AVW93] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[BBP$^+$82] D. L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, and D. H. D. Warren. *DECsystem-10 Prolog User's Manual*, 1982.

[BC93] F. Benhamou and A. Colmerauer. *Constraint Logic Programming — Selected Research.* MIT Press, 1993.

[Bee88] J. Beer. The Occur-Check Problem Revisited. *The Journal of Logic Programming*, 5(3):243–262, September 1988.

[BJCD87] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, August - September 1987. IEEE, Computer Society Press.

[BL89] M. E. Bermudez and G. Logothetis. Simple Computation of LALR(1) Lookahead Sets. *Information Processing Letters*, 31:233–238, 1989.

[Boc76] G. V. Bochmann. Semantic Evaluation from Left to Right. *Communications of the ACM*, 19(2):55–62, February 1976.

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *The Journal of Logic Programming*, 10(1,2,3 and 4):225–300, 1991.

[Bro74] B. M. Brosgol. *Deterministic Translation Grammars.* PhD thesis, Harvard University, Cambridge, Massachusetts, 1974. TR 3-74.

[Bru91] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10(1,2,3 and 4):91–124, 1991.

[BS89a] A. Brodsky and Y. Sagiv. Inference of Monotonicity Constraints in Datalog Programs. In *Eighth ACM Symposium on Principles of Database Systems*, pages 190–199, 1989.

[BS89b] A. Brodsky and Y. Sagiv. On Termination of Datalog Programs. In *First International Conference on Deductive and Object-Oriented Databases*, pages 95–112, Kyoto, Japan, 1989.

[BS91] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Tenth ACM Symposium on Principles of Database Systems*, 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[CFZ82] B. Courcelle and P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes (I and II). *Theoretical Computer Science*, 17(2 and 3):163–191 and 235–257, 1982.

[CH87] J. Cohen and T. J. Hickey. Parsing and Compiling Using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2), 1987.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

[CWA$^+$91] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, Oct 1991.

[CWY91] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *International Conference on Logic Programming*, pages 443–456. MIT Press, 1991.

[Deb89] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM TOPLAS*, 11(3):418–450, July 1989.

[DM93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[DW88] S. K. Debray and D. S. Warren. Automatic Mode Inference for Logic Programs. *The Journal of Logic Programming*, 5:207–229, 1988.

[DW89] S. K. Debray and D. S. Warren. Functional Computations in Logic Programs. *ACM TOPLAS*, 3(3):451–481, July 1989.

[Ear70] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2), 1970.

[FJ88] C. N. Fischer and R. J. LeBlanc Jr. *Crafting a Compiler.* Benjamin-Cummings Publishing Company, Inc, 1988.

[HB88] S. Haridi and P. Brand. Andorra Prolog – An integration of Prolog and Committed Choice Languages. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 745–754, 1988.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, Dec 1990.

[HM89] T. Hickey and S. Mudambi. Global Compilaton of Prolog. *The Journal of Logic Programming*, 7:193–230, 1989.

[Hor90] R. Nigel Horspool. Incremental Generation of LR Parsers. *Computer Languages*, 15(4):205–223, 1990.

[HTSW74] G. Holloway, J. Townley, J. Spitzen, and B. Wegbreit. *ECL Programmer's Manual*, 1974.

[HW90] P. Hudak and P. Wadler, editors. *Report on the Programming Language Haskell.* Yale University, 1990.

[Jan90] G. Janssens. *Deriving Run-Time Properties of Logic Programs by means of Abstract Interpretation.* PhD thesis, Dept of Computer Science, Katholieke Universiteit Leuven, Belgium, 1990.

[JM80] N. D. Jones and C. M. Madsen. Attribute-influenced LR parsing. In N. D. Jones, editor, *Semantics Directed Compiler Generation*, 94, pages 393–407. Springer-Verlag, 1980.

[Joh75] S. C. Johnsson. Yacc—Yet another compiler compiler. Technical Report CSTR 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Jon86] S. L. Peyton Jones. Parsing Distfix Operators. *Communications of the ACM*, 29(2), Feb 1986.

[JOR75] M. Jazayeri, W. F. Ogden, and W. C. Rounds. The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars. *Communications of the ACM*, 18:697–721, 1975.

[Ker89] J. Kerr. On LR Parsing of Languages with Dynamic Operators. Technical Report UCSC-CRL-89-13, UC Santa Cruz, 1989.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, 1990.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] D. E. Knuth. Semantics of Context-Free Languages; Correction. *Mathematical Systems Theory*, 3(1):95–96, 1971.

[Kow74] R. A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP'74*, pages 569–574, Amsterdam, 1974. North-Holland.

[Kow79] R. A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–431, 1979.

[KT91] M. Korsloot and E. Tick. Compilation Techniques for Nondeterministic Flat Concurrent Logic Programming Languages. In *International Conference on Logic Programming*, pages 457–471. MIT Press, 1991.

[Kun87] K. Kunen. Negation in Logic Programming. *The Journal of Logic Programming*, 4(4):289–308, December 1987.

[KW76] K. Kennedy and S. E. Warren. Automatic generation of efficient evaluators. In *Proc. 3rd ACM Conference on Principles of Programming Languages*, pages 32–49, Atlanta, Georgia, 1976.

[Las90] J.-L. Lassez. Querying constraints. In *Ninth ACM Symposium on Principles of Database Systems*, pages 288–298, 1990.

[LdR81] W. R. LaLonde and J. des Rivieres. Handling Operator Precedence in Arithmetic Expressions with Tree Transformations. *ACM TOPLAS*, 3(1), 1981.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LS91] J. W. Lloyd and J. C. Sheperdson. Partial Evaluation in Logic Programming. *The Journal of Logic Programming*, 11(3 & 4):217–242, October/November 1991.

[May81] B. H. Mayoh. Attribute Grammars and Mathematical Semantics. *SIAM J. Comput.*, 10(3):503–518, 1981.

[Mel81] C. S. Mellish. The Automatic Generation of Mode Declarations for Logic Programs. Technical Report DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[Mel85] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2(1):43–66, April 1985.

[Mel87] C. S. Mellish. Abstract Interpretation of Prolog Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horword, Chichester, U.K., 1987.

[MO81] T. Moto-Oka. Challenge for Knowledge Information Processing Systems (Preliminary Report on Fifth Generation Computer Systems). In *International Conference on Fifth Generation Computer Systems, Tokyo*, pages 1–85, 1981.

[MS88] K. Marriott and H. Søndergaard. Bottom-up Abstract Interpretation of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 733–748, Seattle, 1988. ALP, IEEE, The MIT Press.

[MSU86] F. Bancilhon D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.

[Nil86] U. Nilsson. AID: An Alternative Implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.

[odAMT91] R. op den Akker, B. Melichar, and J. Tarhio. Attribute evaluation and parsing. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 187–214, Prague, Czechoslovakia, June 1991. Springer-Verlag.

[O'K84] R. O'Keefe. Draft Proposed Standard for Prolog Evaluable Predicates. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1984.

[O'K90] R. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[Ped91] K. R. Apt D. Pedreschi. Proving termination of general Prolog programs. In *Proceedings of International Conference on Theoretical Aspects of Computer Science*, Sendai, Japan, 1991.

[Pfe92] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[Pla84] D. A. Plaisted. The Occur-Check Problem in Prolog. In *Proc. International Symposium on Logic Programming*, pages 272–280, Atlantic City, 1984. IEEE, Computer Society Press.

[Plü90] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1990.

[PN91] D. Palmer and L. Naish. NUA-Prolog: An Extension to the WAM for Parallel Andorra. In *International Conference on Logic Programming*, pages 429–442. MIT Press, 1991.

[Pos94] K. Post. Mutually Exclusive Rules in Logic Programming. In *Logic Programming — Proceedings of the 1994 International Symposium*. MIT Press, 1994. To appear.

[PVGK93] K. Post, A. Van Gelder, and J. Kerr. Deterministic Parsing of Languages with Dynamic Operators. In D. Miller, editor, *Logic Programming — Proceedings of the 1993 International Symposium*, pages 456–472. MIT Press, 1993.

[Ram91] R. Ramakrishnan. Magic Templates: A Spellbinding Approach To Logic Programs. *The Journal of Logic Programming*, 11(3 & 4):189–216, October/November 1991.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *The Journal of the ACM*, 12(1):23–41, 1965.

[Sah93] D. Sahlin. MIXTUS: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[Sco90] R. S. Scowen. Prolog – Budapest papers – 2 – Input/Output, Arithmetic, Modules, etc. Technical Report ISO/IEC JTC1 SC22 WG17 N69, International Organization for Standardization, 1990.

[Sco92] R. S. Scowen. Draft Prolog Standard. Technical Report ISO/IEC JTC1 SC22 WG17 N92, International Organization for Standardization, 1992.

[Soh93] K. Sohn. *Automated Termination Analysis for Logic Programs.* PhD thesis, UC Santa Cruz, 1993.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[ST85] H. Sawamura and T. Takeshima. Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and their Applications to Prolog Optimization. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 200–207, Boston, Massachusetts, 1985. IEEE, Washington D.C.

[SVG91] K. Sohn and A. Van Gelder. Termination detection in logic program using argument sizes. In *Tenth ACM Symposium on Principles of Database Systems*, pages 216–226, 1991.

[Tic91] E. Tick. *Parallel Logic Programming.* MIT Press, 1991.

[Tom86] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, Massachusetts, 1986.

[Udd88] G. Uddeborg. A Functional Parser Generator. Technical Report 43, Dept. of Computer Sciences, Chalmers University of Technology, Göteborg, 1988.

[Ull89] J. D. Ullman. *Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[UVG88] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[VG91] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 1(3):361–392, 1991.

[VR90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, UC Berkeley, 1990.

[vW76] A. van Wijngaarden, editor. *Revised Report on the Algorithmic Language Algol 68.* Springer-Verlag, 1976.

[War77] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report DAI Research Paper 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1977.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report Tech. Note 309, SRI International, Menlo Park, CA, 1983.

[War86] D. H. D. Warren. Optimizing Tail Recursion in Prolog. In *Logic Programming and its Applications*, pages 77–90. Ablex Publishing, N.J., 1986.

[Wat77] D. A. Watt. The Parsing Problem for Affix Grammars. *Acta Informatica*, 8:1–20, 1977.

Index

$A(p)$: 61
Abstract interpretation: 7, 97
accept action: 14
Action
    accept: 14
    error: 14
    reduce: 14, 35
    Semantic: 15
    shift: 14, 35
Adornments: 22
a_k: 61
Algol-68: 28–29
Ambiguity in operator expressions: 48
Ambiguous grammar: 13–14, 41
Analysis: 5
Andorra: 75
Answer substitution: 19
    Computation of: 20
a_{pk}: 61
Argument path: 77
Artificial Intelligence: 1
Associativity: 27
Atom: 17, 48
Attribute
    Inherited: 44
    Synthesized: 44
Attribute grammar: 44, 57, 61
    Non-circular: 64
    Strongly non-circular: 65
Attributes: 61
    Of a production: 61
Automated Theorem-Proving: 1

b (bound mode): 22
Backarc: 78, 82
Backtracking: 6, 20, 44, 74, 87
Binary numbers: 12
Binding: 18
$body(a)$: 68
Bottom-up execution strategy: 74
Bottom-up parsers: 13, 44
□ (empty goal): 20
Bracket-fix operators: 29

C: 28
c (constant mode): 22, 71, 73, 77
$C(p)$: 67
Circular definitions: 65, 71
Clause: 16
Clause indexing: 91
Colmerauer, A.: 2
“`,`”: 50, 54
Comma operator: 50, 54
`comma2list`: 54
Composite graph: 63
Composition: 19
Compound term: 16, 48
Conflict: 32
    reduce/reduce: 14
    shift/reduce: 14, 34
Constant: 16
Constraint inference: 73, 76
Context-free grammar: 11
Copy symbols: 58
Coroutine: 70
Covering clauses: 95
Critical positions: 79
Cut: 6, 21, 90
    Green: 21, 74
    Red: 21, 74
cutto: 96
C++: 8

d (don't-know mode): 22, 73
$D(p)[\ldots]$: 63
$D(T)$: 62
DCG: 5, 9, 22, 32
DDGEN: 30, 70
Debray, S.: 74–76
Declarative style: 1
Deferred decision parsing: 30, 32
Definite Clause Grammar: 5, 9, 22, 32, 43, 70
δ: 35
Dependency graph: 62
Derivation
    Leftmost: 12

Reverse rightmost: 13
Rightmost: 12
SLD: 19
Derives: 12
$\Rightarrow$: 12
$\overset{*}{\Rightarrow}$: 12
Determinacy: 76
Determinate: 83
determinate(g): 83
Dewey notation: 77
Difference lists: 9
Disjoint: 73, 77
Distfix: 29
"." (dot): 17
Dutch national flag: 92
Dynamic operators: 26, 32

Earley's algorithm: 28
EDB: 78
Edinburgh syntax: 29, 49
EL1: 28–29
Empty goal: 20
Empty list: 17
Empty string: 11
`eof`: 15
ϵ (identity substitution): 18
ε (empty string): 11
Equivalence classes: 76
error action: 14
Evaluation problem: 58
Evaluation rules: 61
Evaluation, postponed: 66
Evaluator
On-the-fly: 66
One-pass: 58
Post-parsing: 58
Expression: 17, 27
Extensional database: 78

f (free mode): 22
Fact: 16
Failed SLD-derivation: 20
FGCS: 1
Fixity: 27, 48
Formal Language Theory: 1
FPG: 59
Function symbol: 17
Functional computations: 6
Functionality: 74, 76, 90
Functor: 17

Goal
Empty: 20
Evaluation: 61
Logic programming: 18
Goal node: 78
Goedel: 8
Grammar
Non-circular: 60
Strongly non-circular: 60
Green cut: 21, 74
Ground: 19

Haskell: 4, 26
hd: 77
Head
Of a clause: 16
Of a list: 17
Of a sequence: 77
Hickey, T.J: 75
Horn clauses: 16

Identity substitution: 18
Imperative programming languages: 1
$\leftarrow$: 16
`:-`: 16
Implicit overloading: 35
Incremental parser generators: 28
Induced grammar: 38
Infix: 27
Inherited attribute: 10, 44, 61
Instance, of an expression: 19

Kerr, J.: 30
Kowalski, R.: 2

L-attributed: 65
L-attributed definitions: 44
L-attributed grammars: 58
Language: 12
Lazy evaluation: 59
Lazy ML: 59

Left-recursive production: 43
Leftmost derivation: 12
Length: 11
Lexical categories in Prolog: 48
lift_right(x, y): 82
LISP: 25
List: 17
  Empty: 17
Literal: 16
  Selected: 20
Local operators: 37
Local variable: 89
Logic program
  Procedural semantics: 19
  Syntax: 16
Logic programming: 16
Look-ahead: 51
Look-ahead token: 34
LR parser: 13–14
  Execution trace: 15
LR(1): 28

Magic set: 8
Mellish, C.S: 76
MIXTUS: 9
ML: 4–5, 26, 37
Mode: 5, 22, 72, 90, 96
Most general unifier: 19
$\mu[S]$: 77
Mudambi, S.: 75
MUTEX_FACTS: 81
Mutual exclusion: 7, 10, 73, 78, 97
  Primitive: 76

NEEDS_PROCESSING: 81
Negation: 9
Nilsson, U.: 44
Non-circular attribute grammar: 60, 64
Nonterminals: 11
nonvar(X): 71
Normal form, of evaluation rule: 61
Nullary operator: 27, 35
Number: 48

Occur check: 6
On-the-fly evaluation: 66
One-pass evaluators: 58
`op/3`: 48
Operator: 4, 27, 48
  Dynamic: 26
  Local: 37
  univ: 54
Operator module: 31
Operator precedence parsing: 28
Operator table: 33
Optimization: 8
Or-parallelism: 75
Overloaded operator: 27, 31
Overloading: 26, 35
Overloading policy: 41

p/n: 16
Parallelization: 7
parents(y): 82
Parse table: 14
Parse tree: 13, 62
parse_action: 34, 45
Parser: 13
  Bottom-up: 44
  Top-down: 43
Parser generator: 4, 14, 27
Parsing: 3
Partial evaluation: 8
Pascal: 6
Peyton Jones, S.L.: 29
π (projection): 77
$\pi_R(S)$: 77
Position, in a term: 77
$\overset{\psi}{\longmapsto}$: 77
$\overset{\psi}{\longmapsto}_{\mathbf{c}}$: 77
Post-parsing evaluators: 58
Postfix: 27
Postponed evaluation: 66
Precedence: 27
Predecessor: 62
$\mapsto$: 62
Predicate symbol: 16
Predictive: 44
Prefix: 27

Prefix operator: 50
Primitive mutual exclusion: 76
Procedural semantics: 19
Procedure: 16
Productions: 11
Projection: 77
Prolog: 25–26, 29, 35, 43, 47
Prolog grammar: 55
Prolog standardization committee: 31
Prolog term: 53
prop(ψ, ψ'): 86
ψ (argument path): 77
Punctuation symbols: 48

Query: 18

RATFOR: 8
`rdtok.pl`: 53
`read.pl`: 47
Reader: 48
Ready(p): 66
Red cut: 21, 74
reduce: 35
reduce action: 14
reduce/reduce conflict: 15
Referenced variables: 71
Refutation: 20
resolve: 34, 45
Reverse rightmost derivation: 13
Right-recursive production: 44
$\rightarrow$: 11
Rightmost derivation: 12
Rule: 16
Rule node: 78
Rule/goal graph: 78

S-attributed: 65
S-attributed definitions: 44
S-attributed grammar: 58
savecp: 96
Sawamura, H.: 76
Scope: 27
Selected literal: 20
Semantic action: 15
Sentence: 12, 53
Sentential form: 12
Sequence
    ψ: 77
    Head of: 77
shift: 35
shift action: 14
shift-reduce parser: 13
shift/reduce conflict: 15, 34
SICSTUS Prolog: 70
SLD-derivation: 19
    Failed: 20
SLD-tree: 20, 24
Standard syntax: 48
Start symbol: 11
Strings: 11
Strong composite graph: 64
Strong non-circular grammars: 60
Strongly non-circular attribute grammar: 65
Subgoals: 16
Substitution: 18
    Applying: 19
    Identity: 18
    More general: 19
Switch-statements: 93
Synthesized attribute: 44, 61
Synthesized functions: 67, 70

Tail
    Of a list: 17
Tail recursion optimization: 3, 74
Takeshima, T.: 76
Term: 16
    Position in: 77
Terminals: 11
Termination analysis: 6
θ (substitution): 18
Token: 11
`%token`: 45
Tomita's algorithm: 28
Top-down execution strategy: 73
Top-down parser: 43
Transformations: 8
Translation process: 3–4
try(x, p, y): 83
Type: 49

Type checking: 5
Type derivation: 5
Type system: 96

Undecidable program properties: 5, 76
Unfolding: 76
Unifiable: 19
Unifier: 19
    Most general: 19
`=..` (univ): 54
$Use(X, a)$: 66

Van Roy, P.: 75
Variable: 16, 48
Variable binding relations: 84
Vocabulary : 11

WAM: 2
Warren, D.H.D: 2
Warren, D.S: 75
Warren, D.S.: 74, 76
$when(Cond, Goal)$: 70
Whitespace: 49

`y.output`: 45
Yacc: 30, 32, 45, 66, 70

$[X|T]$: 8
[] (empty list): 17

`\+`: 51
|| (disjoint): 77

“|”: 11