Deterministic Parsing of Languages with Dynamic Operators*

Kjell Post Allen Van Gelder James Kerr
Baskin Computer Science Center
University of California
Santa Cruz, CA 95064
email: `kjell@cs.ucsc.edu`

UCSC-CRL-93-15

July 27, 1993

Abstract

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. 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 approximately linear time. Conventional implementations based on backtracking parsers can require exponential time.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages; extensible languages*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators; parsing*

General Terms: Grammars, Parsers, Operators

Additional Key Words and Phrases: Parser Generators, Ambiguous Grammars, LR Grammars, Attribute Grammars, Context-Free Languages, Logic Programming

*Originally appeared in "Logic Programming — Proceedings of the 1993 International Symposium (ILPS93)"

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 [21] and Haskell [14], 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 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 paper 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 this paper's contributions. Later sections discuss several of the issues in detail.

1.1 Definitions

We briefly review some standard definitions concerning operators, and specify particular terminology used in the paper. 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. This report uses 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.

[1] Some earlier languages permitted very limited definitions of new operators; see section 1.2

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.

We assume in the rest of this paper some acquaintance with Prolog [5, 6]. A basic knowledge of LR parsing [1, 3] and the parser generator YACC [15] is also helpful.

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 [1, 3, 4, 9, 15]. 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 [8] 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 [27] 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 [11, 13] 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 [18] 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 [28] and EL1 [12], 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 [12]. 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.

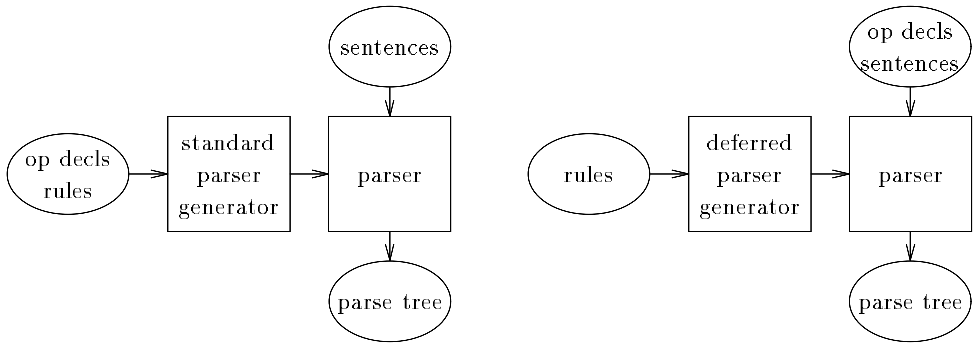

Figure 1: Standard Parser Generator and Deferred Decision Parser Generator.

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 [12]. More recently, Peyton Jones [16] describes a technique for parsing programs that involves user-defined *distfix* operators (*e.g.* `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 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 program code, `read.pl`.

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 [17]. 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 [15]. 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 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 4.

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 [5] 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 [25],

> "These possibilities provide multiple opportunities for ambiguity and the draft standard has therefore defined restrictions on the syntax so that a) expressions can still be parsed from left to right without needing significant backtracking or look-ahead..."

A preliminary report on this work [24] 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 [26].

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 [5]. 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. The thrust of our research has been to correctly handle inherited attributes. This work is discussed in Section 5.

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.

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 (e.g. fig. 2). 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.

	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 2: An Example Run-Time Operator Table.

```
term(T) ::= atom(T).
term(T) ::= var(T).
term(T) ::= '(', term(T), ')'.
term(T) ::= op(Name), term(T1),            { T =.. [Name,T1] }.     % prefix
term(T) ::= term(T1), op(Name), term(T2), { T =.. [Name,T1,T2] }.  % infix
term(T) ::= term(T1), op(Name),            { T =.. [Name,T1] }.     % postfix
term(T) ::= op(T).                                                  % nullary
```

Figure 3: Subset Grammar for Prolog Terms.

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), 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 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 $parse\_action(S,X)$, 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 $parse\_table(S,\ X) = resolve(S',\ A \rightarrow \alpha\ op_A\ \beta)$
> then return $do\_resolve(A \rightarrow \alpha\ op_A\ \beta,\ X)$
> else return $parse\_table(S,\ X)$

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 [2, 3], with extensions to cover cases that cannot be declared in YACC (*cf.* section 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 $\alpha\ op_A\ \beta$ 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 $A \rightarrow \alpha\ op_A\ \beta$, 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 "delta". This treatment guarantees a deterministic grammar if there is no declared overloading (*cf.* section 4) and retains the flexibility of allowing operators to appear as terms.

The overloading of the operators op_A and 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 $\alpha\ op_A\ \beta$	*Possible fixity combinations* (op_A, op_B)
$\alpha \neq \epsilon, \beta \neq \epsilon$	(infix, infix), (infix, postfix)
$\alpha \neq \epsilon, \beta = \epsilon$	(infix, prefix), (infix, null), (postfix, infix), (postfix, postfix)
$\alpha = \epsilon, \beta \neq \epsilon$	(prefix, infix), (prefix, postfix)
$\alpha = \epsilon, \beta = \epsilon$	(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 op_A and op_B have equal scope, then
   (a) If op_A is right-associative, shift[2].
   (b) If op_B is left-associative, reduce.
2. If op_A is either infix or prefix with wider scope than op_B, shift.
3. If op_B is either infix or postfix with wider scope than op_A, reduce.

Example 2.1: We examine the deferred decision parser as it reads `-X+Y*Z!`, using the operators in fig 2. Figure 8 shows the steps involved. At step 4, state 5 contains the shift/reduce conflict $\{term \rightarrow term\bullet$ `op(+)` $term, term \rightarrow$ `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 $\beta = term \neq \epsilon$ we satisfy the requirements for the third rule above and reduce.

[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[26].

At step 8, state 7 contains the shift/reduce conflict $\{term \rightarrow term\bullet$ `op(*)` $term, term \rightarrow term$ `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 $\{term \rightarrow term\bullet$ `op(!)`, $term \rightarrow term$ `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. $\square$

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.

Standard ML is a language with infix operators only, but these can be declared locally in blocks, with accompanying scope rules [21]. 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 the grammar subset in figure 4.

The line `dynop_token(atom(Name), op(Name))` 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)`.

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 reinstantiated 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 reinstantiate old operators.

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 in fig. 5 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.

```
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 4: Grammar Subset for ML Operator Expressions.

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 $i \in 1 \dots k$ 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):

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

   (b) Build a precedence grammar consisting of $k+1$ layers (fig.6). This will serve as a skeleton for the induced grammar. The ith layer ($0 < i \leq k$) holds productions for nonterminal A_i which corresponds to operators with precedence p_i. The 0th layer defines the sentential forms with 0 precedence, provided that there are any in the dynamic operator grammar.

   (c) For $i \in 1 \dots k$:
   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 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.

Figure 5: Algorithm for Induced Grammar.

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

Figure 6: Skeleton for induced grammar. (The production $A_1^N \rightarrow A_0$ is included only if $\Pi_{rem} \neq \emptyset$.)

Example 4.1: Consider the Prolog term grammar and the operator table in fig. 2. Sorting the table gives us three precedence levels (fig. 7). The induced grammar is shown below ($term_i^N$ omitted for brevity). The induced grammar is deterministic (as verified by YACC).

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

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.

The recent ISO draft proposes a different set of restrictions to avoid ambiguities [26]. Probably neither solution is the final word. We hope our parser generator can be a useful tool to explore design alternatives.

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 [7]. However, there are several reasons why a top-down backtracking parser is unsuitable for recognizing sentences such as programming language constructs.

Slot	*Prec*	(*o*, *Fix*, *Assoc*)
1	300	($-$, `pre`, `right`), ($+$, `pre`, `right`), (!, `post`, `left`)
2	400	($*$, `in`, `left`), ($/$, `in`, `left`)
3	500	($-$, `in`, `left`), ($+$, `in`, `left`)

Figure 7: Sorted Operator Table.

- 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 [3]. Also, there is no clearcut way to transform the semantic actions correctly.
- Unless the parser is *predictive* [3] 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, e.g. 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 [22] 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 (DCGs) as attribute grammars, which have been studied extensively in connection with deterministic translation schemes [3]. 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 [3]. Also, it is possible to associate a nonterminal with a function from inherited to synthesized attributes [20].

In the following example we demonstrate how inherited attributes in a bottom-up parser can be implemented in systems with coroutining facilities.

Example 5.1: The following non-L-attributed grammar [3] recognizes C declarations of the form `int a, b[2][5]`.

```
decl    ::= type(T), list(T).
type(T) ::= int, { T = integer }.
type(T) ::= float, { T = float }.
list(T) ::= list(T), ',', name(T).
list(T) ::= name(T).
name(T) ::= name(AT), '[', int(N), ']', { AT = array(N,T) }.
name(T) ::= ident(N), { install(N,T) }.
```

Ignoring for the moment the problem with left-recursion in the grammar, we notice that a top-down parser would in the first production have the type information `T` available *before* it tried to recognize `list`. In a bottom-up parser, however, `T` will not be available until the entire right-hand side has been reduced. As a result of this, the bottom-up parser will not have the type available as it tries to install the identifiers in the symbol table. The grammar is non-L-attributed because attribute `N` is propagated right-to-left in the fifth production, making it difficult even for a top-down parser to compute all the attributes in one pass.

If the host language allows procedures (goals, in Prolog) to be blocked until certain conditions are true it is a straightforward task to postpone the execution of a semantic action until its attribute values have been bound. Using the coroutining predicate `when(`*Condition*`,`*Goal*`)`[3] in SICStus Prolog [19] we can rewrite the above specification as follows.

```
decl    ::= type(T), list(T).
type(T) ::= int, { T = integer }.
type(T) ::= float, { T = float }.
list(T) ::= list(T), ',', name(T).
list(T) ::= name(T).
name(T) ::= name(AT), '[', int(N), ']', { AT = array(N,T) }.
name(T) ::= ident(N), { when((ground(N),ground(T)), install(N,T)) }.
```

An algorithm to do such a transformation automatically would depend on supplied mode declarations, *e.g.* `:- mode install(++,++)`. Procedure calls within actions must then be blocked until `++`-arguments are ground and `+`-arguments are bound.

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 itemsets (*e.g.* `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*.

The source code and manual for the parser generator is available from the authors — send mail to `kjell@cs.ucsc.edu` and request a copy.

7 Concluding Remarks

We have 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.

The technique has been built into an LR style parser-generator that produces deterministic, efficient, and table-driven parsers. Prototype parsers for Prolog [23] 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. We are also interested in handling inherited attributes in the bottom-up evaluation of attribute grammars.

[3] The suggestion of using `freeze`, of which `when` is a variant, was pointed out by one of the ILPS reviewers. Other versions of Prolog have similar features. See also [10] for a similar technique based on lazy evaluation in functional programming.

8 Acknowledgements

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.

The authors are grateful to Richard O'Keefe and Roger Scowen for valuable input on the syntax of Prolog, and Manuel Bermudez for helpful discussions on parser generation. We are also indebted to Lawrence Byrd for suggesting that LR parsing technology be applied to Definite Clause Grammars. Michel Mauny and Pierre Weiss of INRIA supplied production rules for CAML LIGHT. Finally we wish to thank the reviewers on the ILPS'93 committee for their input.

References

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

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

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

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

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

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

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

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

[9] Charles N. Fischer and Richard J. LeBlanc Jr. *Crafting a Compiler.* Benjamin/Cummings Publishing Company, Inc, 1988.

[10] Göran Uddeborg. A Functional Parser Generator. Technical Report 43, Dept. of Computer Sciences, Chalmers University of Technology, Gothenburg, 1988.

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

[12] Glenn Holloway, Judy Townley, Jay Spitzen, and Ben Wegbreit. *ECL Programmer's Manual*, 1974.

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

[14] Paul Hudak and Philip Wadler, editors. *Report on the Programming Language Haskell.* Yale University, 1990.

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

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

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

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

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

[20] Brian H. Mayoh. Attribute Grammars and Mathematical Semantics. *SIAM J. Comput.*, 10(3), 1981.

[21] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, 1990.

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

[23] Kjell Post and Allen Van Gelder. Parsing Prolog. Technical Report UCSC-CRL-93-22, University of California, Santa Cruz, 1993.

[24] Kjell Post, Allen Van Gelder, and James Kerr. Deterministic Parsing of Languages with Dynamic Operators. Technical Report UCSC-CRL-91-31, University of California, Santa Cruz, 1991.

[25] 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.

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

[27] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, 1986.

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

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

Figure 8: Deferred Decision Parsing Example.