A SIMPLE AND PRACTICAL IMPLEMENTATION OF PREDICATES IN CONTEXT-FREE PARSERS

C. N. Fischer
M. Ganapathi
R. J. LeBlanc

Computer Sciences Technical Report #493

April 1983

A Simple and Practical Implementation of Predicates in Context-Free Parsers[1]

C.N. Fischer,[2]

M. Ganapathi,[3]

R.J. LeBlanc[4]

Abstract

It has long been recognized that context-free grammars are inadequate to represent the complete syntax of modern programming languages. A way of extending the expressive power of context-free grammars is to add predicates to productions. Production application (or recognition) is blocked if a predicate evaluates to false. Predicates also can be used to disambiguate parsing decisions, allowing broader classes of languages to be parsed. Unfortunately, standard context-free parser generators do not directly allow the use of predicates. Simple and efficient means of utilizing predicates within the framework of existing parser generators are considered. Experience using these techniques is discussed.

---

[1]Research supported in part by NSF Grant MCS 78-02570

[2]Computer Sciences Department, University of Wisconsin - Madison.

[3]Computer Systems Laboratory, Stanford University.

[4]Department of Information and Computer Science, Georgia Institute of Technology

Introduction

Context-free grammars are almost universally used to represent the syntax of modern programming languages. These grammars represent a very simple and readable formalism that can be automatically processed (by *parser generators* [1], [2]) to produce tables for a parsing module. This use renders the construction of the syntax analysis component of a compiler a routine, and indeed almost trivial, task. Context-free parsing techniques have been adapted to other aspects of compilation, most notably automatic code synthesis [3], [4].

The use of context-free grammars is not without its drawbacks. Syntactic representation is incomplete, and limited to *context-free syntax*. Thus, it is possible to include in a context-free grammar for Pascal [5] rules that prohibit, e.g.,

```
1 := x;
```

but it is not possible to prohibit[5]

```
x := True + False;
```

Non-context-free aspects of program syntax are typically handled by *semantic routines* that are invoked by a parser as productions are recognized. Semantic routines verify that scoping and type compatibility rules (which can't be represented by context-free grammars) are obeyed and issue "semantic errors" if they are violated.

---

[5]True and false are normally treated as predefined, but not reserved, identifiers. Hence they are indistinguishable from other named constants.

Another problem that arises in utilizing context-free grammars is that not all grammar forms are easily parsable. In practice, therefore, only certain classes of grammars (most commonly LL(1) ([6], sect. 5.5) and LALR(1) ([6], sect. 6.5)) are used. Sometimes, certain language constructs can't be represented by a given grammar class. (Pascal's "dangling else" can't be generated by an LL(1) grammar [7]). In other cases, the size of the grammar needed to represent a construct is prohibitive. For example, the size of a grammar that generates a list of N options, in any order, with no option repeated, is exponential in N.

A valuable adjunct to standard context-free grammars is the addition of *contextual predicates* [8] to productions. If a contextual predicate appears in a right hand side, then the predicate must evaluate to true before the production can be used in a derivation or recognized in a parse. Contextual predicates allow productions to be *blocked* when necessary, thus controlling when a production may be applied. For example, expressions involving the "+" operator might be generated by

Expr → Expr + Term

where Expr and Term are non-terminals generating various subexpressions. Using contextual predicates[6] this production becomes

Expr → Expr #TestArithmetic + Term #TestArithmetic

where #TestArithmetic represents a contextual predicate that tests whether the operand preceding it is arithmetic (integer or real). If it

---

[6]Throughout this paper, all predicate symbols will be prefixed by a "#".

isn't, the production can't be applied (or recognized). Thus, expressions such as "True + False" can be recognized as *syntactically* invalid.

Sometimes predicates can be used to aid in making parsing decisions. This aid allows us to use grammars that otherwise couldn't be parsed. Such predicates are often termed *disambiguating predicates* [9] because they resolve parsing decisions that otherwise would be ambiguous.

As an example, consider the following (ambiguous) grammar fragment that generates Pascal's "dangling else" construct:

```
Stmt     → IF Expr THEN Stmt ElsePart
ElsePart → ELSE Stmt
ElsePart → ε
```

(ε represents the null string). This fragment is ambiguous because if more THENs than ELSEs are generated, a given ELSE may be paired with more than one THEN. The "dangling else" rule in Pascal is that an ELSE is always paired with the *nearest* unpaired THEN. This rule can be made explicit using a disambiguating predicates as follows:

```
Stmt     → IF Expr THEN Stmt ElsePart
ElsePart → ELSE Stmt
ElsePart → #TestNextSymbol
```

#TestNextSymbol tests the next input symbol (the lookahead). If it is anything other than an ELSE, the predicate evaluates to true. If the lookahead is an ELSE, the predicate blocks application of the production, ensuring that the ELSE is paired as soon as possible. This modified grammar can be parsed by LL(1)-like techniques, and in fact similar ideas are usually used to allow an LL(1) parser to handle the "dangling else" construct [7].

As another example, consider the problem, noted earlier, of generating a list of options, in any order, with no option repeated. Using disambiguating predicates, the following simple (and compact) grammar (G_1) can be used:

```
OptionList → OptionList , Option
OptionList → Option
Option     → #ANotUsed A
Option     → #BNotUsed B
             .
             .
             .
```

The disambiguating predicates test whether or not a given option has been used yet. Once an option is used, future uses are blocked.

Implementing Predicates

Obviously, predicates are most useful when they are fully integrated into a parser generator package. In fact, a few such generators exist [10], usually including attribute evaluation with parsing.

Unfortunately such packages are not widely available, and often are a good deal more complex than ordinary parser generators. We shall therefore investigate ways of obtaining the power of predicates using standard parser generators. Any computations that are needed can either be formulated in terms of standard parsing issues, or included as a post-processor to the output of a parser generator. The concepts discussed will also be of value in adding predicates to existing "standard" parser generators.

In many ways, a predicate symbol (i.e., a symbol that represents the invocation of a contextual or disambiguating predicate), can be viewed as a variety of terminal symbol. Thus, given

Variable → ID #IsArray [Expression]

the predicate symbol #IsArray represents a "marker" verifying that ID is declared as an array. If ID isn't an array, the marker is absent, and the production is blocked.

This approach is easily implemented. If a predicate symbol can be read in a given parser configuration[7] then the corresponding predicate is evaluated.[8] Arguments to a predicate are found in the semantic stack and symbol table maintained by the compiler.

If the predicate evaluates to true, a token corresponding to the predicate symbol is inserted into the input.[9] This insertion allows the parser to consume the expected symbol, and production recognition can proceed. If the predicate evaluates to false, the predicate symbol is not inserted, and the production is blocked (possibly causing a syntax error to be recognized). In effect, special markers are added to the user's input (as a side-effect of predicate evaluation), providing extra information to the parser.

---

[7]This condition is easily determined by examining the parse table entries corresponding to a given configuration.

[8]All predicates are assumed to be side-effect free.

[9]Lookahead symbols must be saved prior to insertion of the predicate symbol.

If the parser is blocked because a predicate symbol cannot be matched, a semantic error has been detected. An error message, based on the expected predicate symbol, can be issued. The arguments used by the predicate symbol can be used to improve the readability of the diagnostic (e.g., "TotalCost has not been declared").[10] Semantic error recovery, analogous to syntactic error recovery, can also be employed. That is, after issuing the appropriate diagnostics, a recovery routine is invoked. This routine does the necessary fixup (e.g., by flagging values the predicate has found to be illegal), then it inserts the predicate symbol and restarts parsing.

As might be expected, treating predicate symbols purely as terminal symbols can raise difficulties. One problem is that in a given parser configuration, more than one predicate may need to be evaluated. What's worse, more than one predicate may evaluate to true. This case may represent a parsing ambiguity, or (more benignly), a situation in which later context will decide which production to choose. For example, in G_1 (used to generate option lists), predicates evaluate to true if an option hasn't been used yet. An option production is recognized only if the option hasn't previously been used *and* if the option is actually present.

Another issue is the use of predicate symbols as "lookaheads". A lookahead is a terminal symbol that may not be part of the production being matched, but rather part of the context just beyond it. Thus, given the expression A + B * C, the "*" is used as a lookahead to decide whether or

[10]For uniformity, we can assume each predicate returns the text of a diagnostic message if it evaluates to false.

not A + B should be recognized as a subexpression. It is possible for a predicate symbol (if it is treated as a terminal) to "shield" a real lookahead symbol. In

Expr → Expr #TestArithmetic * Term #TestArithmetic

#TestArithmetic follows Expr and can appear as a lookahead for a production derived from Expr. #TestArithmetic shields the "*" that follows it, thus requiring an additional lookahead to determine the proper subexpression to match. Furthermore, #TestArithmetic can't be properly evaluated until the Expr to its immediate left is fully matched (and its type is determined).

From the above, we can conclude that predicates should only be "visible" (and evaluated) when they are part of a production currently being matched. Their use as lookaheads must (as detailed below) be severly limited.

Adding Predicates to LL(1) Parsers

LL(1) parsing is the simplest automatic parsing technique that is powerful enough to handle modern programming languages. The basic idea is remarkably simple. A stack of grammar symbols that must be matched is maintained. This stack is updated as symbols are matched. In particular, if the top unmatched symbol is a terminal, it is compared with the next input symbol. If it matches, the input symbol is consumed, and the stack is popped. If it doesn't match, a syntax error is detected.

If the top unmatched symbol is a non-terminal, A, then a parse table, M, is queried, using A and b, the next input symbol. M[A,b] contains either a production $A \rightarrow X_1 \cdots X_m$ or an error flag. If the error flag is present, a syntax error is detected; otherwise, A is replaced with $X_1 \cdots X_m$.

An especially useful aspect of LL(1) parsers is the inclusion of *action symbols*[11] ([11], sect. 7.3). Action symbols are distinct from the class of terminals and non-terminals. In fact, they are ignored completely in determining parser actions. When an action symbol reaches the top of the unmatched symbol stack, it is immediately popped, and a corresponding semantic routine is called. Thus, in

ExprTail → + Expr @Add ExprTail

the action symbol, @Add, would call a semantic routine after the right operand of a plus operation is matched.

The similarity of predicates to action symbols is obvious, and in fact action symbols are routinely used to check for semantic correctness (i.e., they serve as contextual predicates). However, because action symbols play no part in parsing decisions,[12] they cannot be used as disambiguating predicates.

We will implement all contextual predicates as action symbols. In LL(1), all parsing choices involving non-terminals are made before *any* symbols of the right hand side are matched (i.e., the parser is "predictive"). Thus

---

[11] All action symbols will be prefixed with a "@".

[12] Other than blocking in the presence of semantic errors.

all disambiguating predicates will appear as the leftmost symbol in a right hand side. Disambiguating predicates will be implemented as terminals. This choice will allow non-LL(1) grammars to be disambiguated. We will assume each disambiguating predicate symbol is unique (although different predicate symbols may represent the same predicate). This assumption will guarantee that the inclusion of disambiguating predicates can resolve all parsing conflicts. For example, consider the following grammar fragment that generates various forms of expressions in Pascal:

```
Expr → Variable
Expr → FunctionCall
Expr → Constant
```

This fragment is not LL(1) because Variable, FunctionCall and Constant can all generate an identifier. The addition of predicates can disambiguate this choice:

```
Expr → #TestVarID Variable
Expr → #TestFunctionID FunctionCall
Expr → #TestConstantID Constant
```

Each disambiguating predicate checks whether the next input symbol is an identifier of the expected class (variable, function or constant). Since the predicate symbols are considered distinct terminals by the parser generator, the productions are trivially LL(1).

For each non-terminal, A, we can build a table containing all the disambiguating predicates that appear in productions that have A as the left hand side. This table represents the set of predicates that need to be evaluated when A is expanded (i.e., when A reaches the top of the

unexpanded symbols stack). Each disambiguating predicate associated with A is evaluated in turn.[13] If more than one predicate is true, we have an error (since LL(1) requires a unique prediction of what production to apply). If exactly one predicate is true, we can insert the corresponding predicate symbol into the input, forcing the desired production to be chosen.[14] If no predicate evaluates to true, we simply continue with the normal parsing cycle (and either select a production not involving a predicate or recognize a syntax error).

Handling Predicate Lookaheads in LL(1) Parsers

It is possible that a disambiguating predicate can appear as lookahead for a production other than the one that contains it. For example, we might have

A → B C D
A → x y z
B → #1 a b
B → #2 a c

Now #1 and #2 appear in A's lockahead set, although they are intended to disambiguate B. As discussed earlier, we wish to delay evaluation of predicates until we are ready to match the production in which they occur. We therefore need a means of ignoring all predicate symbols other than those related to the non-terminal being expanded.

---

[13]Since predicates are side-effect free, the order of evaluation is irrelevant.

[14]As an optimization, we can simply pop A, and replace it with the selected right hand side less the predicate symbol.

Let us define *non-local predicates* to be all predicate symbols that appear as lookaheads for some non-terminal A, but which don't begin any production with A as the lefthand side. Analogously, *local predicates* are those that begin productions belonging to the non-terminal under consideration. Our goal is then to eliminate non-local predicates as lookaheads when expanding a given non-terminal.

We first note that a given production cannot contain both local and non-local predicates as lookaheads (since a local predicate implies that the production begins with a terminal – the predicate symbol). Thus, all the productions associated with a given non-terminal can be partitioned into those that employ a local predicate and those that don't. The latter class can be distinguished on purely syntactic grounds (else disambiguating predicates would have been added).

We limit our consideration to those productions that don't employ a local predicate (and thus may have non-local predicates in their lookahead sets). Only one of these productions (at most) can derive ε (otherwise they couldn't be distinguished on syntactic grounds). If the production $A \rightarrow \alpha$ can derive ε (ignoring predicate symbols) and has a non-local predicate in its lookahead set, we will make it a "default production". That is, it will be predicted if a given lookahead predicts no other production. This prediction is feasible because a lookahead that predicts no other production must either be a correct lookahead for $A \rightarrow \alpha$, or an error symbol. In the latter case, a syntax error will be discovered later when the lookahead symbol can't be matched against the stack of expected sym-

bols.

If a production $A \rightarrow \beta$ can't derive ε, and has a non-local predicate in its lookahead, we need only compute the set of terminals (ignoring predicates) that can begin strings derived from β. That is, we compute First(β) where

$$\text{First}(\beta) = \{a \in V_t \mid \beta \Longrightarrow^* a \cdots\}$$

To do this computation, we first read in the productions of the grammar (this information is produced as output by LL(1) parser generators) and eliminate all disambiguating predicates and action symbols. We then compute the set of non-terminals that can derive ε (either directly or indirectly). This computation can be done iteratively: First mark non-terminals that derive ε directly, then mark nonterminals that directly derive a sequence of marked non-terminals. This iteration is continued until no more non-terminals can be marked.

To compute First sets, we can employ the following algorithm:[15]

Algorithm 1 {Computation of First sets}

[1] Start with First(A) = $\{X \mid A \rightarrow X \cdots\}$.

[2] If $A \rightarrow Y_1 \cdots Y_m Z \cdots$ and $Y_1 \cdots Y_m$ derive ε
(i.e., are marked non-terminals)
Then add Z to First(A).

[3] Repeat until no more additions to First set can be made):
If some non-terminal $B \in$ First(A)
Then First(A) := First(A) $\cup$ First(B)

---

[15]This computation is equivalent to taking the transitive closure of a binary relation. See, e.g., [12].

The above computations are easy to include in an LL(1) post-processor. For example, the ε-marking and First computations were included in a Pascal post-processor to the LL(1) generator described in [2]. This post-processor reads the parse tables produced by the LL(1) generator and removes from them all predicates that appear as non-local lookaheads. Only 150 lines of code were needed for the post-processor, and a full Pascal grammar could be processed in 5 seconds on a Vax 11/780.

Preserving LL(1)-ness

One final point to be considered is whether the inclusion of disambiguating predicates in one production can cause prediction conflicts to appear in other productions. This can in fact happen. Consider, e.g.,

S → A B $
A → B x
A → ε
B → ε

This grammar is LL(1). If we change the B production to B → #, then the two A productions have a prediction conflict (on #). Fortunately, this problem can arise only if we add a predicate symbol to a production of the form C→α, where C derives *only* ε (if C derives any terminals symbols, then these too would induce prediction conflicts). Since a non-terminal that derives only ε clearly never needs disambiguation, we will simply require that contextual predicates (implemented as action symbols), rather than disambiguating predicates, be used in such situations.

LL(1) Checklist

A convenient way to employ predicates in an LL(1) grammar is to first add predicate symbols to a context-free grammar. Then the following checklist can be used to determine how to implement the predicates within the framework of an LL(1) parser generator.

(1) All contextual predicates can be implemented as action symbols.

(2) Disambiguating predicates that are local to a production are implemented as terminals. When the corresponding left hand side is expanded, local disambiguating predicates are evaluated. If any evaluates to true, the corresponding predicate symbol is inserted into the input.

(3) Nonlocal disambiguating predicates are removed by using Algorithm 1 to compute the lookaheads that predict the application of a particular production.

(4) Predicates can introduce parsing conflicts only if a disambiguating predicate (implemented as a terminal) is added to a production whose left hand side derives *only* ε. Since disambiguation is never needed in such a situation, the disambiguating predicate can be replaced by a contextual predicate (implemented as an action symbol).

Adding Predicates to LR-type Parsers

We now investigate how to add predicates to the LR family of parsers (SLR(1), LALR(1), LR(1)).[16] In LR-type parsers, action symbols are much more restricted than in the LL(1) case. In particular, they may only appear at the extreme right of a production, and are invoked upon recognition of a production.[17] Predicates that appear anywhere except at the extreme right can be implemented as either terminal symbols, or as new non-terminals that derive only ε.

Implementation of predicates as non-terminals is attractive in that predicate evaluation can be triggered by the usual production recognition mechanism. Further, these non-terminals cause no lookahead problems because they generate only ε.

Unfortunately, not all predicates can be implemented as non-terminals. Parsing conflicts can be introduced (e.g., if various predicates, in different productions, need to be evaluated at the same time). Further, such non-terminals are generally not useful as disambiguating predicates (since they add, rather than remove, reduction actions in a state). We therefore will concentrate on the implementation of predicates as terminal symbols, always assuming that action symbols or non-terminals are used where feasible to implement contextual predicates.

---

[16]See [6] for a review of the fundamental concepts of LR parsing.

[17]This restriction on action symbols is necessitated by the fact that LR-type parsers delay production recognition until the entire right hand side is examined.

As in the LL(1) case, we will evaluate predicates when they can be read (i.e., shifted) as part of a production currently being matched. LR-type parsers allow more than one production to be under consideration at the same time.[18] This possibility implies that more than one predicate might evaluate to true, indicating that a set of productions are still valid candidates for recognition. This possibility is exploited, e.g., in G_1 which uses disambiguating predicates to monitor which options can still be matched.

After predicates are added to a grammar, we can analyze each parse state to determine what predicate symbols can be shifted from that state. This analysis represents the set of predicates that must be evaluated whenever that state is at the top of the parse stack. Sometimes, the predicates are mutually exclusive. If they are not, we require some means of indicating to the parser what set of predicates is true. There are two ways to indicate this set. The first is to create new symbols that represent all the various predicate combinations that might be evaluated, and found true, at the same time. For example, we might have

S → A a x
S → B a y
A → #1 b
B → #2 b

If the predicates represented by #1 and #2 could both evaluate to true, we would introduce a new symbol, #{1,2}, which represents the case in which both predicates are true. Wherever #1 occurs, a new production, with #{1,2} replacing it, is created, and a similar process is performed for

[18]This is the reason why LR-type parsers can handle a broader class of grammars than LL-type parsers.

#2. Thus, the above grammar would become

S → A a x
S → B a y
A → #1 b
A → #{1,2} b
B → #2 b
B → #{1,2} b

Now depending on predicate evaluation, #1, #2, #{1,2} or no predicate symbol at all is inserted.

Naturally, depending on how many predicates can be simultaneously true, the size of a grammar can grow exponentially. In fact, we do not recommend that this approach be used in practice. It is valuable, however, in detecting problems induced by predicates that aren't mutually exclusive. Thus in the above example, we would have a parsing ambiguity (a reduce-reduce conflict) if both predicates were true. This difficulty could be detected by attempting to create a parser for the expanded grammar.

If we know that no parsing conflicts can arise if multiple predicates evaluate to true, we can utilize the original grammar and parse tables. The trick is to maintain the parse stack not as a stack of states, but rather as a stack of sets of states. If more than one predicate evaluates to true, we push successors under each of the appropriate predicate symbols. All states at the top of the stack must agree to shift or do the same reduction (else the extended grammar would have contained parsing conflicts). While parsing G_1, for example, we might have a number of states at the top of the stack, each waiting to shift a particular option.

Sometimes rewriting a grammar a bit allows us to eliminate the possibility of multiple predicates evaluating to true. For example grammar G_1, could be rewritten into

```
OptionList → OptionList , Option
OptionList → Option
Option     → A #ANotUsed
Option     → B #BNotUsed
             .
             .
             .
```

Here we test whether an option has previously been used *after* it is matched. Clearly this form of G_1 is equivalent to the original, and only one predicate is evaluated at any time.

Handling Predicate Lookaheads in LR parsers

As in the LL case, we will remove from the parse table predicate symbols that appear as lookaheads for reduction actions. Only predicates that indicate shift actions will be retained, and these actions will cue predicate evaluation.

Lookahead calculation can be quite involved in LR-type parsers, particularly in the LALR(1) and LR(1) cases. We won't attempt to duplicate these calculations. Rather, the tables produced by an LR-type parser generator (particularly the *action* and *goto* tables) will be analyzed to infer the lookahead symbols (excluding predicates) that indicate a reduction in a given state. This kind of calculation can be conveniently incorporated in a post-processor to a LR-type parser generator, allowing the use of predicates without altering existing software.

LR-type parsers assume a finite number of *parse states*, denoted $s_1, \cdots, s_m$, and a vocabulary V consisting of terminal symbols (V_t) and non-terminal symbols (V_n). The parser action table maps a state s and a terminal symbol a (denoted Action[s,a]) into one of four classes of actions: Shift, Reduce $A \rightarrow \alpha$, Accept, Error.

The Goto table maps a state s and a symbol X (denoted Goto[s,X]) into a new state t, the successor of s under X. The Goto table represents a partial function, as a state need not have successors under all symbols.

From the Goto table, we can create an *ancestor* function, A, that maps states to the set of states that are their immediate predecessors. That is,

$$A(s) = \{t \mid \text{Goto}(t,X) = s \text{ for } X \in V\}$$

This definition can be generalized to a *set* of states, S:

$$A(S) = \bigcup_{s \in S} A(s)$$

Goto is extended to a set of states in an analogous manner. The set of n-th ancestors of a set of states, A(S,n), is defined as:

$$\begin{aligned} A(S,0) &= S \\ A(S,1) &= A(S) \\ A(S,n+1) &= A(A(S),n) \end{aligned}$$

Finally, let R(s) be the set of all productions that might (for some lookahead) be recognized in state s. This information is readily extracted from the Action table.

Assume now that for some state s and some predicate #p we have Action[s,#p] = Reduce $B \rightarrow \beta$. Some states have the property that the only

non-error action associated with them is the reduction of a particular production. In such states lookahead need not even be examined,[19] so the presence of a predicate in the lookahead can be ignored. If there is choice of actions however, we must determine what "real" (i.e., non-predicate) lookaheads are appropriate for a given reduction (say $B \rightarrow \beta$ in state s). To make this determination, we first decide what states we might reach after the production is recognized. This set is defined as

$$\text{Reduce}(s, B \rightarrow \beta) = \text{Goto}(A(s, |\beta|), B)$$

where $|\beta|$ represents the size (or length) of the string β.[20] That is, we first determine the set of states possible after β is reduced ($A(s,|\beta|)$), then we obtain the set of states possible after the left-hand side, B, is read.

Given this set of states, three situations may arise:

(1) A terminal may be read (any such terminal is a valid lookahead).

(2) A predicate symbol may be read (with valid lookaheads following this predicate symbol).

(3) Another reduction may be applied (with valid lookaheads read after this reduction).

To model these possibilities, we define a *closure* function, Close(S). It takes a set of states and defines the set of states that are reachable from that set. Any terminal that can be read from Close(S) is a legal lookahead (since it could be accepted as the next input symbol). Define

---

[19]Such states are often eliminated as a space optimization.

[20]Reduce is extended to sets of states in the obvious manner.

Goto#(S) = {s|t∈S and s = Goto(t,#p) for some predicate #p}

Goto#(S) is the set of states that can be reached from states in S after reading a predicate symbol. Now

$$\text{Close}(S) = \bigcup_{B\to\beta\in R(S)} \text{Close}(\text{Reduce}(S,B\to\beta)) \cup \text{Close}(\text{Goto\#}(S)) \cup S$$

The states reachable after reducing $A\to\alpha$ in state s are

Close(Reduce(s,$A\to\alpha$))

Define

Read(S) = {a∈V_t | Action[s,a] = Shift for s∈S}

Read(S) represents all the terminal symbols that can be read from states in S. Then the "real" lookaheads (excluding all predicate symbols) that indicate Reduce $A\to\alpha$ in state s are

Read(Close(Reduce(s,$A\to\alpha$)))

The above lookahead calculations are fairly easy to implement, especially in languages such as Pascal that allow set manipulation. A postprocessor for the LALR(1) parser generator described in [1] required 330 lines (in Pascal). It was tested on a Pascal grammar containing 191 productions. The grammar was augmented with contextual predicates to enforce semantics and a few disambiguating predicates to resolve parsing conflicts. Where possible, contextual predicates were implemented as action symbols (i.e., those that appeared at the extreme right end of productions). For testing purposes, all other predicates were implemented as terminals (although some contextual predicates could have been implemented as non-terminals). A total of 41 predicate symbols were

added, representing 17 distinct predicates. The postprocessor ran in 66 seconds on a Vax 11/780.

A predicate symbol cannot appear as a lookahead for a reduction action if it immediately follows a terminal. Further, if a (non-predicate) terminal symbol, t, immediately follows a predicate symbol, then the closure algorithm described above isn't really needed. Rather, if the predicate symbol appears as a lookahead for a reduction action, it can be immediately replaced by t.[21] Therefore the only situations in which the closure algorithm is needed are those in which a predicate symbol is bracketed between two non-terminals, or between a non-terminal and an end of the right hand side, or in which the predicate symbol is the entire right hand side (and thus is bracketed by both ends of the production).

"Bracketed" predicate symbols appear to be comparatively rare (they only occurred 5 times in the Pascal grammar described above). If a grammar can be rewritten to remove bracketed predicate symbols, then the closure algorithm need not be used. In the Pascal grammar described above, all 5 occurrences of bracketed non-terminals were easily removed. For example, the following production was used to generate infix expressions involving '+' and '-'.

<SIMPLE EXPR> → <SIMPLE EXPR> <SIGN> #check_arith_or_set <TERM>

Substituting for <SIGN>, we created the following equivalent productions.

<SIMPLE EXPR> → <SIMPLE EXPR> + #check_arith_or_set <TERM>
<SIMPLE EXPR> → <SIMPLE EXPR> - #check_arith_or_set <TERM>

---

[21]Recall that all predicate symbols are assumed to be unique.

The closure algorithm should be viewed as a means of accommodating the widest class of predicated LR grammars. Our experience, including the work described in [13], indicates that bracketed predicate symbols can easily be avoided. This allows simpler (indeed, essentially trivial) post-processors can be employed.

In the lookahead calculations described above, the closure function may include states that aren't actually reachable. For example, given an ε-production, $B\rightarrow\varepsilon$, Reduce(s,$B\rightarrow\varepsilon$) = Goto(A(s,$|\varepsilon|$), B) = Goto(A(s,0),B) = Goto(s,B) = s'.

Now when s' is closed, if Reduce(s',$D\rightarrow\delta$) is computed for $\delta\neq\varepsilon$, then *all* ancestors of s' (rather than just s) are included. This addition can cause unreachable states to be included.

In practice these unreachable states do not appear to cause any problems (i.e., no parse conflicts seem to be introduced by including lookaheads from the extra states). It is possible to extend the above algorithms to exclude the unreachable states. The idea is to compute *sequences* of states representing possible uppermost stack sequences. Goto extends the sequence by appending a state, and the ancestor function removes states from the sequence. In the Close function we exploit the fact that no state need ever be repeated in a sequence.[22] Further, closure of a set of sequences of the form $\{s_1t, s_2t, \cdots, s_kt\}$ where $s_1, s_2, \cdots, s_k$ represent *all* ancestors of t can be replaced by closure of {t}. (If all ancestors of t are to be included, individual sequences involving

[22] I.e., only cycle-free paths through the underlying CFSM need be considered.

each ancestor of t are not needed).

Using these observations, an extended postprocessor can be implemented. Ours required 520 lines in Pascal (vs 330 lines) and required 200 seconds (vs 66 seconds) to remove predicate symbols from lookaheads for our extended Pascal grammar. In all tests both algorithms computed exactly the same lookaheads, and we conjecture that the extended algorithm will rarely, if ever, be needed in practice.

Avoiding New Parse Conflicts

As in the LL(1) case, it is possible that the inclusion of a predicate symbol in an ε-production can introduce a parsing conflict. For example, consider

S → A d e
S → B d f
S → Q A b
S → R A c
A → ε
B → ε
Q → ε
R → ε

This grammar has a parsing conflict involving A→ε and B→ε (both have a lookahead of d). This conflict could be resolved by introducing a disambiguating predicate in the A production, obtaining A→#p. Now, however, we have a new conflict involving Q→ε and R→ε (both now see #p as a lookahead). This isn't a "real" conflict and correct lookaheads will be seen after predicate symbols are purged as lookaheads for Reduce actions (by the post-processor). The problem is that the parser generator will see an

apparent conflict and may not even generate tables (because of apparent errors)! We may need someway to "trick" the generator into seeing an acceptable grammar. A way to implement this trick is to add new "dummy" disambiguating predicates that will disambiguate the apparent lookahead conflicts. Thus, in the above example, two predicates, #p1 and #p2, are added to productions 3 and 4 to obtain:

S → A d e
S → B d f
S → Q #p1 A b
S → R #p2 A c
A → #p
B → ε
Q → ε
R → ε

These two predicates will always evaluate to true, since lookahead (once predicate symbols are removed) suffices to make the necessary parsing decisions. In fact, since these dummy predicates were added solely to "trick" the parser generator into generating tables, we can even eliminate them completely. This elimination is done by merging the states (and parsing actions) that were split by the inclusion of the dummy predicates. That is, if we have states s and s' where Goto(s,#p1) = s', we can merge s and s' into one state, by merging Goto and Action table entries. This merge is safe since we know that all parsing decisions can be made without the evaluation of these dummy predicates.

LR Checklist

A convenient way to employ predicates in an LR-type grammar (i.e., LR(1), LALR(1), SLR(1)) is to first add predicate symbols to a context-free grammar. Then the following checklist can be used to determine how to implement the predicates within the framework of an LR-type parser generator. The simplest alternatives are listed first, and these will suffice in the vast preponderance of cases.

(1) Contextual predicates that appear at the extreme right end of a production can be implemented as action symbols.

(2) Contextual predicates that appear elsewhere in a right hand side can be implemented as non-terminals that derive only ε or as terminals. Non-terminals can be used if no parsing conflicts are introduced; otherwise terminals should be used.

(3) Disambiguating predicates should be implemented as terminal symbols.

(4) Terminals that implement predicates can be removed as lookaheads for reduce actions as follows:

   (a) If a predicate symbol follows a (non-predicate) terminal, then the predicate symbol can't appear in the lookahead of a reduce action.

   (b) If a (non-predicate) terminal follows a predicate symbol, then all occurrences of the predicate symbol as a lookahead for a reduce action can be replaced by the terminal that follows the predicate symbol.

(c) In all other cases, the predicate symbol must be bracketed by non-terminals and/or the ends of the right hand side. In these cases, the grammar can either be rewritten to remove bracketed predicates *or* the closure algorithm can be used to eliminate predicates as lookaheads for reduce actions.

(5) If more than one predicate can evaluate to true at the same time:

(a) New, composite predicate symbols that represent possible combinations of true predicates can be added. This construction can greatly expand the size of a grammar, but is useful for locating parsing conflicts not resolved by disambiguating predicates.

(b) The LR parser driver can be modified to maintain a stack of sets of parse states. If the predicated grammar contains no unresolved parsing conflicts, then lookaheads and/or additional predicates will guarantee that all states in a set will agree on the same action (shift or reduce a particular production).

(c) The grammar can be rewritten to guarantee that only one predicate can evaluate to true at any point. (From our experience this is usually easy to do).

(6) If the addition of predicate symbols (implemented as terminals) introduces new parse conflicts, then "dummy predicates" (which always evaluate to true) can be added. As an optimization, after parse tables are generated, states that were split by the dummy predicates can be merged (along with their corresponding action

table entries).

Conclusion

Context-free grammars and parsers have proved to be remarkably useful tools in the construction of compilers and language processors. Predicates represent a simple way to extend the range and scope of these tools. The techniques we have presented allow the use of predicates with existing bottom-up and top-down context-free parser generators. We believe these techniques are simple, efficient and quite general. They also may be used *within* a parser generator to extend the class of grammars that can be accommodated.

As a final point, the addition of predicates to context-free grammars can open entirely new application areas. For example, most of the results presented in this paper are the result of work in automatic code generation [13], [14], [4]. Here context-free productions represent "templates" that describe various target machine instructions. Since real computers often have numerous instructions that can be used to effect the same result (e.g., add, add immediate, increment by one, etc), predicates are extremely useful in defining how and when a given instruction will be chosen. It is this ability to "wire in" extra information in a production that makes the application feasible. We believe that any application area that utilizes context-free parsing techniques may benefit from the enhanced capabilities that predicates provide.

References

[1] Mauney, Jon and Charles N. Fischer, "ECP - An Error-correcting Parser Generator: user guide," Tech. report #450, University of Wisconsin (October 1981).

[2] Mauney, Jon and Charles N. Fischer, "FMQ - An LL(1) Error-correcting Parser Generator: user guide," Tech. report #449, University of Wisconsin (October 1981).

[3] Glanville, R. S. and S. L. Graham, "A New Method for Compiler Code Generation," *Fifth ACM Symposium on Principles of Programming Languages*, (Jan. 1978).

[4] Ganapathi, M. and Charles N. Fischer, "Automatic Code Generation and Optimization using Attributed Grammar Machine Descriptions," *ACM Transactions on Programming Languages and Systems*, (to appear).

[5] Jensen, K. and N. Wirth, *Pascal User Manual and Report, Second Edition*, Springer-Verlag, New York (1978).

[6] Aho, Alfred V. and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

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

[8] Milton, D. R., "Syntactic Specification and Analysis with Attribute Grammars," PhD thesis, University of Wisconsin-Madison (1977).

[9] Milton, D. R. and C.N. Fischer, "LL(k) Parsing for Attributed Grammars.," *International Colloquium on Automata, Languages and Programming, 1979*, (July. 1979).

[10] Milton, D. R., L. W. Kirchoff, and B. R. Rowland, "An ALL(1) Compiler Generator," *ACM Sigplan Symp. Compiler Construction, Boulder, Colo.*, (Aug. 1979).

[11] Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns, *Compiler Design Theory*, Addison-Wesley (1973 (or 1976)) Reading, Mass..

[12] Aho, Alfred V. and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, N.J (1972).

[13] Ganapathi, M., "Retargetable Code Generation and Optimization using Attribute Grammars," Tech. Report 406, University of Wisconsin-Madison (1980) Ph.D. thesis.

[14] Ganapathi, M. and C. N. Fischer, "Description-driven Code Generation using Attribute Grammars," *Proc. 9th ACM Symp. Principles of Programming Languages* , pp. 108-119 (January 1982).