Comput. Lang. Vol. 14, No. 1, pp. 25–33, 1989
Printed in Great Britain.

0096-0551/89 $3.00 + 0.00

SEMANTIC PREDICATES IN PARSER GENERATORS

MAHADEVAN GANAPATHI
Computer Systems Laboratory, Stanford University, Stanford, CA 94305, U.S.A.

(*Received* 6 *July* 1988; *revision received* 3 *October* 1988)

Abstract—Parser-based pattern matching proves to be attractive to compiler implementors because of its use in compiler-compilers. Semantic predicates can be added to parsers to augment the decision making power provided by syntactic analysis. This paper presents a practical technique to incorporate them within the framework of existing parser generators. A nearly vanilla parser can be used in this manner provided that epsilon productions are added to the grammar. Reduction by epsilon productions triggers the predicates, which as a side effect modify the lookahead symbol(s), thus influencing the progress of the parser at the last possible minute.

Parser generators Code generator generators Disambiguating predicates Conflict resolution Attributed parsing

1. SEMANTIC PREDICATES

LL and LR based parsers may exhibit conflicts when the input grammar fails to fall within the lookahead limits of the parser. In LL (top-down) parsing, production identification takes place before all of the right-hand-side (RHS) components have been processed [1, 2]. Ambiguities that occur in LL parsing are all predict–predict conflicts. In LR (bottom-up) parsing, a reduction takes place only when the entire RHS of a production has been processed. Ambiguities that occur in LR parsing are either shift–reduce or reduce–reduce conflicts. In practice, shifts are taken over reductions, and otherwise the first applicable reduction is taken. One could instead supply nonsyntactic predicates to resolve the conflicts.

For example, consider a reduce–reduce conflict among the following productions:

$$P \rightarrow c$$
$$Q \rightarrow c$$

The user can express his intention to always reduce by the latter production by placing the latter before the former in his input grammar specification. This resolution is determined statically and having selected to always reduce by $Q \rightarrow c$, one can never reduce by the production $P \rightarrow c$.

In general, there can be several productions with similar right-hand-side (RHS) symbols; all these productions may participate in conflicts. Often, it is convenient to select parsing actions dynamically. In the above example, it may be useful to reduce by $P \rightarrow c$ in some cases and by $Q \rightarrow c$ in other situations. To provide such a facility, the parsing algorithm can be modified to accept disambiguating predicates that dynamically control parsing [3, 4]. Usually, predicates are introduced to produce different suffixes in otherwise identical productions. During parsing, these predicates evaluate to true or false only.

Disambiguating predicates serve as a guide to when the production is applicable. They select a production based on context, thereby resolving parsing conflicts. Although, these predicates can be added as the productions are designed, in practice, they are added only after a canonical collection of configuration sets has been computed and found to contain conflicts. Upon the occurrence of a conflict, all relevant predicates are evaluated in the order in which the conflicting productions are specified.

In general, a production may have many predicates. Furthermore, several predicates may evaluate to true simultaneously. To select a production, a hierarchy of predicates or a linear ordering is used. For machine-description grammars [5, 6], productions may syntactically match the intermediate representation from a compiler front-end [7]. However, these grammar productions may not satisfy the semantic restrictions imposed by the target architecture. Such blocking

is termed as semantic blocking. Certain constraints must be imposed to avoid ambiguity or semantic blocking. For a particular conflict, only one predicate must evaluate to true and all others must evaluate to false. This restriction prevents ambiguity. To prevent semantic blocking, at least one predicate must evaluate to true. This criterion ensures that at most one production is selected at any instance. Linear ordering of preference is a simple strategy to break a parsing conflict. All relevant predicates are evaluated in the order in which the productions are specified. Then, the first production whose predicate evaluates to true is selected.

2. THEORETICAL ISSUES

A predicate can be associated directly with the production whose prediction it will determine. Since, before prediction, very little information is available on the RHS symbols, potentially infinite look-aheads are required to select the proper template. Predicates need to consider the non-terminal on top of the parser stack along with these look-aheads.

At the time when a predicate must be evaluated, the left-hand-side (LHS) of the production is unknown. Dependencies of a grammar-variable on its parent is unknown and thus, unavailable. For shallow parse-trees and fairly independent productions, this restriction does not pose any problem. Usually, for non-recursive productions, inter-dependent symbols can be grouped in the same production, eliminating the necessity for information flow from parent to child. All information on RHS symbols, available as semantic attributes of left-context symbols on the attribute-stack, can therefore be accessed to disambiguate multiple production matches and control parsing.

Predicates take a fundamentally different form for LR parsers than for LL parsers. In LR parsing the conflicts are of the shift–reduce or reduce–reduce variety. Moreover, the conflicts are present only in the context of a particular state or configuration set. Thus, while an LL parser bases its decision on a non-terminal and look-ahead, an LR parser bases its decisions on a parse-state and look-ahead. In theory, for LR parsers, predicates must be associated with states and not productions. Furthermore, the top stack symbol, along with its attributes, will typically not provide enough left context for a predicate to perform disambiguation. Left context in bottom-up parsing can only be transmitted up the derivation tree. Thus, the symbol on top of an LR parser stack can only convey information, in its attributes, from the sub-tree it heads. Therefore, in general, predicates need to examine more than one symbol at top of the parse-stack. The state, actually the corresponding configuration set, will determine how many symbols on top of the stack will be available to the predicate for examination.

In general, positions available for invoking semantic routines for LR(k) and SLR(k) parsers have been elaborated by [8]. Similarly, restrictions must be placed on where predicates may appear in the RHS of grammar productions. To be safe, they may appear only at the extreme end of a production. Movement of predicates to the left can potentially yield unresolvable parsing conflicts [4, 9]. In particular, they may shield existing look-aheads and thus, introduce blocking situations. Consider the following examples:

- Movement of conditional semantic action to a predicate in the RHS.
  C → ∈
  ```
      if condition
          then action_routine()
  ```
  The condition could be moved out of the semantic action and represented as a predicate appearing at the extreme right end of the production.
  C → ∈ *condition*
  ```
      action_routine()
  ```
- Movement of predicate to the left in the RHS of a production.
  A → B C *predicate*
  If the predicate is really testing the semantics of non-terminal B alone, then it is possible to move the predicate before C.
  A → B *predicate* C

In the first example, although the grammar with the predicated ϵ production may be conflict free, blocking situations can occur. The ϵ production can introduce fresh parsing conflicts by forcing the parser to look-ahead too soon. In such situations, disambiguating tokens are unavailable to the parser before it needs to resolve a conflicting situation. Similarly, in the second example, the production A → B *predicate* C can block recognition of A → B C.

To prevent blocking, the introduciton of predicates must not shield existing look-aheads. More concisely, using the First and Follow sets of [1, 10],

$$\text{First}(\text{predicate\_symbol}) \subseteq \text{First}(\text{Follow}(\text{predicate\_symbol}))$$
$$\text{First}(\text{predicate\_symbol}) \supseteq \text{Follow}(\text{symbol\_before predicate\_symbol})$$

Alternately, extra productions can be added to prevent blocking situations. Consider the following productions:

P1: A → C
P2: B → C
P3: D → E A B
P4: D → E B A

The reduce–reduce conflict between P1 and P2 can be resolved by a predicate. It selects either A or B. In this grammar, the predicate must guarantee that after recognizing the non-terminal A another A will not immediately be accepted; similarly is the case with recognition of two consecutive occurrences of B. Otherwise, the following productions must also be added:

D → E A A
D → E B B

Intuitively, after the predicate symbol on the RHS, every occurrence of A and B should be replaced by occurrences of a symbol denoting A or B. If this replacement is not carried out, it is possible that a blocking situation may arise. Consider the productions P3 and P4 with predicates in their RHSs:

P3: D → E A *predicate_a* B
P4: D → E B *predicate_b* A

If *predicate_a* evaluates to true and on input C, P1 is selected then P3 will block. Similarly, if *predicate_b* evaluates to true and on input C, P2 is selected then P4 will block.

To enhance the power of predicates, both left-context as well as right-context can be used. When left-context is required, the grammar-writer can use existing knowledge of a configuration-set at the position where a predicate must be evaluated. With reference to this point, the stored position of a needed left-sibling on the parse-stack can be calculated. This relative offset is used to access the left-sibling. Consider similarly-sized productions. Usually, the desired left-sibling will be at a constant offset from the top of the stack for all items in the configuration-set. However, it is the grammar-writer's responsibility to ensure that the constant stack-offset is correct for all items in the configuration-set. The writer must also understand the details of the parser to write predicates that interrogate the stack for their arguments.

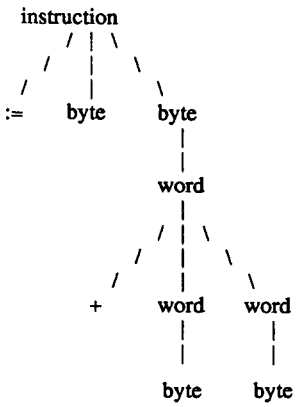
To resolve conflicts using right-context, look-ahead predicates must be used. Such predicates examine the look-ahead symbol, if any, already provided by the parser. In some situations, disambiguation is necessary before the end of a production. Predicates must therefore appear in the middle of the RHS of the production. Invariably, such predicates include look-ahead symbols as their arguments to make a correct selection and prevent syntactic blocking of the code generator for a valid input. Consider input := byte + byte byte and the following productions:

instruction → := byte byte
instruction → := word word
word → + word word
byte → + byte 1
byte → word
word → byte

In this conflicting situation, the correct selection would be to reduce by the production word → byte instead of a shift of byte in the production byte → + byte 1. To ensure this selection, the look-ahead predicate *LAshift* is used as follows:

byte → + byte *LAshift*(1) 1
word → byte ¬*LAshift*(1)

The predicate *LAshift* evaluates to true if the look-ahead symbol already provided by the parser happens to be one among its arguments. Thus, if the second operand to the + operator is not 1, a reduction is selected instead of a shift. The parse tree for the above input becomes:



Sometimes, in ambiguous specifications, look-ahead predicates can yield better parse solutions. Both shifting as well as reducing can give correct solutions but, one may be a better choice than the other in terms of semantic action. For example, consider machine-code generation [5, 6]. To prevent sub-optimal choices, look-ahead predicates can be used to check ambiguous situations and select between a reduction and a shift. Consider a shift–reduce conflict among the following productions (these grammar productions model comparison and branching on modern architectures):

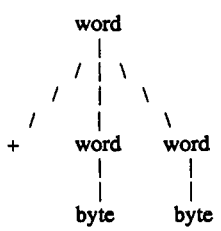
word → − word 1
instruction → > c d label
instruction → := word − word 1 > c d label

We add look-ahead predicate *LAshift* and its converse ¬*LAshift* to these conflicting grammar productions as follows:

instruction → := word − word 1 *LAshift*(>) > c d label
word → − word 1 ¬*LAshift*(>)

The predicate *LAshift* evaluates to true if the lookahead symbol already provided by the parser happens to be one among its arguments. In this example, the look-ahead symbol must be '>' in order that the shift be taken in place of the reduction.

To exemplify the advantages of lookahead predicates, consider addition on the PDP-11/70. It has an increment-byte instruction, but does not have an add instruction that takes byte operands. Thus, in the case where one operand is the constant one and the other is a temporary storage location, an increment byte instruction must be selected. Otherwise, bytes must be coerced to words so that an add-word instruction can be issued. The parse tree for the coercive input becomes:



We add lookahead predicate *LAcoerce* to handle conditional coercion:

word → + word word
word → byte *LAcoerce*(¬1, ¬*temporary*)

3. PRACTICAL TECHNIQUES

Ordinary parser generators pose problems when grammar writers utilize predicates to control parsing. They are not designed to handle predicates and controlled ambiguities in large grammars. Of course, new parser generators could be written to include these components [3, 9]. An attractive alternative is to recast these components in terms that conventional parser generators can handle [4]. The following paragraphs illustrate two strategies for implementation of predicates in context-free parsers. The first strategy calls for a modification of the parser generator itself. Predicate evaluation and selection is integrated in the parser generator. In addition to terminal and non-terminal symbols, predicate symbols are also included in the construction of sets of items in a given configuration of the parser-generator's characteristic finite-state machine.

The bottom-up parser with disambiguating predicates employs the standard LR(k) parsing loop with added code to call disambiguating predicates:

```
PROGRAM parser;

State  := 0;
Action := Shift;

    SWITCH (Action) OF

        CASE Shift:
            Push(State);
            (* determine next action *)
            Actset := Next_action(State,Token);
            IF Actset is single-valued
                THEN   Action := Actset
            ELSE
                Action := Disambiguate(State, Actset);
            (* determine next state *)
            State := Next_state(State, Token);
        END; (* case shift *)

        CASE Reduce:
            SWITCH (LhsProd) OF

                CASE nonterminal:
                    Pop(RhsProd);
                    State := Stack[1]; (* top of stack *)
                END; (* case nonterminal *)

            END; (* switch *)
            Actset := Nextaction(State,LhsProd);
            IF Actset is single-valued
                THEN Action := Actset
            ELSE
                Action := Disambiguate(State, Actset);
            State  := Next _state(State, LhsProd);
        END;   (* case reduce *)

        CASE Accept:   halt, accepting;
        END;   (* case accept *)

        CASE Error:    halt, rejecting;
        END;   (* case error *)
    END; (* switch *)
END. (* end program parser *)
```

Alternate to modifying the parser-generator, the second strategy involves modification of the parser-driver alone. The driver output by the parser generator is modified to create parsers for predicate-directed parsing. Conflicting productions in the grammar are expanded with distinct terminal symbols and an ∈ production is introduced to invoke a predicate-routine and select a production. Consider the example in the previos section. A reduce/reduce conflict is caused

by P1 and P2. To disambiguate this conflict, a non-terminal symbol V, a semantic routine disambiguate and distinct terminals Ta and Tb are added as follows:

- P1: $A \rightarrow C\ V\ Ta$
- P2: $B \rightarrow C\ V\ Tb$
- P5: $V \rightarrow \epsilon$ *disambiguate*(*Ta*, *Tb*)

The decision to select P1 or P2 is made by *disambiguate* when a reduction by P5 occurs; P5 triggers the predicate. The predicate looks at the context, or uses conditions programmed by the user, to select the production. Consequently, it inserts either token Ta or Tb in the parser's input stream as an indication of its choice. This method always works because *disambiguate* inserts either of Ta or Tb only, thus, ensuring that either P1 or P2 is selected.

This insertion is complicated by the possibility of the parser already having read in a look-ahead symbol. Thus, a modification is required in the parser-driver routine. The parser may have already read in a look-ahead token. In this case, the disambiguating token must be inserted before any look-ahead token. Thus, a lexical-token buffer is necessary. If the token inserted by the routine disambiguate is consumed before another similar insertion, then this buffer need only be a single global location. In general, for a k-token look-ahead parser, the buffer must be a queue of depth k so that all k parser look-aheads can be pushed onto the buffer-queue before disambiguate inserts any token into the input stream. The following code illustrates this technique:

```
PROCEDURE disamb(Token1, .. ,Tokenn)
    BEGIN
        IF predicate1 evaluates to true
            THEN   insert(Token1)
        ELSE
            IF predicate2 evaluates to true
                THEN   insert(Token2)
            ELSE
                .....
    END; (* procedure disamb *)

PROCEDURE insert(Token)
    BEGIN (* check if parser look-ahead already exists *)
        IF parser look-ahead token exists
            THEN
            (* save look-ahead in a buffer,
             * global to the parser driver
             * and this routine
             *)
            save(look-aheads);
        look-ahead := Token
    END; (* procedure insert *)

PROCEDURE save(Tokens)
    BEGIN (* save tokens in buffer *)
        Buffers := Tokens

    END; (* procedure save *)

The parser-driver is modified as follows:

SWITCH (Action) OF
    CASE Shift:
        IF no look-ahead token
            THEN   Symbol := Lexicalanalyzer()
        ELSE
            Symbol := look-ahead;
        State := Nextstate(State, Symbol);
        restore(Symbol);
    END; (* case shift *)

    CASE Reduce:
        (* reduce by appropriate production *)
    END; (* case reduce *)

    CASE Accept:
        (* halt; accept and stop parsing *)
    END; (* case accept *)
```

```
    CASE Error:
        (* halt rejecting; report error *)
    END; (* case error *)

PROCEDURE restore(Symbol)
    BEGIN (* check if buffer is empty *)
        IF Buffer <> Empty THEN
            BEGIN
                Symbol := Buffer;
                Buffer := Empty
            END
    END; (* procedure restore *)
```

It is not advisable to insert disambiguating terminals directly into the scanner's input buffer for the following reasons:

- If the look-ahead is already present in the scanner's buffer, it is usually in its token form. In order to un-scan the symbol, it must be recast into the original symbolic form. This re-mapping can only be done by looking up a table that contains input symbols versus token numbers. Usually, the scanner has no knowledge of the correspondence between the input symbol and its token number.
- The scanner might have done a look-ahead itself, and that must also be saved.

4. IMPLEMENTATION

These ideas were implemented for YACC [11] on a VAX 11/780 running Unix†. When the parser driver determines that it has done a valid shift, it sets a variable so that the next action will call the scanner for a new token. The modification checks if the disambiguating buffer is non-empty. If it contains a token, then on a valid shift, the variable is set to the contents of the buffer; otherwise it is set to call the scanner. The disambiguating buffer is then emptied.

The following code details the parser-driver modifications in a Unix/C environment. Similar techniques have worked for other parsers.

Filename	commands,C-code
Makefile	yacc grammar.y ed y.tab.c < yacc.make

yacc.make	/ set variable to call the scanner/
	c
	variable = buffer ? buffer : -1;
	buffer = 0;
	.
	w
	q

grammar.y	extern int buffer;

disambiguate.c	int buffer = 0;
	
	insertinput (token)
	{ /* insert token in yacc's input stream
	preserve existing look-ahead symbol, if any */
	
	buffer = variable >= 0 ? variable : 0;
	
	variable = token;
	
	}
	
	disambiguate (predicate, token1, token2)
	{ /* predicate will decide to insert token1 or token2 */
	
	if (predicate)
	insertinput (token1)
	else
	insertinput (token2)
	
	}

†Unix is a trademark of AT&T Bell Laboratories.

5. CONCLUSIONS

Conflicts in parsers such as predict–predict, shift–reduce and reduce–reduce conflicts are normally resolved by parser generators in favor of a fixed production depending on look-ahead. Often, it becomes necessary to select different productions form a conflicting set under different contexts. Disambiguating predicates are used to dynamically resolve such conflicts. Conflict resolution is delayed from parser generator time to parsing time. This refinement overcomes the shortcomings of parser generators and enhances parser-based pattern matching to allow more information to be used to drive the parse. The inclusion of semantics and other context-sensitive information in the form of grammar predicates contributes to an overall gain in convenience. Furthermore, resolution by a linear ordering of predicates permits incremental addition of productions in a grammar.

The modification requires a few lines of C code. Usually, table sizes increase about 20% for a grammar of about one thousand productions. In top-down parsing, predicates are production oriented. In contrast, in bottom-up parsing, predicates are state oriented. If p is the maximum number of productions participating in any one conflict, c is the total number of distinct conflicting configurations, and $T\nabla$, $V\nabla$ and $P\nabla$ are corresponding increments in terminals, non-terminals and productions respectively, then

- One extra terminal is required for each grammar production that participates in a conflict; the maximum number of productions in any conflicting situation indicates the increment in the number of terminals required.
- One ϵ production with a distinct nonterminal on the LHS is needed for each of c conflicting situations; for clarity, each combination of this nonterminal and a conflict resolving terminal is represented as another distinct nonterminal.
- The increment in the number of grammar productions is one plus p for each conflicting configuration.

Thus,

$$T\nabla = p$$
$$V\nabla = c + p$$
$$P\nabla = c + p*c$$

From practical experiments with large grammars ($\sim$1000 productions), $P\nabla$ ranges from about 50 to 100. This increase in the number of productions can be reduced by factoring predicates among productions. However, predicate-sharing could cause shielding of existing look-aheads [4]. This problem causes fresh blocking situations that would otherwise not occur. In particular, ϵ productions force the parser to look-ahead sooner than decisions can be made by predicates. Thus, they introduce premature parsing conflicts that are unresolvable. Alternately, to get rid of cross products, distinctly different predicates, d1, ..., dn, can be composed into a common predicate d. When d is invoked, constituent predicates d1, d2, ..., dn, in turn, are called in sequence.

There is an observable degradation on parser run-time but this effect is not crucial in practice. Predicate-directed parser generation takes about four minutes to process grammar sizes of one thousand productions. Usually, predicates are small procedures containing few lines ($\sim$10) of code, containing no loop constructs. They may be evaluated even if no conflicts are present. To guarantee any limits on run-time degradation, restrictions must be placed on the behavior of predicate routines. In comparison, parser tables can be post-processed once the generator builds them. Thus, instead of modifying existing code associated with the parser-generator, this post-processor can work in conjunction with the parser-generator. Typically, such a program is about three hundred lines of Pascal code with a running time of about 66 seconds on a VAX [4].

In closing, this paper provides a simple technique to extend off-the-shelf parsers with predicates. Also, an implementation is suggested representing a practical technique to incorporate semantic predicates within existing parsers. The technique involves modification of parser drivers alone, without modifying parser generators. Enhancing the context-free parsing structure with predicates will be interesting to compiler writers trying to implement attribute evaluation based on existing

parser generators. This modification should prove useful in syntactic analysis, error correction, code generation and peephole optimization phases of compilers [6].

REFERENCES

1. Aho A. V. and Ullman J. D., *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Englewood Cliffs, N.J. (1973).
2. Lewis P. M., Rosenkrantz D. J. and Stearns R. E., *Compiler Design Theory*. Addison-Wesley, Reading, Mass. (1976).
3. Milton D., Kirchoff L. and Rowland B. An ALL(1) compiler generator. In *Proceedings of the '79 Symposium on Compiler Construction*, ACM, Colorado (1979).
4. Fischer C. N., Ganapathi M. and Leblanc R. J. A simple and practical implementation of predicates in context free parsers. Technical Report No. 493, Computer Science Department, University of Wisconsin, Madison, (1983).
5. Ganapathi M. and Fischer C. N., Affix grammar driven code generation. *ACM Trans. Prog. Lang. Sys.* **7**, 4 (1985).
6. Ganapathi M. and Fischer C. N., Integrating code generation and peephole optimization. *Acta Inform.* **25**(1), 85–109 (1988).
7. Ganapathi M. and Fischer C. N., Attributed linear intermediate representations for retargetable code generators. *Software–Practice Exper.* **14**(4), 347–364 (1984).
8. Purdom P. and Brown C. A., Semantic routines and LR(k) parsers. *Acta Inform.* **14**(4), 299–316 (1980).
9. Watt D. A., The parsing problem for affix grammars. *Acta Inform.* **8**, 1–20 (1977).
10. Aho A. V., Sethi R. and Ullman J. D., *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass. (1986).
11. Johnson S. C., YACC—Yet another compiler compiler. *Unix System Programmer's Manual*. AT&T Bell Labs. (1975).

About the Author—MAHADEVAN GANAPATHI is affiliated with the Computer Systems Laboratory at Stanford University. His interests are in Programming Languages, Compiler Construction and Computer Architecture. His research centers on the design of intermediate representations, automatic code generation and code optimization. Much of his work includes implementation of stylish code generators for various target architectures. Related research involves automatic code generation and peephole optimization algorithms. His additional interests include theory of parsing and translation, attribute grammars and attributed parsing. His recent research efforts involve interprocedural analysis and code optimization, and also efficient implementations of functional languages.