SYNTAX PROGRAMMING

Stefan Feyock
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23185

ABSTRACT

This paper describes a new programming technology that is to syntax analysis as formal logic is to logic programming, and which we have accordingly named syntax programming. The table-driven nature of bottom-up parsers provides this approach with a number of attractive features, among which are compactness, portability, and introspective capability. Syntax programming has been successfully used for a number of applications, including expert system construction and robot control as well as non-AI problems.

1. Introduction

This paper describes a new programming technology that is to syntax analysis and parser construction as formal logic is to logic programming (LP), and which we have accordingly named syntax programming (SP). This approach is made plausible by the strong formal similarity of BNF (Backus-Naur Form) productions to Horn clauses, and attractive by the power and elegance of present-day parser construction technology.

Investigation of the syntax programming/logic programming analogy has led to results which we have found intriguing as well as encouraging. Like logic programming, SP provides a production-oriented programming framework similar to LP's Horn clauses, with the attendant inducements to orderly task composition and hierarchical program structure. More interesting, however, are the aspects of SP that differ from LP. SP programs are inherently table-driven, handle information propagation differently, and do not backtrack automatically. It thus appears that LP and SP are not direct competitors, but rather represent tailored approaches to specific types of problems. We will describe some of the strengths of SP, particularly its efficiency and capability for self-reference, and discuss work in progress toward a synthesis of SP and LP.

The research reported in this paper was supported in part by the Intelligent Systems Research Laboratory (Grant NCA 1-53), the Institute for Computer Applications in Science and Engineering, and NASA Grant NAG-1-469, all of NASA/Langley Research Center.

2. Overview of parser technology

Our development of SP has been concerned with bottom-up parsing techniques using some version of LR parsing. Top-down techniques were considered in the early phases of this project, and quickly rejected: most grammars occurring "naturally" are not parsable by the top-down approach, and must undergo various transformations to make them acceptable to such a parser. These transformations, which are usually acceptable in programming language parsing, are not feasible in many AI applications. Expert systems, for example, are frequently required to display their rules to the user in order to explain their reasoning. If these rules have been distorted for the benefit of the parser, they may no longer be comprehensible to the user. We have found no instances, on the other hand, where such distortions were necessary when using the more powerful LR (specifically LALR) parsing technique as a basis for SP.

We begin with a brief overview of the parser construction technology underlying our work. We will assume that the reader is familiar with BNF notation, and that this overview constitutes a review rather than an introduction to the basic concepts of compiler construction for him; if not, [1] is an appropriate source.

2.1. The MYSTRO Parser Generator

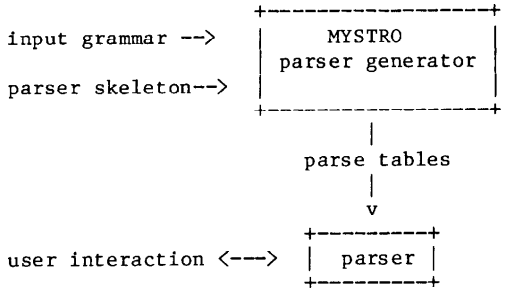
Our research has been performed using the MYSTRO parser generation system developed at the College of William and Mary [6]. This system is written in Pascal, and was therefore easily modified as required for this project. Since MYSTRO's basic organization is typical of the operation of parser generators, our exposition will consist of a description of this system.

As shown in Fig. 1, MYSTRO is a program development tool that takes as input the BNF specification of the language to be processed, along with code specifying the semantics, i.e. the operations to be performed when a particular syntactic construct is encountered.

The input to MYSTRO consists of a series of BNF productions and their associated semantics, as shown in Fig. 2.

MYSTRO analyzes this grammar and, if no errors are detected, produces a set of parse tables which are subsequently utilized by the parser to process its input. This parser is constructed by MYSTRO by inserting declarations, scanner routines, and the semantics for each production into a preexisting skeleton parser program.

An LR parser reads and stacks input text until the parse stack contains the right-hand side (rhs) of the appropriate production, as determined by the parse tables. At that time the semantics associated with that production are executed. The semantics stack referred to above is a parallel stack to the parse stack. The semantics stack elements are typically records containing a field for



The operation of MYSTRO

Fig. 1

```
* Lines with * in column 1 are comments.
* A < in column 1 signifies the beginning
* of the left-hand side (lhs)
* of a BNF production.
*
<declaration> ::= DECLARE <id> INTEGER;
* Here come the semantics,
* denoted by a blank in column 1.
   <declaration>.location :=  get_free_word;
* The notation   <symbol>.attribute
* occurring in the semantics is translated
* by MYSTRO into a semantics stack reference:
* sem_stack[stack_ptr].location := get_free_word;
   <declaration>.type := integer;
   <declaration>.value := <id>.value;
```

Typical BNF Production and associated semantics

Fig. 2

each item of interest, such as location, type, value, etc. Upon completion of the semantics code the rhs on the parse stack is replaced by the lhs of the production.

2.2. Ambiguity Resolution

It frequently happens that the grammar that is given to the parser generator is not acceptable to the parsing scheme used by the parser (MYSTRO is a LALR parser generator). MYSTRO allows the use of disambiguating predicates to deal with such contingencies. When the parser reaches a parse table entry corresponding to a shift/reduce conflict the decision is always to shift; this has worked well in practice. Upon encountering a reduce/reduce conflict, the disambiguating predicates associated with the conflicting productions are evaluated in order, and the first one whose predicate evaluates to true is used. (If none of the predicates evaluates to true, the last production in sequence, which must not have a predicate associated with it, is used.) The robot controller given in the Appendix makes extensive use of such disambiguating predicates, which are denoted by a / in column 1.

It should be noted that the disambiguating predicates play the role of metarules: when more than one rule "fires", these predicates are used to establish priority. They have access to the entire parsing environment, including the parse stack, present parse state, and the parse tables themselves, and thus can be used to do extensive introspection if desired.

3. A Syntax Programming Example

3.1. Optimizer Expert System

It is important to note that syntax programming, like logic programming, is a general-purpose software construction methodology rather than an AI-specific tool. Like LP, however, SP appears particularly suited to the requirements of a variety of AI problems. Our first example is accordingly the SP version of an expert system currently under development by J. Rogers at NASA/Langley Research Center. It represents a consultation system to be sent to prospective users of the ADS-1 General Purpose Optimization Program [11], a large package of FORTRAN-based optimizer programs for structural optimization. To use this package the user must make a number of decisions that depend on the nature of his optimization problem. In particular, he must decide on the strategy, optimizer, and type of one-dimensional search to be used. These decisions can require considerable expertise; the SP of Fig. 3 is an excerpt from an SP expert system that provides consultation to aid the user in making this decision.

This example serves to illustrate a number of points regarding syntax programming. Perhaps the most obvious is the extreme simplicity of this program. While many of the productions that constitute the actual system have been omitted, that system differs from our excerpt only in the number of productions. It should be emphasized that this system is an actual application that is to be sent out to ADS-1 users, not a contrived toy problem.

It is interesting to note that much of the simplicity results from the fact that an LR parser maintains a large amount of information relevant to the problem automatically in its states and parse tables. Consider, for example, the item set corresponding to state 3, depicted in Fig. 4. (Readers not familiar with item sets: back to [1]!) As can be seen, this state automatically records the fact that "problem is gradient" is known, as well as indicating clearly the facts yet to be established. This automatically engendered knowledge maintainance facility explains the scarcity of explicit semantics associated with most of the productions.

The second important point of this example is that it is typical of applications that require two of the fundamental advantages of syntax programs: compactness and portability. If an expert consultant system is to be sent to a user community accustomed to FORTRAN packages, it is not usually feasible to write the expert system in, say, INTERLISP or one of the expert system generators based on it, and then send the resulting system, which is apt to be quite large, to the user along with the blythe directive "Just put this system up on your machine and you'll be all set." On the other hand, the arguments against writing an expert system ad hoc in FORTRAN or other algorithmic languages are well known.

Syntax programs bypass both sets of difficulties in an elegant manner. As depicted in Fig. 1, the output of the parser generator is a set of

```
? AMBIGUOUS+ XREF- MAXIMA- SCAN- ECHO+
<answer> ::= <optimizer_choice>
  writeln(' END OF SESSION. ');
<optimizer_choice> ::= <strat:2,opt:2,ld_search:4>
* MYSTRO attaches no special significance to most special characters
* such as underscore, comma, or colon.
 writeln(' STRATEGY IS 2, OPTIMIZER IS 2, AND 1D-SEARCH IS 4. ')
*
<optimizer_choice> ::= <strat:2,opt:4,ld_search:4>
 writeln(' STRATEGY IS 2, OPTIMIZER IS 4, AND 1D-SEARCH IS 4. ')
*
*  (*** etc. --- other <optimizer_choice> alternatives not shown ***)
*
<strat:2,opt:2,ld_search:4> ::=  <strat:2> <opt:2> <search:4>
  ;
<strat:2> ::= <?ask,problem,is,1st-order> <strat:2_or_4>
  ;
<strat:2_or_4> ::=
+  <?ask,problem,is,unconstrained>
+  <?ask,problem,has,more,than,50,design,variables>
+  <?ask,problem,is,gradient>
+  <?ask,no,feasible,starting,points,can,be,found>
* + in column 1 denotes continuation of production
 ;
* (*** etc. --- remaining productions not shown ***)
```

Typical Productions from SP Optimizer Expert System

Fig. 3

```
State 3 <?ask,problem,is,gradient>
  shift [ 7] <opt:3>     ::=
       <?ask,problem,is,gradient> . <?ask,problem,is,large,or,very,large>
  shift [ 9] <opt:5>     ::=
       <?ask,problem,is,gradient> . <?ask,problem,is,medium,or,small>
                              .......
```

Portion of Typical Item Set

Fig. 4

tables, which are plugged into a parser skeleton to produce a running parser that embodies the expert system. This parser skeleton can be in whatever language is desired; we currently have a parser skeleton in Pascal, one in LISP, and are working on a FORTRAN version. Moreover, these parser skeletons are quite compact: the Pascal skeleton has fewer than 800 lines, while the LISP skeleton is less than half as large.

3.2. Turning a Parser into an SP Processor

We now turn to an important technical aspect of this example. Consider the production

```
<strat:2> ::=
   <?ask,problem,is,lst-order> <strat:2_or_4>
```

Recall that entities enclosed in < -- > are considered to be single (nonterminal or pseudoterminal) symbols, regardless of their length. The symbol <strat:2_or_4> is a nonterminal defined on the right-hand side of a separate production. The symbol <?ask,problem,is,lst-order>, on the other hand, is a pseudoterminal of a kind unique to SP. The scanner that is part of the parser skeleton has been modified to give pseudoterminals beginning with the character sequence "<?" special treatment: such pseudoterminals are deemed to be procedure calls to the (boolean) procedure named after the "?". The pseudoterminal

```
<?ask,problem,is,lst-order>
```

is thus handled by the scanner as if it were the procedure call

```
ask('problem is lst-order');
```

The procedure ask simply queries the user regarding its input string; in this case it would generate

```
IS IT TRUE THAT   problem is lst-order ?
```

If the user's response is "yes", the effect is as if the pseudoterminal had indeed been encountered as head-of-input, and the parser proceeds accordingly; if not, the scanner seeks to establish the presence (truth) of alternate pseudoterminals, as directed by its tables. The pseudoterminals are read in as part of these tables.

The parser/scanner modification we have just described is critical to SP. It effectively transforms the parsing environment from

```
input text --> scanner --> tokens --> parser
```

to

```
data base <--> scanner --> tokens --> parser
```

It is this generalization of the scanning mechanism that transforms a parser from a language processing device to a powerful general-purpose programming tool.

4. Argument passing and control flow

We now turn to a comparison of two important aspects in which SP and LP differ: argument passing and flow of control.

Information in LP is propagated up and down the "parse tree", i.e. the tree of subgoals at a given point in the computation, by means of instantiation of argument variables as forced by unification. We assume the reader is familiar with this mechanism; if not, [3] and [7] are the standard references.

We have experimented with two approaches to argument manipulation and information propagation in connection with SP. One is the methodology we have described in our overview of parsing technology, which involves associating with each production certain semantic actions which consist of essentially unrestricted code written in the language of the parser (Pascal or LISP in our case) performing arbitrary manipulation of the environment and database at the time a reduction is pending.

This information propagation scheme has several disadvantages. One of these is the aforementioned lack of discipline: the programming environment is that of the semantics language. For example, if the semantics language is Pascal, the semantics consist of essentially arbitrary Pascal code.

A second disadvantage lies in the fact that information residing in the semantics of symbols on the parse stack below the left-hand side of the current production is not accessible in an orderly fashion; in other words, all attributes are synthesized attributes.

4.1. Affix Grammars

These disadvantages have led us to investigate the feasibilty of using affix grammars [9], which promised to provide a highly disciplined information propagation method that has strong formal similarity to the arguments used for information propagation in logic programs. This approach has proven to be highly productive, and has been used in a number of SP programs. The Appendix contains an SP program with affix arguments that implements a robot controller similar to the one presented in [10]. A similar program has been used (after preprocessing as described in [9]) to control one of the robot arms in NASA/Langley Research Center's Intelligent Systems Research Laboratory.

The theory underlying LR parsing of affix grammars is far too extensive to present in this space; we must confine ourselves to a very brief overview. Consider the production

```
<puton>!object,support ::=
   <getspace>!object,support ^place
   <putat>!object,place
```

occurring in the robot controller. Intuitively, each production may be considered to be a procedure, with the inherited attributes, denoted by !, and synthesized (also called "derived") attributes, denoted by ^, playing the roles of input and output parameters respectively. Argument values are maintained on an affix stack; each "procedure" obtains its input arguments from this stack and leaves its output arguments on the stack after completion. In this case <readconds>, which occurs in an earlier production (see the Appendix), has left OBJECT and SUPPORT on the stack for <puton> to pick up.

4.2. Control Flow and Backtracking in SP

Since, like most logic programmers, we had become accustomed to thinking of backtracking as a way of life, it was with some surprise that we noted that SP's lack of backtracking caused no difficulties in the problems we attacked. This was true even though these problems had not been chosen for their lack of backtracking requirements; rather, they were research problems that were "in the air" at NASA/Langley Research Center. Nonetheless there are many problems for which a backtracking solution is natural, and it is desirable for SP programs to be able cope with them.

4.3. Semantic Backtracking

One obvious solution lies in the fact that the semantics of an SP program can contain calls to arbitrary procedures written in or callable from the language in which the parser is written. As indicated, we have implemented a parser skeleton in LISP, as well as one in Pascal that can call Lispkit LISP code [5]. Thus arbitrary LISP, Lispkit, or Pascal functions can be invoked, in particular functions that implement Prolog-like capabilities. PiL [8] provides such a function in (full) LISP, while [2] describes a purely applicative version suitable for a Lispkit LISP implementation. By this means backtracking can be confined to situations where it is necessary for searching the solution space, and need not be used in roles which are more aptly filled by other control structures.

4.4. Transforming Backtracking into Database Queries

There is a further class of situations which are implemented by means of backtracking in LP, but which turn out to be easily implementable as straightforward database searches. Consider this example:

```
sibling(x,y) :- parent(z,x),parent(z,y).
/* sibling here actually refers to
   sibling or half-sibling */
parent(z,x) :- mother(z,x).
parent(z,x) :- father(z,x).
```

Suppose the query

```
sibling(john,jane)?
```

is entered, and that the database consists of

```
mother(ann,john),
father(harry,john),
father(harry,jane)
```

(in that order). Then the subgoal parent(ann,john) will be tried first, but will have to be retracted, since parent(ann,jane) cannot be established. An LR parser-driver SP program cannot perform such a backup. A query such as this would be handled by expressing it in terms of a database query. In relational terms, the given query is equivalent to the retrieval (in an idealized relational query language)

```
parent_john intersect parent_jane
 where parent_john = {z | father(z,john) }
                   union {z | mother(z,john) }
 and   parent_jane = {z | father(z,jane) }
                   union {z | mother(z,jane) }
```

A retrieval such as this (or its equivalent in the semantics language) would then appear as part of the semantics of the SP program. We have found that it is frequently possible to eliminate backup by means of such a transformation.

5. Discussion

Having presented the concepts underlying syntax programming, we now examine some of the implications of this method of programming. These derive largely from the fact that the behavior of an LR-parser based syntax program is driven by the parse tables it reads in. We have already discussed the fact that SP programs thus inherit the compactness and high speed exhibited by LR parsers in general. We have not yet emphasized, however, one of the most important implications of this fact: since an SP program's behavior is determined by its parse tables, and since these parse tables can form part of the data base accesssed by the program, any SP program has the potential for extensive introspection into its own operation. In particular it appears straightforward to provide the user with the capability to to ask questions such as " what is your present state?" and "what are the presently legal inputs?", and have the responses generated automatically on the basis of the parse tables and parse stack. ([4] discusses the implementation of a similar capability for transition diagrams.) We consider this capability to be one of the most exciting consequences of our method, and are actively pursuing this aspect of SP.

5.1. Explanation of reasoning process

The ability to explain its reasoning to the user is an indispensable feature of expert systems. SP-based expert systems achieve this effect very neatly: since their mode of operation is based on parsing, it is trivial for them to display their parse tree, which is a representation of their "reasoning process" so far.

6. Conclusion

Syntax programming has been successfully applied to a number of problems in addition to those presented in this paper. These problems include the Tower-of-Hanoi problem, a graph manipulator, an expert system to diagnose robot end effector malfunctions, as well as a NASA-funded project to apply SP to the construction of an in-flight pilot aid system to provide malfunction consultation. This last project is currently in progress and typifies the sort of problem for which SP is well suited: the construction of rule-based expert systems that feature the compact size and high execution speed inherent in table-driven LR parsing technology.

REFERENCES

1. Aho, A., and J. Ullman, Principles of Compiler Construction, Addison-Wesley, 1977.

2. Carlsson, M., On Implementing Prolog in Functional Programming, Proc. of the 1984 International Symposium on Logic Programming, pp. 154-159 Atlantic City, NJ, February 1984.

3. Clocksin, W., and C. Mellish, Programming in Prolog, Springer-Verlag, 1981.

4. Feyock, S., Transition Diagram-based CAI/HELP Systems, International Journal of Man-Machine Studies 9, pp. 399-413, 1977.

5. Henderson, P, Functional Programming, Prentice-Hall, 1980.

6. Noonan, R., and R. Collins, The MYSTRO Parser Generator PARGEN User's Manual, Internal Report, Dept. of Computer Science, College of William and Mary, Willimamsburg, VA.

7. Kowalski, R., Logic for Problem Solving, North-Holland, 1979.

8. Wallace, R., An Easy Implementation of PiL (Prolog in LISP), SIGART Newsletter, No. 85, pp. 29-32, July 1983.

9. Watt, D. The Parsing Problem for Affix Grammars, Acta Informatica, v. 8, pp. 1-20 (1977).

10. Winston, P., and B. Horn, LISP, Addison-Wesley, 1981.

11. Vanderplaats, G., et al., ADS-1: A New General-Purpose Optimization Program, Proceedings of the AIAA/ASME/ASCE/AHS 24th Structures, Structural Dynamics, and Materials Conference, pp. 117-123, Lake Tahoe, Nevada, May 1983.

Appendix

```
<goal> ::= <readconds>^object,support <puton>!object,support  <eof>
 (apop 2) ; clear the affix stack
* This syntax program uses the LISP skeleton.
*
<readconds>^object,support ::=
 (terpri) (print "object to move?,") (apush (read)) (terpri)
 (terpri) (print "support?,") (apush (read)) (terpri)
*
<puton>!object,support ::=
   <getspace>!object,support ^place  <putat>!object,place
 (apop 3) ;
*
<putat>!object,place ::=
    <grasp>!object
    <moveobject>!object,place
    <ungrasp>!object
 (apop 3) ;
*
***** Several productions have been omitted here for brevity
*
<notsupported> ::=
* epsilon productions often cause ambiguity (usually intentional)
/ (! notsupported)
* If (! notsupported) returns true, this production fires.
 ;
***** Remaining productions have been omitted for brevity.
***** Complete program available upon request.
```