NASA Contractor Report 177955

ICASE REPORT NO. 85-36

NASA-CR-177955
19850026208

ICASE

THE PARSER GENERATOR AS A GENERAL PURPOSE TOOL

Robert E. Noonan

W. Robert Collins

Contract No. NAS1-17070

July 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NF00711

1 Report No NASA CR-177955 ICASE Report. No. 85-36	2 Government Accession No	3 Recipient's Catalog No
4 Title and Subtitle The Parser Generator As a General Purpose Tool		5 Report Date July 1985
		6 Performing Organization Code
7 Author(s) Robert E. Noonan and W. Robert Collins		8 Performing Organization Report No 85-36
		10 Work Unit No
9 Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		11 Contract or Grant No NAS1-17070
		13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14 Sponsoring Agency Code 505-31-83-01

15 Supplementary Notes
Langley Technical Monitor: J. C. South, Jr.
Final Report

Submitted to
Computing Surveys

16 Abstract

The parser generator has proven to be an an extremely useful, general purpose tool. It can be used effectively by progammers having only a knowledge of grammars and no training at all in the theory of formal parsing. Some of the application areas for which a table-driven parser can be used include interactive, query languages, menu systems, translators, and programming support tools. Each of these is illustrated by an example grammar.

17 Key Words (Suggested by Author(s)) parser generators, grammars translators, menu systems	18 Distribution Statement 61 - Computer Programming & Software Unclassified - Unlimited

19 Security Classif (of this report) Unclassified	20 Security Classif (of this page) Unclassified	21 No of Pages 36	22 Price A03

THE PARSER GENERATOR AS A GENERAL PURPOSE TOOL

Robert E. Noonan[1,2]
College of William and Mary
and
Institute for Computer Applications in Science and Engineering

W. Robert Collins[1,3]
College of William and Mary

ABSTRACT

The parser generator has proven to be an an extremely useful, general purpose tool. It can be used effectively by progammers having only a knowledge of grammars and no training at all in the theory of formal parsing. Some of the application areas for which a table-driven parser can be used include interactive, query languages, menu systems, translators, and programming support tools. Each of these is illustrated by an example grammar.

[1]Research was supported in part by NASA Langley Research Center under Grant NSG-1435.

[2]Research was supported in part by the National Aeronautics and Space Administration under NASA Contract No. NAS1-17070 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

[3]Research was supported in part by NASA Langley Research Center under Grant NAG-1-534.

N85-34521 #

1. INTRODUCTION

In the last decade there has been a great deal of research in the area known as software engineering. Two goals of this work have been the improvement in quality of the programs written and a gain in programmer productivity (however measured). Much of the resulting research has concentrated on better programming methodologies.

In the commercial area a gain in both programmer productivity and program quality has been achieved by the use of so-called 4th generation languages, such as Mantis, Nomad, etc. These languages are largely nonprocedural in nature and succeed because they restrict themselves to a fairly restricted and well understood domain. Other examples of successful application generators include screen generators, statistical packages, spreadsheet packages, etc.

It is our thesis that parser generators are another useful, yet often neglected, application generator. In the last ten years parser generators have become generally available on a large number of computers. A partial list of these would include BOBSW [Berger 1978], Lila, LR [Wetherell 1981], LRParse, MetaWare (Tm) [DeRemer 1981], Mystro [Collins 1980], and YACC [Johnson 1979].

We intend to show a number of applications of table-driven parsers, excluding compilers. These applications are not especially novel, but do not appear to be widely known. We hope to show that table-driven parsers can profitably be used by programmers with only a cursory understanding of the underlying theory. We do assume that the reader is familiar with context-free (BNF) grammars, or equivalently, syntax charts.

2. BASIC NOTIONS OF PARSING

The primary use of parsers has traditionally been in the so-called "front-end" of compilers, in which the parser is responsible for recognizing the basic constructs of the language, including statements, expressions, etc. Basic compiler theory, including parsing, is well covered in the texts [Aho 1977], [Barrett 1979], and [Waite 1983], while an introduction to just (LR) parsing is given in [Aho 1974]. However, for our purposes the reader need only understand the basic grammar and parsing material in these references, and not the details of parser construction (for example, sections 1-3 of [Aho 1974]).

A dictionary definition of the verb "parse" is:

> To resolve into its elements, as a sentence, pointing out the several parts of speech and their interrelation; to analyze and describe grammatically, as a word.

For our purposes a parser is merely a grammar-based pattern recognizer.

Consider the following BNF grammar for computing arithmetic expressions in "+" and "-", with the usual precedence and associativity:

```
<calc> ::= <expr> <end>
<expr> ::= <expr> + <term>
<expr> ::= <term>
<term> ::= <term> * <factor>
<term> ::= <factor>
<factor> ::= ( <expr> )
<factor> ::= <number>
```

The terminal symbol <end> denotes the end of the expression, while the terminal symbol <number> denotes an arbitrary number.

Grammars are useful not just for defining what sentences or strings are legal in a language, but more importantly because a derivation or parse tree imposes a definite structure on legal strings. In a parse tree each nonterminal serves as the root of a subtree, where the subtree is derived from one of the productions or rules for that nonterminal. A given string is a legal sentence of the language if it has at least one tree derivable from the goal or start symbol. The grammar is ambiguous if there exists some string with two distinct parse trees.

Consider the string "5 + 4 * 3". The only parse tree for this string according to the above grammar is:

```
          <calc>
            |
          <expr>
            |
   +--------+-------------+
   |        |             |
 <expr>     +          <term>
   |                      |
   |              +-------+------+
 <term>           |       |      |
   |            <term>    *  <factor>
<factor>          |              |
   |           <factor>      <number>
<number>          |             =3
   =5          <number>
                  =4
```

In bottom up or LR parsing, the input is recognized from left-to-right and the tree constructed from the bottom to the top. Thus, first, the <number> 5 is reduced to <factor>, then the <factor> reduced to a <term>, etc. After the seventh reduction

<term> ::= <term> * <factor>

the tree would appear as:

```
 <expr>     +          <term>
   |                      |
   |              +-------+------+
 <term>           |       |      |
   |            <term>    *  <factor>
<factor>          |              |
   |           <factor>      <number>
<number>          |             =3
   =5          <number>
                  =4
```

In order to be able to use an LR parser, one need not be familiar with the underlying theory. It is important to know only the order in which the reductions or rules will be applied in recognizing the input. In using an automatically generated parser, it is usually a simple matter to construct a sample input and to have the parser print out the order in which symbols are recognized and rules applied. Additionally, of course, a programmer must have some familiarity with grammars and have a parser generator available.

In addition, a programmer must know that the front end of a compiler is divided into three phases [Aho 1977]: scanner, parser, and semantics phase. The scanner is responsible for reading the input and coalescing characters into tokens, including keywords, identifiers, numbers, etc. Tokens are passed in a stream to the parser, which in turn constructs a parse tree of the input. As the end of a rule is recognized, the parser invokes the semantics routine, passing as arguments the number of the rule recognized and the "semantics" associated with each symbol in the right hand side of the rule.

In our university, parser-based programs are commonly built by students who have had neither a course in compiler construction nor even a course in programming languages (CS15 and CS8, respectively, in Curriculum 78 [Austing 1979]). Such programs include constructing an assembler in computer organization, a simulator in computer architecture, a database query language in database systems, etc.

More generally, table-driven parsers have been used in:

1. Compilers
   a. Parse phase
   b. Code optimization
   c. Code generation [Glanville 1978; Ganapathi 1982]
2. Other translators
3. Command languages [Campbell 1984]
4. Editors
   a. Traditional [Noonan 1985]
   b. Language-based [Fischer 1984]
5. Query languages
6. Language-based tools
7. Software design tools
8. Miscellaneous

In the sections which follow, we show some typical examples from these areas, beginning with query languages. For the sake of concreteness, we will show the grammars and associated semantics as they would be input to the Mystro system [Collins 1980; Noonan 1984]. All of these examples were adapted from larger, production applications. Most of them could easily be adapted to another parser generator.

3. QUERY LANGUAGES

One of the simplest and most popular uses of parsers is in the development of interactive, query languages. One example of this might be a desk calculator program based on an expanded version of the grammar given in the previous section. An added advantage to using a grammar is that it forces the developer to explicitly specify the user interface, often leading to a simpler and more consistent interface.

However, before proceeding with developing the desk calculator, we must explain some of the conventions used. Grammars will be given in standard BNF. Each production or grammar rule may be followed by zero or more lines of semantics in Pascal [Jensen 1975], each of which will be indented slightly. These semantic lines may refer directly to grammar symbols, using notation adapted from [Aho 1977]; Mystro provides for the translation of these references to references to a runtime semantics stack.

Our completed desk calculator program with appropriate Pascal semantics is shown in Figure 1. Three of the rules have no semantics, namely, the fourth, sixth, and last rules; this results from the fact that the nonterminal on the left will automatically inherit any value or meaning associated with a single symbol on the right (whether terminal or nonterminal). To get a running application, the desk calculator grammar need only be input to Mystro and the resulting parser compiled and linked.

Mystro requires as additional input a skeletal program or skeleton into which is inserted certain generated constants, types, Pascal code, etc. These programs are called skeletons because they need to be enhanced with application dependent code. Over the years a number of distinct skeletons for various application areas have been developed. When used with a grammar with no semantics, they result in a program that can be compiled and run without modification. Such programs are useful in debugging the grammar, determining the order in which rules will be applied, creating a set of validation tests, etc.

Table-driven parsers are commonly used as interfaces to application programs with complex input command languages, such as database query languages. More recently, parser generators have been used for the construction of menu-based systems.

```
<calc> ::= <expr> <end>
   writeln ('= ', <expr>.val);
<expr> ::= <expr> + <term>
   <expr>.val := <expr>.val + <term>.val;
<expr> ::= <term>
   ; (* <expr> inherits <term>'s val automatically *)
<term> ::= <term> * <factor>
   <term>.val := <term>.val * <factor>.val;
<term> ::= <factor>
<factor> ::= ( <expr> )
   <factor>.val := <expr>.val;
<factor> ::= <number>
   ; (* the scanner must provide for <number>.val *)
```

Figure 1: Desk Calculator

4. MENU-BASED SYSTEMS

In recent years many interactive systems have used menus rather than a more traditional command language. Here again a grammar can be used to both specify such an interface, and to implement it [Noonan 1985]. In some cases the user interface is a very small part of the total system and is used partially to hide a very large and complex system. In other cases the grammar and associated semantic code constitute the entire application.

An example of the latter type occurs when one computer is used to generate batch jobs for another. A menu interface can be used to hide the idiosyncracies of the command language of the batch computer. In one such system a DEC VAX computer is used to generate jobs for a CDC VPS-32 supercomputer; complicating the situation is the fact that a CDC Cyber computer is used as a front-end to the VPS. Of course, the VAX, Cyber, and VPS all have distinct command languages. Thus, scientists on the VAX (who are primarily mathematicians) wanting to use the VPS must know three command languages!

In this particular instance, the VAX is used to generate a job on the Cyber, which in turn generates a batch job on the VPS. Unfortunately, to save output on the Cyber or return it to the VAX, the VPS must in turn generate a batch job for the Cyber. Thus, a typical job stream contains three batch jobs in two different command languages, consisting of three pages of commands.

In order to minimize this complexity as much as possible, a menu-based program was developed, based on a BNF grammar and its supporting LR parser. The skeleton used is novel in that it does not contain a scanner. Instead the notions of lazy input [Kaye 1980] and syntactic error recovery are combined. The parser does not request any input until it reaches a state in which a token is required. At this point, the "error recovery" routine lists on the terminal the set of legal input tokens (menu options) and reads the user's response. If the user selects a valid menu choice, then the error recovery routine returns with the token chosen; otherwise the user is reprompted for input. (This approach requires the use of "default" reductions [Anderson 1973] in the parser tables; fortunately, this is a widely used space optimization technique in parser generators.)

In this particular application, for example, program data can exist on the VAX, the Cyber, or the VPS. Usually, the data is kept on the VAX if the program is (relatively) fixed and the data changes from one run to another. Conversely, the data is usually kept on the Cyber if it is large in size and infrequently changed. Data is only rarely kept on the VPS due to the limited disk space available and the inability to edit VPS files directly. The grammar rules defining the data menu appear below:

```
<data_menu> ::= C(yber <cyber_data> <get_unit> <data_menu>
<data_menu> ::= I(case <icase_data> <get_unit> <data_menu>
<data_menu> ::= V(ps <vps_data> <get_unit> <data_menu>
<data_menu> ::= N(o-more
```

In this particular implementation, the first letter of each menu option is used to select the option; entering a "C" or "c" selects the Cyber option. Unfortunately, both VPS and VAX start with the same letter, so the location of the VAX machine (ICASE) was used instead. The nonterminals <cyber_data>, <icase_data>, and <vps_data> are used to prompt for the permanent filename of the data file and generate the appropriate batch commands, while the nonterminal <get_unit> is used to prompt for the local VPS filename or Fortran unit number of the file. Right recursion is used in the first three rules to provide a loop. The "N(o-more" option is used to exit from this loop.

A similar, but more complicated structure is used for program output files. Each such file can be printed, mailed (via electronic mail), or saved. If not mailed, the file can be printed or saved on any of the three machines. The grammar rules defining these menus appear as follows:

```
<output_menu> ::= P(rint <what_file> <print_menu> <output_menu>
<output_menu> ::= M(ail <what_file> <mail_file> <output_menu>
<output_menu> ::= S(ave <what_file> <save_menu> <output_menu>
<output_menu> ::= N(o-more

<print_menu>  ::= C(yber <print_cyber>
<print_menu>  ::= I(case <print_icase>
<print_menu>  ::= V(ps <print_vps>
<print_menu>  ::= E(xit

<save_menu>  ::= C(yber <save_cyber>
<save_menu>  ::= I(case <save_icase>
<save_menu>  ::= V(ps <save_vps>
<save_menu>  ::= E(xit
```

In this instance, the "E(xit" is provided so that a choice can be undone, if desired. As before, the <what_file> nonterminal prompts for the local filename or Fortran unit number of the output file. The "print" nonterminals (excluding <print_menu>) generate the appropriate batch commands. The "save" nonterminals (excluding <save_menu>) prompt for permanent filenames and generate the appropriate batch commands.

The grammar for this application is about half the size of a grammar for Pascal, counting the number of terminal symbols, nonterminal symbols, grammar rules, and parser states. A prototype was developed rapidly and enhanced as the developers better understood the batch command languages and the users requested more options.

Such menu-based interfaces are merely special cases of systems which implement a transition diagram. One interesting example is the development of an intelligent cockpit aid for aircraft [Collins 1985]. For such applications a translator program named TD Converter was constructed which permits stylized natural language in describing the transition diagram, which is then converted automatically to a grammar. TD converter was, of course, built using a grammar-based parser.

5. TRANSLATORS

Another common application of parser generators is to build translators of various kinds, including compilers. For example, students at our university often are required to build an assembler in a course on computer organization. The only exposure of these students to grammars or syntax charts is limited to learning Pascal; despite this, they use a parser generator to construct their assemblers. We have found this approach to be superior to both the manual construction of assemblers and to the use of meta-assemblers [Collins 1983].

Assembly languages are quite distinct in structure from high level languages, and thus, pose distinct, but easily solved problems. First, assembly languages typically have a single statement per line, with explicit continuation conventions. This problem is easily solved by having the grammar represent the structure of only a single statement rather than the entire program, with the parser being called once per statement. One advantage of this approach is that it simplifies syntactic error recovery. Explicit continuation of statements across line boundaries is easily handled in the scanner.

A second problem is that assembly languages are typically fixed format, with the following general form:

```
<label>  <opcode>  <operands>  <comment>
```

Sometimes the opcode, operands, and optional comment are constrained to start in fixed columns. However, this problem is again easily handled in the scanner.

A problem with several distinct solutions is that traditional assembly languages have no reserved words, yet knowledge of the specific opcode is often crucial to further processing of the statement. One approach often used in PL/I grammars is to allow opcode names to appear explicitly in the grammar, and also to map the nonterminal <name> into both the terminal symbol <ident>, as well as any otherwise "reserved" names:

```
<operand> ::= <name>
  ...
<name> ::= <ident>
<name> ::= LOAD
<name> ::= STORE
 ...
```

An alternative approach is to map the opcodes and pseudo ops into angled terminals in the scanner:

```
<opcode> ::= <MACRO>
<opcode> ::= <LOAD>
<opcode> ::= <STORE>
 ...
```

Yet another approach is to map the opcodes into classes in the scanner based on the number and type of operands it requires. Although all these approaches will work, we prefer the last one since it results in a significantly smaller grammar and corresponding parse tables.

The last problem is that the number and type of operands depend on the opcode. For example, IBM 370 instructions [Struble 1975] are divided into storage classes, called register-register (RR), register-storage (RX), another register-storage (RS), storage-immediate (SI), and storage-storage (SS). The opcode-operand formats for each of these classes is as follows:

```
<instr> ::= <RR_op> <register> , <register>
<instr> ::= <RX_op> <register> , <storage>
<instr> ::= <RS_op> <register> , <register> , <storage>
<instr> ::= <SI_op> <storage> , <immediate>
<instr> ::= <SS_op> <storage> , <storage>
```

We can combine all of these notions to produce a grammar for (a part of) the assembly language for the IBM 370, as shown in Figure

2. To this grammar we have added some simple semantics to update the program counter, to define labels in pass 1, to generate code in pass 2, etc. One of the advantages we have found to this approach is that grammars are inherently structured and modular. Thus, it is a simple matter to start with a subset grammar, validate it on a subset of the test cases, and then expand either the grammar or the semantics to include more and more of the language being translated.

Other translators that have been developed using a parser generator include a code generator language [Donegan 1979], the UNIX (Tm) make utility [Feldman 1979], a database machine interface [Fishwick 1983], a utility for typesetting mathematics [Kernighan 1975], etc.

```
<bal> ::= <stmt> <eoln>
   pc := NewPC;
<stmt> ::= <comment>
<stmt> ::= <label> <instr>
   if pass = 1 then
   begin
      NewSymbol (symbol, pc, <label>.idname);
      DefineSymbol (symbol);
   end;
<stmt> ::= <instr>
<instr> ::= <RR_op> <register> , <register>
   begin
      NewPC := pc + 2;
      if pass = 2 then
         GenerateRR (<RR_op>.opcode, <register-1>.register,
                     <register-2>.register);
   end;  (* RR *)
<instr> ::= <RX_op> <register> , <storage>
   begin
      NewPC := pc + 4;
      if pass = 2 then
         GenerateRX (pc, <RX_op>.opcode, <register>.register,
                     <storage>.index, <storage>.base,
                     <storage>.address);
   end;  (* RX *)
```

```
<instr> ::= <RS_op> <register> , <register> , <storage>
   begin
      NewPC := pc + 4;
      if pass = 2 then
         GenerateRS (pc, <RS_op>.opcode, <register-1>.register,
                     <register-2>.register,
                     <storage>.base, <storage>.address);
   end;  (* RS *)
<instr> ::= <SI_op> <storage> , <immediate>
   begin
      NewPC := pc + 4;
      if pass = 2 then
         GenerateSI (pc, <SI_op>.opcode, <immediate>.val,
                     <storage>.base, <storage>.address);
   end;  (* SI *)
<instr> ::= <SS_op> <storage> , <storage>
   begin
      NewPC := pc + 6;
      if pass = 2 then
         GenerateSS (pc, <SS_op>.opcode, <storage-1>.length,
                     <storage-1>.base, <storage-1>.address,
                     <storage-2>.base, <storage-2>.address);
   end;  (* SS *)
<instr> ::= <unknown_op>
   error (IllegalOpcode, <unknown_op>.idname);
<register> ::= <number>
```

```
   if pass = 2 then
      <register>.register := <number>.val;
<storage> ::= <ident>
   if pass = 2 then
   begin
      FindSymbol (<storage>, <ident>.idname);
      <storage>.base := BaseReg;
      <storage>.index := 0;
   end;
<storage> ::= <ident> ( <base?> , <index?> )
   if pass = 2 then
   begin
      FindSymbol (<storage>, <ident>.idname);
      <storage>.base := <base?>.register;
      <storage>.index := <index?>.register;
   end;
<storage> ::= <ident> ( <base> )
   if pass = 2 then
   begin
      FindSymbol (<storage>, <ident>.idname);
      <storage>.base := <base>.register;
      <storage>.index := 0;
   end;
<base?> ::=
   if pass = 2 then
      <base?>.register := BaseReg;
```

```
<base?> ::= <base>

<base> ::= <register>

<index?> ::=

   if pass = 2 then

      <index?>.register := 0;

<index?> ::= <register>
```

Figure 2: IBM Assembly Language Grammar

6. PROGRAMMING LANGUAGE TOOLS

As noted by Glass [1982], the typical programming environment contains very few tools. The average programmer has at his or her disposal only compilers, editors, linkage editors, and document formatters (or word processors). However, with the aid of a parser generator a given installation can easily improve the situation. LALR(1) grammars exist for all the major programming languages, including Cobol and Fortran.

One such parser-based tool is a pretty printer [Oppen 1980], which is perhaps most useful in the maintenance phase, particularly, of older, heavily modified programs. Another useful tool is a cross-reference program, which can easily be developed from a simple parser and can be enhanced to compute various software metrics, violations of installation standards, etc. However, if too many enhancements are wanted, such a project can require almost as many resources as the building of a compiler. In fact, any set of enhancements which requires the building of a complete symbol table for the language being supported may be too ambitious for a simple support tool.

Fortunately, Browne [1978] has shown a simple, yet effective way out of this dilemma. For the language under consideration you define a set of relations which capture the important semantic aspects of the language. Then, a simple table-driven parser can be used to build a relational database for the program being analyzed. The query language for the relational database system can then be used to check for violations of installation standards, find uses of particular identifiers, produce management reports, etc. Such an approach is surprisingly effective.

Consider the case of building a simple cross referencer for Pascal. One of the problems in Pascal is that a name, such as "x," may not be unique; there may be several variables, a type, a procedure, etc., all named "x." One simple way to handle this is to keep a very simple symbol table in the database constructor. The symbol table itself can be kept as a stack of binary search trees, with one tree per scope level. Scopes are created (deleted) by pushing (popping) a level onto (off of) the stack. Other than the identifier defined, almost no other information need be kept. Then each identifier can be referenced by a (scope, identifier) pair, where each scope is given a unique number.

Similarly, statements cannot be uniquely identified by the line they are on, since Pascal allows multiple statements per line. It, thus, seems best to uniquely number each statement, as well as recording the exact position (line and column) of the statement in the program.

Figure 3 gives a grammar fragment for building some of the needed relations. The "define" routines hide the details of the interface to the relational database system used. The "lookupname" routine searches the symbol table for the identifier and returns the unique scope number for the identifier referenced.

Other uses of a language parser include augmenting the language with some needed facility. These might include adding relational database constructs, graphics primitives, etc. Such additions would then be mapped into calls to various runtime support routines. This approach allows for hiding the complexity of dealing with certain subroutine packages.

```
<while_stmt> ::= <while_clause> do <stmt>
<while_clause> ::= <while> <expr>
   begin
      definestmt (stmtno, <while>.lineno, <while>.column,
                whileclause, staticscope);
        (* relation stmt(stmt#, line#, column#, stmt_type,
                         staticscope#)  *)
      stmtno := stmtno + 1;
   end;
<while> ::= while
   usagekind := reference;
 ...
<expr> ::= <simple_expr>
<expr> ::= <simple_expr> <relop> <simple_expr>
 ...
<variable> ::= <ident>
   begin
      lookupname (<ident>.name, scope);
      definenameusage (<ident>.name, scope, stmtno,
         <ident>.lineno, <ident>.column, usagekind);
        (* relation nameusage (name, staticscope#, stmt#,
                        line#, column#, usage_kind)  *)
   end;
```

Figure 3: Pascal Cross Reference

7. CONCLUSIONS

In our environment we have found the parser generator to be extremely useful as a general application generator. It has been used to develop both end-user application programs and various software tools. We have found that a parser generator can be effectively used by programmers having only a knowledge of grammars and no training at all in the theory of formal parsing.

In particular, much of the power of the parser generator has been unleashed by the development of skeletal parsers targeted at particular application areas. At our university these include:

-- two compiler skeletons,

-- a two pass assembler,

-- a three pass assembler,

-- the query skeleton,

-- the menu skeleton,

-- two skeletons used for code generation.

We have enhanced these skeletons to facilitate debugging by those unfamiliar with parsing theory. Our students have found that the time and effort needed to learn to use these tools are well worth the investment.

NOTES

8. REFERENCES

1. Aho, Alfred V., and Johnson, Steven C. LR parsing, Computing Surveys, 6, (June 1974), 99-124.

2. Aho, Alfred V., and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley, 1977.

3. Anderson, T., Eve, J., and Horning, J. J. Efficient LR(1) parsers. Acta Informatica, 2 (1973), 12-39.

4. Austing, Richard H., et. al. Curriculum '78: recommendations for the undergraduate program in computer science -- a report of the ACM Curriculum Committee on Computer Science. CACM, 22 (March 1979), 147-165.

5. Barrett, William A., and Couch, John D. Compiler Construction: Theory and Practice. SRA, 1979.

6. Berger, W. F. BOBSW 3.0 -- a parser generator. Univ. of Texas at Austin Technical Report 87, (November 1978).

7. Browne, J. C., and Johnson, David B. FAST: a second generation program analysis system. Proceedings of 3rd Intl. Conf. on Software Engineering. (May 1978), 142-148.

8. Campbell, Roy H., and Kirslis, Peter A. The SAGA project: a system for software development. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, (April 1984).

9. Collins, W. Robert, Knight, John C., and Noonan, Robert E. A translator writing system for micro-computer high-level languages and assemblers. NASA-AIAA Workshop on Aerospace Applications of Microcomputers, (November 1980), 179-186.

10. Collins, W. Robert, Noonan, Robert E., Gregory, Samuel T., Knight, John C., and Hamm, Roy W. Comprehensive tools for assembler construction. Software -- Practice and Experience, 13, (1983), 447-451.

11. Collins, W. Robert, and Feyock, Stefan. Syntax programming, expert systems, and real-time fault diagnosis. Proceedings of the 1985 Eastern Simulation Conference, (March 1985).

12. DeRemer, F., and Pennello, T. J. The MetaWare (Tm) TWS User's Manual. MetaWare, Santa Cruz, Ca., 1981.

13. Donegan, Michael K., Noonan, Robert E., and Feyock, Stefan. A code generator generator language. SIGPLAN Symposium on Compiler Construction, (August 1979), 58-64.

14. Feldman, S. I. Make -- a program for maintaining computer

programs, Software -- Practice & Experience, 9 (April 1979), 255-265.

15. Fischer, Charles N., et. al. The Poe language-based editor project. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, (April 1984).

16. Fishwick, Paul A. HILDA: The Flexible Design and Implementation of a Database Machine Executive. MS thesis, College of William and Mary, 1983.

17. Ganapathi, Mahadevan, and Fischer, Charles N. Description-driven code generation using attribute grammars. Ninth Annual ACM Symposium on Principles of Programming Languages, (January 1982), 108-119.

18. Glanville, R. Steven, and Graham, Susan L. A new method for compiler code generation. Fifth Annual ACM Symposium on Principles of Programming Languages, (January 1978), 231-240.

19. Glass, Robert L. Recommended: a minimum standard software toolset. ACM Software Engineering Notes, 7 (October 1982), 3-13.

20. Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer-Verlag, 1975.

21. Johnson, S. C. Yacc -- Yet another compiler-compiler. _UNIX Programmer's Manual_. Bell Laboratories, (January 1979).

22. Kaye, Douglas R. Interactive Pascal input. _SIGPLAN Notices_, 15 (January 1980), 66-68.

23. Kernighan, Brian K., and Cherry, Lorinda L. A system for typesetting mathematics. _CACM_, 18 (March 1975), 182-193.

24. Noonan, Robert E., and Collins, W. Robert. The Mystro System (v. 7.2): Parser Generator Guide. College of William and Mary, 1984.

25. Noonan, Robert E., and Collins, W. Robert. Construction of a menu-based system, _Software -- Practice & Experience_, (to appear, 1985). Also ICASE Report No. 85-16, ICASE, NASA Langley Research Center, Hampton, VA.

26. Oppen, Derek C. Prettyprinting. _Trans. on Programming Languages and Systems_, 2 (October 1980), 465-483.

27. Struble, George W. _Assembler Language Programming: the IBM System/360 and 370_, 2nd edition. Addison-Wesley, 1975.

28. Waite, William M., and Goos, Gerhard. _Compiler Construction_. Springer-Verlag, 1983.

29. Wetherell, C., and Shannon, A. LR -- automatic parser generator and LR(1) parser. IEEE Trans. Soft. Engr., 7 (May 1981), 274-278.

End of Document