1986

LPARSER, an LL (1) parser generator /

James A. Femister
Lehigh University

LPARSER: AN LL(1) PARSER GENERATOR

by

James A. Femister

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1986

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Sept. 24, 1986
(date)

Samuel L. Gulden
Professor in Charge

Donald J. Hillman
Head of CS Division

[illegible]
for Chairman of CSEE Department

Table of Contents

Abstract

LPARSER, an LL(1) based parser generation system developed by the author and implemented in Turbo Pascal is discussed. Functional details and some implementation details are given. The system consists of a table generator and a skeleton parser (along with a lexical analyzer). The table generator reads in a grammar description from a text file and creates several files that are compiled along with the skeleton parser. Several aspects of LL(1) and LR(1) parsing are also discussed. LPARSER is compared with YACC, an LALR(1) based parser generator. An application of the system (an expression evaluator) is demonstrated.

Introduction

Considerable work has been done in the area of automatic, table driven parser generation. Most of this work has concentrated on LR(1) parsing techniques. LL(1) parsing, although not as widely used, has several advantages over LR techniques. LPARSER is an LL(1) based parser generator that can be used for many different applications. It is implemented in Turbo Pascal and provides a self contained parser generation system.

LL(1) vs LR(1) Parsing

The primary reason that LR(1) parsing has received so much attention by researchers is that a larger set of languages can be defined by LR grammars than by LL grammars. It has been proven that the set of LL languages (languages that can be defined by an LL(k) grammar for any $k > 0$) is a proper subset of the set of LR languages [1]. LR parsers tend to need very large parsing tables and much of the research has been involved with reducing table size. This has led to the development of SLR (Simple LR) and LALR (Look Ahead LR) grammars, which are both subsets of LR but require smaller parsing tables. LL parsers require relatively small tables that are easy to generate. The maximum number of entries in an LL(1) parse table is V x T where V is the total number of symbols in the grammar and T is the number of terminal symbols (tokens). The tables are typically sparse. Several trials yielded load factors ranging from 8% to 22%.

It has been argued that the power available in LR parsing techniques is unnecessary and that language features that require non-LL(1) constructs tend to produce grammars that are difficult to comprehend [9].

Functional Description

Overview

The LPARSER system consists of two programs: an LL(1) table generator and a skeleton parser (which includes a built in lexical analyzer), both written in Turbo Pascal. The table generator reads in an ASCII text file that contains a language definition and generates several files that contain tables and executable Pascal code. These files are automatically included in the source code of the skeleton parser when it is compiled. The resulting executable program can read a file (or computer keyboard) containing a sentence in the given language, parse it, and perform any semantic actions specified in the language definition. If the parser encounters an error it will display an appropriate error message and then halt.

Format of the Language Description File

There are five sections to the language description file:

- Terminal Symbol Declaration
- Nonterminal Symbol Declaration
- Start Symbol Declaration

- Additional Code Declaration
- Productions

Each section starts with a keyword of the form %xxxxx and must appear in the order listed above. Tokens are case sensitive. Tokens may be any length, but only the first twenty characters are significant. Text may be entered free format as long as identifiers are separated by at least one white space character (blank, tab, carriage return). Any text enclosed by the symbols (* and *) is considered a comment and is ignored. Comments may appear anywhere that white space is allowed.

Terminal Declaration

This section is used to declare names for terminal symbols that will be used in the productions. The only terminal symbols that have to be declared here are those that cannot be represented by a literal string, (i.e. identifier, integer, etc.). In some cases, it may be desirable to assign a name to a terminal or set of terminals that can by represented by a literal string. For example, the arithmetic operators, +, -, *, and / could each be treated as separate literal symbols or could by given a name, such as arithoper, that would be returned by the lexical analyzer along with a value that would indicate which of the four symbols was read. The symbol arithoper would then have to be declared in this section. Also, long literals could be assigned shorter names.

Nonterminal Symbol Declarations

All nonterminal symbols must be listed here. This is mainly to enable checks for misspelled symbols in the productions.

Start Symbol Declaration

The start symbol for the grammar is listed here. If this section is missing the first nonterminal listed in the above section is assumed to be the start symbol.

Additional Code Declaration

Any declarations (variables, constants, procedures, etc.) that are needed by the action routines are listed here enclosed by braces. All text between the braces will be copied verbatim into an insert file that will be included in the skeleton parser.

Productions

The productions of the grammar are listed here. Although all nonterminals must be declared above, literal terminals may be used freely by enclosing them in double quotes. The code for semantic actions is enclosed by braces and may be included anywhere within the right hand side of a production. To help minimize typing, multiple productions that have the same nonterminal on the left hand side (lhs) can be combined into a single compound production with vertical bars separating the different right hand sides (rhs).

Using Semantic Actions

Although LPARSER could be used to generate a parser that has no semantic actions associated with its productions, it could do little but decide whether a given input is generable by the given grammar. Most parsers must build and maintain symbol tables and other structures and in the case of a compiler, generate some sort of code. All of these tasks are accomplished by interspersing semantic actions within the grammar productions. Every production in an LPARSER grammar may have a semantic action assigned to it by placing the action code between braces and inserting it before the first symbol of the rhs of the production. Whenever the production is expanded by the parser its semantic action code is executed first before the parse stack is updated. Semantic actions that appear between symbols or at the end of the production are handled by creating a new, unique, nonterminal symbol called an action symbol and a new production for each such action. The production is always a null production with the action symbol on the lhs. The action symbol is then inserted in the original production in place of the semantic action which is then associated with the new production. Whenever the action symbol gets to the top of the parse stack it is always expanded by the null production after its semantic action is executed, effectively removing it from the stack. If a semantic stack is needed during the parse, it must be declared along with any supporting procedures in the Additional Code Declaration section and maintained through semantic action code.

Resolution of Conflicts

If the grammar that is input to LPARSER is truly LL(1) then each entry in the parse table it generates will be uniquely defined. If it is not LL(1) then one or more elements of the table will contain expand actions that reference more than one production. LPARSER uses a simple rule to decide between two conflicting productions; the production which appears first in the grammar definition file will always be selected for expansion. By carefully selecting the order of the productions in the file the grammar writer can resolve ambiguities in the most logical manner. A classical example of grammatical ambiguity is the dangling else problem [2]. As an example of this problem, consider the grammar in Fig 1 and the program fragments in Fig 2.

```
statement ::= IF condition THEN statement elsepart |
              other_statement
elsepart ::= ELSE statement | ^
```

Fig 1.

```
IF condition1 THEN                IF condition1 THEN
   IF condition2 THEN                IF condition2 THEN
      statement1                        statement1
   ELSE                           ELSE
      statement2                     statement2
```

(a) (b)

Fig 2.

The problem arises when trying to expand the symbol elsepart when the current input symbol is ELSE. Since ELSE is in both FIRST(elsepart) and FOLLOW(elsepart) and elsepart is nullable, either of the two expansions is valid. By defining the elsepart production as shown in Fig 1 with the "ELSE statement" clause listed first, the ELSE in a compound IF statement will always be associated with the innermost IF as indicated in Fig 2a. This is the desired association in almost all current programming languages. Switching the order of the two clauses in the elseif production will cause the ELSE to be associated with the outermost IF, as shown in Fig 2b.

LPARSER will always generate a parse table by resolving all conflicts that arise regardless of how many conflicts are present in the grammar.

Implementation Description

Data Structures

The most basic data type that is manipulated by the system is the symbol, which is represented in LPARSER by a Turbo Pascal char. This limits the number of symbols in a grammar to 256 but allows the use of the Turbo Pascal string type to represent strings of grammar symbols. The built in string manipulation capabilities of Turbo Pascal (concatenation, string copying, etc.) are used extensively. The LPARSER table generator could be modified to represent symbols with integers and implement string manipulation facilities with user defined procedures. This would increase the maximum number of symbols allowed in a grammar to 32767.

The two main data structures that are used by the table generator are the production table and the symbol table. The symbol table has an entry for every symbol in the input grammar. The internal value of a symbol is its index into the symbol table (which has an index type of char).

Two fields are defined in the symbol table:

1. symbol name - a twenty character string
2. symbol type - terminal, nonterminal, or null (the lambda symbol is neither a terminal or non terminal)

The production table is an array of records, each of which has three fields:

1. lhs - the single nonterminal on the lhs of the production
2. rhs - a twenty symbol string representing the rhs of the production
3. rhsset - a set of symbols that contains the same symbols as rhs. Although this field is redundant, it helps to speed up operations that are set oriented (e.g. checking to see if a certain symbol is present in the rhs of a production)

Director Set Generation

LPARSER calculates the FIRST set for each symbol of the grammar and the FOLLOW set for each nonterminal to be used later in generating the LL(1) parse table and other supporting tables. It first calculates the set of nullable nonterminals, which is used in the FIRST set calculation.

A standard bottom up algorithm, illustrated in Fig 3, is used to find the set of nullable nonterminals [3]. This algorithm, like the others that follow it, is written in Pascal-like pseudo code. The algorithm repeatedly scans through the list of productions. When a production is found that has a rhs composed entirely of already discovered nullable nonterminals (or is empty) the lhs of the production is added to the list of nullable nonterminals. The algorithm halts when no new symbols are added to the set during a complete scan of the productions.

An efficient two step algorithm is used for computing the FIRST and FOLLOW sets [4]. The algorithm uses a relation matrix with the rows and columns representing grammar symbols. The matrix is implemented as an array of sets (of 0..255). Each set represents a row in the matrix with 256 binary elements, providing a very compact Pascal implementation. The transitive closure algorithm used in the FIRST and FOLLOW set computation is a variation of Warshall's algorithm [5] that ORs entire rows of the matrix together [6]. This method has the advantage of exploiting the low level parallelism of the OR operation on most modern cpus. ORing two rows together is implemented in Pascal by performing a set union on the two sets that represent the rows. This is a built-in Turbo Pascal operation which executes quickly.

The algorithm for computing FIRST sets is shown in Fig 4. First, the boolean relation matrix, R, is initialized to false; certain entries are set to true; and then a transitive closure is done on the entire matrix. The FIRST set for a nonterminal S is found by scanning row S of the matrix R for true values in the terminal symbol columns. If S is nullable, then lambda should be added to its FIRST set.

The algorithm for computing FOLLOW sets is similar to the one for FIRST sets. The FOLLOW set algorithm uses the FIRST sets to set entries in the relation matrix.

```
nullable := []
number_of_scans := 0
repeat
   old_nullable := nullable
   for each production
      if rhs of production <= old_nullable then
         nullable := nullable + [lhs of production]
   number_of_scans := number_of_scans + 1;
until old_nullable = nullable or
      number_of_scans = number of nonterminals
```

Fig 3 - Algorithm for Finding Nullable Nonterminals

```
initialize R to false
for each production p
   i := 1
   continue := true
   while i <= length(p.rhs) and continue
      R[lhs, rhs[i]] := true
      if rhs[i] is not nullable then
         continue := false
      i := i + 1
compute transitive closure of R
```

Fig 4 - Algorithm for Calculating FIRST Sets

```
initialize R to false
R[start symbol, lambda] := true
for each production p
   for i := 1 to length(p.rhs) do
      if rhs[i] is a nonterminal then
         F := first(substr( rhs[i+1]..rhs[length(rhs)]))
         for all terminal symbols, s
            if s in F then
               R[rhs[i], s] := true
         if lambda in F then
            R[rhs[k], lhs] := true
compute transitive closure of R
```

Fig 5 - Algorithm for Calculating FOLLOW Sets

Error Reporting

The LPARSER system uses a simple table driven method for reporting errors encountered during a parse. When the parser encounters an error entry in the parsing table it calls an error routine and passes it the symbol on top of the stack and the current input symbol. The top of stack symbol is used to index a table that contains a list of all the possible input symbols that could legally appear at that point in the parse. The error routine then generates a message of the form:

```
Error at line n
Expected S1 S2 S3 ... Sn
But found  inp
```

where n, the current line number, is supplied by the lexical analyzer; S1 to Sn are the expected input symbols taken from the table; and inp is the actual current input symbol. While this method of error reporting does not provide an explanatory message tailored to each error, it has the advantage of being automatically generated from the grammar. Therefore, the grammar writer does not have to anticipate all errors that may occur, or run the risk of having some errors not handled at all [3].

Insert File/Table Description

LPARSER reads in a grammar file and generates seven insert files containing declarations, data tables, and executable code. The first five files are used by the parser and the last two are used by the lexical analyzer.

DECL.INS is the simplest of the insert files. It is a verbatim copy of the Additional Code Declaration section of the grammar input file. It is inserted in the declaration section of the skeleton parser and provides a means for the implementer to declare constants, variables, and procedures that will be used by the semantic actions. Any legal Turbo Pascal declaration code can be included here as long as no identifier name conflicts arise with the skeleton parser or lexical analyzer code.

The LL(1) parse table is contained in the file PARSETAB.INS (Fig 6). The table is implemented as a matrix of records. Each record contains an action field with one of four actions (pop stack, expand production, accept, error) and a production number that is only used with the expand production action. The array declaration is included in the insert file rather than the skeleton parser because its dimensions are determined by the grammar file. Comments marking the rows and columns with the symbols they represent are included for debugging purposes.

When the parser encounters an 'expand production' action it pops the top symbol from the parse stack and pushes the rhs of the specified production onto the stack. An array containing the rhs of each production is contained in the file PRODTAB.INS (Fig 7). Each element of the array is a string; each character in the string has the internal value of one of the symbols in the grammar. The symbolic equivalents of each production are included as comments. Before the stack is altered, any semantic actions for the production are executed. All semantic actions are contained in a case statement in the file ACTIONS.INS (Fig 8). The case statement is indexed by production numbers, each case containing the semantic action code for that production.

The table used by the error reporting routine is contained in the file ERRORTAB.INS (Fig 9). Each symbol in the grammar is assigned an element in the table which contains a string of grammar symbols. The string is made up of all the terminal symbols that are valid input symbols when the given symbol is on top of the parse stack. For a terminal symbol the string will only contain itself. For nonterminals the string will contain all the nonterminal's FIRST symbols and, if lambda is one of the FIRST symbols, it will contain all of the nonterminal's FOLLOW symbols also.

The lexical analyzer that is included in the skeleton parser requires certain constants that are derived from the input grammar. The constants are contained in the file CONSTS.INS. These constants include internal symbol values for special symbols (such as integers and identifiers) and array limits. The file LEXVALS.INS contains a lookup table of reserved symbols that is used by the lexical analyzer.

```
const
   maxparse  = 20;
type
   parse_table_type = array[1..maxparse,0..8] of
                         record
                            act:char;
                            prodno:integer
                         end;
const
   m : parse_table_type =(
(* # Row *)
(
(act:'A'; prodno:0 ),    (* # *)
(act:'E'; prodno:0 ),    (* integer *)
(act:'E'; prodno:0 ),    (* ( *)
(act:'E'; prodno:0 ),    (* ) *)
(act:'E'; prodno:0 ),    (* + *)
(act:'E'; prodno:0 ),    (* - *)
(act:'E'; prodno:0 ),    (* * *)
(act:'E'; prodno:0 )   (* / *)
),
(* ; Row *)
(
(act:'E'; prodno:0 ),    (* # *)
(act:'E'; prodno:0 ),    (* integer *)
(act:'E'; prodno:0 ),    (* ( *)
(act:'E'; prodno:0 ),    (* ) *)
(act:'E'; prodno:0 ),    (* + *)
(act:'E'; prodno:0 ),    (* - *)
(act:'E'; prodno:0 ),    (* * *)
(act:'E'; prodno:0 )   (* / *)
)
```

Fig 6 - LL(1) Parse Table - PARSETAB.INS

```
const
   maxprod  = 15;
type
   prod_table_type = array[0..maxprod] of string[15];
const
   prod_table : prod_table_type =
(
(*   0 START ::= PROGRAM # *)
#1#18,
(*   1 PROGRAM ::= EXPRESSION ; ACT1 *)
#20#2#14,
(*   2 ACT1 ::= *)
''
)
```

Fig 7 - Production Table - PRODTAB.INS

```
case prodno of 0:;
1:begin
     sp := 0; isp := 0
  end;
2:begin writeln(istk[isp]) end;
9:begin
     op := pop;
     op1 := istk[isp-1];
     op2 := istk[isp];
     case op of
        '+' : op3 := op1 + op2;
        '-' : op3 := op1 - op2;
        '*' : op3 := op1 * op2;
        '/' : op3 := op1 div op2;
     end;
     isp := isp - 1;
     istk[isp] := op3;
  end;
10:begin
      isp := isp + 1; istk[isp] := value;
   end;
12:begin push('+') end;
13:begin push('-') end;
14:begin push('*') end;
15:begin push('/') end;
end;
```

Fig 8 - Action Case Statement - ACTIONS.INS

```
const
   maxerr = 20;
type
   errtabtype = array[0..maxerr] of string[10];
const
   errtab : errtabtype =
(
'',
#1,
#2,
#9,
#3#4,
#3#4,
#3#4,
#2#5#6#7#8#9,
#3#4,
#2#5#6#7#8#9,
#1
);
```

Fig 9 - Error Table - ERRORTAB.INS

```
const
   POUND = 1;
   START = 10;
   IDENT = -11;
   INTEGER1 = 3;
   NUMTERM = 9;
```

Fig 10 - Index Constants - CONSTS.INS

```
const
   maxlex = 9;
type
   keywdtabletype = array[0..maxlex] of string30;
const
   keywdtable : keywdtabletype =
( '',
'#',
';',
'integer',
'(',
')',
'+',
'-',
'*',
'/'
);
```

Fig 11 - Lexical Values - LEXVALS.INS

Applications

The LPARSER system provides a general parsing facility that can be used in a variety of situations. Possible applications are : the front end of a compiler or interpreter, a calculator program, or a source code formatter. Fig 12 shows the grammar definition file for an expression evaluator that will read in an arithmetic expression followed by a semicolon and display its numerical value. It should serve as a simple but concrete example of how a grammar input file is actually written.

The tokens used by the expression grammar are: integers, the arithmetic operators + - * /, parentheses and the semicolon. Except for the integers, all of these symbols can by represented in the productions by literal strings and therefore do not have to be listed in the terminal declaration section. All the nonterminals used in the productions are listed in the nonterminal declaration section. The symbol ACT is not part of the expression grammar itself but is used as a semantic action symbol. All other semantic actions are inserted directly in the productions. The symbol ACT is explicitly declared and used in several places in the grammar to avoid duplicating its associated semantic

```
(* Expression Evaluator Grammar Definition *)

%terminals integer ;

%nonterminals
    FACTOR TERM TERMTAIL EXPRESSION EXPTAIL ADDOP MULOP
    PROGRAM ACT ;

%start PROGRAM

%declarations
{
   type

      string10 = string[10];

   var

      stk : array[1..20] of char;
      isp, sp : integer;
      istk : array[1..20] of integer;
      op : char;
      op1, op2, op3 : integer;

   procedure push(s:string10);

   begin
      sp := sp + 1;
      stk[sp] := s;
   end;

   function pop:string10;

   begin
      pop := stk[sp];
      sp := sp - 1;
   end;
}
```

Fig 12 - Calculator Grammar File

(continued on next page)

```
%productions

   PROGRAM ::= { sp := 0; isp := 0 } EXPRESSION ';' {
                 writeln(istk[isp]) } ;

   EXPRESSION ::= TERM EXPTAIL ;

   EXPTAIL ::= ADDOP TERM ACT EXPTAIL |  ;

   TERM ::= FACTOR TERMTAIL ;

   TERMTAIL ::= MULOP FACTOR ACT TERMTAIL |  ;

   ACT ::=
   {
      op := pop;
      op1 := istk[isp-1];
      op2 := istk[isp];
      case op of
         '+' : op3 := op1 + op2;
         '-' : op3 := op1 - op2;
         '*' : op3 := op1 * op2;
         '/' : op3 := op1 div op2;
      end;
      isp := isp - 1;
      istk[isp] := op3;
   } ;

   FACTOR ::= { isp := isp + 1; istk[isp] := value; }
                integer  |  '(' EXPRESSION ')' ;

   ADDOP ::=  { push('+') } '+'|
              { push('-') } '-';

   MULOP ::=  { push('*') } '*' |
              { push('/') } '/' ;

$END
```

Fig 12

action code.

The declarations section of the file is used to declare the variables and procedures that are needed to implement a pushdown stack for evaluating expressions. Similar code could be used to implement a semantic stack in a compiler front end.

Comparison With YACC

One of the best known and most widely used parser generators in existence is YACC, written by Steve Johnson at Bell Laboratories [7]. It reads in grammar description files similar to those used by LPARSER and generates LALR(1) parsers written in C.

Semantic actions in YACC are used the same way as they are in LPARSER with one main exception. Due to the bottom up nature of LALR parsing it is possible to automatically maintain a semantic stack that runs in parallel with the parse stack. Semantic actions in YACC can directly access and modify this semantic stack. The top down parsing mechanism of LPARSER requires that the implementer declare and maintain his own semantic stack which grows and shrinks independent of the parse stack.

The LPARSER system includes a lexical analyzer that can be modified by the user. YACC comes without a lexical analyzer but is designed to be used in conjunction with LEX, a lexical analyzer generator [10].

Future Work

Work is currently being done to extend LPARSER in several directions. One extension is to increase the maximum number of symbols allowed in an input grammar in order to accommodate compiler front ends for languages like Pascal or Modula-2. The current limit of 256 symbols is sufficient for defining the syntax of these languages but is too small to accommodate the large number of action symbols that are needed to perform semantic analysis and code generation.

Another extension is the implementation of a table driven error correction and recovery mechanism [3,8]. When confronted with a parsing error the parser will repair the error by deleting symbols from and adding symbols to the input string. Which symbols to add and delete is determined by calculating a least cost function based on a set of terminal symbol costs assigned by the parser designer.

The source code for the implementation of LPARSER is on file with Professor Samuel L. Gulden, Dept of C.S.E.E., Lehigh University.

References

[1] Aho, A.V. and J.D. Ullman. The Theory of Parsing, Translation and Compiling, Vol 2:Compiling, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] Aho, A.V., R. Sethi, and J.D. Ullman. Compilers - Principles, Techniques, and Tools, Addison-Wesley, Reading, Mass., 1986.

[3] Tremblay, J.-P. and P.G. Sorenson. The Theory and Practice of Compiler Writing, McGraw-Hill, New York, N.Y., 1985

[4] Gulden, S.L. Personal communication, 1986.

[5] Warshall, S. "A Theorem on Boolean Matrices," J. ACM, 9(1962), pp. 11-12.

[6] Reingold, E.M., J. Nievergelt, and N. Deo. Combinatorial Algorithms - Theory and Practice, Prentice-Hall, Englewood Cliffs, N.J., 1977.

[7] Johnson, S.C. "Yacc - Yet Another Compiler-Compiler," Computing Science Technical Report #32, AT&T Bell Laboratories, Murray Hill, N.J., 1978.

[8] Fischer, C.N., D.R. Milton, and S.B. Quiring. "Efficient LL(1) Error Correction and Recovery Using Only Insertions," ACTA INFORMATICA, Vol 13, No. 2, February 1980, pp. 141-154.

[9] Machanick, P. "Are LR Parsers Too Powerful?," SIGPLAN NOTICES, Vol 21, No. 6, June 1986, pp. 35-40.

[10] Lesk, M.E. "Lex - a lexical analyzer generator," Computing Science Technical Report #39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

Vita

James A. Femister was born on September 24, 1959, in Newark, N.J. to James and Dolores Femister. He earned a BS in Computing Science from Lehigh University in 1981 and worked as a research assistant at the University of Illinois Urbana-Champaign from 1981-82. Before returning to Lehigh in 1985 he worked in New York City as a consultant for Business Logic, Inc., a computer consulting firm.