dissertation, Dep. Comput. Sci., Univ. of Southwestern Louisiana, Lafayette, LA, Dec. 1978.

[17] P. Y. Ma, "Optimizing microcode produced from a high level language," Ph.D. dissertation, Dep. Elec. Comput. Eng., Oregon State Univ., Corvallis, OR.

[18] M. Tabendeh and C. V. Ramamoorthy, "Execution time (and memory) optimization in microprograms," in *Proc. 7th Annu. Workship on Microprogramming* (preprints supplement), pp. 119-127.

[19] M. Tsuchiya and C. V. Ramamoorthy, "A high level language for horizontal microprogramming," *IEEE Trans. Comput.*, vol. C-23, pp. 791-802, Aug. 1974.

[20] M. Tsuchiya and M. J. Gonzalez, "An approach to optimization of horizontal microprograms," in *Proc. 7th Workshop on Microprogramming*, Palo Alto, CA, Sept. 1974.

[21] S. S. Yau, A. C. Schowe, and M. Tsuchiya, "On storage optimization of horizontal microprograms," in *Proc. 7th Annu. Workshop on Microprogramming* (preprints), pp. 98-106.

[22] P. Ma and T. G. Lewis, "Design of a machine independent, optimizing system for emulator development," *ACM TOPLAS*, vol. 2, Apr. 1980.

[23] T. G. Lewis, K. Malik, and P. Ma, "Firmware engineering using a high level microprogramming system to implement virtual instruction set processors," presented at IFIPS Workshop, Linz, Austria, Apr. 1980.

[24] T. G. Lewis, P. Ma, K. Malik, and C. Liu, "On the problem of portable microprogramming," Dep. Comput. Sci., Oregon State Univ., Corvallis, OR, Tech. Rep. TN79-3.

[25] M. H. Halstead, *Elements of Software Science.* North-Holland: Elsevier, 1977.

Perng-Yi Richard Ma was born in Taiwan, on March 25, 1951. He received the B.S.E.E. degree from Taiwan Maritime College in 1972, the M.S.E.E. degree in 1976 and the Ph.D. degree in 1978, both in electrical engineering from Oregon State University, Corvallis.

From 1978 to 1979 he worked in the Space Division at Rockwell. Currently, he is with TRW Systems Group, Redondo Beach, CA. His present research interests include static and dynamic allocation in distributed computing systems, closed loop design among application tasks, software mapping systems, and network architecture.

Ted G. Lewis received the B.S. degree in mathematics from Oregon State University, Corvallis, in 1966, and the M.S. degree and the Ph.D. degree in computer science from Washington State University, Pullman, in 1970 and 1971, respectively.

He taught at the University of Missouri at Rolla and the University of Southwestern Louisiana, Lafayette, before joining Oregon State University, Corvallis, in 1976. He has been an officer in ACM's SIGMINI, SIGSMALL, and SIGMICRO organizations, and is the author of ten books and 35 papers on topics ranging from data structures to personal computing. His current activities include serving as an Associate Technical Editor of IEEE Computer Societies COMPUTER Magazine, and director of Project FS which conducts research in microprogramming, distributed operating systems, and software engineering.

LR—Automatic Parser Generator and LR(1) Parser

CHARLES WETHERELL AND ALFRED SHANNON

Abstract—LR is an LR(1) parser generation system. It is written entirely in portable ANS1 standard Fortran 66 and has been successfully operated on a number of computers. LR uses a powerful algorithm of Pager's to generate a space efficient parser for any LR(1) grammar. Generated parsers have been used in a variety of compilers, utility programs, and applications packages.

Index Terms—Compiler construction, LR(1) parsing, programming languages.

Manuscript received June 1, 1979; revised January 22, 1980. This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under Contract W-7405-ENG-48. This report was prepared as an account of work sponsored by the U.S. Government. Neither the United States nor the U.S. Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights. Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

C. Wetherell was with the Computing Science Group, Department of Applied Science, University of California at Davis, Livermore, CA 94550. He is now with Bell Laboratories, Murray Hill, NJ 07974.

A. Shannon is with the Lawrence Livermore Laboratory, Livermore, CA 94550.

LR is a pair of programs—an automatic parser generator and an LR(1) parser. The parser generator reads a context-free grammar in a modified BNF format and produces tables which describe an LR(1) parsing automaton. The parser is a suite of subroutines that interpret the tables to construct a parse of an input stream supplied by a (locally written) lexical analyzer. The entire system may be used to generate parsers for compilers, utility routines, command interpreters, and the like. **LR** and its predecessors have been in use at Lawrence Livermore Laboratory (LLL) for ten years. **LR**'s outstanding characteristic is the ease with which new tables can be generated to reflect a change in the language to be parsed. This flexibility is prized by programmers writing utilities and command interpreters whose input languages typically grow and change during program development. **LR** is written entirely in ANSI standard Fortran 66 [1] and requires only minor changes when moved to a new computer.

Usage

Input to **LR** is a context-free grammar. The grammar is written in a variation of Backus-Naur form (BNF) that allows a little more flexibility in the choice of symbol names. Some minor restrictions are placed on the input grammar: all productions for a nonterminal must be grouped together; terminals are exactly those symbols that have no productions; the start symbol is deduced from context; some special characters have escaped representations. Input is free-form and the modified BNF is easy to read and write. Commands to control the processing may be included in the grammar, as may comments. Fig. 1 shows typical input.

Output comes in distinct sections. Most of the sections can be turned on or off by the user. Errors are announced as soon as found. For example, the first output section is a direct echo of the input file. Any mistakes in BNF format are noted immediately. At the end of the section, errors in the formation of a legitimate grammar (e.g., too many start symbols) are also announced.

The second output section is a sorted list of the grammar symbols with index numbers assigned. The indexes are used in the generated parser during table lookups. Next comes a pretty reformatted version of the grammar. This serves as a publication copy and also tells the user *exactly* what grammar the analyzer *thinks* it is analyzing. The second service is important because mistakes and oversights in grammar specification are common. The productions are numbered and a special starting production is added. The last informational section is a grammar cross-reference table, often quite useful in grammar debugging. It is common to generate informational printouts several times before parser construction; while a grammar is growing, pretty output can be a design aid. The analysis stages can be turned off with a simple switch. Examples of these printouts are seen in Fig. 2.

LR's formal analysis phase provides the next two output sections. The first is a list of all the nonterminals that can generate the null string and a list for each nonterminal of all possible terminals that can be leftmost in a generated phrase. This information is needed in the grammar analysis and has proved useful in grammar debugging. Fig. 3 shows these tables. The second section is a state-by-state listing of the parsing automaton with kernel configurations, context sets, transitions, reductions, and look-ahead sets noted. When the grammar is LR(1), this information is largely academic since a working parser will be generated and presumably users do not care about the parser's internal details. Fig. 4 is an entire parser listing.

If the grammar is not LR(1), offending states will have error messages describing the symptoms. Unfortunately, grammar debugging is more an art than a science; experience and intelligence are usually necessary to convert a non-LR(1) grammar into an equivalent LR(1) grammar. Once offending productions are isolated and characterized, some standard methods for modifying grammars are well-known (for example, see Aho and Ullman [2, sect. 2.4.2]). On the brighter side, we have found that if a candidate grammar is not LR(1), it is almost certainly ambiguous. (This is an observation about how people write grammars; theoretically, the grammar might fall into any number of non-LR(1) but still not ambiguous classes.) Thus, debugging usually becomes a matter of spotting and removing phrases with multiple parses, a task simplified by the fact that at least some of the productions mentioned in the error flagged states will be involved in the multiple parses.

```
&c
&c   this little example grammar is given by deremer.
&2 &c   turn on null nonterminals and leftmost terminals.
&j &k &c generate fortran 66 tables
&c
<EXPRESSION> <EXPRESSION> + <TERM> &a
     <TERM> &p
<TERM> <FACTOR> ↑↑ <TERM> &a
     <FACTOR> &p
<FACTOR> <IDENTIFIER> &a
     ( <EXPRESSION> ) &p &g
```

Fig. 1. Input for a famous small grammar. The character & is used as special punctuation: notice that &a ends alternates and &p ends the productions for a nonterminal. The nonterminal that begins a block of productions is not separated from the productions by any punctuation. A generic terminal like <IDENTIFIER> may be surrounded by < >'s; it is identified as a terminal because it has no productions.

```
           t e r m i n a l s                     n o n   t e r m i n a l s

1 (                                           7 <EXPRESSION>
2 )                                           8 <FACTOR>
3 +                                           9 <SYSTEM GOAL SYMBOL>
4 <IDENTIFIER>                               10 <TERM>
5 END
6 ↑↑

                        t h e   p r o d u c t i o n s

  1 <SYSTEM GOAL SYMBOL> ::= END <EXPRESSION> END

  2 <EXPRESSION> ::= <EXPRESSION> + <TERM>
  3                | <TERM>

  4 <TERM> ::= <FACTOR> ↑↑ <TERM>
  5          | <FACTOR>

  6 <FACTOR> ::= <IDENTIFIER>
  7            | ( <EXPRESSION> )

          a   v o c a b u l a r y   c r o s s - r e f e r e n c e

(         7
)         7
+         2
<IDENTIFIER>        6
END       1     1
↑↑        4
<EXPRESSION>        1     2    -2    -3     7
<FACTOR>      4     5    -6    -7
<SYSTEM GOAL SYMBOL>           -1
<TERM>        2     3     4    -4    -5
```

Fig. 2. Informational printout for example grammar. Notice the extra production that is added at the beginning of the grammar: this production makes it easy to start and stop the parser in a standard state. A "–" in front of a line number in the cross reference indicates a definition of the symbol.

```
potentially null non-terminals

 the thead sets
   <EXPRESSION>  ( <IDENTIFIER>
   <FACTOR>  ( <IDENTIFIER>
   <SYSTEM GOAL SYMBOL>  END
   <TERM>  ( <IDENTIFIER>
```

Fig. 3. Null nonterminals and T-head sets. The example grammar has no null nonterminals. The T-head sets are listed with the nonterminal first and the leftmost terminals following.

Parser tables form the final output section. These come in three flavors: ANSI Fortran 66 DATA statements, our local LRLTRAN DATA statements, and a pure data file format. In each case, a few parameters describing the structure of the generated parser come first followed by lengthy arrays of data. Most of the information is numeric; one array is the vocabulary expressed as strings. The pure data version of the tables is to be used when the parser is to be rewritten in a language other than Fortran 66. It is a local responsibility to write a small program which will translate the data tables into a format ac-

```
Configuration Set:     1
    1 <SYSTEM GOAL SYMBOL> ::= . END <EXPRESSION> END   { END }
  the transitions:              2
Configuration Set:     2
    1 <SYSTEM GOAL SYMBOL> ::= END . <EXPRESSION> END   { END }
  the transitions:              3     4     5     6     7
Configuration Set:     3
    7 <FACTOR> ::= ( . <EXPRESSION> )   { ) + END ↑↑ }
  the transitions:              3     4     8     6     7
Configuration Set:     4
    6 <FACTOR> ::= <IDENTIFIER> .   { ) + END ↑↑ }
  the reductions:              6   ) + END ↑↑
Configuration Set:     5
    2 <EXPRESSION> ::= <EXPRESSION> . + <TERM>   { + END }
    1 <SYSTEM GOAL SYMBOL> ::= END <EXPRESSION> . END   { END }
  the transitions:              9    10
Configuration Set:     6
    5 <TERM> ::= <FACTOR> .   { ) + END }
    4 <TERM> ::= <FACTOR> . ↑↑ <TERM>   { ) + END }
  the transitions:             11
  the reductions:              5   ) + END
Configuration Set:     7
    3 <EXPRESSION> ::= <TERM> .   { ) + END }
  the reductions:              3   ) + END
Configuration Set:     8
    2 <EXPRESSION> ::= <EXPRESSION> . + <TERM>   { ) + }
    7 <FACTOR> ::= ( <EXPRESSION> . )   { ) + END ↑↑ }
  the transitions:             12     9
Configuration Set:     9
    2 <EXPRESSION> ::= <EXPRESSION> + . <TERM>   { ) + END }
  the transitions:              3     4     6    13
Configuration Set:    10
    1 <SYSTEM GOAL SYMBOL> ::= END <EXPRESSION> END .   { END }
  the reductions:              1   END
Configuration Set:    11
    4 <TERM> ::= <FACTOR> ↑↑ . <TERM>   { ) + END }
  the transitions:              3     4     6    14
Configuration Set:    12
    7 <FACTOR> ::= ( <EXPRESSION> ) .   { ) + END ↑↑ }
  the reductions:              7   ) + END ↑↑
Configuration Set:    13
    2 <EXPRESSION> ::= <EXPRESSION> + <TERM> .   { ) + END }
  the reductions:              2   ) + END
Configuration Set:    14
    4 <TERM> ::= <FACTOR> ↑↑ <TERM> .   { ) + END }
  the reductions:              4   ) + END
```

Fig. 4. Parsing automaton for example grammar. The listing has been condensed for printing by eliminating some blank lines and by moving context sets to lie within { }'s on the same lines as configurations. However, all information is exactly as generated by **LR**. In the listing a reduction is followed by its lookahead set. A transition is the name of a new state; the symbol driving the transition can be inferred from the configurations in the new state.

ceptable to the parser's host language. Figs. 5 and 6 show some data tables in ANSI standard Fortran 66 format.

When a set of tables is complete and correct, the tables are inserted into the parser subroutines to create a parser. The parser will generate a right-canonical parse of any input strings presented to it. If a syntactic error occurs in the input, the parser is guaranteed to stop as soon as any parser with one symbol look-ahead could possibly detect the error. There is no error correction or recovery capability, but many of the error correction schemes described in the literature might be grafted onto the **LR** parser with little difficulty.

Although it is not necessary, a generated parser is usually built into a one-pass translator. During execution, the parser requests tokens from the lexical analyzer (which must be locally written to recognize the language at hand), does the syntactic analysis, and as *each* production is recognized notifies semantic routines of the reduction about to occur. The parser maintains a pushdown stack of state and symbol pairs; normally the semantic routines add parallel columns to the stack to retain semantic information. The extra columns are managed by the parser, but the semantic information is totally ignored by the parser. Use of automatic parsers and the connection of parsers to lexical and semantic routines is well explained by McKeeman, Horning, and Wortman [5] and by Aho and Ullman [3].

Time and space requirements for parser generation are hard

```
      DIMENSION V(  66), VOC( 11), TRAN(   24), FTRN(   15)
      DIMENSION ENT(   14), FRED(   15), NSET(    7), PROD(     7)
      DIMENSION LHS(  7), LEN(  7), LSET(    4), LS(    8)
c
      DATA V(   1)/1H(/,V(   2)/1H)/,V(   3)/1H+/,V(   4)/1H</
      DATA V(   5)/1HI/,V(   6)/1HD/,V(   7)/1HE/,V(   8)/1HN/
      DATA V(   9)/1HT/,V(  10)/1HI/,V(  11)/1HF/,V(  12)/1HI/
      DATA V(  13)/1HE/,V(  14)/1HR/,V(  15)/1H>/,V(  16)/1HE/
      DATA V(  17)/1HN/,V(  18)/1HD/,V(  19)/1H↑/,V(  20)/1H↑/
      DATA V(  21)/1H</,V(  22)/1HE/,V(  23)/1HX/,V(  24)/1HP/
      DATA V(  25)/1HR/,V(  26)/1HE/,V(  27)/1HS/,V(  28)/1HS/
      DATA V(  29)/1HI/,V(  30)/1HO/,V(  31)/1HN/,V(  32)/1H>/
      DATA V(  33)/1H</,V(  34)/1HF/,V(  35)/1HA/,V(  36)/1HC/
      DATA V(  37)/1HT/,V(  38)/1HO/,V(  39)/1HR/,V(  40)/1H>/
      DATA V(  41)/1H</,V(  42)/1HS/,V(  43)/1HY/,V(  44)/1HS/
      DATA V(  45)/1HT/,V(  46)/1HE/,V(  47)/1HM/,V(  48)/1H /
      DATA V(  49)/1HG/,V(  50)/1HO/,V(  51)/1HA/,V(  52)/1HL/
      DATA V(  53)/1H /,V(  54)/1HS/,V(  55)/1HY/,V(  56)/1HM/
      DATA V(  57)/1HB/,V(  58)/1HO/,V(  59)/1HL/,V(  60)/1H>/
      DATA V(  61)/1H</,V(  62)/1HT/,V(  63)/1HE/,V(  64)/1HR/
      DATA V(  65)/1HM/,V(  66)/1H>/
c
      DATA VOC(  1)/   1/
      DATA VOC(  2)/   2/,VOC(  3)/   3/,VOC(  4)/   4/,VOC(  5)/  16/
      DATA VOC(  6)/  19/,VOC(  7)/  21/,VOC(  8)/  33/,VOC(  9)/  41/
      DATA VOC( 10)/  61/,VOC( 11)/  67/
```

Fig. 5. Basic declarations and vocabulary tables. This portion of the ANSI Fortran 66 tables gives the basic definitions for all the variables and provides the data for the vocabulary symbols. One of the sources of inefficiency is the loose packing of characters required for portability.

```
      DATA TRAN(    1)/     2/,TRAN(    2)/     3/,TRAN(    3)/     4/
      DATA TRAN(    4)/     5/,TRAN(    5)/     6/,TRAN(    6)/     7/
      DATA TRAN(    7)/     3/,TRAN(    8)/     4/,TRAN(    9)/     8/
      DATA TRAN(   10)/     6/,TRAN(   11)/     7/,TRAN(   12)/     9/
      DATA TRAN(   13)/    10/,TRAN(   14)/    11/,TRAN(   15)/    12/
      DATA TRAN(   16)/     9/,TRAN(   17)/     3/,TRAN(   18)/     4/
      DATA TRAN(   19)/     6/,TRAN(   20)/    13/,TRAN(   21)/     3/
      DATA TRAN(   22)/     4/,TRAN(   23)/     6/,TRAN(   24)/    14/
c
      DATA FTRN(    1)/     1/,FTRN(    2)/     2/,FTRN(    3)/     7/
      DATA FTRN(    4)/    12/,FTRN(    5)/    12/,FTRN(    6)/    14/
      DATA FTRN(    7)/    15/,FTRN(    8)/    15/,FTRN(    9)/    17/
      DATA FTRN(   10)/    21/,FTRN(   11)/    21/,FTRN(   12)/    25/
      DATA FTRN(   13)/    25/,FTRN(   14)/    25/
      DATA FTRN(   15)/    25/
c
      DATA ENT(    1)/  3/,ENT(    2)/  5/,ENT(    3)/  1/
      DATA ENT(    4)/  4/,ENT(    5)/  7/,ENT(    6)/  8/
      DATA ENT(    7)/ 10/,ENT(    8)/  7/,ENT(    9)/  3/
      DATA ENT(   10)/  5/,ENT(   11)/  6/,ENT(   12)/  2/
      DATA ENT(   13)/ 10/,ENT(   14)/ 10/
c
      DATA FRED(    1)/     1/,FRED(    2)/     1/,FRED(    3)/     1/
      DATA FRED(    4)/     1/,FRED(    5)/     2/,FRED(    6)/     2/
      DATA FRED(    7)/     3/,FRED(    8)/     4/,FRED(    9)/     4/
      DATA FRED(   10)/     4/,FRED(   11)/     5/,FRED(   12)/     5/
      DATA FRED(   13)/     6/,FRED(   14)/     7/
      DATA FRED(   15)/     8/
c
      DATA NSET(    1)/     2/,NSET(    2)/     1/,NSET(    3)/     1/
      DATA NSET(    4)/     3/,NSET(    5)/     2/,NSET(    6)/     1/
      DATA NSET(    7)/     1/
c
      DATA PROD(    1)/     6/,PROD(    2)/     5/,PROD(    3)/     3/
      DATA PROD(    4)/     1/,PROD(    5)/     7/,PROD(    6)/     2/
      DATA PROD(    7)/     4/
c
      DATA LHS(  1)/  9/,LHS(  2)/  7/,LHS(  3)/  7/,LHS(  4)/ 10/
      DATA LHS(  5)/ 10/,LHS(  6)/  8/,LHS(  7)/  8/
c
      DATA LEN(  1)/  3/,LEN(  2)/  3/,LEN(  3)/  1/,LEN(  4)/  3/
      DATA LEN(  5)/  1/,LEN(  6)/  1/,LEN(  7)/  3/
c
      DATA LSET(  1)/     1/
      DATA LSET(  2)/     4/,LSET(  3)/     8/,LSET(  4)/     9/
c
      DATA LS(     1)/     2/,LS(     2)/     3/,LS(     3)/     5/
      DATA LS(     4)/     2/,LS(     5)/     3/,LS(     6)/     5/
      DATA LS(     7)/     6/,LS(     8)/     5/
```

Fig. 6. Tables describing the parser state structure. The remainder of the tables encode the parser state structure into vectors of integers. When used on a machine where data can be packed in words, the space required for the tables shrinks drastically.

to describe for two reasons. First, both the time and the space required for parser generation depend on the number of states in the generated parser. Unfortunately, there is no known way to predict the number of states needed as some simple function of the input grammar size. Even worse, there are grammars that can cause the number of states to grow exponentially as a function of the number of symbols in the grammar. However, practical experience shows that parsers for programming languages never exhibit this cancerous state growth and that the number of states in a parser is usually about twice the number of productions in the grammar. In practice, we allow about 20 000 16-bit words for work-space and this is usually adequate; if it is not, a new run with a larger work-space usually suffices to handle even the biggest grammars. The time necessary for parser generation, by the previous argument, is at least a linear function of the grammar size. The actual **LR** implementation has a runtime dominated more by lookup algorithms and output generation than by the basic parser generation algorithm. Because of the storage regimen imposed by the Fortran implementation, we estimate that the actual running time is bounded by the square of the grammar size; that is, doubling the size of the grammar can cause as much as a factor of four increase in the runtime. Because **LR** is so portable, more precise time estimates depend on the quality of the Fortran compiler and basic machine speeds at each installation. However, we are not aware of any installation where **LR** is unacceptably slow for Pascal size grammars.

Parser running time is guaranteed to be linear in the length of the input by basic LR(1) parsing theory. Here too, though, table lookup probably dominates the time taken in the parser. The **LR** tables are arranged for easy comprehension and not for particularly fast lookup; some obvious lookup optimizations will undoubtedly occur to any user. However, any discussion of parser execution time is largely academic because it is our experience that the other parts of any translator take far more runtime than the parser ever does. Hence, runtime optimization of an **LR** generated parser is almost always a futile exercise unless all other translator parts have first been completely optimized.

Just as the parser tables have not been optimized for speed, they have not been optimized for space. For example, the tables for the Easy language (Wetherell [8]), with a grammar similar to Pascal's, take about 4000 bytes. About a quarter of this space is devoted to storing the grammar symbols in text form; this space obviously depends on the wordiness of the language designer. By taking advantage of machine-dependent packing, about 40–50 percent of this space might be saved while retaining the basic table structure. On medium to large computers, 4000 bytes of table storage is probably insignificant; on small computers, **LR** users will undoubtedly have to exercise considerable ingenuity to fit parsing tables in small spaces.

Benefits

Some virtures of the **LR** system are common to all parser generators: grammars are easy to verify and modify; parser construction is not ad hoc; given correct implementation of the generator, parser correctness is guaranteed; parsers need not be rewritten for each new application; programmers learn a standard parser system once and for all. But **LR** has some unique virtues. First, it is written in ANSI standard Fortran 66. This has forced us to use an awkward data space allocation method in the generator and to handle characters inefficiently, but in return portability is high. Only two procedures need to be changed during a move; a week will more than suffice to make **LR** run in almost any environment.

The second benefit depends more on properties of LR(1) grammars than on **LR** itself. Every deterministic language has an LR(1) grammar. Now the technical definition of LR(1) is complicated (see Aho and Ullman [2]), but informally a grammar is LR(1) if every parsing decision can be made by peeking no more than one symbol ahead during a left-to-right scan of the input. Our experience shows that language designers tend naturally to write grammars with this property. Some other classes of deterministic grammars for which parser generators have been written (e.g., LL(1), simple precedence, MSP, SLR(1), LALR(1)) place more complicated restrictions on the acceptable grammars. Since programming languages almost always have deterministic grammars, LR(1) grammars combine maximum expressiveness with broadest coverage. There are occasionally reasons to use nondeterministic grammars in programming languages, but then neither **LR** nor any other standard parser generator technique will suffice.

Third, because **LR** uses a generation algorithm due to Pager [6], a full-sized LR(1) parser is built only where the grammar warrants. In the past, LR(1) systems have built large parsers or they have saved space at the cost of accepting only a restricted class of grammars (e.g., SLR(1), LALR(1)). The Pager algorithm might be termed adaptive because it adds states to the parser only where states are necessary to distinguish similar syntactic constructions. In addition, the parser's state transitions have a list representation (DeRemer [4]) rather than a full matrix representation (Aho and Ullman [2]) at a considerable space saving. The generated parsers often have a minor flaw shared with SLR(1) and LALR(1) parsers; some errors will be announced only after some further reductions have been made. The parser will *never* read erroneous input, but the reductions may be misguided. This has not proved to be a practical difficulty and often actually aids error correction by suppressing irrelevant detail near the error location.

Experience

The XPL compiler writing system (McKeeman, Horning, and Wortman [5]) included the first widely available parser generator. The original version was based on precedence analysis and DeRemer built a revision based on his discovery of SLR(k) grammars [4]. DeRemer's program was a model for a new parser generator used by one of us during dissertation research. This generator was informally distributed from Cornell University and also brought to LLL. When its host computer left LLL, we decided to recode the analyzer in portable Fortran and to use Pager's improved algorithm. The **LR** system is the result. The parser subroutines have changed little because there has been little need to change the parser table formats.

Our choice of Fortran 66 as an implementation language may seem strange at first glance. However, pure Fortran 66 is the one language available on almost all computer systems, from smallest to largest. Although Fortran 66 made some portions of the parser generator awkward to write (and perhaps slower in execution than they might be), these difficulties were outweighed by the portability gained. We anticipate that the average user will run the generator without any concern for its structure or implementation. The parser subroutines will almost certainly be recoded into a local language or dialect;

program		size		percentage of
name	function	words	percentage	execution time
XPORT	file handler	2610	13	< 5
DDT	dynamic debugger	2600	7	10
FRAMIS	data base handler	3072	5	< 1

Fig. 7. A summary of **LR** generated parsers in use at LLL. Size information is decimal number of words in the parser on a CDC-7600 including packed tables. Parser sizes could double or triple if the tables are not packed.

here, Fortran 66 is perfectly capable of conveying the content of the short (several hundred lines) subroutine package clearly. In sum, we do not recommend Fortran 66 for its expressiveness, but its portability served us well.

Over the ten years at LLL, **LR** and its predecessors have been used by students and professional programmers for a variety of applications. Since LLL writes few compilers, most compiler usage has been by students. However, several programming languages under development have been analyzed during design by **LR** and **LR** predecessors have also been used to study programming language usage [7]. More important has been **LR**'s contribution to utility and application command languages. Several major utilities (file handler, dynamic debugger, relational database manager, picture generation language), the user interface of a timesharing system, and a number of data analysis packages have front-ends entirely controlled by **LR** generated parsers. Information on the parsers used in some of these programs is summarized in Fig. 7. The ease with which modifications can be made to a command language, the standard interface, and the speed of parser generation have all helped **LR**'s acceptance as a standard tool.

Program Availability

The **LR** system is available from

National Energy Software Center
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
(312) 972-7250

under the title **LR**. Copies of NESC software packages are generally available without charge to eligible organizations (e.g., DOE laboratories and NRC offices) or to other organizations for the cost of reproduction and distribution of the package [9].

At installation, only two procedures within **LR** need be changed. Subroutine INIT reads the command line and attaches input and output files; it must be tailored to the local environment. Subroutine CHRIND converts its character argument to an integer so that the value may be used in indexing computations; the exact computation depends on how the local Fortran system packs character values into words. Since the parser is a set of subroutines, it must be supplied with a main program and a lexical analyzer before it can function. Presumably, any practical use would also add semantic processing routines. **LR** has been checked by several Fortran 66 portability checkers and has been successfully installed under at least nine systems; we are sure of its portability.

References

[1] *American National Standard Programming Language Fortran*, ANSI X3.9-1978, Amer. Nat. Standards Inst., New York, 1978.

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

[3] —, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1978.

[4] F. L. DeRemer, "Simple LR(k) grammars," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 453-460, 1971.

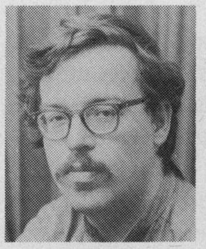
[5] W. M. McKeeman, J. J. Horning, and D. B. Wortman, *A Compiler Generator*. Englewood Cliffs, NJ: Prentice-Hall, 1970.

[6] D. Pager, "A practical general method for constructing LR(k) parsers," *Acta Informatica*, vol. 7, pp. 249-268, 1977.

[7] C. S. Wetherell, "Problems of error correction for programming languages," Ph.D. dissertation, Cornell Univ., 1975.

[8] —, *Etudes for Programmers*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

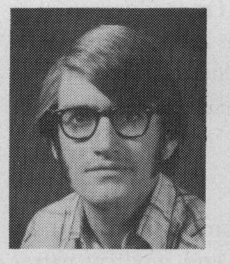
[9] C. S. Wetherell and A. S. Shannon, "Letter to the editor," *SIGPLAN Notices*, vol. 14, p. 3, Mar. 1979.



Charles Wetherell received the A.B. degree in applied mathematics from Harvard University, Cambridge, MA, in 1967 and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, in 1975.

Until the Summer of 1979, he held a joint appointment at the University of California, Davis, as an assistant professor and at the Lawrence Livermore Laboratory as a computer scientist. His research interests include programming language design and implementation. He is currently employed at Bell Laboratories, Murray Hill, NJ, where he is building a prototype compiler for Ada.

Dr. Wetherell is a member of the Association for Computing Machinery and the IEEE Computer Society.



Alfred Shannon received the B.S. degree in applied mathematics from California State University, Hayward, in 1974 and the M.S. degree in computing science from the University of California, Davis, in 1976, where he is continuing doctoral work.

He is a computer scientist at Lawrence Livermore Laboratory and works in the compiler group. His current responsibilities include code generation techniques for the Cray-1 computer. His doctoral research is centered on parser generation and automaton of compiler production.

Mr. Shannon is a member of the Assocation for Computing Machinery.