Generators for High-Speed Front-Ends

Josef Grosch
GMD Forschungsstelle für Programmstrukturen an der Universität Karlsruhe
Haid-und-Neu-Str. 7, D-7500 Karlsruhe 1, West Germany

Abstract

High-speed compilers can be constructed automatically. We present some existing tools for the generation of fast front-ends.

Rex (Regular EXpression tool) is a scanner generator whose specifications are based on regular expressions and arbitrary semantic actions written in one of the target languages C or Modula-2. As scanners sometimes have to consider the context to unambiguously recognize a token the right context can be specified by an additional regular expression and the left context can be handled by so-called start states. The generated scanners automatically compute the line and column position of the tokens and offer an efficient mechanism to normalize identifiers and keywords to upper or lower case letters. The scanners are table-driven and run at a speed of 180,000 to 195,000 lines per minute on a MC 68020 processor.

Lalr is a LALR(1) parser generator accepting grammars written in extended BNF notation which may be augmented by semantic actions expressed by statements of the target language. The generator provides a mechanism for S-attribution, that is synthesized attributes can be computed during parsing. In case of LR-conflicts, unlike other tools, *Lalr* provides not only information about an internal state consisting of a set of items but it prints a derivation tree which is much more useful to analyze the problem. Conflicts can be resolved by specifying precedence and associativity of operators and productions. The generated parsers include automatic error reporting, error recovery, and error repair. The parsers are table-driven and run at a speed of 400,000 lines per minute. Currently parsers can be generated in the target languages C and Modula-2.

Ell is a LL(1) parser generator accepting the same specification language as *Lalr* except that the grammars must obey the LL(1) property. The generated parsers include automatic error reporting, recovery, and repair like *Lalr*. The parsers are implemented following the recursive descent method and reach a speed of 450,000 lines per minute. The possible target languages are again C and Modula-2

A comparison of the above tools with the corresponding UNIX tools shows that significant improvements have been achieved thus allowing the generation of high-speed compilers.

1. The Scanner Generator Rex

The scanner generator *Rex* has been developed with the aim to combine the powerful specification method of regular expressions with the generation of highly efficient scanners. The name *Rex* stands for *regular expression tool,* reflecting the specification method.

A scanner specification consists in principle of a set of regular expressions each associated with a semantic action. Whenever a string constructed according to a regular expression is recognized in the input of the scanner its semantic action which is a sequence of arbitrary statements written in the target language is executed. To be able to recognize tokens depending on their context *Rex* provides start states to handle left context and the right context can be specified by an additional regular expression. If several regular expressions match the input characters, the longest match is preferred. If there are still several possibilities, the regular expression given first in the specification is chosen.

Rex generated scanners automatically provide the line and column position of each token. For languages like Pascal and Ada where the case of letters is insignificant tokens can be normalized to lower or upper case. There are predefined rules to skip white space like blanks, tabs, or newlines.

The generated scanners are table-driven deterministic finite automatons. The tables are compressed using the so-called comb-vector technique [ASU86]. Whereas the generator *Rex* is implemented in Modula-2 it can generate scanners in the languages C and Modula-2. Currently *Rex* is available for PCS Cadmus/UNIX and SUN/UNIX workstations.

The most outstanding feature of *Rex* is its speed. The generated scanners process nearly 200,000 lines per minute without hashing of identifiers and up to 150,000 lines per minute if hashing is applied. This is 4 times the speed of *Lex* [Les75] generated scanners. In typical cases *Rex* generated scanners are 4 times smaller then *Lex* generated ones (around 15 KB). Usually *Rex* takes only 1/10 of the time of *Lex* to generate a scanner. All figures have been measured on a MC 68020 processor.

In the following we will demonstrate the powerful specification method provided by *Rex* and present a comparison with other scanner generators.

1.1. Structure of Specification

A complete scanner specification is structured like shown in Figure 1. The regular expressions may be preceded by six sections containing arbitrary target code, which may contain declarations to be used in the semantic actions or statements for initialization and finalization of data structures. The DEFINE and START sections serve to abbreviate regular expressions by identifiers and to declare start states (see below). A complete definition of the

```
EXPORT { external declarations }
GLOBAL { global   declarations }
LOCAL  { local    declarations }
BEGIN  { initialization code   }
CLOSE  { finalization   code   }
EOF    { end of file    code   }
DEFINE   definition of regular expressions
START    definition of start states
RULE     regular expressions and semantic actions
```

Fig. 1: Structure of Specification

specification language can be found in the user manual [Gro87].

1.2. Right Context

There are languages where the strategy of the longest match fails. For example in Modula-2 the input `1..` has to be recognized as tokens "1" and "..", not as "1." and ".", which are also two legal Modula tokens. The problem can be solved using an additional regular expression to describe this situation where the right context of a token leads to an exception in the longest match strategy. Figure 2 shows the syntax used in *Rex* for regular expressions and semantic actions to describe the 4 tokens involved in the above problem. The character '/' separating two regular expressions specifies to recognize a sequence of digits only if it is followed by two dots.

```
{0-9} +                : { return SymDecimal; }
{0-9} + / ".."         : { return SymDecimal; }
{0-9} + "." {0-9} *    : { return SymReal    ; }
".."                   : { return SymRange   ; }
"."                    : { return SymDot     ; }
```

Fig. 2: Scanner Specification Using Right Context

1.3. Start States

To handle tokens whose recognition depends on the left context or to process even tokens which cannot be specified by regular expressions the scanners can have several start states. In every start state a different set of regular expressions is recognized. There is a special statement to change the current start state (yyStart). For example nested comments like in Modula can be scanned as shown in Figure 3.

```
GLOBAL  {VAR NestingLevel: CARDINAL;}

BEGIN   {NestingLevel := 0;}

EOF     {IF yyStartState = Comment THEN Error ("unclosed comment"); END;}

DEFINE  CmtCh   = - {*(\t\n}.

START   Comment

RULES
           "(*" : {INC (NestingLevel); yyStart (Comment);}

#Comment#  "*)" : {DEC (NestingLevel);
                   IF NestingLevel = 0 THEN yyStart (STD); END;}

#Comment#  "(" | "*" | CmtCh + : {}
```

Fig. 3: Scanner Specification Using Start States

1.4. Ada Quote Problem

The Ada quote problem can also be solved using start states. The problem is to scan for example

```
t'(',',',',',') as
t   '   (   ','   ,   ','   ,   ','   )  and not as
t   '('   ,   ','   ,   ','   ,   '   )
```

which are both possible sequences of Ada tokens. The correct solution again violates the longest match strategy. A careful study of the language definition reveals that single quotes only appear behind identifiers and closing parentheses. Figure 4 shows the structure of a solution. After recognizing one of these two tokens we switch to start state QUOTE which recognizes among other tokens single quotes. After all the other tokens we switch to the predefined start state STD where quotes are only accepted as delimiters for character literals. More examples of scanner specifications can be found in [Gro88a].

```
LOCAL   {char Word [256]; int  L;}

DEFINE  character = {\ -~}.
        letter    = {A-Z a-z}.
        digit     = {0-9}.

START   QUOTE

RULES

#STD# ' character ' : {
      L = GetWord (Word);
      Attribute.vChar = Word [1];
      return SymCharacterLiteral;}

#QUOTE# ' : {
      yyStart (STD);
      return SymApostrophe;}

"(" : {yyStart (STD); return SymLParenthesis;}

")" : {yyStart (QUOTE); return SymRParenthesis;}

letter (_? (letter | digit)+ )* : {
      yyStart (QUOTE); L = GetLower (Word);
      Attribute.vSymbol = MakeSymbol (Word, L);
      return SymIdentifier;}
```

Fig. 4: Scanner Specification Solving the Ada Quote Problem

1.5. Comparison of Scanner Generators

Figure 5 compares *Rex* to the classical UNIX scanner generator *Lex* [Les75] and to the new public domain remake of *Lex* called *Flex* [Pax88] (for fast *Lex*). The table compares the specification technique and the performance of the generators as well as of the generated scanners. The specification dependent numbers for generation time and scanner size are for a

	Lex	Flex	Rex
specification language	regular expressions	regular expressions	regular expressions
semantic actions	yes	yes	yes
right context	yes	yes	yes
start states	yes	yes	yes
conflict solution	longest match first rule	longest match first rule	longest match first rule
source coordinates	line	-	line + column
case normalization	-	yes	yes
predefined rules to skip white space	-	-	yes
several solutions (REJECT)	yes	yes	-
adjustment of internal arrays	by hand	automatic	automatic
scanning method	table-driven	table-driven	table-driven
table compression	comb-vector	comb-vector	comb-vector
implementation language	C	C	Modula
target languages	C	C	C, Modula
speed [lines/min.]			
without hashing	36,400	139,000	182,700
with hashing	34,700	118,000	141,400
table size [bytes]	39,200	57,300	4,400
scanner size [bytes]	43,800	64,100	11,200
generation time [sec.]	73.7	7.2	4.9
availability	UNIX	UNIX	PCS/UNIX SUN/UNIX VAX/UNIX BSD 4.2

Fig. 5: Comparison of Scanner Generators (speed measured on MC 68020 processor)

Modula-2 scanner.

2. The Parser Generator Lalr

The parser generator *Lalr* has been developed with the aim to combine a powerful specification technique for context-free languages with the generation of highly efficient parsers. As it processes the class of LALR(1) grammars we chose the name *Lalr* to express the power of the specification technique.

The grammars may be written using extended BNF constructs. Each grammar rule may be associated with a semantic action consisting of arbitrary statements written in the target language. Whenever a grammar rule is recognized by the generated parser the associated semantic action is executed. A mechanism for S-attribution (only synthesized attributes) is provided to allow communication between the semantic actions.

In case of LR-conflicts a derivation tree is printed to ease in locating the problem. The conflict can be resolved by specifying precedence and associativity for terminals and rules. Syntactic errors are handled fully automatically by the generated parsers including error reporting, recovery, and repair. The mentioned features are discussed in more detail in the following chapters.

The generated parsers are table-driven. Like in the case of *Rex* comb-vector technique is used to compress the parse tables. The generator *Lalr* is implemented in the language Modula-2. Parsers can be generated in the languages C and Modula-2. The generator uses the algorithm described by [DeP82] to compute the look-ahead sets although the algorithm published by [Ive86] promises to perform better. Currently *Lalr* is available for PCS-Cadmus/UNIX and SUN/UNIX workstations.

Parsers generated by *Lalr* are twice as fast as *Yacc* [Joh75] generated ones. They reach a speed of 400,000 lines per minute on a MC 68020 processor excluding the time for scanning. The size of the parsers is only slightly increased in comparison to *Yacc* (e. g. 37 KB for Ada), because there is a small price to be paid for the speed.

In the following we will discuss some features of *Lalr* in detail and present a comparison to other parser generators. Further information about the implementation of *Lalr* can be found in [Gro88b].

2.1. Structure of Specification

The structure of a parser specification follows the style of a *Rex* specification as shown in Figure 6. Again, there may be five sections to include target code. The TOKEN section defines the terminals of the grammar and their encoding. In the OPER (for operator) section precedence and associativity for terminals can be specified to resolve LR-conflicts. The RULE section contains the grammar rules and semantic actions. A complete definition of the specification-language can be found in the user manual [Vie88].

```
EXPORT { external declarations }
GLOBAL { global   declarations }
LOCAL  { local    declarations }
BEGIN  { initialization code   }
CLOSE  { finalization   code   }
TOKEN    coding of terminals
OPER     precedence of operators
RULE     grammar rules and semantic actions
```

Fig. 6: Structure of Specification

2.2. S-Attribution

Figure 7 shows an example for the syntax of grammar rules and semantic actions. The semantic actions may access and evaluate attributes associated with the nonterminals and terminals of the grammar rules. This attributes are currently denoted in the less readable "numeric" style of *Yacc* [Joh75].

```
expr : expr '+' expr { $0.value := $1.value + $3.value; } .
expr : expr '*' expr { $0.value := $1.value * $3.value; } .
expr : '(' expr ')'  { $0.value := $2.value; } .
expr : number        { $0.value := $1.value; } .
```

Fig. 7: Parser Specification Using S-Attribution

2.3. Ambiguous Grammars

The grammar of Figure 7 as well as the example in Figure 8 are typical examples of ambiguous grammars. Like *Yacc* we allow to resolve the resulting LR-conflicts by specifying precedence and associativity for terminals in the OPER section. Figure 9 gives an example. The lines represent increasing levels of precedence. LEFT, RIGHT, and NONE denote left-associativity, right-associativity, and no associativity. Rules can inherit the properties of a terminal with the PREC suffix.

```
stmt : 'IF' expr 'THEN' stmt                PREC LOW
     | 'IF' expr 'THEN' stmt 'ELSE' stmt    PREC HIGH .
```

Fig. 8: Ambiguous Grammar (Dangling Else)

```
OPER  LEFT '+'
      LEFT '*'
      NONE LOW
      NONE HIGH
```

Fig. 9: Resolution of LR-Conflicts Using Precedence and Associativity

2.4. LR-Conflict Message

To ease in locating the reason for LR-conflicts we adopted the method proposed by [DeP82]. Besides reporting the type of the conflict and the involved items (whatever that is for the user) like most LR parser generators a derivation tree is printed. Figure 10 shows an example. It shows how the items and the look-ahead tokens get into the conflict situation. In general there can be two trees if the derivations for the conflicting items are different. Each tree consists of 3 parts. An initial part begins at the start symbol of the grammar. At a certain node (rule) two subtrees explain the emergence of the item and the look-ahead.

Every line contains a right-hand side of a grammar rule. Usually the right-hand side is indented to start below the nonterminal of the left-hand side. To avoid line overflow dotted edges also refer to the left-hand side nonterminal and allow to shift back to the left margin. In Figure 10 the initial tree part consists of 5 lines (not counting the dotted lines). The

```
State 266

read reduce conflict

program End-of-Tokens
'PROGRAM' identifier params ';' block '.'
.................................:
:
labels consts types vars procs 'BEGIN' stmts 'END'
.....................................:
:
stmt
'IF' expr 'THEN' stmt 'ELSE' stmt
                 :
                 'IF' expr 'THEN' stmt
                 :
reduce   stmt -> 'IF' expr 'THEN' stmt.  {'ELSE'}  ?
read     stmt -> 'IF' expr 'THEN' stmt.'ELSE' stmt  ?
```

Fig. 10: Derivation Tree for an LR-Conflict (Dangling Else)

symbols 'stmt' and 'ELSE' are the roots of the other two tree parts. This location is indicated by the "unnecessary" colon in the following line. After one intermediate line the left subtree derives the conflicting items. The right subtree consists in this case only of the root node (the terminal 'ELSE') indicating the look-ahead. In general this can be a tree of arbitrary size. The LR-conflict can easily be seen from this tree fragment. If conditional statements are nested like shown there is a read reduce conflict (also called shift reduce conflict).

2.5. Error Recovery

The generated parsers include information and algorithms to handle syntax errors completely automatically. We follow the complete backtrack-free method described by [Röh76, Röh80, Röh82] and provide expressive reporting, recovery, and repair. Every incorrect input is "virtually" transformed into a syntactically correct program with the consequence of only executing a "correct" sequence of semantic actions. Therefore the following compiler phases like semantic analysis don't have to bother with syntax errors. *Lalr* provides a prototype error module which prints messages as shown in Figure 11. Internally the error recovery works as follows:

- The location of the syntax error is reported.
- All the tokens that would be a legal continuation of the program are computed and reported.
- All the tokens that can serve to continue parsing are computed. A minimal sequence of tokens is skipped until one of these tokens is found.

Source Program:

```
program test (output);
begin
   if (a = b] write (a);
end.
```

Error Messages:

```
3, 13: Error       syntax error
3, 13: Information expected symbols: ')' '*' '+' '-' '/' '<' '<='
                   '=' '<>' '>' '>=' 'AND' 'DIV' 'IN' 'MOD' 'OR'
3, 15: Information restart point
3, 15: Repair      symbol inserted : ')'
3, 15: Repair      symbol inserted : 'THEN'
```

Fig. 11: Example of Automatic Error Messages

- The recovery location is reported.
- Parsing continues in the so-called repair mode. In this mode the parser behaves as usual except that no tokens are read from the input. Instead a minimal sequence of tokens is synthesized to repair the error. The parser stays in this mode until the input token can be accepted. The synthesized tokens are reported. The program can be regarded as repaired, if the skipped tokens are replaced by the synthesized ones. With leaving the repair mode parsing continues as usual.

2.6. Comparison of Parser Generators

Figure 12 compares *Lalr* with:

- Yacc	well known from UNIX [Joh75]
- Bison	public domain remake of *Yacc* [GNU88]
- PGS	Parser Generating System also developed at Karlsruhe [GrK86, KlM88]
- Ell	recursive descent parser generator described in chapter 3.

The language dependent numbers exclude time and size for scanning and refer to experiments with a Modula-2 parser.

The measurements of the parser speed turned out to be a hairy business. The results can be influenced in many ways from:

- The hardware: We used a PCS Cadmus 9900 with a MC68020 processors running at a clock rate of 20 MHz.
- The compiler: We used the C compiler of PCS.
- The language: We used Modula-2.
- The size of the language: In the case of *Lalr* the size of the language or the size of the grammar does not influence the speed of the parser because the same table-driven algorithm and the same data structure is used in every case. This can be different for other

	Bison	Yacc	PGS	Lalr	Ell
spec. language	BNF	BNF	EBNF	EBNF	EBNF
grammar class	LALR(1)	LALR(1)	LALR(1) LR(1) SLR(1)	LALR(1)	LL(1)
semantic actions	yes	yes	yes	yes	yes
S-attribution	numeric	numeric	symbolic	numeric	-
L-attribution	-	-	-	-	planned
conflict message	state, items	state, items	state, items	derivation-tree	-
conflict solution	precedence associativity	precedence associativity	modification	precedence associativity	
chain rule elim.	-	-	yes	-	-
error recovery	by hand	by hand	automatic	automatic	automatic
error repair	-	-	yes	yes	yes
parsing method	table-driven	table-driven	table-driven	table-driven	recursive descent
table compression	comb-vector	comb-vector	comb-vector	comb-vector	-
impl. language	C	C	Pascal	Modula	Modula
target languages	C	C	C Modula Pascal Ada	C Modula	C Modula
speed [lines/min.]	105,000	184,000	200,000	385,000	437,000
table size [bytes]	8,004	10,364	11,268	11,795	-
parser size [bytes]	11,136	12,548	17,616	17,416	14,344
gen. time [sec.]	5.0	19.6	69.5	29.6	6.4
availability	UNIX	UNIX	PCS/UNIX VAX/UNIX BSD 4.2 SIEMENS/ BS2000	PCS/UNIX SUN/UNIX	PCS/UNIX SUN/UNIX

Fig. 12: Comparison of Parser Generators (speed measured on a MC 68020 processor)

parsers. For example the speed of directly coded parsers decreases with an increasing grammar size. PGS stores states in one byte if there are less than 256 states and in two bytes otherwise. This increases the speed for small grammars, too, at least on byte-addressable machines.

- The grammar style, the number of rules, especially chain rules and the like: We used the same grammar for most experiments which had as few chain rules as possible and which caused as few reduce actions as possible. This means e. g. we specified expressions in an ambiguous style like shown in Figure 7. Exceptions are *Ell* which needs an LL(1) grammar and PGS, because modifications are inelegant to resolve many ambiguities.

- The test input: We used the same large Modula program as test data in every case, of course. Nevertheless the programming style or the code "density" influence the resulting speed. This effect could be eliminated by selecting tokens per minute as measure. In spite of this we chose lines per minute as measure because we find this to be more expressive. (In the average there are 4 tokens in a line).
- The timing: We measured CPU-time and subtracted the total time and the scanner time to get the parser time.
- The semantic actions: We specified empty semantic actions for all rules in order to simulate the conditions in a realistic application. This has more consequences as one might think. It disables a short cut of *Yacc* and the chain rule elimination [WaG84] of PGS, decreasing the speed in both cases. A further experiment with PGS revealed even more problems. To allow chain rule elimination we deleted the empty semantic actions for chain rules. Surprisingly, instead of getting faster the parser was slower. The reason is that chain rule elimination increases the number of states. Accidentally we exceeded the number of 256. Now states have to be stored in two bytes instead of one. The additional memory accesses are more expensive than the win from the chain rule elimination.

3. The Parser Generator Ell

The parser generator *Ell* processes LL(1) grammars which may contain extended BNF constructs and semantic actions and generates a recursive descent parser. A mechanism for L-attribution (inherited and synthesized attributes evaluable during one preorder traversal) is to be added. Like *Lalr* syntax errors are handled fully automatic including error reporting from a prototype error module, error recovery, and error repair. The generator *Ell* is implemented in Modula-2 and can generate parsers in C and Modula-2. Those satisfied with the restricted power of LL(1) grammars may profit from the high speed of the generated parsers which lies around 450,000 lines per minute. For a detailed comparison see Figure 12.

4. Conclusion

We presented the tools *Rex*, *Lalr*, and *Ell* that allow the generation of efficient compiler front-ends. The combination of generated scanners and parsers reach speeds of more than 100,000 lines per minute or almost 2,000 lines per second. As scanning itself is one of the dominating tasks in a compiler we belief that compilers with a total performance of 1,000 lines per second can be generated automatically. Our current work concentrates on tools for semantic analysis based on attribute grammars and code generation based on pattern matching.

Acknowledgement

The author implemented *Rex* and contributed the parser skeletons in C and Modula-2 for *Lalr*. The generator program *Lalr* was written and debugged by Bertram Vielsack who also provided the experimental results for the parser generators. The parser generator *Ell* was programmed by Doris Kuske.

References

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[DeP82] F. DeRemer and T. Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 615-649.

[GNU88] GNU Project, *Bison - Manual Page*, Public Domain Software, 1988.

[GrK86] J. Grosch and E. Klein, *User Manual for the PGS-System*, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1986.

[Gro87] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[Gro88a] J. Grosch, Selected Examples of Scanner Specifications, Compiler Generation Report No. 7, GMD Forschungsstelle an der Universität Karlsruhe, Mar. 1988.

[Gro88b] J. Grosch, LALR - Generates Efficient Table-Driven Parsers, Compiler Generation Report No. 10, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1988.

[Ive86] F. Ives, Unifying View of Recent LALR(1) Lookahead Set Algorithms, *SIGPLAN Notices 21*, 7 (1986), 131-135.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[KlM88] E. Klein and M. Martin, The Parser Generating System PGS, *to appear in Software—Practice & Experience*, , 1988.

[Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[Pax88] V. Paxson, *Flex - Manual Pages*, Public Domain Software, 1988.

[Röh76] J. Röhrich, Syntax-Error Recovery in LR-Parsers, in *Informatik-Fachberichte*, vol. 1, H.-J. Schneider and M. Nagl (ed.), Springer, Berlin, 1976, 175-184.

[Röh80] J. Röhrich, Methods for the Automatic Construction of Error Correcting Parsers, *Acta Inf. 13*, 2 (1980), 115-139.

[Röh82] J. Röhrich, Behandlung syntaktischer Fehler, *Informatik Spektrum 5*, 3 (1982), 171-184.

[Vie88] B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[WaG84] W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, NY, 1984.