Lalr - A Generator for Efficient Parsers

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Lalr - A Generator for Efficient Parsers

Josef Grosch

Oct. 7, 1988

Report No. 10

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Lalr - A Generator for Efficient Parsers

J. Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
grosch@karlsruhe.gmd.de

SUMMARY

***Lalr* is a parser generator that generates very fast and powerful parsers. The design goals have been to generate portable, table-driven parsers that are as efficient as possible and which include all the features needed for practical applications. Like *Yacc* it accepts LALR(1) grammars, resolves ambiguities with precedence and associativity of operators, generates table-driven parsers, and has a mechanism for S-attribution. Unlike *Yacc* it allows grammars to be written in extended BNF, includes automatic error reporting, recovery, and repair, and generates parsers in C or Modula-2. In case of LR-conflicts, a derivation tree is printed instead of the involved states and items in order to aid the location of the problem. The parsers are two to three times faster as those of *Yacc*. Using a MC 68020 processor, 35,000 tokens per second or 580,000 lines per minute can be parsed. The sources of Lalr exist in C and in Modula-2. We describe in detail the development steps of the generated parsers. We show how software engineering methods like pseudo code and stepwise refinement can turn a parsing algorithm from a textbook into a complete and efficient implementation. We present the details of the generated parsers and show how the performance is achieved with a relatively simple and short program. We discuss important features of the generator and finally present a comparison of some parser generators.**

KEY WORDS syntactic analysis parser generator LALR(1) grammar

INTRODUCTION

The parser generator *Lalr* has been developed with the aim of combining a powerful specification technique for context-free languages with the generation of highly efficient parsers. As the parser generator 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 aid the location of 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 one of the following sections.

The generated parsers are table-driven. The comb-vector technique [ASU86] is used to compress the parse tables. The sources of the generator *Lalr* exist in the languages C and Modula-2. Parsers can be generated in the languages C and Modula-2, too. The generator uses the algorithm described by DeRemer and Pennello [DeP82] to compute the look-ahead sets. Currently *Lalr* runs on several workstations under UNIX.

Parsers generated by *Lalr* are two to three times as fast as *Yacc* [Joh75] generated ones. They reach a speed of 35,000 tokens per second or 580,000 lines per minute on a MC 68020 processor, excluding the time for scanning. The sizes of the parsers are 25 to 40% larger than those produced by *Yacc* (resulting in e. g. 37 KB for Ada). The reason is mainly due to additional code for error recovery, as well as a small space penalty for the increase of speed.

Recently some researchers report that very fast LR parsers can be achieved by generating direct code, in which the parse tables are converted into executable statements. Pennello [Pen86] generates assembly code and reports that the resulting parsers run six to ten times faster than table-driven parsers generated by an unspecified generator. He measured a speed of 500,000 lines per minute on a computer similar to a VAX 11/780 and 240,000 lines per minute on an Intel 80286. A disadvantage of this solution is the increase of the parser size by a factor of two to four, mainly because a second parser is needed for error recovery.

Whitney and Horspool [HoW89, WhH88] generate C code and report the generation of parsers between five and seven times faster than those produced by *Yacc*, resulting in a speed of 95,500 to 142,000 tokens per second or 700,000 to 2 million lines per minute for parsers for C and Pascal. The parser size lies in the same range of *Yacc*. Currently, error recovery, S-attribution, and semantic actions are not provided. In the last section, we discuss the problems with this kind of measurement and conclude that the results are not comparable.

Currently, table-driven parsers and direct code schemes are hard to compare. First, direct code schemes do not as yet implement error recovery, S-attribution, or semantic actions. However, for realistic applications these features are necessary and of course cost some time (see below). Second, the speed of the direct code schemes decreases with the grammar size. We also made experiments with direct code parsers [KlM89] and found for example in the case of Ada that an efficient table-driven parser was superior both in speed and space. A further problem is that the code directly generated for large grammars may be too big to be translated under the restrictions of usual compilers.

According to Aßmann [Ass88] a typical compiler spends 25% of the time for lexical analysis, 15% for syntactic analysis, 35% for symbol table handling and semantic analysis, and 25% for code generation. In our own experiments, syntactic analysis took 5 to 10% when all compiler parts were constructed as efficiently as possible. Therefore, improving the speed of the syntactic analysis phase can reduce the total compilation time by only a few percent. However, an engineer wants all the compiler parts to be as good as possible. An engineer also wants a parser generator to have all the features needed for practical applications, such as error recovery, S-attribution, and semantic actions.

From this engineering point of view we decided to follow the table-driven approach. This avoids the disadvantages of the direct code parsers like assembly code, large parsers, and trouble with compiler restrictions for big programs. On the other hand, fast table-driven parsers can be generated, and can include error recovery, S-attribution, and semantic actions with as few space and time expenses as possible.

This paper shows that a speed-up between two and three times faster than *Yacc* is possible using a table-driven implementation programmed in C. With this approach, the code size increases only by 25 to 40%, mainly because of added features like error recovery. The improvement is possible by carefully designing the encoding of the parser actions, the details of the parsing algorithm, and the structure of the parse tables. In the following we will discuss the implementation of the generated parsers as well as some important features of the generator *Lalr* and present a comparison to other parser generators.

THE GENERATED PARSER

This section describes the parsers generated by *Lalr*. We develop the parsing algorithm step by step as given below. We will use a pseudo code notation except in the last step where we introduce C code.

- basic LR-parsing algorithm
- LR(0) reductions
- encoding of the table entries
- semantic actions and S-attribution
- table representation and access
- error recovery
- mapping pseudo code to C

To be able to formally handle scanners, stacks, and grammars, we assume three modules. We only give the headings of the exported procedures consisting out of the procedure name and the types of the parameters and results.

```
MODULE scanner
   PROCEDURE GetToken     (): Vocabulary
   PROCEDURE GetAttribute (): <any type>
```

Each call of the function GetToken yields the next token of the input. If the input is read completely, GetToken yields the special value EndOfInput. A call of the function GetAttribute returns the attributes of the current token, such as symbol tables indices for identifiers or values of numbers.

```
MODULE stack
   PROCEDURE Push         (Stack, Element)
   PROCEDURE Pop          (Stack): Element
   PROCEDURE Top          (Stack): Element
```

A stack is defined as usual.

```
MODULE grammar
   PROCEDURE Length       (Productions): Integer
   PROCEDURE LeftHandSide (Productions): Nonterminals
   PROCEDURE Semantics    (Productions): <action statements>
```

The function Length returns the length of the right-hand side of a production. The function LeftHandSide returns the nonterminal on the left-hand side of a production. The function Semantics maps every production to some action statements. These statements should be executed whenever the associated production is recognized or reduced.

Basic LR-Parsing Algorithm

We start by looking in a textbook on compiler construction [ASU86, WaG84]. (Following [DeP82] we use the notion *read* action for what is usually called *shift* action.) An LR-Parser is controlled by a parse table implementing the following function

$$\text{Table} : \text{States} \times \text{Vocabulary} \rightarrow \text{Actions}$$

where

$$\text{Actions} = (\ \text{read} \times \text{States}\ ) \cup (\ \text{reduce} \times \text{Productions}\ ) \cup \{\ \text{halt, error}\ \}$$

Algorithm 1: Basic LR parser

```
BEGIN
   Push (StateStack, StartState)
   Terminal := GetToken ()

   LOOP
      CASE Table (Top (StateStack), Terminal) OF

      read t:     Push (StateStack, t)
                  Terminal := GetToken ()

      reduce p:   FOR i := 1 TO Length (p) DO
                     State := Pop (StateStack)
                  END
                  Nonterminal := LeftHandSide (p)

                  CASE Table (Top (StateStack), Nonterminal) OF
                     read t: Push (StateStack, t)
                  END

      error:      ErrorRecovery ()

      halt:       EXIT

      END
   END
END
```

The table controls a general LR parsing algorithm (Algorithm 1). This is a pushdown automaton which remembers the parsing of the left context in a stack of states. Depending on the state on top of the stack and on the actual look-ahead symbol, it accesses the parse table and executes the obtained action.

There are two places where the table is accessed. Depending on the second argument of the function Table, we will call these places *terminal* and *nonterminal* accesses. At a terminal access, all four actions are possible. Nonterminals appear after reductions, only a read action is possible at a nonterminal access: no error action can occur. Nevertheless we use a CASE statement at a nonterminal access to decode the action, because there will be more cases in the next step.

LR(0) Reductions

The textbooks also tell us about LR(0) reductions or read-reduce actions [WaG84]. For most languages 50% of the states are LR(0) reduce states, in which a reduce action is determined without examining the look-ahead token. The introduction of a read-reduce action is probably one of the best available optimizations. This saves many table accesses and considerable table space.

$$\text{Actions} = (\text{read} \times \text{States}) \cup (\text{reduce} \times \text{Productions}) \cup (\text{read-reduce} \times \text{Productions}) \cup \{\text{halt}, \text{error}\}$$

As we did not find an LR parsing algorithm that uses read-reduce actions in the literature we present our version in Algorithm 2. The character '_' stands for a value that doesn't matter. A read-reduce action can occur at both places of table access. In the terminal case, we combine a read and a reduce action. In the nonterminal case we have to "virtually" read the nonterminal and then to execute a reduction. This is accomplished by the inner LOOP

Algorithm 2: LR parser with LR(0) reductions

```
BEGIN
   Push (StateStack, StartState)
   Terminal := GetToken ()

   LOOP
      CASE Table (Top (StateStack), Terminal) OF

      read t:        Push (StateStack, t)
                     Terminal := GetToken ()

      read-reduce p: Push (StateStack, _)
                     Terminal := GetToken ()
                     GOTO L

      reduce p:  L:  LOOP
                        FOR i := 1 TO Length (p) DO
                           State := Pop (StateStack)
                        END
                        Nonterminal := LeftHandSide (p)

                        CASE Table (Top (StateStack), Nonterminal) OF

                           read t:        Push (StateStack, t)
                                          EXIT

                           read-reduce p: Push (StateStack, _)

                        END
                     END

      error:         ErrorRecovery ()

      halt:          EXIT

      END
   END
END
```

statement. A reduce action can be followed by a series of read-reduce actions. The inner LOOP statement is terminated on reaching a read action.

This solution with an inner LOOP statement has two advantages: First, as there are only two cases within the second CASE statement, it can be turned into an IF statement. Second, there is no need to differentiate between read and read-reduce with respect to terminal or nonterminal table access, as these different kinds of access occur at two different places.

Encoding of the Table Entries

The next problem is how to efficiently represent the table entries. Conceptually, these entries are pairs consisting of an action indicator and a number denoting a state or a production. A straightforward representation using a record wastes too much space and is too hard to decode for interpretation. It is advantageous to represent the table entries by simple integers in the following way:

Table : States $\times$ Vocabulary $\rightarrow$ INTEGER

where

action	integer value	constant name
error	0	NoState
read 1	1	
...		
read n	n	
read-reduce 1	n + 1	FirstReadReduce
...		
read-reduce m	n + m	
reduce 1	n + m + 1	FirstReduce
...		
reduce m	n + m + m	
halt	n + m + m + 1	Stop

The constant n stands for the number of states and the constant m for the number of productions. As it is not possible to "read-reduce" every production, not all numbers between n+1 and n+m are used. The advantage of this solution is that the table entries do not take much space, and that to decode them the CASE statements can be turned into three simple comparisons as shown in Algorithm 3. We neglect the actions error and halt for the moment, and reintroduce them in later sections.

Semantic Actions and S-Attribution

For the implementation of an S-attribution which is evaluated during LR parsing, the parser has to maintain a second stack for the attribute values. This stack grows and shrinks in parallel with the existing stack for the states. Algorithm 4 shows how these features have to be added.

In order to be able to access the attributes of all right-hand side symbols, we need a stack with direct access, because the attribute for the i-th symbol (counting from 1) has to be accessed by

```
AttributeStack [StackPointer + i]
```

The action statement can compute an attribute value for the left-hand side of the production which has to be assigned to the variable SynAttribute (for a synthesized attribute). After executing the action statements, this value is pushed onto the attribute stack by the parser to reestablish the invariant of the algorithm. The attribute values for terminals have to be provided by the scanner.

Algorithm 3: LR parser with actions encoded by integers

```
BEGIN
   Push (StateStack, StartState)
   Terminal := GetToken ()

   LOOP
      State := Table (Top (StateStack), Terminal)

      IF State >= FirstReadReduce THEN          /* reduce or read-reduce? */

         IF State < FirstReduce THEN            /* read-reduce */
            Push (StateStack, _)
            Terminal := GetToken ()
            Production := State - FirstReadReduce + 1
         ELSE                                   /* reduce */
            Production := State - FirstReduce + 1
         END

         LOOP                                   /* reduce */
            FOR i := 1 TO Length (Production) DO
               State := Pop (StateStack)
            END
            Nonterminal := LeftHandSide (Production)

            State := Table (Top (StateStack), Nonterminal)

            IF State < FirstReadReduce THEN     /* read */
               Push (StateStack, State)
               EXIT
            ELSE                                /* read-reduce */
               Push (StateStack, _)
            END
         END

      ELSE                                      /* read */
         Push (StateStack, State)
         Terminal := GetToken ()
      END
   END
END
```

Algorithm 4: LR parser with action statements and S-attribution

```
BEGIN
   Push (AttributeStack, _)
   Push (StateStack, StartState)
   Terminal := GetToken ()

   LOOP
      State := Table (Top (StateStack), Terminal)

      IF State >= FirstReadReduce THEN          /* reduce or read-reduce? */

         IF State < FirstReduce THEN            /* read-reduce */
            Push (AttributeStack, GetAttribute ())
            Push (StateStack, _)
            Terminal := GetToken ()
            Production := State - FirstReadReduce + 1
         ELSE                                   /* reduce */
            Production := State - FirstReduce + 1
         END

         LOOP                                   /* reduce */
            FOR i := 1 TO Length (Production) DO
               Dummy := Pop (AttributeStack)
               State := Pop (StateStack)
            END
            Nonterminal := LeftHandSide (Production)

            Semantics (Production) ()

            State := Table (Top (StateStack), Nonterminal)

            IF State < FirstReadReduce THEN     /* read */
               Push (AttributeStack, SynAttribute)
               Push (StateStack, State)
               EXIT
            ELSE                                /* read-reduce */
               Push (AttributeStack, SynAttribute)
               Push (StateStack, _)
            END
         END

      ELSE                                      /* read */
         Push (AttributeStack, GetAttribute ())
         Push (StateStack, State)
         Terminal := GetToken ()
      END
   END
END
```

As many of the terminals don't bear any attribute, most of the associated Push operations could be replaced by pushing a dummy value: that is, an increment of the stack pointer would be enough. To be able to distinguish between two kinds of Push operations, two kinds of read actions would be necessary. To make the decision would cost an extra check for every token. We did not use this optimization because we believe that the extra checks cost as much as the saved assignments.

How do we implement the mapping of a production to the associated action statements? Of course, the natural solution is a CASE statement. The access to the Length and the LeftHandSide also depends on the production. One choice would be to access two arrays. As array access is relatively expensive, we can move these computations into the CASE statement which we already need for the action statements. In each case, these computations are turned into constants. The FOR loop disappears anyway, because it suffices to decrement the stack pointers. The code common to all reductions is not included in the CASE statement but follows afterwards. We also factor out the code common to all parts of the IF statement at the end of each reduction (Algorithm 5).

Parts of the code in the case alternatives can be evaluated during generation time. In Algorithm 5, p and q stand for arbitrary productions. Whereas in the case of p the code has just been carried over from Algorithm 4, the

Algorithm 5: LR parser with CASE statement

```
BEGIN
   Push (AttributeStack, _)
   Push (StateStack, StartState)
   Terminal := GetToken ()

   LOOP
      State := Table (Top (StateStack), Terminal)

      IF State >= FirstReadReduce THEN          /* reduce or read-reduce? */

         IF State < FirstReduce THEN            /* read-reduce */
            Push (AttributeStack, GetAttribute ())
            Push (StateStack, _)
            Terminal := GetToken ()
         END

         LOOP                                   /* reduce */
            CASE State OF

            Stop:       HALT

            ...

            p+FirstReadReduce-1, p+FirstReduce-1:

               FOR i := 1 TO Length (p) DO
                  Dummy := Pop (AttributeStack)
                  State := Pop (StateStack)
               END
               Nonterminal := LeftHandSide (p)
               Semantics (p) ()

            q+FirstReadReduce-1, q+FirstReduce-1:

               AttributeStackPointer -:= m
               StateStackPointer     -:= m
               Nonterminal            := n

               <action statements for q>        /* Semantics (q) */

            ...

            END

            State := Table (Top (StateStack), Nonterminal)
            Push (AttributeStack, SynAttribute)
            Push (StateStack, State)

            IF State < FirstReadReduce THEN EXIT END /* read */
         END

      ELSE                                      /* read */
         Push (AttributeStack, GetAttribute ())
         Push (StateStack, State)
         Terminal := GetToken ()
      END
   END
END
```

case of q contains the code after applying constant folding: the FOR loop reduces to a decrement of the stack pointers and the precomputation of the left-hand side of q yields a constant:

m = Length (q)
n = LeftHandSide (q)

The two stacks could use one common stack pointer if this were an array index. As we want to arrive at real pointers in a C implementation, we have to distinguish between the two stack pointers.

The halt action (Stop) can be treated as a special case of a reduction. It occurs when the production S’ $\rightarrow$ S # is recognized. We have augmented the given grammar by this production where

S	is the original start symbol
S'	is the new start symbol
#	is the end of input token

By this we assure that the complete input is parsed and checked for syntactical correctness.

Table Representation and Access

After developing the principle algorithm for LR-parsing, the question of how to implement the function Table has to be discussed before we turn to the details of the implementation. The most natural solution might be to use a two-dimensional array. For large languages like Ada this array would become quite big:

(95 terminals + 252 nonterminals) * 540 states * 2 bytes = 374,760 bytes

This amount may be bearable with todays main memory capacities. However, we have chosen the classical solution of compressing the sparse matrix. Using this compression the storage required for the Ada parser is reduced to 22,584 bytes, including additional information for error recovery. The decision of how to compress the table has to take into account the desired storage reduction and the cost of accessing the compressed data structure.

An advantageous compression method is the comb-vector technique described in [ASU86] which is used for example in *Yacc* [Joh75] and in the latest version of PGS [KlM89]. This technique combines very good compression quality with fast access. The rows (or columns) of a sparse matrix are merged into a single vector called Next. Two additional vectors called Base and Check are necessary to accomplish the access to the original data. Base contains for every row (column) the offset where it starts in Next. Check contains for every entry in Next the original row (column) index. The resulting data structure resembles the merging of several combs into one. The optional combination with a fourth array called Default allows further compression of the array, as parts common to more than one row (column) can be factored out. Algorithm 6 shows how to access the compressed data structure if rows are merged. For more details see [ASU86].

Algorithm 6: access to the compressed table (comb-vector)

```
PROCEDURE Table (State, Symbol)
   LOOP
      IF Check [Base [State] + Symbol] = State THEN
         RETURN Next [Base [State] + Symbol]
      ELSE
         State := Default [State]
         IF State = NoState THEN RETURN error END
      END
   END
END
```

With this kind of compression it is possible to reduce the table size to less than 10%. The space and time behaviour can be improved even more, yielding in the case of Ada a size reduction of 6%. Algorithm 6 may return the value "error". However, if Symbol is a nonterminal, no errors can occur, as nonterminals are computed during reductions and are always correct. Therefore, in the case of nonterminal access, if Default is omitted the vector Check can be omitted, too. In order to implement this it is necessary to split the function Table in two parts, one for terminal and one for nonterminal access:

TTable	: States $\times$ Terminals	$\rightarrow$ INTEGER
NTTable	: States $\times$ Nonterminals	$\rightarrow$ INTEGER

The terminal Table TTable is accessed as before using Algorithm 6. The access to the nonterminal Table NTTable can be simplified as shown in Algorithm 7. Only the vectors NTNext and NTBase are necessary in this case.

Algorithm 7: access to the nonterminal table

```
PROCEDURE NTTable (State, Symbol)
   RETURN NTNext [NTBase [State] + Symbol]
END
```

Splitting the table into two parts has several advantages: the storage reduction is improved because the two parts pack better if handled independently. The storage reduction by omitting Default and Check is greater than using these two vectors. The table access time in the nonterminal case is improved significantly.

A further possibility to reduce space is to make the arrays Next and Check only as large as the significant entries require. Then a check has to be inserted to see if the expression 'Base [State] + Symbol' is within the array bounds. We decided to save this check by increasing the arrays to a size where no bounds violations can occur.

Error Recovery

The error recovery of *Lalr* follows the complete backtrack-free method described by [Röh76, Röh80, Röh82]. The error recovery used by *Lalr* is completely automatic and includes error reporting, recovery, and repair. From the user's point of view it works as described in a later section.

There is only one place where an error can occur: in an access to the terminal table where the lookahead token is not a legal continuation of the program recognized so far. The algorithm for error recovery replaces in the access of the terminal table (Algorithm 6) the return of the value "error". The parsing algorithm has two modes: the *regular mode* and the *repair mode* used during error repair. A problem is that during repair mode reductions and the associated semantic actions have to be executed. To avoid duplication of this code we use the same code during both modes. In order to avoid the immediate generation of new errors in repair mode error messages and skipping of tokens are disabled in this mode. Repair mode is exited when we can accept a token at one of the two places where tokens are read. We add an instruction at these places to leave repair mode.

It might be argued that this additional instruction to leave repair mode could be avoided. This is true, if either the code for reductions and semantic actions were duplicated or turned into a procedure which could be called twice. The first solution would increase the code size significantly. This is avoided in the second solution, but we have to pay for a procedure call at every reduction. This is more expensive than a simple assignment to change the mode for every input token, because the number of input tokens is usually about the same as the number of reductions.

Mapping Pseudo Code to C

The remaining implementation decisions are how to map the pseudo code instructions into real programming language statements. This concerns primarily the operations Push and Top and the table access. The following arguments hold only for the language C, as things are different in Modula-2.

The communication of the attribute value of a token from the scanner is not done by the procedure GetAttribute but by the global variable Attribute.

Stacks are implemented as arrays which are administered by a stack pointer. If the stack pointer is a register variable a Push operation could be accomplished by one machine instruction on an appropriate machine if auto increment is used. We preferred post increment, because we use a MC 68020 processor.

```
Push (AttributeStack, Attribute)   ⇒   * yyAttrStackPtr ++ = Attribute;
```

The operation Top turns into dereferencing a pointer.

```
Top (StateStack)   ⇒   * yyStateStackPtr
```

A dummy value is easily pushed by an increment instruction.

```
Push (StateStack, _)   ⇒   yyStateStackPtr ++;
```

The vector NTBase does not contain integer values but direct pointers into the vector NTNext to indicate where the rows start in NTNext. These pointers are initialized after loading of the program. This saves the addition of the start address of NTNext during the outer array access. The start address is already preadded to the elements of the vector NTBase. This kind of access is probably cheaper than the access to an uncompressed two-dimensional array which usually involves multiplication. The same technique is applied to the vector Base of the terminal table.

```
State := NTNext [NTBase [Top (StateStack ())] + Symbol]   ⇒
yyState = * (yyNTBasePtr [* yyStateStackPtr] + yyNonterminal);
```

The terminal table access is handled as follows. Instead of implementing the vectors Next and Check as separate arrays, we used one array of records. The array is called Comb and the records have two fields called Check and Next. This transformation saves one array access, because whenever we have computed the address of the Check field we find the Next field besides it.

```
LOOP
   IF Check [Base [State] + Symbol] = State THEN
      RETURN Next [Base [State] + Symbol]
   ELSE
      State := Default [State]
      IF State = NoState THEN <error recovery> END
   END
END

⇒

for (;;) {
   typedef struct { int Check, Next; } yyTCombType;
   register yyTCombType * TCombPtr;

   TCombPtr = yyTBasePtr [yyState] + yyTerminal;
   if (TCombPtr->Check == yyState) {
      yyState = TCombPtr->Next;
      break;
   };
   if ((yyState = yyDefault [yyState]) != yyNoState) continue;

   < code for error recovery >
}
```

To check for stack overflow we have to add the code below at every read (terminal) action. Read-reduce and reduce actions can only cause the stacks to grow by at most one element and therefore don't need an overflow check. We have implemented the stacks as flexible arrays. In case of stack overflow the sizes of the arrays are increased automatically. Therefore from the users point of view stack overflow never occurs.

```
if (yyStateStackPtr >= yyMaxPtr) < allocate larger arrays and copy the contents >
```

Of course we have used the register attribute for the most used variables:

```
yyState, yyTerminal, yyNonterminal, yyStateStackPtr, yyAttrStackPtr, TCombPtr
```

The variables yyTerminal and yyNonterminal have the type `long` because they are involved in address computations. These have to be performed with long arithmetic in C. The `long` declaration saves explicit machine instructions for length conversions (at least on MC 68020 processors). The variable yyState has the type `short` in order to be compatible with the type of the table elements. These have the type `short` to keep the table size reasonable small.

Continue statements have been changed to explicit gotos, because some compilers don't generate optimal jump instructions.

The appendix shows an example of a parser generated by *Lalr*. The error recovery, which constitutes most of the code, and some other parts have been omitted. Some details may deviate from the above description which concentrated primarily on the principles. For example all identifiers carry the prefix 'yy' to avoid conflicts with identifiers introduced by the user into the semantic actions. Also, the sizes of the arrays are greater than really necessary as we used a non-dense coding for the terminal symbols.

THE PARSER GENERATOR

This chapter will discuss important features of the generator *Lalr* from the user's point of view.

Structure of Specification

The structure of a parser specification is shown in Figure 1. There may be five sections to include arbitrary target code, which may contain declarations to be used in the semantic actions or statements for initialization and finalization of data structures. 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 [GrV88].

```
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. 1: Structure of Specification

Semantic Actions and S-Attribution

A parser for practical applications has more to do than just syntax checking: it must allow some kind of translation of the recognized input. In the case of LR parsing, action statements can be associated with the productions. These statements are executed whenever the production is reduced.

Additionally, during LR parsing it is possible to evaluate an S-attribution (only synthesized attributes). This mechanism works as follows: every terminal or nonterminal of the grammar can be associated with an attribute. After a reduction (that is during the execution of the action statements) the attributes of the symbols on the right-hand side of the reduced production can be accessed from an attribute stack using the notation $i. The action statement can compute an attribute value for the left-hand side of the production which has to be assigned to the special variable $$. After executing the action statements, this value is pushed onto the attribute stack by the parser to reestablish the invariant of the algorithm. The attribute values for terminals have to be provided by the scanner. Figure 2 shows an example for the syntax of grammar rules, semantic actions, and S-attribution.

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

Fig. 2: Grammar Rules with Semantic Actions and S-Attribution

Ambiguous Grammars

The grammar of Figure 2 as well as the example in Figure 3 are typical examples of ambiguous grammars. Like *Yacc* we allow resulting LR-conflicts to be resolved by specifying precedence and associativity for terminals in the OPER section. Figure 4 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. 3: Ambiguous Grammar (Dangling Else)

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

Fig. 4: Resolution of LR-Conflicts Using Precedence and Associativity

LR-Conflict Message

To help a user locate 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 do, the system also prints a derivation tree. Figure 5 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 three 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.

```
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. 5: Derivation Tree for an LR-Conflict (Dangling Else)

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 a shift back to the left margin. In Figure 5 the initial tree part consists of 5 lines (not counting the dotted lines). The 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 as shown, then there is a read reduce conflict (also called shift reduce conflict).

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 6. 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.
- 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. Upon leaving repair mode, parsing continues as usual.

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. 6: Example of Automatic Error Messages

COMPARISON OF PARSER GENERATORS

Finally we present a comparison of *Lalr* with some other parser generators that we have access to. We are comparing the features of the generators and the performance of the generated parsers. Tables 1 to 4 contain the results and should be self explanatory. Besides *Lalr* we examine:

- Yacc well known from UNIX [Joh75]
- Bison public domain remake of *Yacc* [GNU88]
- PGS Parser Generating System also developed at Karlsruhe [GrK86, KlM89]
- Ell recursive descent parser generator described in [Gro88].

Table 1: Comparison of Specification Techniques for Parser Generators

	Bison	Yacc	PGS	Lalr	Ell
specification 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	-	-	-	-	symbolic

Lalr, PGS, and *Ell* accept grammars written in extended BNF whereas *Yacc* and *Bison* require grammars in BNF. The tools *Lalr*, PGS, *Yacc*, and *Bison* process the large class of LALR(1) grammars but can only evaluate an S-attribution during parsing. *Ell* on the other hand processes the smaller class of LL(1) grammars but generates parsers which are 50% faster (see below) and can evaluate a more powerful L-attribution during parsing.

Table 2: Features for Grammar Conflicts and Error Recovery

	Bison	Yacc	PGS	Lalr	Ell
conflict message	state, items	state, items	state, items	derivation- tree	involved productions
conflict solution	precedence associativity	precedence associativity	precedence associativity modification	precedence associativity	first rule
chain rule elimination	-	-	yes	-	-
error recovery	by hand	by hand	automatic	automatic	automatic
error repair	-	-	yes	yes	yes

In case of grammar conflicts, *Lalr* has the advantage of providing derivation trees to support the location and solution of this kind of problems. The tools *Lalr*, PGS, and *Ell* provide automatic error recovery and repair in case of syntax errors.

Table 3: Comparison of Implementation Techniques and Languages

	Bison	Yacc	PGS	Lalr	Ell
parsing method	table-driven	table-driven	table-driven	table-driven	recursive descent
table compression	comb-vector	comb-vector	comb-vector	comb-vector	-
implementation languages	C	C	Pascal	C Modula	C Modula
target languages	C	C	C Modula Pascal Ada	C Modula	C Modula

All the LALR(1) tools generate table-driven parsers and use the comb-vector technique for table compression. *Ell* produces directly coded recursive descent parsers. Whereas *Yacc* and *Bison* are implemented in C and generate C code, the sources of *Lalr* and *Ell* exist in Modula-2 and in C and the tools generate Modula-2 as well as C. PGS is implemented in Pascal and generates parsers in even more languages.

The parser measurements (Table 4) exclude time and size for scanning and refer to experiments with Modula-2 parsers. The grammar for the LALR(1) tools had 69 terminals, 52 nonterminals, and 163 productions. The grammar for *Ell* was in extended BNF and contained 69 terminals, 45 nonterminals, and 91 productions. The timings result from analyzing a big Modula-2 program containing 28,302 tokens, 7867 lines, or 193,862 characters with the generated parsers. For this input the parser generated from *Lalr* executed 13,827 read, 14,476 read-terminal-reduce, 11,422 read-nonterminal-reduce, and 18,056 reduce actions. The terminal Table TTable was accessed 46,358 times and the nonterminal Table NTTable 43,953 times.

Table 4: Comparison of Parser Speeds and Sizes

	Bison	Yacc	PGS	Lalr	Ell
speed [10^3 tokens/sec.]	8.93	15.94	17.32	34.94	54.64
speed [10^3 lines/min.]	150	270	290	580	910
table size [bytes]	7,724	9,968	9,832	9,620	-
parser size [bytes]	10,900	12,200	14,140	16,492	18,048
generation time [sec.]	4.92	17.28	51.04	27.46	10.48

In the presented experiment *Lalr* generated parsers are twice as fast as those of *Yacc*. In general, we observed speed-ups between two and three depending on the grammar. The sizes of the parse tables are nearly the same in all cases. The parser generated by *Lalr* is 35% larger then the *Yacc* parser, mainly because of the code for error recovery. The generation times vary widely. The reasons can be found in the different algorithms for computing the look-ahead sets and the quality of the implementation. *Lalr* spends a considerable amount of time in printing the derivation trees for LR-conflicts. PGS generated parsers turned out to be faster in comparison to *Yacc*, but *Bison* performed considerably slower. Parsers generated by *Ell* were found to be the fastest exceeding the speed of *Lalr* by 50%.

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 processor running at a clock rate of 16.7 MHz.
- The loader: our timings include the time to load the parser which is almost zero.
- 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 parsers. For example the speed of directly coded parsers decreases with the grammar size. One reason is that linear or binary-tree search is done to determine the next action. Another reason can be that long jump instructions become necessary instead of short ones. PGS stores states in one byte if there are less than 256 states and in two bytes otherwise. This decreases the speed for large 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 except for *Ell* 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 2.
- 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 when it is measured in lines per minute.
- The timing: we measured CPU-time and subtracted the scanner time from the total time (scanner and parser) to get the parser time. We used the UNIX shell command *time* to get the time and added user and system time.
- The measure: we selected tokens per second as well as lines per minute as measures. The first one is independent of the density of the input and therefore more exact. The second one has been used by other authors and it is more expressive for a user.
- 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 than one might think. It disables a short cut of *Yacc* and the chain rule elimination [WaG84] of PGS, decreasing the speed in both cases.

Our conclusion from the numerous problems with the measurement of parser speed is that results from different environments or from different people can not be compared. There are too many conditions that influence the results and usually only a few of these conditions are reported.

CONCLUSION

We showed that table-driven, portable LR(1) parsers can be implemented efficiently in C or similar languages. Following the presented ideas the parser generator *Lalr* generates parsers that are two to three times faster than parsers generated by *Yacc*. They are vary fast and reach a speed of 35,000 tokens per second or 580,000 lines per minute. The generated parsers have all the features needed for practical applications such as semantic actions, S-attribution, and error recovery.

We have shown how to develop an efficient parsing algorithm in a systematic way. The starting point was a basic algorithm from a textbook. In a stepwise manner we turned it into a relatively short yet efficient algorithm mainly using pseudo code. Target code was introduced only in the last step.

We presented the main features of the parser generator *Lalr* from the user's point of view. A valuable feature of *Lalr* is that it prints a derivation tree in case of LR-conflicts to aid the location of the problem. We finally compared the features and performance of some parser generators.

ACKNOWLEDGEMENTS

Whereas the author contributed the parser skeletons in C and Modula-2, the generator program *Lalr* was written and debugged by Bertram Vielsack who also provided the experimental results. Valuable suggestions to improve this paper are due to Kai Koskimies and Nigel Horspool. Nick Graham deserves many thanks for transforming my text into idiomatic English.

REFERENCES

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

[Ass88] W. Aßmann, A Short Review of High Speed Compilation, *LNCS 371*, (Oct. 1988), 1-10, Springer Verlag.

[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.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[HoW89] R. N. Horspool and M. Whitney, Even Faster LR Parsing, Report Dept. of Computer Science-114-IR, University of Victoria, Department of Computer Science, May 1989.

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

[KlM89] E. Klein and M. Martin, The Parser Generating System PGS, *Software—Practice & Experience 19*, 11 (Nov. 1989), 1015-1028.

[Pen86] T. J. Pennello, Very Fast LR Parsing, *SIGPLAN Notices 21*, 7 (July 1986), 145-151.

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

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

[WhH88] M. Whitney and R. N. Horspool, Extremely Rapid LR Parsing, in *Proceedings of the Workshop on Compiler Compiler and High Speed Compilation*, D. Hammer (ed.), Berlin, GDR, Oct. 1988, 248-257.

APPENDIX

Example of a Generated Parser

Grammar:

```
L       : L S | .
S       : 'i' '=' E .
E       : E '+' E
        | E '-' E
        | E '*' E
        | E '/' E
        | 'i'
        | 'n'
        | '(' E ')' .
```

Parser:

```
# include "DynArray.h"

# define yyInitStackSize        100
# define yyNoState                0
# define yyTableMax             122
# define yyNTableMax            119
# define yyLastReadState         13
# define yyFirstReadReduce       14
# define yyFirstReduce           20
# define yyStartState             1
# define yyStopState             20

typedef short   yyStateRange;
typedef struct  { yyStateRange Check, Next; } yyTCombType;
typedef struct  { tScanAttribute Scan; } tParsAttribute;

static  yyTCombType *   yyTBasePtr      [yyLastReadState + 1];
static  yyStateRange *  yyNBasePtr      [yyLastReadState + 1];
static  yyStateRange    yyDefault       [yyLastReadState + 1];
static  yyTCombType     yyTComb         [yyTableMax + 1];
static  yyStateRange    yyNComb         [yyNTableMax + 1];
static  int             yyErrorCount;

int Parse ()
{
   register yyStateRange     yyState         ;
   register long             yyTerminal      ;
   register yyStateRange *   yyStateStackPtr ;
   register tParsAttribute * yyAttrStackPtr  ;
   register bool             yyIsRepairing   ;
            unsigned long    yyStateStackSize = yyInitStackSize;
            unsigned long    yyAttrStackSize  = yyInitStackSize;
            yyStateRange *   yyStateStack    ;
            tParsAttribute * yyAttributeStack;
            tParsAttribute   yySynAttribute  ;       /* synthesized attribute */
            yyStateRange *   yyMaxPtr;

   yyState           = yyStartState;
   yyTerminal        = GetToken ();
   MakeArray (& yyStateStack, & yyStateStackSize, sizeof (yyStateRange));
   MakeArray (& yyAttributeStack, & yyAttrStackSize, sizeof (tParsAttribute));
   yyMaxPtr          = & yyStateStack [yyStateStackSize];
   yyStateStackPtr   = yyStateStack;
   yyAttrStackPtr    = & yyAttributeStack [1];
   yyErrorCount      = 0;
   yyIsRepairing     = false;

ParseLoop:
   for (;;) {
      if (yyStateStackPtr >= yyMaxPtr) {             /* stack overflow? */
         int StateStackPtr   = yyStateStackPtr - yyStateStack;
```

```
         int AttrStackPtr    = yyAttrStackPtr - yyAttributeStack;
         ExtendArray (& yyStateStack, & yyStateStackSize, sizeof (yyStateRange));
         ExtendArray (& yyAttributeStack, & yyAttrStackSize, sizeof (tParsAttribute));
         yyStateStackPtr     = yyStateStack + StateStackPtr;
         yyAttrStackPtr      = yyAttributeStack + AttrStackPtr;
         yyMaxPtr            = & yyStateStack [yyStateStackSize];
      };
      * yyStateStackPtr = yyState;

TermTrans:
      for (;;) {   /* SPEC State = Table (State, Terminal); terminal transition */
         register    yyStateRange * TCombPtr;

         TCombPtr = (yyStateRange *) (yyTBasePtr [yyState] + yyTerminal);
         if (* TCombPtr ++ == yyState) { yyState = * TCombPtr; break; };
         if ((yyState = yyDefault [yyState]) != yyNoState) goto TermTrans;

         /* error recovery */
      };

      if (yyState >= yyFirstReadReduce) {            /* reduce or read-reduce? */
         if (yyState < yyFirstReduce) {              /* read-reduce */
            yyStateStackPtr ++;
            (yyAttrStackPtr ++)->Scan = Attribute;
            yyTerminal = GetToken ();
            yyIsRepairing = false;
         };

         for (;;) {
            register long yyNonterminal;

            switch (yyState) {
   case yyStopState: /* S' : L End-of-Tokens . */
      ReleaseArray (& yyStateStack, & yyStateStackSize, sizeof (yyStateRange));
      ReleaseArray (& yyAttributeStack, & yyAttrStackSize, sizeof (tParsAttribute));
      return yyErrorCount;
   case 21:
   case 19: /* L : L S . */
      yyStateStackPtr -= 2; yyAttrStackPtr -= 2; yyNonterminal = 111; break;
   case 22: /* L : . */
      yyStateStackPtr -= 0; yyAttrStackPtr -= 0; yyNonterminal = 111; break;
   case 23: /* S : 'i' '=' E . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 112; break;
   case 24: /* E : E '+' E . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 113; break;
   case 25: /* E : E '-' E . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 113; break;
   case 26:
   case 17: /* E : E '*' E . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 113; break;
   case 27:
   case 18: /* E : E '/' E . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 113; break;
   case 28:
   case 14: /* E : 'i' . */
      yyStateStackPtr -= 1; yyAttrStackPtr -= 1; yyNonterminal = 113; break;
   case 29:
   case 15: /* E : 'n' . */
      yyStateStackPtr -= 1; yyAttrStackPtr -= 1; yyNonterminal = 113; break;
   case 30:
   case 16: /* E : '(' E ')' . */
      yyStateStackPtr -= 3; yyAttrStackPtr -= 3; yyNonterminal = 113; break;
            };

            /* SPEC State = Table (Top (), Nonterminal); nonterminal transition */
            yyState = * (yyNBasePtr [* yyStateStackPtr ++] + yyNonterminal);
            * yyAttrStackPtr ++ = yySynAttribute;
            if (yyState < yyFirstReadReduce) goto ParseLoop;
         };                                          /* read-reduce? */

      } else {                                       /* read */
         yyStateStackPtr ++;
```

```
         (yyAttrStackPtr ++)->Scan = Attribute;
         yyTerminal = GetToken ();
         yyIsRepairing = false;
      };
   };
};
```

Contents