Even Faster LR Parsing

R. NIGEL HORSPOOL AND MICHAEL WHITNEY

Department of Computer Science, University of Victoria,
P.O. Box 1700, Victoria, B.C. V8W 2Y2, Canada

June 1990

SUMMARY

Conventional LR parser generators create tables which are used to drive a standard parser proc parsers can be obtained by compiling the table entries into code that is directly exec with a directly executable parser is its large size. In this paper, we introduce optimiz the parsing speed even further while simultaneously reducing the size of the parser.

KEY WORDS Parsing LR parsers Compilers

INTRODUCTION

The syntax analysis phase of a compiler can represent a significant proportion of the entire compilation time. Waite and Carter[1] give figures showing that a recursive descent parser consumed 24% of the entire compilation time in a Pascal compiler that they studied. Of course, this number is strongly dependent on the parsing technique and on the nature of the compiler. But it does serve as a warning that parsing should not be ignored when the goal is to achieve fast compilation rates.

There are two families of parsing methods in widespread use. One family corresponds to a top-down approach to parsing, and the class of grammars that are accepted is LL(1). One method of implementing LL(1) parsers is by recursive descent, but a more efficient technique is to use a table-driven parser. In fact, Wite and Carter managed to reduce the proportion of time spent parsing in their Pascal compiler from 24% to 7% by substituting a table-driven LL(1) parser coded in assembly language.

The second family of methods is based on bottom-up, shift-reduce, parsing. The classes of grammars that are normally used are SLR(1), LALR(1) or LR(1) depending on the parser generator employed. W will use the term LR parsing to refer to this collection of methods. Until recently, LR parsers have all been implemented using the table-driven approach. The parser generator transforms the grammar into tables whose entries must be interpretively executed by a driver program.

Because LALR(1) grammars have more expressive power than LL(1) grammars, the syntax rules for most programming languages are presented in LALR(1) form, whereas LL(1) grammars are provided for only a few. Therefore, it can be argued that a general-purpose parser generator should accept either the

LALR(1) or LR(1) grammar class, rather than be restricted to the LL(1) class. A discussion of the pros and cons of the LL and LR methods can be found in Reference 2, pages 196-200.

There has been considerable research into reducing the storage requirements of parsers,[3] but there has been comparatively little work directed towards increasing parsing speed. Table-driven LR parsers can be made to execute very much faster through careful coding of the driver program. Grosch[4,5] has followed this approach and can parse C source code at a rate of approximately 400,000 lines per minute on a Motorola 68020 processor. This is approximately twice as fast as a parser generated by *yacc*,[6] the standard parser generator supplied with UNIX.

An alternative approach, used by Pennello,[7] is to convert table-driven LR parsers into directly-executed code. Instead of having a driver program access a table entry and interpret its actions, each table entry can be 'compiled' into low-level statements that perform the actions directly. By compiling LR tables into assembly language, Pennello increased the speed of various parsers by a factor of 6.5, achieving a processing rate of 240,000 lines per minute on an Intel 80286 processor with a 8MHz clock rate. A speed-up by a factor of 10 for a COBOL parser on a proprietary architecture was also reported. As might be expected, the conversion into code came at the expense of an increase in memory requirements – a growth factor of 3.6 was reported. It is also possible to compile the entries of a LL(1) parser into code,[9] and Gray has implemented such a scheme. Gray chose to compile the LL(1) parser into C code, and therefore his parser generator has the advantage of being portable. On the other hand, the C language is somewhat less flexible than assembly language and some loss of coding efficiency inevitably occurs (especially when **switch** statements are used in the C source code).

We believe that a directly executable parser generated from LL(1) tables will usually be faster than one generated from LR(1) tables. If the LL(1) parser is implemented by recursive-descent or by a table-driven equivalent of recursive-descent with an explicit control stack, it will perform much less stack manipulation than the LR(1) equivalent. This form of LL(1) parser pushes and pops an item onto its stack no more than once for each application of a production rule. The number of stacking operations is reduced even further if iteration is used instead of recursion when recognizing constructs defined by right-recursive production rules. A LR parser, however, pushes and pops an item onto its state stack for every symbol that is read as well as for each application of a production rule.

In this paper, we introduce some optimization techniques that considerably reduce the number of stack operations performed by directly executable LR parsers, and we will describe a parser generator that implements these techniques. Our parser generator is compatible with *yacc* and generates a parser in either C source code form or in the assembly language of the SUN3 computer. Retargeting the parser generator to a different language is straightforward. The stack optimizations have the benefit of reducing the size of the parser while simultaneously increasing its speed. When combined with some other, simpler, optimizations, we can generate parsers that are only slightly larger than their table-driven counterparts, but execute up to eight times faster.

The following sections of this paper give an example of an unoptimized directly executable parser, introduce some optimizations that reduce stack use, explain some additional simpler optimizations, and give performance results achieved by our implementation. No specific knowledge of LR parser generation techniques is assumed. The interested reader can refer to texts on LR parsing[10] or on compiler construction.[2]

for this information.

The results reported here represent a re-implementation of and an extension to some earlier work[11,12]

AN EXAMPLE

Before examining optimization of directly executable parsers, it would be helpful to begin with a small example of the parse tables produced by a LR parser generator, and to look at how the tables might be translated into C code.

A Grammar and its LALR(1) Parse Tables

Here is a small grammar for arithmetic expressions containing infix addition and multiplication operators.

0. `S` → `E`
1. `E` → `E + E`
2. `E` → `E * E`
3. `E` → `( E )`
4. `E` → `id`

The symbol `id` represents an identifier. With declarations to specify that the `+` and `*` operators are left-associative and that `*` has higher precedence than `+`, the grammar would be acceptable to the *yacc* parser generator. (Some grammar transformations would be necessary before the grammar would be acceptable to parser generators that do not support such declarations.) This grammar describes arithmetic expressions such as

```
a + b
a + b * c
( a + b ) * ( c )
```

and so on.

A LALR(1) parser generator would convert the above grammar into tables that encode the actions of a LR parser for this grammar. The main parser tables are sometimes called the T table and the N table. The columns of the T table are indexed by terminal symbols of the grammar, whereas columns of the N table are indexed by non-terminal symbols. Figures 1 and 2 show the tables for our example grammar. The terminal symbol `EOF` represents an end-of-input marker. We note that a SLR(1) or LR(0) parser generator would generate similar tables but containing fewer blank entries. A LR(1) parser generator would normally create larger tables with more rows. However, all four parsing methods use the same kinds of table entries and process the table entries in exactly the same way.

A parser that interprets the actions in these tables maintains a stack of state numbers. The top state number on this stack represents the current state of the parser. The parser selects an action from the T table based on the current state and on the current input symbol. There are five different kinds of entry used in the T table.

- An entry like ‘s7’ indicates that a shift to state 7 should occur. The new state number, 7, is pushed onto the state stack and a new input symbol is read.

	id	+	*	(	)	EOF
0	sr4			s3		
1						acc
2		s5	s4			r0
3	sr4			s3		
4	sr4			s3		
5	sr4			s3		
6		s5	s4		sr3	
7		r1	s4		r1	r1

Figure 1: T-Table

	S	E
0	s1	s2
1		
2		
3		s6
4		sr2
5		s7
6		
7		

Figure 2: N-Table

- An entry like ‘r4’ indicates that a reduction using production number 4 should occur. There are three parts to a reduction. First, if there is any semantic action associated with rule number 4, it should be executed. Second, as many entries are popped off the state stack as there are symbols on the righthand side of rule number 4. Third, the symbol that appears on the lefthand side of the production rule is used to select an action from the Ntable (see below).

- An entry like ‘sr5’ represents a composite shift-reduce action. It is equivalent to the pair of actions: ‘s*k*;r5’ where *k* represents an arbitrary state number. The value of *k* is immaterial because the reduce action pops and discards the value immediately after it is pushed. The use of shift-reduce actions allows many states to be eliminated from the parser and hence makes the tables considerably smaller.

- A blank entry indicates that a syntax error has been detected. A table-driven parser would normally report the error and then attempt to resume the parsing process after executing a syntactic error recovery algorithm.

- Finally, the entry ‘acc’ (accept) indicates that the parser should halt and report a successful parse.

There are three kinds of entry in the Ntable. Entries in the table are selected by the current state (the topmost state number on the stack) and by a non-terminal symbol.

- An entry like ‘s3’ indicates that state number 3 is pushed onto the state stack. Parsing would then continue by reverting to use of the Ttable (where the next action is determined by the new state on top of the stack and by the current input symbol).

- An entry like ‘sr2’ again represents a composite shift-reduce action. The effect is the same as executing a entry like s*k* from the Ntable immediately followed by ‘r2’ from the Ttable. This implies that another Ntable action is executed immediately after the ‘sr2’ action.

- A blank entry represents a “don’t care” entry because it can never be accessed, regardless of whether the input token sequence is syntactically correct or not.

Directly-Executable Parser Code

A straightforward translation of the T and N tables into directly executable C code is easy to accomplish. If we make only a minimal attempt to generate good code, the code might look like that shown in Figure

```
      /* Code for T Table Actions */
S0:   token = scan();
P0:   push( 0 );
      if (token == id) goto SR4;
      if (token == '(') goto S3;
      goto Error;
S1:   token = scan();
P1:   push( 1 );
      if (token == EOF) return;
      goto Error;
S2:   token = scan();
P2:   push( 2 );
      if (token == '+') goto S5;
      if (token == '*') goto S4;
      goto R0;

      code for states 3-6 omitted

S7:   token = scan();
P7:   push( 7 );
      if (token == '*') goto S4;
      goto R1;
```

```
      /* Code for Rule Reductions */
SR0:  push( -1 );
R0:   pop( 1 ); lhs = S; goto NXT;
SR1:  push( -1 );
R1:   pop( 3 ); lhs = E; goto NXT;
SR2:  push( -1 );
R2:   pop( 3 ); lhs = E; goto NXT;
SR3:  token = scan(); push( -1 );
R3:   pop( 3 ); lhs = E; goto NXT;
SR4:  token = scan(); push( -1 );
R4:   pop( 1 ); lhs = E; goto NXT;

      /* Code for N Table Actions */
NXT:  switch( top() ) {
      case 0:
         if (lhs == E) goto S1;
         goto P2;
      case 3:
         goto P6;
      case 4:
         goto SR2;
      case 5:
         goto P7;
      }
Error:
         ... report the error
```

Figure 3: Directly Executable Parser

3.

Each row of the T table is translated into a standard block of code with two entry points. The row for state number *n* is converted into code with the entry labels S*n* and P*n*. The former calls the *scan* function to read a new token whereas the latter does not. The code continues with a statement to push the current state number onto the stack and then performs a sequence of tests on the current input symbol. A minor optimization, seen in states 2 and 7, is to eliminate some comparisons by making a rule reduction into a default action. The only consequence of this optimization is to delay detection of a syntax error in the input until after the rule reduction has been performed.

Each row of the N table translates into a clause in a **switch** statement. The row for state number *n* is converted to code with a label of the form **case** *n*, followed by a sequence of tests. The subject of these tests is the symbol that appeared on the lefthand side of the previous rule reduction. An action like 's2' is coded as a transfer to label *P2*, rather than to *S2*, because the parser should not read a new symbol. Since blank entries in the N table cannot be accessed, empty rows need not be converted into code. For the same reason, there need not be an explicit test for the last possibility in a sequence of tests on the lefthand-side symbol.

Each rule in the grammar is translated into a standard block of code with two entry labels. The label with the form R*n* handles a reduce by rule *n* (corresponding to an entry of the form 'r*n*' in the T table).

The code for a reduce first pops k values off the state stack, where k is the length of the righthand side of production rule number n. Then it sets the *lhs* variable to the lefthand side symbol of rule n, and transfers control to the **switch** statement. If a semantic action is associated with the production rule, the code for this action should be included.

The label with the form SRn handles a shift-reduce action by rule n (corresponding to an entry of the form 'sr$n$' in either the T or N table). If the entry occurs in the T table, a new symbol must be read. A fictitious state number is pushed onto the stack, and then the reduce action code can be executed. The SRn label and the push operation may be omitted if there are no actions of the form sr$n$ in either of the T or N tables.

No doubt the reader will have noticed that the code in Figure 3 can be considerably improved. One observation is that states 3, 4 and 5 in the T-table produce nearly identical code (differing only in the state number that is pushed onto the stack). A second point is that not all the labels defined in the code are referenced – implying that these labels and several lines of code may be deleted. The use of C as a target language is also a source of some minor inefficiencies. For example, if the target language permitted statement labels to be used as first-class objects (as in Fortran, PL/I or assembly language), we could implement the state stack by a stack of statement labels and thus replace the **switch** statement labelled *NXT* by an indirect branch. Pennello's directly-executable parser[7] used this approach.

We will return to issues of code quality later. First, we will consider optimizations that reduce usage of the state stack.

STACK ACCESS OPTIMIZATIONS

Many of the push and pop actions performed during a LR parse are redundant. We present, below, the "Minimal Push" optimization technique that eliminates these redundant actions. In addition, we present two additional optimizations that have the effect of increasing the applicability of minimal push optimization.

Minimal Push Optimization

Suppose that the grammar contains a production rule

```
A → a b c d
```

where a, b, c and d represent terminal symbols. Further suppose that there are no alternative productions for A that begin with a. If the LR parser begins recognizing an instance of A in a context where there are no possibilities other than A, the parser should contain a sequence of states like those shown in Figure 4. (If state 5 has been eliminated by shift-reduce optimization, the argument given below requires only minor changes.)

While recognizing A from state 1, the parser reads the symbols a, b, c, d and follows the sequence of state transitions 1 → 2 → 3 → 4 → 5 in the figure. The numbers of each of these states are successively pushed onto the state stack. At state 5, the parser performs a reduction action using the rule `A → a b c d`. The action causes state numbers 2, 3, 4 and 5 to be popped off the stack and discarded without ever having been used! The state number that is uncovered by the four pop operations (state 1) is used,

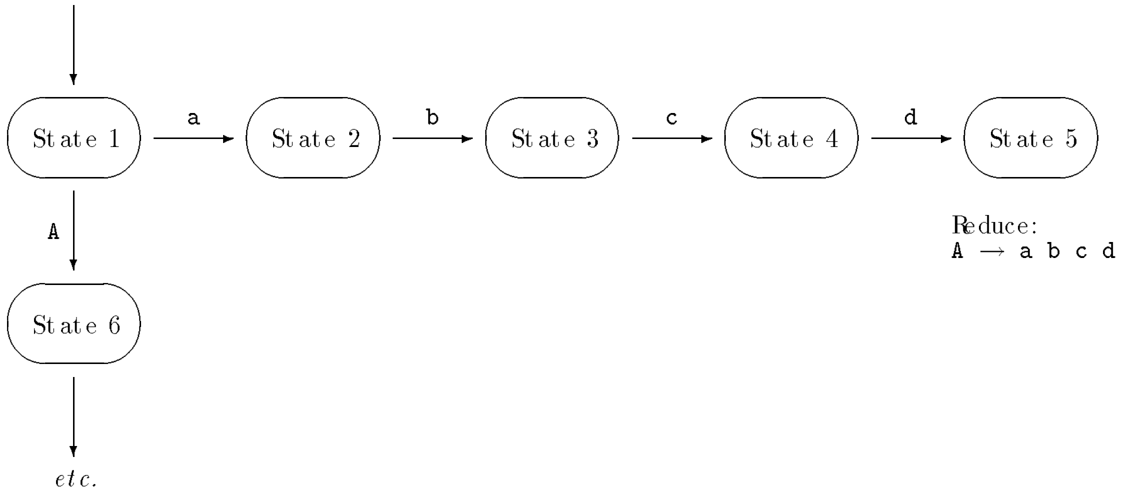

Figure 4: State Structure for `A` → `a b c d`

however. It and the lefthand-side symbol, A, are needed to determine that a transition to state 6 is required after the rule reduction.

We make the observation that only states with non-terminal transitions (like state 1 in the figure) need be pushed onto the stack. These state numbers are the only ones which might be consulted after a rule reduction. For the parser whose tables are given in Figures 2 and 3, the states which need to be pushed are 0, 3, 4 and 5. These are the only states which have non-empty rows in the N table.

The obvious optimization to make to a directly executable parser, therefore, is to generate statements for pushing the state number only in states that have non-terminal transitions. In the code generated for a rule reduction, we also need to modify the number of items to be popped off the state stack. For example, the rule reduction code for `A` → `a b c d` in State 5 would have to be changed from popping four items to popping zero items. Determination of the correct number of items to pop at a reduction is not difficult. However, we will provide a moderately formal explanation.

We use the notation $S_i \xrightarrow{X} S_j$ to indicate that the LR parser has a transition from state S_i to state S_j on symbol X. For a state S_n where a reduction $X \rightarrow X_1 X_2 \ldots X_n$ is performed, we define the *reduction path* set as:

$$\left\{ S_1 S_2 \ldots S_n \middle| S_1 \xrightarrow{X_1} S_2,\ S_2 \xrightarrow{X_2} S_3,\ \ldots S_{n-1} \xrightarrow{X_n} S_n \right\}$$

Given a particular reduction path $S_1 S_2 \ldots S_n$, the number of items to pop is equal to the number of states with non-terminal transitions among $S_2, S_3 \ldots S_n$. (Note that S_1 is not included in the list.)

Two problems arise. The first problem is that two different states may perform a reduction by the same production, and the count of items to pop in one state may disagree with the number of items to pop in the other state. The second problem is that a particular reduction in some state may have more than one reduction path, and some of these paths may require differing numbers of items to be popped.

The first problem, where there is a conflict between reduction actions in two different states, can be resolved by replicating the rule reduction code in the directly executable parser. Each replica can be

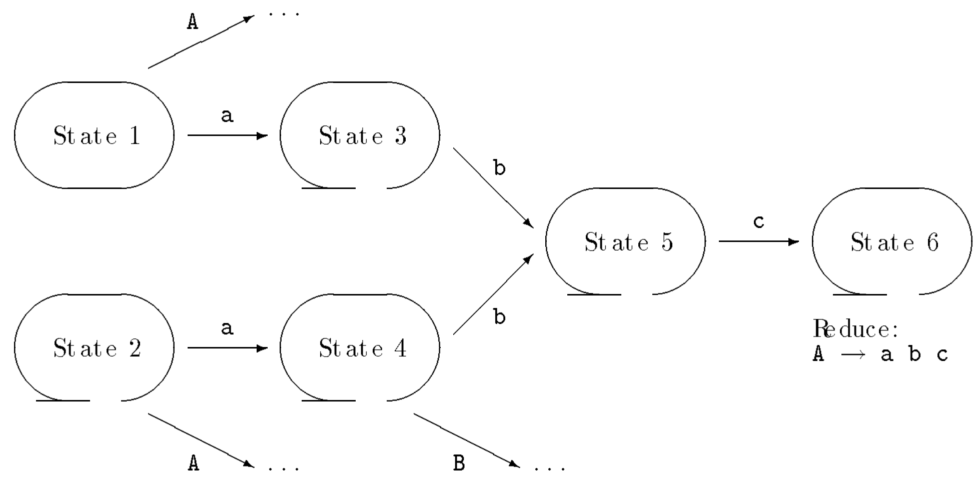

Figure 5: A "Pop-Count" Conflict

modified to pop a different number of items off the stack. It is then an easy matter to transfer control to the appropriate version of the rule reduction code.

The second problem, where a reduction in a single state does not have an unambiguous number of items to pop, is illustrated in Figure 5. In the figure, states 1, 2 and 4 have non-terminal transitions and cause items to be pushed onto the stack. If state 6 is entered by the path 1 → 3 → 5 → 6, the rule reduction should not pop any items off the stack. But if state 6 is entered by the path 2 → 4 → 5 → 6, one item (state number 4) needs to be popped. We call this situation a "pop-count" conflict.

There are at least two possible solutions to such a conflict. One possibility is to insert some extra stack pushes into the reduction paths which have fewer pushes. A solution along these lines must exist because, at worst, we can cause every state to push its state number on entry – reverting to the behaviour of the unoptimized LR parser. An alternative solution, and the one actually implemented in our parser generator, is to duplicate the states involved in the conflict. For example, the configuration of Figure 5 would be transformed into the configuration shown in Figure 6. By this means, the states 6 and 6′ are given distinct numbers of items to pop from the stack when a reduction by the rule A → a b c is to be performed. In other words, the second problem with pop counts is reduced to an instance of the first problem

In practice, we have found pop count conflicts to be rare. Only a very few states need to be duplicated to eliminate the conflicts. Some figures appear towards the end of this paper in the section on experimental results.

For a typical grammar, the proportion of stack pushes that are eliminated by the minimal push optimization alone is not very high. We observed that only about 30% of pushes are eliminated, and this figure does not appear high enough to make the optimization worthwhile. As one referee of this paper observed, two or more adjacent terminal symbols in the righthand side of a rule are necessary for the optimization to be applicable. Typically, terminal symbols are used to separate non-terminal symbols and they do not often appear in adjacent positions. However, the optimization becomes highly effective when

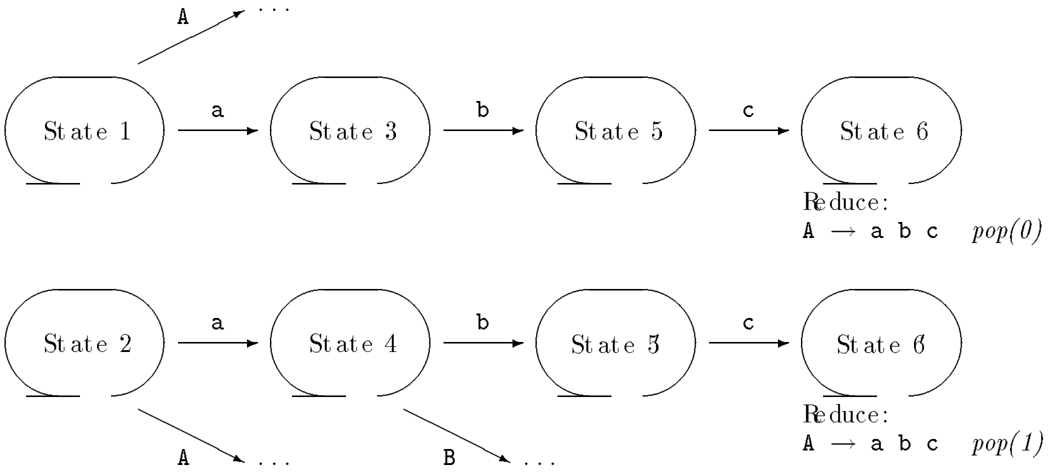

Figure 6: The Pop-Count Conflict Resolved

used in conjunction with the "direct goto" optimization, described next.

An alternative method of eliminating some redundant stacking operations from the parser would be to implement left-corner parsing.[8] With this technique, LR parsing is used for parsing the righthand-sides production rules until it becomes unambiguous as to which righthand-side is involved. At this point, LL(1) parsing is used to recognize the remainder of the rule. Since we can implement LL(1) parsing with fewer stacking operations than for the LR methods, some savings should occur.

Direct Goto Determination

A table-driven LR parser uses the N table to determine how to continue after a rule reduction. The state stack is popped and the top state on the stack is used to select a row in the table. The lefthand side of the rule determines the column, and thus the appropriate shift or shift-reduce action is selected. The equivalent actions in the directly executable parser (as seen in Figure 3) are to select some code to execute based on the top state on the stack. Then a series of tests on the lefthand side symbol is performed.

If a rule reduction in some state always causes the parse to be continued in exactly the same way, all the work of selecting a clause in the **switch** statement and the tests on the lefthand side symbol can be suppressed. The rule reduction can be implemented by popping the stack, if required, and then making a direct transfer of control to the appropriate continuation code.

We will now give a more formal explanation of minimal push optimization and of the direct goto optimization introduced in this section. In order to simplify the formal explanation, we will assume that the parser does not contain any combined shift-reduce actions. (It is straightforward, but not instructive, to remove this assumption.)

First, we will introduce some more terminology. Let *States* be the set of parser states and *Rules* be the set of production rules which may be used in reduction actions. If a state S may perform a reduction using the rule R = $X \rightarrow X_1X_2\ldots X_n$, we define an *origin state* for this reduction as being a state S_1 such that

$S_1 S_2 \ldots S_n S$ is a reduction path for rule R in state S. We also define the *destination state* associated with S_1, R and S as being the state, D, reached by the non-terminal transition on symbol X (the lefthand-side symbol of rule R) from state S_1. In other words, D represents one of the possible states in which parsing resumes after a reduction by rule R in state S.

For the entire parser, we can define the set of *Reductions* as $\{ <Q, D, R, S> \}$ where Q is an origin state associated with each rule reduction R performed in each state S, and $Q \rightarrow D$ is a transition labelled by the lefthand-side symbol of R.

We can optimize the reduction by rule R in state S if the set of possible goto destinations

$$\{ d \mid <q, d, R, S> \in Reductions,\ q \in States,\ d \in States \}$$

contains exactly one element. In this situation, the parser need not consult the state stack after the rule reduction and can simply transfer control to the unique destination state. We call this particular optimization *direct goto* optimization.

Since direct goto optimizations eliminate some accesses to the state stack, the minimal push optimization can be extended. It is frequently the case that direct goto optimization eliminates the need for pushing many more states.

The exact conditions under which a state should now be pushed are the following. State Q must be pushed onto the stack in a parser to which both minimal push and direct goto optimization are applied only if

$$(1)\quad \exists R, S, D: \qquad <Q, D, R, S> \in Reductions, \quad \text{and}$$

$$\begin{aligned}(2)\quad \nexists\ R, S, D: \qquad & R \in Rules,\ S \in States, \\ & <Q, D, R, S> \in Reductions, \\ & \left| \{ d \mid <q, d, R, S> \in Reductions,\ q, d \in States \} \right| > 1\end{aligned}$$

The first of the two conditions requires that state Q has at least one non-terminal transition. The second condition requires that at least one use of the non-terminal transition after a rule reduction cannot be eliminated by direct goto optimization.

If we were to apply minimal push optimization and direct goto optimization to the parser shown in Figure 3, the following changes would be made. First, stack push operations would be suppressed in states 0, 2, 6 and 7. Second, reductions by rules 0 and 4 would be implemented by direct control transfers. Third, reductions by rules 1, 2 and 3 would require only one item to be popped (instead of the three items required by a conventional LR parser).

Right-Recursive Rule Optimization

Consider how lists of identifiers separated by commas may be described by grammatical productions. One possibility is to use *right-recursive* production rules, as follows.

```
        L → id LL
```

```
LL →
LL → , id LL
```

An alternate possibility is by using *left-recursive* production rules, as follows.

```
L → id
L → L , id
```

The left-recursive formulation cannot be used with non-backtracking, LL(1)-based parsing methods. The parser is unable determine which of the two rules to use when it begins to match the symbols on the righthand side. The right-recursive formulation, however, works very well with recursive-descent or table-driven LL(1) parsers. It is particularly effective if the tail-recursion implied by the third production rule is optimized into an iterative loop.

A parser based on one of the LR methods can be used with either formulation. (This is one of the reasons why the LR methods are often preferred to the LL methods.) But the left-recursive formulation is much preferable to the right-recursive form. A LR parser, constructed by the standard method from the right-recursive grammar, will perform a shift action for every identifier and comma in the list that is being parsed. It will not perform any reductions until a shift action has been performed for the final identifier in the list. Thus, if the list contains n identifiers, $2n - 1$ states will be pushed on to the state stack without any intervening pop operations. The unfortunate implication is that the amount of storage allocated to the stack restricts the maximum length of a list that can be parsed by a normal shift-reduce parser using a right-recursive grammar. In contrast, if the left-recursive formulation is used, a list of indefinite length can be parsed with a stack that has space for just three items.

The minimal push optimization, introduced above, reduces stack use for the right-recursive grammar somewhat. For example, a list containing n identifiers would require only n stack pushes, because pushes of the state where the comma separator is recognized are suppressed. However, it would definitely be advantageous to reduce stack use even further.

As the following example will illustrate, the push operations occurring during recognition of a construct defined by a right-recursive rule may indeed be unnecessary. To simplify the explanation, we use a slightly simpler grammar than the one given above. Consider a grammar that includes the following three rules.

```
1.   S → ...  L ...
2.   L → a , L
3.   L → b
```

The dots indicate that some symbols in the first rule are not shown. The non-terminal L generates lists of `a` symbols terminated by a `b` symbol and separated by commas. If the grammar contains no other references to the non-terminal `L`, the LR parser will include the states shown in Figure 7.

In this parser, the minimal push optimization technique will determine that only states 1 and 5 (both with non-terminal transitions on `L`) need to be pushed. However, state 5 is located in a two-state cycle that is traversed once for each occurrence of the symbol `a`.

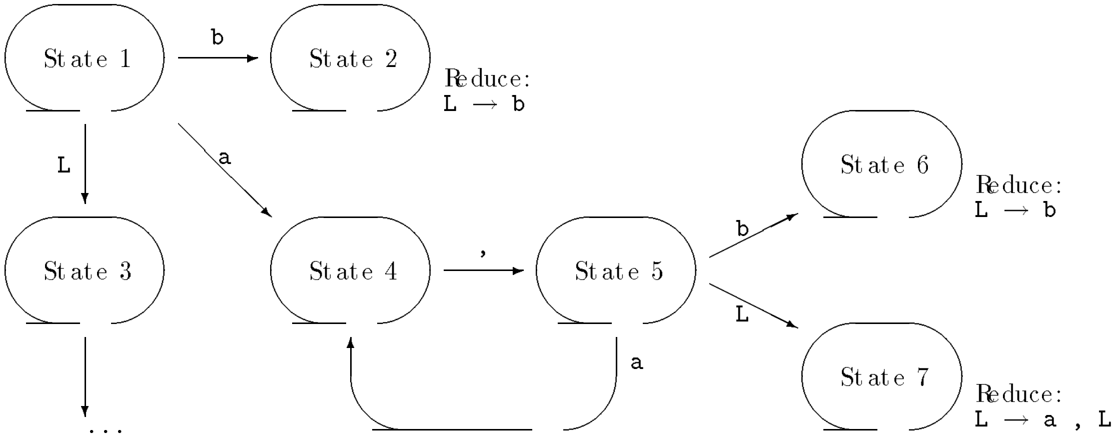

Figure 7: Parser with Right-Recursive Rules

Analysis performed for direct goto optimization reveals that the set of reduce tuples for the parser contains the following three elements:

$$<State_5,\ State_7,\ \texttt{L} \rightarrow \texttt{b},\ State_6>,$$

$$<State_1,\ State_3,\ \texttt{L} \rightarrow \texttt{a,L},\ State_7>,$$

$$<State_5,\ State_7,\ \texttt{L} \rightarrow \texttt{a,L},\ State_7>$$

We note that the reduction by the rule `L` → `b` in state 6 is a candidate for direct reduction optimization. However, the critical observation to make is the following. If a reduction by the rule `L` → `a , L` in state 7 uncovers state number 5 on the stack, control is immediately passed back to state 7 again. That is, the destination state and reduction state in the reduction quadruple are the same. If there are no semantic actions associated with a reduction by this rule, such a reduction has no more effect than popping one element off the stack.

When parsing the list *a,a, a, a, b*, the parser will push state 1 followed by four occurrences of state 5 onto the stack. Then the top four items will be popped off the stack (as four reductions by rule number 2 occur) and control will be transferred to state 3. If state 5 is simply popped off the stack without having any other effect on the parse and is not accessed in any state other than state 7, it need not have been pushed in the first place.

The minimal push optimization technique can be extended to eliminate the push of state 5 in the example. We call this extension *right-recursive rule* optimization.

The amended specification of when a state should be pushed onto the state stack in a parser that implements minimal push optimization, direct goto optimization, and right-recursive rule optimization is the following. A state *Q* need be pushed only if

(1) $\exists\ R, S, D$: $R \in Rules,\ S, D \in States,$

$<Q, D, R, S> \in Reductions$, and

(2) $\not\exists\ R, S, D$: $R \in Rules,\ S, D \in States,$

$$<Q, D, R, S> \in Reductions,$$
$$\left|\{ <q, d, R, S> \mid q, d \in States, d \neq S \textbf{ or } R \text{ has semantics} \}\right| > 1$$

The only change from before is that we ignore a reduce tuple in the second condition if the destination state is identical to the reduction state and if the production rule has no associated semantic actions.

Unit Rule Elimination

Grammars typically contain many production rules of the form `A` $\rightarrow$ `X`, where X represents either a terminal or non-terminal symbol. Such a production is called a *unit rule.*

LR parsers generated from grammars containing unit rules usually have many states like state S_2. The following scenario. Consider state S_1 which has an outgoing transition labelled by symbol X, which can be either a terminal or non-terminal symbol. When state S_1 accepts X, control is passed to state S_2 where a reduction by rule `A` $\rightarrow$ `X` takes place. This causes a goto action, corresponding to a transition away from state S_1 to state S_3 on symbol A, to take place.

If there are no semantic actions associated with the unit rule, there is no reason why state S_1 should not simply have a transition to S_3 labelled by X. And the transition to S_2 labelled by X should be deleted. This transformation is known as unit-rule elimination or chain rule elimination and is a standard optimization technique for LR parsers[10, 13, 14]. We mention it here because it is an optimization that eliminates many push and pop operations from the parser. The effect on a directly executable parser is to reduce both space and execution time requirements.

Although we have specifically implemented unit rule elimination in our generator for directly executable parsers, branch-chaining optimization (described below) often achieves the same result.

LOW-LEVEL OPTIMIZATIONS

Careful attention to the code patterns is necessary if we wish to maximize speed while maintaining an acceptable size for the parser. A good peephole optimizer[2] would perform some of the code transformations described below. But, since these transformations are important for obtaining good performance, they are explicitly implemented by our generator for directly executable parsers.

Code Sharing

Many states in the parser have identical or nearly identical sets of outgoing transitions. For example, states 0, 3, 4 and 5 of the parser shown in Figure 1 have identical T-Table actions. Very significant reductions in parser size can be achieved with little or no penalty in terms of increased execution time if the code for such states is shared.

We have found it convenient to categorize state similarity into four different classes, and associate different code transformations with each. Note that for the purposes of finding similar states, we consider T-Table actions and N-Table actions independently.

The first category occurs when two states have identical sets of actions. The obvious transformation in this case is to replace one copy of the code by a branch to the other copy.

The second category covers the case when the actions in one state form a subset of the actions in another state. In this case, the obvious transformation is for the two states to share the code for their common actions. For example, the state which has more actions could contain just the code for the actions which are peculiar to it and this code is followed by a branch to the other state. We note, however, that the code sharing transformation applied to a sequence of conditional tests may increase the execution time if the sequences are subsequently converted into binary searches (see below).

The third category occurs when two states have similar, but not identical, actions and neither set of actions is a subset of the other. A small-scale example appears in Figure 1 with states 2 and 6. Again, the obvious transformation is for both states to contain code for the actions which they do not have in common followed by a branch to some shared code. (The second category can be seen as a special case of the third category.) In the case of states 2 and 6 in Figure 1, the benefit of them sharing code is small. In practice, it is desirable to require that two states must have a minimum number of actions in common before a code sharing transformation is applied.

The fourth category occurs when several (more than two) states have similar, but not quite identical, actions. As a very small scale example, suppose that state *S1* includes actions { A2, A3 }, which are performed when the input symbol is *t1* or *t2*, respectively. Similarly, state *S2* includes actions { A1, A3 }, performed when the input symbol is *t1* or *t3*, and state *S3* includes actions { A1, A2 }, performed when the input symbol is *t1* or *t2*. Any two of these states can share code using the technique described above, but a different strategy must be used if all three states are to share code efficiently. This situation frequently arises with practical grammars, but on a larger scale. For example, there are many states associated with recognition of expression structure. They are differentiated only by the precedences of the expression operators that have been recognized so far. These states accept similar, but not identical, sets of terminal tokens and have similar actions associated with the tokens. Therefore, it is worthwhile to perform a code transformation that allows the code to be shared. Our solution is to group the common actions together and to test a vector element in each state to determine if control should be passed to the combined action group. The code for the small-scale example might have the form

```
S1: if ( bvector1[token] == 1 ) goto G;
    ... other actions for state 1
S2: if ( bvector2[token] == 1 ) goto G;
    ... other actions for state 2
S3: if ( bvector3[token] == 1 ) goto G;
    ... other actions for state 3
G:  switch( token ) {
    ... code for actions A1, A2 and A3
    }
```

The data storage required for the vectors and the execution time cost of performing the indexed vector test make this transformation worthwhile only if a relatively large number of each state's actions can be included in the group. In practice, the same vector may be tested in several states and this reduces the storage cost somewhat.

Conditional Sequences

The T-table actions for each state normally require the current input token to be compared against several possible values. Similarly, the N-table actions for each state require comparisons of the non-terminal symbol that appeared on the lefthand-side of a reduced production rule against various possible values. When the number of possibilities is small, say 7 or fewer, a series of comparison tests, as used in the sample code of Figure 3, is reasonably efficient. However, when the number of possibilities is larger, more efficient code should be used. (With the C grammar that was used in our experiments, one state has 30 possibilities for the next terminal symbol and several states allow 29 possibilities.)

As Pennello observed[7], a faster approach is to organize the tests as a binary search. An alternative method, suitable when the number of tests is large and the range of possible values is reasonably compact, is to use a jump table, corresponding to the usual implementation of a **switch** statement in C or a **case** statement in Pascal. Depending on the number of possible values, our parser generator selects whichever of these schemes appears to be most appropriate.

Given information about the dynamic behaviour of the parser, it would also be possible to order the tests so as to minimize expected execution time. However, this possibility is ignored in our current implementation of the parser generator.

Branch Chaining

When the target of a branch statement is another branch statement, the first branch can be re-targetted to transfer control to the destination of the second. This particular transformation is commonly included in the repertoire of peephole optimizers[15], and should therefore be performed by an optimizing compiler.

However, we stress the importance of the branch-chaining optimization because it can have a similar effect to other, apparently more sophisticated optimizations, when applied to a directly-executable parser. For example, consider the following grammar for arithmetic expressions.

```
0.   S → E
1.   E → E + T
2.   E → E - T
3.   E → T
4.   T → T * F
5.   T → T / F
6.   T → F
7.   F → ( E )
8.   F → id
```

There are states in the parser for this grammar where the following sequence of unit rule reductions can occur: `F → id`, `T → F`, `E → T`. These are unit reductions, as discussed earlier. If there are no semantic actions associated with these unit rules, then after direct goto optimization, the code for one of these states should be similar to the following. (We assume, for sake of example, that this state happens to be numbered 7.)

```
S7:   token = scan();
P7:   push( 7 );
      if (token == id) goto SR8;
      ... omitted code
SR3:  goto P27;
      ... omitted code
SR6:  goto SR3;
      ... omitted code
SR8:  goto SR6;
```

If branch chain optimization is applied, the test in state 7 would be simplified to

```
      if (token == id) goto P27;
```

This is similar to what one would expect if unit rule elimination optimization had been applied to the parse tables.

IMPLEMENTATION AND EXPERIMENTAL RESULTS

Structure of the Parser Generator

A generator for directly executable parsers has been implemented in four phases:

1. LALR(1) parse table generation.

2. Stack use optimizations and PM code generation.

3. PM code peephole optimization.

4. PM code to target language translation.

The first phase reads a grammar specification given in the same notation as supported by *yacc*.[6] Every construct of yacc except for the semantic stack (the stack whose elements are accessed by '$$', '$1', ... notation) and support for error recovery are provided. The semantic stack is not supported because the minimal push optimization causes positions in the state stack to lose their one-to-one correspondence with positions in the semantic stack. The user can, of course, use explicitly declared stacks in the semantic action code. The lack of support for error recovery may be a more serious deficiency, depending on the application in which the parser will be used. A discussion of how error recovery could be supported appears at the end of this paper.

The second phase of the parser generator reads the parse table into memory and applies the optimizations described in third section of this paper. The implementation does not assume that the tables are generated by a LALR(1) algorithm; thus it would be relatively straightforward to substitute a SLR(1) or LR(1) generation algorithm for the first phase. The output of the second phase is a directly executable parser, described in an idealized and very compact notation that we call *PM-code* (short for *parser machine code*). PM code can be viewed as the assembly language for a hypothetical machine which has a state stack, a register named *T* which holds the last input token and a register named *L* which holds the a non-terminal symbol. The machine has instructions for testing the T register, L register or the top of stack, for reading a new token into the T register, for storing a value in the L register, pushing and popping the state stack, and so on.

The third phase applies the optimizations described in fourth section of this paper. Even though branch chain optimization is performed by many compilers (and thus would be applied to the generated parser), we perform this optimization because it reduces the volume of PM code and because it makes code sharing optimizations easier to apply. The output from the third phase is in the PM code format, except that a greater variety of instructions is used. For example, after this phase, the PM code may contain bit-vector tests, whereas none would have been present in the input.

The fourth phase is constructed as a small, simple, program so that retargeting to different languages is relatively easy. We currently have two implementations. One performs a simple translation from PM code to the C language, the other translates to SUN-3 assembly language. (The SUN-3 computer series use the Motorola 68020 and 68030 CPUs.) Comparing parsers generated by the two versions allows us to estimate the extra overhead introduced by the use of C.

The C version of the translator generates C code as described in this paper. The assembly language version attempts to generate the best possible code. It implements the stack of states as a stack of labels, and uses the system stack for this purpose. There are three benefits of using the system stack instead of a separate stack. The first is that the CPU provides efficient instructions for pushing label addresses onto the system stack, both 'jsr' and 'pea' can be used for this purpose on the MC68020. The second benefit is that the stack size is limited only by the maximum stack size of the Unix process. The third benefit is that stack overflow checks are implicitly performed by the machine.

Experimental Data

Grammars for the C and Pascal languages were used to generate directly executable parsers, and timing experiments were performed on these parsers. Comparisons with parsers created by the *yacc* parser generator[6] were also performed, but it should be realized that yacc was not designed with fast parsing in mind.

Two grammars for C were used. The first grammar was a slightly modified version of a published grammar [16]. The minor changes to the grammar comprised, for the most part, the inclusion of semantic actions to process declarations of type identifiers. The C language separates the expression operators into 15 different precedence levels and the first grammar uses a different non-terminal symbol to represent each precedence level. The second C grammar is based on the first, except that the precedence levels of the operators are not defined by the grammar rules. Instead, precedence declarations are supplied to the parser generator and are used by the algorithm that constructs the LR(0) states. The second grammar eliminates almost all unit rule reductions that occur when expressions are parsed. Parsers based on the second grammar would usually execute faster than parsers created for the first grammar. The two grammars are named *C* and *C/prec*, below.

Similarly, two Pascal grammars were used. Again, the first grammar is similar to one that has been

	C	C/prec	Pasc	Pasc/prec
Rules	238	214	172	167
Non-Terminals	93	69	71	66
Terminals	84	84	61	61
LR(0) States	345	342	305	312
LR(0) States (after SRopt.)	172	216	163	181
Push States	74	96	79	91
Pop Count Conflicts	9	10	0	0

Table 1: Characteristics of the Test Grammars

published[17]. (Some minor manipulation was required to expand extended BNF notation into production rules suitable for use with a LR parser.) The first grammar embodied the operator precedence levels in the production rules. For comparison purposes, a second grammar where the precedence levels are declared separately was also created. The two grammars are named *Pasc* and *Pasc/prec*, below.

The overall characteristics of the four grammars and of their LR parse tables are summarized in Table 1. The fifth line in the table shows how many LR(0) states remain after shift-reduce optimization is applied. The sixth line shows the number of states which need to be stacked after the minimal-push and direct-goto optimizations described in this paper have been performed. The last line in the table shows how many states needed to be duplicated to eliminate pop count conflicts associated with the minimal-push optimization technique.

For each of the four grammars, three parsers were generated. Our own parser generator was used to create two of the parsers – one created as a C program and the other as an assembly language program. The third parser was created by *yacc*[6].

The C source code file used for timing the three C parsers comprised 1395 lines containing 38561 characters. (The file was the source code for the suntools program included with version 3.5 of the SunOS UNIX operating system.) But some of these lines were directives to the C preprocessor that caused several thousand more lines of C code to be included. The file size after preprocessing was 10193 lines, containing 122880 characters or 26738 lexical elements. However, a large proportion of the lines in the output from the preprocessor were empty (because they were empty in the original source files or contained only comments or contained preprocessor directives in the original files). If empty lines and lines containing file-name/line-number directives (used by the C compiler to correlate error messages with positions in the original input files) are stripped from the file, the size is reduced to 3169 lines or 105197 characters. Since it is unclear which file size should be used when computing the speed of a C parser, we avoid stressing speeds measured in units of source lines or source characters per minute.

The Pascal source file used for timing the Pascal parsers comprised 4462 lines containing 100684 characters or 18526 lexical elements. (This large Pascal program is the source code for a lexical analyzer generator called *aardvark*.) Since standard Pascal[17] does not have a preprocessor, these figures are unambiguous. Stripping all comments and blank lines from the file would reduce its size to 3934 lines or 89241 characters.

The sizes and the execution times for each of the 12 parsers are given in Table 2. The execution times do not include lexical analysis (or preprocessing time for the C source files). However, the C parsing

	C	C/prec	Pasc	Pasc/prec
C Parser				
Source code size (lines)	1704	1872	1173	1305
Object code size (bytes)	8256	8400	6032	6928
Execution time (seconds)	0.54	0.28	0.13	0.13
Assembler Parser				
Source code size (lines)	2614	2759	1848	2171
Object code size (bytes)	7688	8240	5408	6248
Execution time (seconds)	0.44	0.23	0.13	0.14
Yacc Parser				
Source code size (lines)	601	632	417	500
Object code size (bytes)	5832	6536	4184	4840
Execution time (seconds)	2.95	1.54	0.91	0.87

Table 2: Sizes and Speeds of the Parsers

	Optimizations Applied			
Minimal Push	no	no	yes	yes
Direct Goto	no	yes	no	yes
Push States	168	168	98	74
Size of Parser (lines)	2004	1937	1805	1704
Object Code Size (bytes)	9472	9264	8616	8256
Execution Time (seconds)	0.66	0.66	0.62	0.54

Table 3: Effect of Stack Optimizations

times do include some overhead entailed in recognizing declarations of type identifiers and entering these identifiers into a table that is accessed by the lexical analyzer. Other than the code needed to handle type identifier declarations, no semantic actions were performed. Each time given in the table is an average over ten measurements. If the parsing rates are converted into lines per minute, the figures for the directly executable parsers range from 800,000 lines per minute to over 2 million lines per minute, depending on how the file size is determined.

The directly executable parsers are five to eight times faster than the equivalent parsers generated by *yacc*[6], at the expense of a modest increase in memory requirements. Unfortunately, it is not possible to compare the parsing speeds with those observed by Pennello[7]. (He used a different grammar, gave his timings for a different CPU and gave no comparisons with parsers generated by *yacc*.)

How effective were the different optimizations in reducing the space and time requirements of the parsers? Table 3 shows the effect of disabling the minimal push and direct goto optimizations on the parser generated for grammar *C* Right-recursion optimization is not included in the table because our sample grammars contain very few right-recursive production rules. All the simpler optimizations were still performed. The importance of the simpler optimizations is hard to quantify because some of these optimizations may or may not be automatically performed when the parser is assembled or compiled. Without question, they are important for reducing the storage requirements to a size comparable to that of a table-driven parser. As an illustration of their importance, we generated an assembly language parser for the *C* grammar, above, where execution of the third phase of the parser generator was suppressed. This caused every sequence of conditional tests to be implemented by a jump table and no code sharing optimizations were performed. The size of the generated parser was 13291 lines of SUN-3 assembler source code, which assembled into 28144 bytes of object code – more than three times larger than the optimized parser. The execution speed was almost identical to that observed for the optimized assembly language parser generated for the same grammar. That is, the average time needed to parse the sample file was 0.44 seconds. We can conclude from this that the optimizations performed in the third phase of the parser generator are neutral in their effect on execution speed.

A question which is sure to arise is 'How much longer does it take to generate a directly executable parser than to generate a table-driven parser?'. Here is a partial answer to the question. The time required by our parser generator when processing the C grammar is 27.3 CPU seconds on a SUN-3/280 system while the corresponding time required by the *yacc* tool on the same computer is 15.4 CPU seconds. For the smaller Pascal grammar, there is much less difference. Our parser generator requires 6.7 CPU seconds and *yacc* requires 5.3 CPU seconds.

SUMMARY AND FURTHER WORK

We have described three different optimization techniques for directly executable parsers which simultaneously increase their speed while decreasing their storage requirements. When used in conjunction with other, simpler, storage optimizations, the resulting parsers can recognize their input at an incredibly fast rate. The parsers are five to eight times faster than the equivalent table-driven parsers generated by *yacc*, while requiring only a modest amount of extra memory.

A possible application area for high-speed parsing is in code generation. Some code generation methods[18, 19] use a parser to perform pattern matching against intermediate code. Typically, each pattern represents a machine instruction of the target computer that is emitted when the pattern is recognized. However, a standard parser is not quite powerful enough to perform the job. It is necessary to attach semantic predicates to some production rules that have the effect of disallowing use of the rule unless the predicate evaluates to true. A generator of directly executable parsers would need to be extended to provide support for semantic predicates. A simple technique to provide such support for the *yacc* parser generator has been described[20] and there is no reason why similar approach should not be used with directly executable parsers.

We have definitely not reached the ultimate in parsing speed. Our directly executable parsers simply represent a reasonable compromise between storage and time efficiency. It is possible to generate a parser with more states that performs fewer rule reductions. For example, the grammar could be preprocessed by substituting the righthand sides of productions for occurrences of the lefthand sides. An alternative approach with a similar effect is to unroll cycles in the LR parser by duplicating states. If carried to its ultimate conclusion, the parser would approximate a (very large) finite state automaton. Such a parser would rarely need to push or pop the stack.

There are at least two possibilities for improving the parsing speed without significantly increasing the storage cost. One approach would be to optimize the order in which equality tests on the current input symbol are performed. The tests should be ordered according to the actual frequencies with which the symbols are encountered in the various states of the LR parser. However, some preliminary experiments indicate that the effect on parsing speed is not very large. A second way (suggested by one of the referees) in which parsing speed might be increased is to borrow an idea from Reference 21. After a rule reduction, only a subset of the non-terminal transitions used in the goto action are possible. With some analysis of the LR items that generate each state, it should be possible to split the non-terminal transitions into the appropriate subsets, so that the number of tests needed to find the correct transition is minimized. On the other hand, the splitting process is likely to interfere with code sharing optimizations and lead to an increase in total memory requirments.

The current implementation of our parser generator provides no support for error recovery. In an interactive environment where the compiler invokes a source editor at the position of the first syntax error, no automatic recovery scheme is necessary. Nor would error recovery be needed if the parser is used in the code generation phase of a compiler. However, if the parser generator is to become useful in other situations, some recovery scheme should be implemented. Pennello observed that a stack of standard LR(0) state numbers can be re-created and with a table that contains the standard LR parsing actions, the usual LR recovery techniques can be applied. The same observation is true of our parsers too, although a little more computation would be needed to re-create the state stack (due to our minimal push optimization technique). We are investigating the feasibility of including Röhrich's recovery method[22] into the directly executable parsers without incurring a large storage cost for extra tables. The important observation to make is that two consecutive items on the stack of the directly executable parser define a path through the recognizer's states. From the path through the parser's states to the state in which a syntax error is detected, it is possible to construct the minimum cost continuation for the error state. If the parser and the lexical analyzer interact appropriately, it would be possible to insert tokens from the continuation sequence into the input stream, and thus implement Röhrich's recovery scheme.

ACKNOWLEDGEMENTS

Some valuable suggestions were received from the two referees of this paper. Financial support from the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

REFERENCES

1. W.M. Waite and L.R. Carter, 'The Cost of a Generated Parser', *Software – Practice and Experience*, **15**, (3), 221-237 (1985).

2. C.N. Fischer and R.J. LeBlanc Jr., *Crafting a Compiler*, Benjamin/Cummings, Menlo Park, Calif., 1988.

3. P. Dencker, K. Durre, and J. Heuft, 'Optimization of Parser Tables for Portable Compilers', *ACM Trans. on Prog. Lang. and Systems*, **6**, (4), 546-572 (1984).

4. J. Grosch, 'LALR - A Generator for Efficient Parsers', *Tech. Report 10*, GMD, University of Karlsruhe, Oct. 1988.

5. J. Grosch, ‘Generators for High-Speed Front Ends’, *Tech. Report 11*, GMD, University of Karlsruhe, Sept. 1988.

6. S.C. Johnson, ‘YACC – Yet Another Compiler Compiler’, *UNIX Programmer’s Manual, 7th Edition*, **2B**, (1979).

7. T.J. Pennello, ‘Very Fast LR Parsing’, *Proc. 1986 Symposium on Compiler Construction, ACM SIGPLAN Notices* **21**, (7), 145-151 (1986).

8. A.J. Demers, ‘Generalized Left Corner Parsing’, *Proc. Fourth Annual ACM Symposium on Principles of Programming Languages*, 170-182 (1977).

9. R.W. Gray, ‘Automatic Error Recovery in a Fast Parser’, *Proc. 1987 Summer USENIX Conference*, 337-346, (1987).

10. N.P. Chapman, *LR Parsing: Theory and Practice*, Cambridge University Press, Cambridge, U.K., 1987.

11. M.J. Whitney, *Optimization of Directly Executable LR Parsers*, M.Sc. thesis, Dept. of Computer Science, University of Victoria, 1988.

12. M.J. Whitney and R.N. Horspool, ‘Extremely Rapid LR Parsing’, *Proc. Workshop on Compiler-Compiler and High-Speed Compilation*, Berlin, G.D.R., 248-257 (1988).

13. D. Pager, ‘Eliminating Unit Productions from LR Parsers’, *Acta Informatica*, **9**, 31-59 (1979).

14. L. Schmitz, ‘On the Correct Elimination of Chain Productions from LR Parsers’, *Intl. J. of Computer Mathematics*, **15**, 99-116 (1984).

15. J.W. Davidson and C.W. Fraser, ‘The Design and Application of a Retargetable Peephole Optimizer’, *ACM Trans. on Prog. Lang. and Systems* **2**, (2), 191-202 (1980).

16. S.P. Harbison and G.L. Steele Jr., *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

17. D. Cooper, *Standard Pascal: User Reference Manual*, Norton, New York, 1983.

18. R.S. Glanville and S.L. Graham, ‘A New Method for Compiler Code Generation’, *Proc. Fifth Annual ACM Symposium on Principles of Programming Languages*, 231-240 (1978).

19. M. Ganapathi and C.N. Fischer, ‘Affix Grammar-Driver Code Generation’, *ACM Trans. on Prog. Lang. and Systems*, **4**, (7), 560-599 (1985).

20. M. Ganapathi, ‘Semantic Predicates in Parser Generators’, *Comput. Lang.*, **14**, (1) 25-33 (1989).

21. G.H. Roberts, ‘Recursive Ascent: An LR Analog to Recursive Descent’, *ACM SIGPLAN Notices* **23**, (8), 23-29 (1988).

22. J. Röhrich, ‘Methods for the Automatic Construction of Error Correcting Parsers’, *Acta Informatica*, **13**, (2), 115-139 (1980).