Optimizing Directly Executable LR Parsers

Peter Pfahler
Universität-GH Paderborn
Warburger Straße 100
D 4790 Paderborn, West Germany

Abstract

Traditionally, LR parsers are implemented as table interpreters. A parser generator creates tables whose entries are interpreted by the parser driver. Recent research shows that much faster LR parsers can be obtained by converting the table entries into directly executed code.

This paper introduces new techniques for optimizing directly executable parsers. The optimization methods proposed here are based on the analysis of the characteristical properties of large programming language grammars. They include a new structure for the parsing algorithm, an adaptation of the classical chain rule optimization and a systematic approach to stack access minimization. A parser generator based on these techniques was developed. It generates directly executable LR parsers running up to seven times faster than comparable table interpreting parsers.

1 Introduction

There are two classes of parsing methods for the syntax analysis phase of a compiler. The first class, called LL parsing, represents a top-down approach: LL parsers recognize the input starting at the root of the parse tree. They can be implemented either by a set of recursive procedures ("recursive descent") or by table interpretation. The second class, called LR parsing, corresponds to a bottom-up construction of the parse tree. LR parsers, also called shift-reduce parsers, are traditionally implemented by a fixed driver routine interpreting parser tables.

Since LR grammars have more expressive power than LL grammars, most programming language grammars are specified in LR form. Thus, the implementation of syntax analysis for these languages requires an LR parser generator to construct the parser tables. During syntax analysis the table entries have to be interpretively executed by the driver routine. Due to the interpretative nature of LR parsers and the fact that LR parsing requires far more stack accesses than LL parsing, there is clearly a speed disadvantage in LR parsing.

This fact has recently lead to the investigation of directly programmed, non interpreting LR parsers. Pennello [Pen86] proposes to translate the parsers finite state control into assembly code. Each parser state becomes a code memory address. The table entries for each state are converted into low level statements which perform the shift and reduce actions directly. Pennello reports a speed-up factor of 6.5 compared to an interpretative LR parser implemented in Pascal. On the other hand the directly programmed assembler parser was nearly four times larger than the interpretative version.

Horspool and Whitney [HoW90], [WhH88] present a directly programmed parser which is similar in structure to Pennellos. It is implemented in the C language (or in assembler). Horspool proposes several optimization methods for the directly executable parser. These optimizations aim at both the reduction of parser size and the increase of speed. There are low level optimizations like branch chaining and the translation of conditional sequences into binary search or switch statements. Very much like in parser table compression techniques, the code for states with equal or similar actions is shared, leading to a substantial decrease in size. Furthermore, the number of states which have to be pushed onto the parser stack is minimized. This push minimization eliminates the push operation for 271 of 345 states in Horspools C language parser example. A speed-up factor of 5 to 8 is reported compared to an interpretative parser generated by yacc [Joh75]. The increase of the source code size is by a factor of almost 3 (object code size increased by about 40 percent).

In this paper we propose some new optimization techniques for directly executable LR parsers. Based on the analysis of several "real-life" grammars we propose a new algorithmic structure of the parsing program. This program structure differs from Pennellos and Horspools and leads to both speed-up and size reduction. This parser organization is presented in chapter 3 after a short introduction to LR parsing in chapter 2.

In chapter 4 we describe the classical parser optimization of chain rule elimination and its benefits for the directly executable parser. By chain rule elimination our parser generator is able to remove a substantial amount of states from the directly executable parser. Furthermore, chain rule elimination considerably increases the parser speed by removing all redundant, generation time computable state transitions.

As an extension of Horspools minimal push optimization, we present a systematic approach to minimize stack access. This optimization is supported by the proposed parser structure, chain rule elimination and a new transformation called "rule splitting". It can be modeled as a solution of the well-known vertex cover problem on undirected graphs. Stack access minimization is described in chapter 5.

It should be stressed here that none of the transformations described below changes the parser's behavior with respect to production reductions and the semantic actions that might be associated with them. For the sake of brevity of this paper, we omit the discussion of low level coding optimizations and size reduction by code sharing. The interested reader can refer to [Pen86], [HoW90] and to work on parse table optimization like [DDH84]. Furthermore, we do not discuss the problems of integrating error recovery schemes like Röhrichs [Röh80] in directly executable LR parsers with push minimization. These problems are still investigated. However, we believe that it is possible to reconstruct the original parsers state stack to find a minimal cost continuation from an error state.

2 LR Parsing

Before investigating the structure of directly programmed LR parsers we introduce the principles of table driven LR parsing with the help of the small example grammar Expr:

```
p0 : S --> E              p4 : T --> F
p1 : E --> E + T          p5 : F --> a
p2 : E --> T              p6 : F --> ( E )
p3 : T --> T * F
```

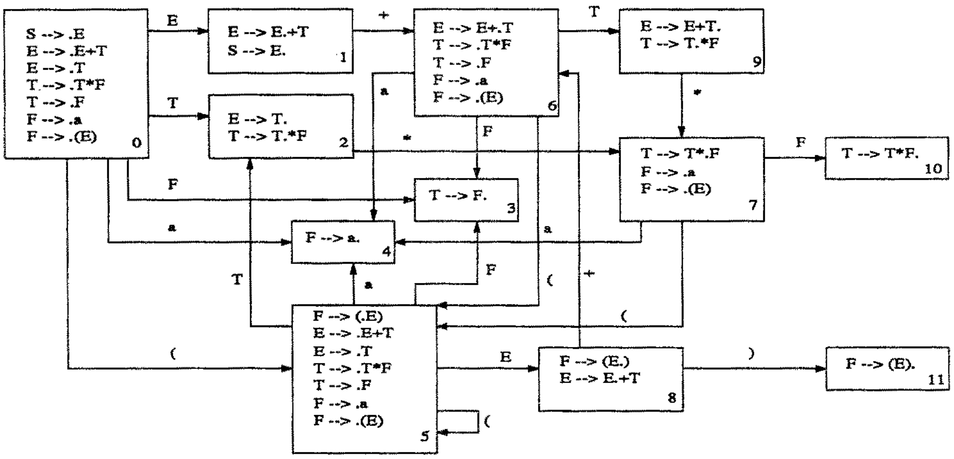

Figure 1: LR automaton for the Expr grammar

A LR parser generator (in this case yacc [Joh75]) constructs from this grammar a set of parser states and a state transition function. These states and transitions are depicted in the LR automaton shown in Figure 1. Each state contains a set of one or more so-called items. An item is a grammar rule with a dot in the right hand rule side. Intuitively, this dot specifies how much of a rule has already been recognized in a given state. So, a dot on the left of a right hand side means that it can be expected to recognize this rule starting from this state. A dot at the right end of a rule means that the recognition of this rule is completed and the parser might reduce according to this rule.

The parser generator converts the state transition function into the parser tables shown in Figure 2. Yacc created default entries for single r-entries in a T-table row by placing them in all free columns. This modification is frequently applied by parser generators and is guaranteed not to change the parsers behavior with respect to the location where input errors are detected. Shift/reduce transitions are not generated by yacc.

The table interpreting parser driver uses a state stack whose top element corresponds to the current state. It performs the following actions depending on the current state s and the next input symbol t:

- if T[s,t] = s_i : "shift to state i"
  Read the next input symbol, push state i and make it the current state
- if T[s,t] = r_j : "reduce according to rule j"
  Execute the semantic action associated with rule j. Pop the stack lrhs(j) times (lrhs = length of the right hand rule side). The *origin state* for this rule reduction is now on the top of the stack. Determine the *continuation state* by consulting the N-Table. Entry N[origin,lhs(j)] specifies the continuation state (lhs(j) is the nonterminal on the left hand side of rule j). Push the continuation state onto the stack and make it the current state.
- if T[s,t] = undefined : "error"
  Report a syntax error, try to recover from it or exit.
- if T[s,t] = acc : "accept"
  Accept the input as syntactically correct and halt.

	a	+	*	(	)	EOF	E	T	F
0	s4			s5			1	2	3
1		s6				acc			
2	r2	r2	s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	r5	r5	r5	r5	r5	r5			
5	s4			s5			8	2	3
6	s4			s5				9	3
7	s4			s5					10
8		s6			s11				
9	r1	r1	s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r6	r6	r6	r6	r6	r6			
	T-TABLE						N-TABLE		

Figure 2: Parser Tables for the Expr grammar

In the following we will use the notion of *reduction path* for reduce entries: A reduction path consists of the origin state of a reduction, the states to be popped (including the current one) and the continuation state. We write " O (p1 p2 .. pn) C" to mean: pop states pn (the current one) to p1 off the stack, find origin state O on top of the stack and goto continuation state C. For every reduce entry (reduction item) in the parser table (LR automaton) we can compute a set of associated reduction paths. This set of reduction paths will be named *reduction situation.*

Example: The reduction situation in state 10 in the Expr parser has the following three reduction paths:

6 (9 7 10) 9
5 (2 7 10) 2
0 (2 7 10) 2

To give an impression of the complexity and table size of real programming language LR parsers, Figure 3 contains some characteristic data for a ANSI-C grammar, a Modula2 and a Modula3 grammar compared to our small Expr example.

3 Directly Executable LR Parsers

A straight forward translation of LR tables into directly executable code generates a block of code for each state. This code is labeled by the state number. It contains actions for reading the input (if the state is accessed by a terminal transition) and pushing the state number onto the parser stack. Then the actions in the parser table row for this state are coded.

Example: State 2 of the Expr parser is coded as follows:

```
S2 : scan();
     push(2);
     if (token == '*') goto S7;
     goto red2;    /* E --> T */
```

	ANSI-C	MOD2	MOD3	Expr
Rules	213	240	296	6
Terminals	82	70	128	5
Nonterminals	64	123	111	3
States	367	420	521	12
Shift entries	1902	759	4792	13
Reduce entries	218	287	350	6
N-Table entries	1572	642	2814	9
Origin states	114	154	154	4
Continuation states	184	236	247	6

Figure 3: Characteristic data for programming language LR parsers

In case of a reduction according to rule r the parser jumps to the label generated for this rule. There the parser pops the stack, remembers the left hand nonterminal of r and, if necessary, executes semantic actions associated with rule r. Then it branches to the code that implements the nonterminal transitions.

Example: The reduction according to rule 2 for the Expr parser is coded as follows:

```
red2 : pop(1); lhs = E;
       outprod("E --> T");    /* semantic action */
       goto continuation;
```

Depending on the topmost state s on the stack the parser branches to the translation for row s in the N-table. The continuation state is found by comparing the left hand nonterminal of rule r with all symbols for which there is a goto entry in state s.

Example: The N-table for the Expr parser is coded as follows:

```
continuation  :
       switch (top)
         { case 0 : {if (lhs == E) goto S1;
                     if (lhs == T) goto S2;
                     goto S3;}
           case 5 : {if (lhs == E) goto S8;
                     if (lhs == T) goto S2;
                     goto S3;}
           case 6 :  {if (lhs == T) goto S9;
                     goto S3;}
           case 7 :  goto S10;
         }
```

This program organization which is the basis for both Pennellos and Horspools directly programmed parsers is summarized in the abstract program representation in Figure 4.

Using the grammar characteristics from Figure 3, we can give an impression of both size and operation of the straight forward and unoptimized parsing program. The average number of shift and reduce actions per state corresponds to the worst case number of token comparisons in our linear conditional sequence for every state. The average number of generated lhs comparisons

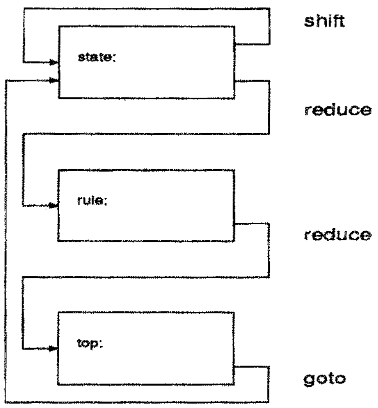

Figure 4: Program structure for directly executable LR parsers

for determination of the continuation state is computed by the number of goto entries divided by the number of states containing goto entries. It corresponds to the maximal number of lhs comparisons that have to be executed for the average reduction. The following table shows the results for our example grammars (the last branch in a conditional sequence is unconditional and not counted):

	ANSI-C	MOD2	MOD3	Expr
token comparisons	4.7	1.4	8.8	0.6
lhs comparisons	12.7	3.2	17.2	1.25

A closer look at the reduction action shows, that for every reduction according to rule r the following steps have to be taken on the average:

- 1 jump ("goto redr;")
- 1 pop stack
- 1 lhs assignment
- 1 semantic action
- 1 jump ("goto continuation;")
- 1 switch(top)
- #goto entries / #origin states -1 lhs comparisons (maximal)
- 1 jump to the continuation state

Note that the lhs comparisons cannot be combined for a given origin state since every nonterminal transition leads to a different continuation state. Also, there is exactly one *default goto* action which can be used instead of the last comparison (the "-1" in the computation above).

We propose a different program organization for the directly executable parser: First we observe that most rules have only one state they are reduced in. Therefore, we can save the additional jump to the middle code block of Figure 4 by integrating the reduction actions into the first code block. Due to the rules which are reduced in more than one state this transformation slightly

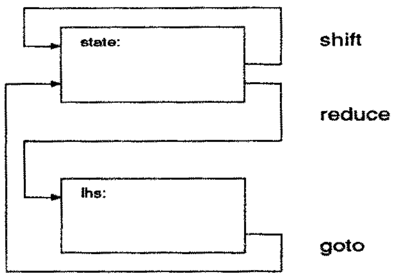

Figure 5: Modified program structure for directly executable LR parsers

increases the size of the parser program, but there are other important reasons (described in section 5) for treating multiple rule reductions individually.

The second modification can easiest be described in terms of the parser table: By inverting the access to the N-table, we can find the continuation state by inspecting the column for the left hand nonterminal in the row for the top of stack state. Transferring this simple inverting of N-table access ("reverse goto") and the individual reduce actions to the directly executable parser leads to the program structure shown in Figure 5. Reduce actions are integrated in the first program block ("action part"). For determining continuation states we create a program block for every nonterminal in the second block ("goto part").

A reduction in this modified program organization takes:

- 1 pop stack
- 1 semantic action
- 1 jump ("goto redlhs;")
- #goto entries / #nonterminals - #equaltargets top comparisons (maximal)
- 1 jump to the continuation state

The top of stack comparisons for a given nonterminal can clearly be combined since in practice many continuation entries for a nonterminal are identical. This fact is also used in the default goto action in which we can combine all origin states which lead to the most frequent continuation state ("#equaltargets"). For the Expr example grammar the continuation state determination by reverse goto looks as follows:

Example:

```
redE : if (top == 5) goto S8;
       goto S1;
redT : if (top == 6) goto S9;
       goto S2;
redF : if (top == 7) goto S10;
       goto S3;
```

We notice that all continuation states can be determined by exactly one top comparison. The table in Figure 6 lists the necessary comparisons for the ANSI-C grammar. It shows that the continuation state for 75 percent of the nonterminals can be determined by 2 or less top

# top comparisons	0	1	2	3	4	5	6	7	8	13	16	18	32	42
# nonterminals	22	16	10	1	3	1	2	2	2	1	1	1	1	1

Figure 6: Top comparisons for the nonterminals of the ANSI-C grammar

comparisons (0.75 on the average). The nonterminal which requires 42 top comparisons for 43 different continuation states is the symbol "identifier" with the single rule "identifier → name". This nonterminal is clearly a candidate for chain rule elimination. This optimization obviates the need for continuation computation by determining the continuation states individually in the origin states during parser construction. Chain rule elimination is described in section 4.

For the ANSI-C example the reverse goto transformation leads to a decrease in the program size for the continuation determination from 1802 source lines to 357 lines. The object code size for this program part decreases from 33114 bytes to 13906 bytes.

4 Chain Rule Elimination

A chain rule is a grammar production with a right hand side of length 1. Chain rules are most often used for descriptions in expression contexts (like "expr → term") or for the collection of related language concepts (like "statement → if-stat | while-stat | assign-stat ..."). As Waite and Goos [WaG84] point out, chain rules are semantically meaningless and "reductions according to chain productions simply waste time". They report that the parse of the statement "a := b" of their example language LAX requires 11 reductions of length 1 before the form "name → expr" is reached.

We propose to integrate chain rule elimination into a parser generator for directly executable LR parsers. Chain rule elimination is a special case of the classical shift/reduce optimization for LR parsers: If there is a transition from state s1 to state s2 on symbol x (terminal or nonterminal) and the only action in state s2 is the reduction according to rule r, this reduction can immediately be performed in state s1 without entering s2. Thus, shift/reduce optimization saves one state transition and (in table driven LR parsers) one push operation for state s2. States like s2 are often called LR(0) reduction states.

In the directly programmed parser structure proposed here (figure 5) shift/reduce optimization has the disadvantage of drastically increasing the size of the action code block since the pop action and the code for semantic actions for reductions in LR(0) reduction states must be copied to all their predecessor states (The ANSI-C grammar e.g. yields 169 LR(0) reduction states which together have 1768 predecessors). The alternative reduction code block in the middle of figure 4 avoids this copying but reintroduces the jump that was to be saved by the shift/reduce optimization.

In contrast to shift/reduce optimization, the chain rule elimination we propose considers only LR(0) reduction states that reduce according to a chain rule. One advantage is clearly that (normally) no semantic actions need to be copied to predecessor states because most chain rules are semantically irrelevant. Furthermore, no pop operation is required since the "pop(1)" for chain rules is eliminated by not pushing LR(0) reduction states. The other, most important advantage is that the parser doesn't have to consult the N-table to determine the continuation state of a LR(0) chain rule reduction. The continuation state can be determined at parser construction time, since the origin state is always equal to the predecessor of a LR(0) chain reduction state (due to the right hand side length of 1).

Repeated continuation computation for "chains" of chain rules finally yields a non LR(0) chain reduction state for continuation. The code blocks for LR(0) chain reduction states can be eliminated from the directly executable parser. Thus, chain rule elimination saves program size as well as execution time by avoiding redundant branches.

Example: In the Expr example the code for states 3 and 4 can be removed. The chain rule optimized version of state 0 looks as follows (semantic actions are included to illustrate the "invisible" steps):

```
S0 : if (term == 'a')
        {outprod("F --> a");
         outprod("T --> F");
         goto S2CT;}
     if (term == '(')
        {goto S5;}
     error();
```

S2CT stands for "S2 chain target", which is a label immediately above S2's code block where an additional scan() operation (the one from S4) is executed.

Chain rule elimination works similar for nonterminal transitions. If the continuation state c for a reduction with origin state o is a LR(0) chain reduction state reducing according to rule r the parser can directly branch to the next continuation state N[o,lhs(r)]. This redundant branch optimization is also applied repeatedly.

The situation is different if the continuation state c is determined by a default goto action combining several origin states. In this case the next continuation state cannot be uniquely determined. Still the directly programmed parser can be optimized by directly branching to the continuation code for lhs(r).

Example: After chain rule elimination the continuation code for nonterminal F of the Expr grammar looks as follows:

```
redF : if (top == 7) goto S10;
       goto redT;
```

For the ANSI-C grammar chain rule elimination eliminated 73 parser states. 1610 redundant branches could be eliminated. The longest chain of reductions that could be removed at parser construction time had a length of 3. The Modula2 and Modula3 parsers contained 61 and 126 LR(0) chain reduction states.

5 Stack Access Minimization

In conventional LR parsers every state which is entered during parsing is pushed onto the parser stack. As Horspool and Whitney [HoW90] point out, most of these states on the stack are simply popped off again without ever being consulted for the determination of a continuation state. Since only potential origin states are used for finding continuations, it is immediately obvious that only origin states need to be pushed. For our grammar examples this simple parser optimization saves the pushes for 253 of 367 states for ANSI-C, for 266 of 420 states for Modula2 and for 367 of 521 states for Modula3. In the Expr parser we can reduce the states to be pushed from 12 to 4.

Clearly stack push minimization requires a modification of the pop actions associated with rule reductions. The number of states to be pushed is no longer equal the the right hand production side. Instead, the so-called pop count is determined by the number of pushed states in the pop lists of the reduction paths associated with each reduce situation.

Two kinds of problems can occur: If a rule is reduced in more than one state its pop count may not be equal in all of these states. This poses no problem in our parser implementation because reductions are handled individually in the states they occur in. The other, more serious so-called *pop count conflict* arises if there are different pop counts for a single reduction situation. This occurs if on different paths to a reduction state a different number of states is pushed. Horspool proposes to duplicate the states on the reduction path to achieve an unambiguous pop count. Another solution was implemented in our parser generator: By inspecting the state stack the parser can determine the correct number of entries to be popped.

Example: If a reduction situation consists of the reduction paths

10 (120) 64
11 (17 120) 70
12 (120) 70

the generated pop code looks as follows:

```
pop(1); if (top == 17) pop(1);
```

Pop count conflicts occur rather rarely in programming language parsers. For the ANSI-C grammar we encountered 3 conflict situations for the push minimization described so far. Thus the time and space penalty for solving pop count conflicts can be neglected.

In the following we present a systematic method to further decrease the number of states that have to be pushed.

One source for further minimization is the chain rule elimination already described. If the origin state of a chain rule reduction has only one nonterminal transition, this origin state clearly doesn't have to be stacked. Another source of push minimization is due to the default goto entries in the continuation code block. Suppose in all reduction situations according to rule r that have a state s as an origin state, the continuation state N[s,lhs(r)] is determined by a default goto action. Then state s is obviously never needed on top of the stack. Its push operation can be eliminated if this elimination does not uncover a conflicting origin state on top of the stack. In the latter case s has to be pushed to "hide" its stack predecessor.

The additional push minimization described above is heavily dependent on both a clever choice of default goto actions and on the amount of continuation states that can be determined by default actions.

The latter, although appearing fixed by the parser grammar, can be increased by a transformation we call *reduction splitting* : Suppose a rule r: X $\rightarrow$ w is reduced in more than one parser state. It can easily be shown that the sets of origin states for these different reduction situations are disjoint. Therefore, the continuation state can be determined by inspecting only individually relevant origin states. In terms of the directly executable parser this leads to a split of the continuation code block for nonterminal X into blocks X_1, X_2 .. X_n, one X_i for each individual reduction situation. This reduction splitting clearly speeds up the continuation determination as compared to a combined continuation code block for nonterminal X. Furthermore, the amount of continuations that are determined by default actions is increased.

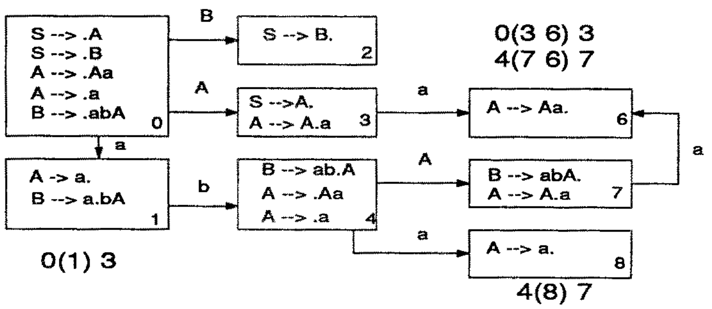

Figure 7: Example of reduction splitting

Figure 7 shows a trivial example for reduction splitting: The two reduction situations for rule "A → a" and the reduction situation for "A → Aa" lead to the creation of the reduction code blocks

```
red1A : goto S3;
red2A : if (top == 0) goto S3;
        goto S7;
```

(The reduction in state 8 is removed by chain rule elimination. For this trivial example the code for "red1A" is not generated by the parser generator because the single continuation state can be branched to directly from state 1.)

The problem of stack access minimization can be formulated as follows: Consider all reduction situations for which the continuation state has to be determined at parse time. These are all reduction situations after reduction splitting that are not eliminated by chain rule optimization and lead to more than one continuation state. Find the minimal number of origin states that have to be stacked in order to be able to determine the continuation state in all these reduction situations.

To determine the globally optimal solution for this minimization problem, we construct the so-called *push graph*. It contains one node for every origin state and an undirected edge between two nodes if there is a reduction situation where the corresponding origin states lead to different continuation states. Informally, an edge between two origin states means that at least one of these states must be pushed in order to be able to determine the continuation. (The states that have to be pushed to hide conflicting predecessor origin states are initially marked as push states.)

The push minimization problem now corresponds exactly to the well-known *vertex cover problem*:

Given: A graph G = (V,E)

Problem: Find a minimal vertex cover, i.e. a subset $V_1 \subseteq V$, such that for each edge $\{u,v\} \in E$ at least one of u and v belongs to V_1 and for all other V_2 with this property: $|V_2| \geq |V_1|$.

The vertex cover problem is known to be NP-complete. A simple approximation strategy covers the nodes in the order of falling vertex degree. The integration of this strategy into our parser generator resulted in a considerable reduction of the number of states to be pushed. Nevertheless, there is one severe problem with this approach: The push graph does not reflect the fact that for reasons of parser size and speed the most frequent continuation state should be accessed by a default goto action. In fact there were cases where the vertex cover algorithm resulted in very bad continuation code blocks. (e.g. a code block with two continuation states, one accessed by 12 different origins, one by only one origin state o1. Since o1 was decided not to be pushed, the continuation code block had to contain 12 top comparisons followed by one default goto for o1.)

We propose to solve this obvious trade-off between stack access minimization and code quality in favor of the latter: The algorithm for push minimization implemented in our parser generator first decides to push all origin states that do not lead to one of the most frequent continuation states of a reduction situation. For the remaining states the push graph is constructed. The vertex cover for this graph which is already partially covered by the first step is completed by the approximation algorithm sketched above. While only slightly increasing the number of stack states, this algorithm generates short and fast continuation code blocks.

The results of the stack access minimization for the example grammars are summarized in the following table. This table also illustrates the importance of reduction splitting for stack access minimization

	ANSI-C	MOD2	MOD3	Expr
origin states	114	154	154	4
stack states	103	86	98	3
reduction splits	5	47	51	0

6 Experimental Results and Conclusion

The implemented parser generator uses the yacc [Joh75] parser generator as a front end. The yacc generated parser states and transitions are read. After the various transformations described above the directly executable parser is generated as a C language program.

The following results were obtained for the directly executable ANSI-C parser run on a Sun4 sparc station. We measured the execution time of the generated parser for three different input programs:

lexer: the source code of the (table driven) lexical analyzer for ANSI-C

pcl: a collection of "typical" otherwise unrelated C language program parts

fapa: the source code of the parser generator for directly executable LR parsers (excluding parser table input).

The characteristics of the three input programs are summarized in the table below.

	Size in bytes	Size in lines	Size in tokens
lexer	26422	1823	10520
pcl	44440	4040	17373
fapa	25229	1435	7049

We compared the run times of a yacc generated parser to the unoptimized straight forward directly executable parser from section 3 figure 4 and three versions of the parser generated by our parser generator: The basic "reverse goto" version, the chain rule eliminated version and the additionally stack minimized version. The following table shows the results. The times measured are for the parser procedure and do not include the time for lexical analysis. There were no semantic actions. Each experiment was run 10 times. Times are given in milliseconds.

	yacc	unopt. direct	reverse goto	chain elim	minpush
lexer	732	686	237	127	110
pcl	904	539	338	183	156
fapa	368	227	145	69	69

These results indicate a speed-up factor between 5.3 and 6.6 compared to the table driven parser. The most considerable speed-up is due to the "reverse goto" organization and the chain reduction elimination while the effect of push minimization is less significant.

Note that our prototype parser generator does not yet include low level coding optimizations like application of binary search instead of linear conditional sequences for the determination of continuation states. A further increase in speed could be expected from such optimizations. Furthermore, our generator version does not stress the aspect of code size reduction in the action part. The only transformation it applies is merging states which are completely identical (including chain rule reductions). This leads to parser sizes which are 7 times larger than the yacc parser with respect to source lines. The size of the directly executable parser's binary (including a table driven scanner generated by *lex*) is about 50 % larger than the lex/yacc executable. Following Horspools results we can expect a further size reduction by applying quite simple optimizations. Horspool achieved this size reduction without a significant parsing speed penalty.

Both space and time measurements were optained on the base of unoptimized compilation of the parser sources. With RISC target processors this lead to a large amount of unfilled delayed branching slots for the conditional sequences of the directly executable parser. Enabling a simple postpass peephole optimization resulted in a size reduction of 22 % for the directly executable parser binary and a speedup of about 10 %. The yacc generated parser's size and speed could not be improved by this optimization.

For the future we plan the integration of both code sharing and simpler code saving transformations. Furthermore, we believe that the parser speed can still be increased. One source for further speed-up surely lies in an intelligent arrangement of token comparisons. By placing the comparisons for the most probable terminals first, a significant reduction of token comparisons can be expected. This does not necessarily require profiling experiments, but can also be achieved by a reasonable terminal code assignment following well known frequency statistics (e.g. [Wai86]).

A similar transformation could be successful in the continuation code blocks: There is no need for the used mapping of state labels to pushed state numbers (as long as labels are no first class objects). By an intelligent state number mapping the fast continuation determination could be supported for many reduction situations. First ideas include the mapping of origin states leading to equal continuations to mutually disjoint integer intervals. Thus, the continuation state could be determined by range checking instead of single top comparisons.

7 References

[DDH84] Dencker, P., Durre, K. and Heuft, J., *Optimization of Parser Tables for Portable Compilers*, ACM Trans. on Prog. Lang. and Systems, 6 (1984), 546–572.

[HoW90] Horspool, R. N. and Whitney, M., *Even faster LR Parsing*, Software - Practice and Experience (to appear) (1990).

[Joh75] Johnson, S. C., *Yacc - yet another compiler compiler*, Computing Science Technical Report 32, AT&T Bell Labs, Murray Hill, N.Y. (1975).

[Pen86] Pennello, T. J., *Very fast LR Parsing*, Proc. 1986 Symp. on Compiler Construction, ACM SIGPLAN Notices 21 (1986), 145–151.

[Röh80] Röhrich, J., *Methods for the automatic Construction of Error Correcting Parsers*, Acta Informatica 13 (1980), 115–139.

[Wai86] Waite, W. M., *The cost of lexical analysis*, Software - Practice and Experience 15 (1986), 473–488.

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

[WhH88] Whitney, M. and Horspool, R. N., *Extremely Rapid LR Parsing*, Proc. Workshop on Compiler-Compiler and High-Speed Compilation, Berlin G.D.R (1988).