Very Fast YACC-Compatible Parsers (For Very Little Effort)

Achyutram Bhamidipaty   Todd A. Proebsting

TR 95–09

Abstract

We have developed a yacc-compatible parser generator that creates parsers that are 2.5 to 6.5 times faster than those generated by yacc or bison. Our tool, mule, creates directly-executable, hard-coded parsers in ANSI C; yacc produces interpreted, table-driven parsers. Hard-coding LR parsers for speed is not a new idea. Two attributes distinguish mule from other parser generators that create hard-coded LR parsers: mule is compatible with yacc (including yacc's peculiar error recovery mechanisms), and mule does absolutely none of the complex automata analysis of previous hard-coded-parser generators. Mule creates simple, fast parsers after very little analysis.

September 22, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

We have developed a `yacc`-compatible parser generator that creates parsers that are 2.5 to 6.5 times faster than those generated by `yacc` [Joh75] or the Free Software Foundation's `yacc` clone, `bison`. Our tool, `mule`, creates directly-executable, hard-coded parsers in ANSI C; `yacc` produces interpreted, table-driven parsers. (We will use `yacc` as the prototypical generator of table-driven LALR(1) parsers, unless a distinction from others (e.g., `bison`) is relevant.) A counter-intuitive advantage of `mule` is that it requires *less* analysis to generate parsers than `yacc` because `mule` does not produce any tables and therefore does not require any table compression techniques. `mule` is compatible with `yacc`, except for the obscure `YYBACKUP` macro.

Hard-coding LR parsers for speed is not a new idea. Two attributes distinguish `mule` from other parser generators that create hard-coded LR parsers: `mule` is compatible with `yacc` (including `yacc`'s peculiar error recovery mechanisms), and `mule` does *absolutely none* of the complex automata analysis of previous hard-coded-parser generators. Error recovery is notable because it is necessary to be `yacc`-compatible, and because it was thought to be very complicated in hard-coded parsers — the mechanism in `mule` is trivial to implement. The complete lack of optimizations is notable because `mule`'s parsers show impressive speedups.

One criticism of hard-coded LR parsers has been that they create parsers that are much bigger than the compressed-table interpretive systems. No attempt was made to reduce the size of `mule`'s parsers and yet they are, on average, only about twice the size of table-based systems on a modest grammar for a subset of C. `mule` increased the size of `gcc`'s parser many-fold, but that only represented an increase of less than 75KB.

The `mule` project began to determine how fast a very simple hard-coded parser would be, and the answer was a resounding "very fast." Given the small increase in size, we believe that hard-coding represents a much better method of encoding `yacc`-compatible parsers than the current table-driven method.

2 Previous Work

We know of three previous LR-parser generators that create hard-code. `mule` differs from all the others in its utter lack of sophisticated optimizations, while still producing impressively fast parsers. Furthermore, `mule` differs from each system in at least one of the following ways:

- `mule` produces ANSI C, not assembler.
- `mule` accepts `yacc/bison` input.
- `mule` does `yacc`-style error recovery.
- `mule` manages a semantic stack for user-code access (e.g., `$$`, `$1`, `$2`, etc.).

2.1 Pennello

Pennello created a parser generator that produced hard-coded parsers in assembly language [Pen86]. His hard-coded parsers showed a 6 to 10-fold improvement in speed over his table-driven system. His system was not `yacc`-compatible. Size increased by a factor of 2 to 4 — enabling error recovery was responsible for the larger increases. Because the hard-coded parsers directly manipulated jump addresses as data, they could not have been directly expressed in ANSI C. Furthermore, he exploited the low-level assembler for purposes of register allocation and stack manipulations. His parsers could directly use the run-time stack as the parser stack, defering stack management to the operating system. The parser-generator was responsible for analyzing the transitions at any given state to determine if they were best realized as a linear search, a binary search, or a jump table. Jump tables were compressed in much the same way as ordinary state-transition tables in a table-driven parser [FL88]. To eliminate unnecessary checks for stack overflow, the parser generator employs a heuristic to break cycles in the characteristic finite state machine's strongly connected components — the optimal solution is an NP-complete problem.

In contrast, `mule` generates ANSI C, *always* checks for stack overflow, and *always* uses C's `switch` statement to compute transitions (defering its translation to the compiler). After eliminating *all* overflow checks from `mule`'s parsers, no test case's speed increased by more than 17%. Also, Pennello's system does not accept `yacc` specifications, it does not maintain a semantic stack, and its error-recovery differs from `yacc`'s.

2.2 Horspool and Whitney

Horspool and Whitney developed many additional optimizations for hard-coded LR parsers [HW90]. Their system accepts `yacc` input, but does not support `yacc`'s semantic stack or error recovery. The parser generator is retargetable and can generate both C and assembler from a special intermediate representation. The parsers show a 5 to 8-fold increase in speed over their table-driven counterparts. The size increased by only 40%. They detail many optimizations relevant to hard-coded LR parsers.

Many of their optimizations can be characterized as *global* optimizations, which require an analysis of the entire finite state machine, rather than just individual states. Their *minimal push* optimization seeks to avoid pushing states on the stack whenever possible. Unfortunately, this optimization is complicated and not obviously compatible with a parser-controlled semantic stack (like `yacc`'s). Similarly, the *right-recursive rule* optimization requires nontrivial analysis and is incompatible with `yacc`'s semantic stack. *Unit-rule elimination* is a valuable optimization in table-driven parsers, but poses complications with respect to semantic actions. The *direct goto determination* optimization eliminates many unnecessary nonterminal state-transition computations.

In addition to the global optimizations, Horspool and Whitney list low-level coding optimizations that increase speed and decrease size. Like Pennello, they attempt to optimize decision sequences via linear searches, binary searches, or jump tables. They also incorporate branch-chaining into their system rather than relying on the peephole optimizer for this simple optimization. To eliminate duplicate code, they attempt to do as much code-sharing as possible — including finding code sequences that *almost* match and carefully encoding the program so that the matching code can be shared.

In contrast, `mule` supports a semantic stack, supports error recovery, defers all low-level coding decisions to the C compiler, and avoids all complex global optimizations.

2.3 Pfahler

Pfahler created a parser-generator that accepts `yacc` specifications and creates a hard-coded parser in ANSI C [Pfa90]. The generated parser cannot do error recovery. The hard-coded parsers are 5.3 to 6.6 times faster than `yacc`-generated parsers, and at least 50% bigger. Pfahler does not concentrate on doing low-level optimizations, but does invent a new parser-design and some new global optimizations.

His parser-generator creates hard-coded parsers that have a simpler structure than Pennello's and Horpool's. The structure is both time and space efficient, although it may require replicating semantic action routines multiple times. One part of the structure is an *inverted table* which requires a piece of code for each nonterminal for computing the state transition on that nonterminal and the current state. The inverted table optimization decreased code size by 58% and increased speed by a factor of 1.6-2.9.

Pfahler's global optimizations require nontrivial analysis of the finite state machine. His *chain-rule elimination* optimization is a special case of the well-known shift/reduce optimization employed in table-driven parsers and gives about a 2-fold speedup, while requiring some growth in code size due to code replication. Additionally, the parser-generator does some complex analysis for *stack access minimization*. This optimization requires heuristics for attacking an NP-complete problem and results in 0 to 20% speed improvements.

In contrast, `mule` supports error recovery, and avoids all complex optimizations. Chain-rule elimination may require replicating semantic actions multiple times within the parser, which may cause code explosion or, worse, incorrect code when labels are declared within the replicated code. Therefore, `mule` does not attempt this powerful optimization. (This is unfortunate since this optimization appears to be the simplest/best of the known global optimizations.) `mule` does, however, adapt Pfahler's inverted tables into its new design because of their simplicity and reported speed and size advantages. The adaptation avoids replicating semantic action routines. When Pfahler's system uses inverted tables, but does none of the error-recovery bookkeeping required of `mule`, its speedups are only 3-fold compared with table-driven techniques, whereas `mule`'s are 2.5 to 6.5-fold *with* bookkeeping.

3 LR-Parsing Mechanics

We briefly explain the fundamentals of shift-reduce parsing (which represents the LR(1) family) without going into any more detail than necessary for subsequent exposition.

LALR(1) parsers like `yacc` simulate, either directly or indirectly, a very simple automaton with a stack of automaton states [FL88]. (Parsers generated by `yacc` also maintain a *semantic* stack, but since that stack grows in parallel with the state stack, we only describe the use of the state stack here.)

Simulating the automaton requires two mechanisms: one for determining the *action*, which is determined by the current input symbol and the state on the top of the stack, and one for determining *state transitions* based on the current top of stack and a grammar symbol. At parser-generation time LALR(1) grammar analysis builds these tables, called *action* and *goto*, respectively. (The analysis is necessary regardless of whether a table-driven or hard-coded parser is desired.) Functionally, these tables have the following signatures.

$$goto\colon\ state \times symbol \rightarrow state$$
$$action\colon\ state \times token \rightarrow \{shift, reduce_Y, accept, error\}$$

There are only four possible actions: reduce, shift, accept, and error. Reduce actions are parameterized by the grammar production being reduced. Actions are described below — let *TOS* be the state on the top of the stack, and let *token* be the current lookahead token.

shift A shift pushes *goto[TOS,token]* onto the stack, and updates *token* by advancing the lexical analyzer.

reduce$_Y$ A reduction processes production $Y : X \rightarrow x_1 \ldots x_n$, which requires popping n states off the stack, followed by pushing *goto[TOS, X]*. (The semantic action of the parser relating to this production would be executed prior to popping states off the stack.)

accept An accept signals a successful parse.

error An error requires error reporting and/or recovery.

4 Simple Implementation

`mule` creates a single parsing routine, `yyparse()`, that simulates the LALR(1) parser directly in ANSI C, without interpreting any tables. The routine has five simple parts: initialization, automata states, reduction actions, nonterminal transitions, and error recovery. Although very similar to the inverted table structure in [Pfa90], this structure avoids the duplication of semantic action routines. Another difference is the `yacc`-compatible error recovery. The structure is simple, with all code being generated from a tiny set of small, well-defined templates that directly mirror the grammar or LALR(1) automaton.

Since both the state stack and the semantic stack grow in unison, we wrap the stack entries into a single structure, `StackType`.

4.1 Initialization

The initalization phase simply sets up bookkeeping and data structures for subsequent automata simulation. It is grammar-independent.

```
#define YYABORT return -1
#define YYACCEPT return 0
#define yyclearin token = yylex()
#define yyerrok yyerrorstatus = 3
#define YYERROR goto user_error_handler
#define YYRECOVERING() (yyerrorstatus <= 2)

typedef struct stackType {
     int state;                          // State stack element.
     YYSTYPE semantic;                   // Semantic stack element.
} StackType;

YYSTYPE yylval;                          // Semantic value computed by yylex().

int yyparse(void) {
     int token = yylex();                // Get first token.
     unsigned yyerrorstatus = 3;         // Initialize error-recovery counter.
     YYSTYPE yyredval;                   // Variable holds semantic value of $$.
     StackType stack_start[MAX_STACK];   // Stack.
     StackType *stack = stack_start;     // Stack pointer.
     StackType *EOS = stack_start + MAX_STACK;// End of Stack.

     goto state_0;                       // Start state.
```

4.2 Hard-coded States

For each automata state, `mule` creates code responsible for simulating the action of that state based on the current input token. All transitions *into* a given state are labeled with the same grammar symbol. States labeled with a token are called *shift states* and they require extra code to advance the lexical analyzer. The template of this code for state N is

```
state_N:
    stack->state = N;

    The subsequent 3 lines appear iff N is a shift state.
    stack->semantic = yylval;              // Put lexical semantic entry on stack.
    token = yylex();                       // Advance lexical analysis.
    yyerrorstatus++;                       // Update error-recovery counter.

    if (++stack == EOS) goto stack_overflow;
state_action_N:                            // Error-recovery entry point.
    switch (token) {
    case Q:  goto state_X;                 // iff shift = action[N,Q], X = goto[N,Q]
    case R:  goto reduce_Y;                // iff reduce_Y = action[N,R]
    case S:  goto error_handler;;          // iff error = action[N,S]
    case T:  YYACCEPT;                     // iff accept = action[N,T]
    ⋮
    // The action table determines the default action for N:
    default:  goto error_handler;
    or
    default:  goto reduce_Z;
    }
```

The state number is stored in the stack, followed by possibly invoking the lexical analyzer. The three optional lines store the semantic value of the current token, advance the lexical analyzer, and do error-recovery bookkeeping. Incrementing the stack pointer completes the push. The case arms of the `switch` are determined by the *action* table computed by the LALR(1) analysis; for each condition met in the comments, a case arm must be generated. Default actions were developed for compressing table-driven parsers, and can be similarly employed here for generating the `switch`'s default [FL88].

4.3 Reduction Actions

One piece of code is generated for each production. Its template is given below.

```
reduce_M:                                  // Production M: P ⇒ x_1 ... x_n
     { User code for production M. }       // iff user code exists.
    or
    yyredval = (stack-n)->semantic;        // default semantic action, $$=$1, iff n > 0.

    stack -= n;                            // Pop RHS symbols from stack.
```

```
     stack->semantic = yyredval;                // Copy ($$) onto semantic stack.
     goto nonterminal_P;                        // Compute transition on production's LHS.
```

User actions are associated with reductions, and the code corresponding to a given production is expanded in-place. After the user code, the symbols associated with right-hand side of the production are popped, followed by copying `$$` onto the semantic stack. Finally, there is a jump to the code that will compute the appropriate state given the left-hand side symbol of this production.

4.4 Nonterminal Transitions

For each nonterminal, code is produced to compute (and jump to) the appropriate state given the current state. This simple `switch` statement is given below.

```
nonterminal_J:
     switch ((stack-1)->state) {                // Top of stack.
     case K:  goto state_L;                     // iff L = goto[K,J]
     ⋮
     }
```

The case arms of the `switch` statement are taken directly from the *goto* table that was computed by the LALR(1) grammar analysis. Because this `switch` cannot fail, no default entry is needed. However, making the most common case arm the default is a trivial time and space optimiztion.

4.5 Error Recovery

`yacc`'s error recovery mechanism is rather idiosyncratic. In fact, examining two books, [LMB92] and [ASU86], and the output generated by `yacc` yields three different descriptions of the recovery mechanism. We have tried to be faithful to the output of `yacc`.

Fortunately, the mechanism has few consequences to the generation of the rest of the hard-coded parser. The only change to the parser is the maintenance of the variable, `yyerrorstatus`. Although relatively short, the code below is very subtle — like the explanation of `yacc`'s error recovery mechanism. The code is given only for completeness.[1]

```
error_handler:
     if (yyerrorstatus > 2) {
          yyerror("syntax error");
     }
user_error_handler:
     if (yyerrorstatus == 0) {
          if (token == 0) YYABORT;              // End of input.
          token = yylex();
          switch ((stack-1)->state) {
```

[1]Our error recovery implementation assumes that at most `MAX_UINT-3` tokens will ever be shifted between syntactic errors — given that this number is 4,294,967,293 on a 32-bit machine, we feel this is safe. Even then, the mechanism is only flawed for the last 3 tokens out of every `MAX_UINT` tokens, and furthermore, the assumption cannot disturb a correct parse, only error-recovery processing. Making it completely safe would be trivial, but would require a conditional increment at each shift, which we consider too costly for the benefit.

```
        case 0:  goto state_action_0;
        case 1:  goto state_action_1;
        ⋮
        }
} else {
        yyerrorstatus = 0;
        while (stack != stack_start) {
                switch ((stack-1)->state) {
                case N:  goto state_M;        // iff M = goto[N, error].
                ⋮
                }
                stack--;
        }
        YYABORT;                              // Empty stack.
}
```

The case arms are the only part of the code that depends on the automaton. Any state that has an outgoing transition on `yacc`'s special `error` symbol will have a case arm in the second `switch` statement.

5 Experimental Results

5.1 Implementation

Our prototype implementation of `mule` is an adaptation of `bison`. `bison`'s grammar analysis remains unchanged. From `bison`'s internal tables, `mule` directly produces hard-code.

5.2 Generated-Parser Statistics

We have tested `mule`'s generated parsers on two hardware platforms, using different C compilers. For out initial tests, we used a grammar for subset of C. To isolate parser costs, we have a trivial lexical analyzer that reads token numbers from a pre-initialized array. Therefore, lexical analysis costs only a procedure call, a table lookup, and an index increment per token. No semantic actions are invoked during timings.

We tested `bison`'s, `yacc`'s, and `mule`'s parsers on the following platforms, using both `gcc` and the vendor's C compiler.

- DEC AlphaStation. 233MHz.
- SPARC Station 10.

Table 1 summarizes the results of parsing 8,790,000 tokens (879 tokens, 10,000 times) with the different hardware/compiler combinations. Compiles used the `-O` flag and any necessary flags to increase the basic block limit of the optimizer — `mule` creates a really huge `yylex` function. The "*worst*" columns indicate the speedup and expansion of `mule`'s parsers relative to the better of `yacc` or `bison` — in other words, they give the ratios that conservatively indicate `mule`'s behavior. (`bison` always produced smaller but slower parsers than `yacc`.)

Machine	Compiler	Speed (in sec.)				Size (in bytes)			
		`bison`	`yacc`	`mule`	*worst* speedup	`bison`	`yacc`	`mule`	*worst* increase
Alpha	`gcc`	14.45	8.90	2.22	4.0	7856	10176	13968	78%
	`cc`	10.82	7.70	2.00	3.8	7728	10000	15952	106%
SPARC	`gcc`	18.48	17.44	5.72	3.0	10270	15135	21103	105%
	`cc`	20.15	18.19	7.20	2.5	9792	14885	22315	128%

Table 1: Results

The results show speedup factors that range from 2.5 to 6.5, and size increases up to 128%. While a 128% increase in size may seem high, this is an increase of less than 13KB for a medium-sized grammar.

Tests show that `gcc`'s parser grew by less than 75KB when built with `mule`. This represented approximately a five-fold size increase. Because it is difficult to run `gcc`'s parser in isolation, we do not yet have a speed comparison, but we expect comparable speedups to those for the subset grammar, unless instruction-cache effects hurt the larger `gcc` parser.

6 Conclusion

We implemented the simplest `yacc`-compatible hard-coded-parser generator that we could imagine and it creates very fast parsers from a trivial translation of the input grammar and the LALR(1) automaton. Despite the simplicity, `mule`'s parsers are 2.5 to 6.5 times faster than `yacc`'s or `bison`'s, while growing by less than 75KB — a very reasonable tradeoff in many situations.

We believe the complexity of previous work in the area of hard-coding LR parsers unfairly prejudiced implementors against hard-coding `yacc`'s parsers — on the belief that heavy-duty optimizations were necessary to get reasonable time/speed behavior; we hope that our experinece with `mule` will remove this prejudice.

References

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

[FL88] Charles N. Fischer and Richard J. Leblanc, Jr. *Crafting a Compiler.* Benjamin/Cummings, Menlo Park, California, 1988.

[HW90] R. N. Horspool and M. Whitney. Even faster LR parsing. *Software Practice and Experience*, 20(6):515–535, June 1990.

[Joh75] Steven C. Johnson. Yacc — yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly & Associates, Inc., 1992.

[Pen86] Thomas J. Pennello. Very fast LR parsing. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 145–151, 1986.

[Pfa90] Peter Pfahler. Optimizing directly executable LR parsers. In *Compiler Compilers: Third International Workshop CC'90*, pages 179–192, October 1990.