How to improve the performance of YECC-generated Erlang (JAM) parsers

SERC-0050 Rev. B
Torbjörn Törnkvist
971212

1 Introduction

YECC is a parser generator written in Erlang [1]. YECC - like the C YACC - takes a LALR(1) grammar as input. YECC generates a parser implemented in Erlang as output. In this text we will explore some ideas described in a paper by Bhamidipathy & Proebsting [2]. The results indicates a 10 times speed up in the generated parsers is possible.

2 Related work

In the paper mentioned earlier [2], the authors describe how they achieve a significant speed up for YACC generated parsers using their method. What they have done is to generate directly, executable, hard-coded parsers. This is in contrast to the standard YACC method of generating interpreted, table-driven parsers. This alternative way of generating parsers seems to be very promising for parsers running in the Erlang (JAM) machine. The results from this work will be presented in the following text.

3 Implementation

The current YECC implementation generates two main functions, one for the action table, and one for the goto table. Each function consist of a (possibly large) number of clauses. For example, the action clause consist of seven arguments where the first argument is an integer representing the state and the second argument representing the token. The correct clause is chosen by the built-in pattern matching mechanism. Since the Erlang JAM system doesn't implement hash indexing on the arguments, we suspected that the alternative way of generating the parser could be advantageous. This alternative method is explored in this paper.

We changed the way the code is generated in YECC. So for example, the originally YECC generated action function was split into one function per state in our modified version. In our modified version each state therefore only needed six arguments, having the token as the first argument. The two code extracts below show the originally YECC generated code compared to our modified YECC generated code.

```
yeccpars2(1, atom, __Ss, __Stack, __T, __Ts, __Tzr) ->
   yeccpars1(__Ts, __Tzr, 3, [1 | __Ss], [__T | __Stack]);
yeccpars2(1, '(', __Ss, __Stack, __T, __Ts, __Tzr) ->
   yeccpars1(__Ts, __Tzr, 1, [1 | __Ss], [__T | __Stack]);
```

```
yeccpars2(1, __Cat, __Ss,  __Stack, __T, __Ts, __Tzr) ->
   __Val =  nil,
   yeccpars2(5, __Cat, [1 | __Ss], [__Val | __Stack], __T, __Ts, __Tzr
yeccpars2(2, '$end', _, __Stack, _, _, _) ->
   {ok, hd(__Stack)};
yeccpars2(2, _, _, _, __T, _, _) ->
   yeccerror(__T);
```

Figure 1. Two (small) states in the old style

```
state_1(atom, __Ss, __Stack, __T, __Ts, __Tzr) ->
   yeccpars1(__Ts, __Tzr, state_3, [state_1 | __Ss], [__T | __Stack]);
state_1('(', __Ss, __Stack, __T, __Ts, __Tzr) ->
   yeccpars1(__Ts, __Tzr, state_1, [state_1 | __Ss], [__T | __Stack]);
state_1( __Cat, __Ss,  __Stack, __T, __Ts, __Tzr) ->
   __Val =  nil,
   state_5( __Cat, [state_1 | __Ss], [__Val | __Stack], __T, __Ts, __T

state_2('$end', _, __Stack, _, _, _) ->
   {ok, hd(__Stack)};
state_2(_, _, _, __T, _, _) ->
   yeccerror(__T).
```

Figure 2. Two (small) states in the alternative style

The generation of the GOTO function was changed in a similar way.

4 Result

In order to measure the results, we made use of three grammars of different size. The table in Figure 3 shows the number of rules for each grammar and the number of tokens which were sent into the parser.

	No of rules	No of tokens
SMALL	9	9
MEDIUM	112	97
LARGE	197	551

Figure 3. Grammar sizes

We measured the time for generating the parser, compiling the parser, and running the parser. The results are shown in the Figures 4-6 below.

	Gen (old)	Gen (new)	Speedup
SMALL	115	105	1.1
MEDIUM	2730	2815	0.97
LARGE	5860	6215	0.94

Figure 4. Generation Time

	Comp (old)	Comp (new)	Speedup
SMALL	630	695	0.91
MEDIUM	8675	8715	0.99
LARGE	14300	14600	0.98

Figure 5. Compilation Time

	Exec (old)	Exec (new)	Speedup
SMALL	0.24	0.23	1.0
MEDIUM	31.9	2.6	12.3
LARGE	464	119.5	3.9

Figure 6. Execution Time

As can be seen from the tables above, the generation and the compilation times are almost equivalent between the two YECC variants. However, the execution times all decreased. The new mechanism's provided up to 12 times speed up.

5 Conclusion

The results of our measurements clearly indicates that the alternative way of generating YECC parsers is to be preferred. However, if a future Erlang JAM compiler should introduce argument indexing, the benefits will vanish. This was demonstrated when we ran the test using the Erlang BEAM system (which uses argument indexing), where the old YECC style of generating parsers outperformed the alternative method described here.

An other interesting result from this experiment is that also huge grammars can be generated and compiled. In particular the latter has been a problem so far. For example a complete SQL grammar with ~1500 rules resulted in 40000 lines of generated code. With the original way of generating the parser the Action and Goto functions become too large for the Erlang compiler, and could not be compiled. With the alternative way of generating, the resulting Erlang code was compiled in a couple of minutes.

6 References


```
[1]  -  Concurrent Programming In Erlang,
        Armstrong, Virding, Williams, Wikström,
        Prentice-Hall, 2:ed, ISBN 0-13-508301-X

[2]  -  Very Fast YACC-Compatible Parsers,
        Bhamidipaty, Proebsting,
        University of Arizona, Department of Comp.Sci.
        Technical report: TR 95-09
```