Build code with lex and yacc, Part 2: Development and troubleshooting

Get started actually building something

Peter Seebach August 24, 2004

The second article of this two-part series explores more advanced lex/yacc development and introduces basic troubleshooting techniques. See e-mail headers parsed before your very eyes! Marvel at cryptic error messages! See a computer actually compute something!

Building on the ground we covered in Part 1 of this two-part series, this article covers a couple of concrete applications of lex and yacc and discusses some common pitfalls. The examples -- a simple calculator and a program to parse e-mail messages -- are simple enough.

Lex-only code

While lex can be used to generate sequences of tokens for a parser, it can also be used to perform simple input operations. A number of programs have been written to convert simple English text into dialects using only lex, for instance. The most famous is probably the classic Swedish Chef translator, widely distributed on the Internet. These work by recognizing simple bits and pieces of patterns, and acting on them immediately. A good lexer example can help a lot with learning how to write a tokenizer. A link to the Swedish Chef program (sources are available, so you can see exactly how it's done) is in the Related topics section of this article.

Most simple programming projects, of course, can get by with very trivial lexers. A bit of care put into designing a language can help simplify this substantially, by guaranteeing that tokens can be recognized reliably without needing to know about context. Of course, not all languages are this congenial; for instance, lex has a hard time with C string constants, where special characters modify other characters.

Toy example: A calculator

Here's a simple example of a program that can be written in a page of yacc and a half-page of lex code. It illustrates a few interesting points, and it's even sort of handy... occasionally.

Listing 1. Lexer for a simple calculator

```
%{
#include "calc.h"
#include "y.tab.h"
%}
NUMBER  [0-9]+
OP [+-/*]
%%
{NUMBER} { yylval.value = strtol(yytext, 0, 10); return NUMBER; }
({OP}|\n) { return yytext[0]; }
.  { ; }
```

All this does is match numbers, operators, and newlines. Numbers have the special token `NUMBER` returned; everything else is just returned as plain characters. Unmatched characters are silently ignored.

Listing 2. Parser for a simple calculator

```
%{
#include <stdio.h>
#include "calc.h"
%}
%union {
 long value;
}
%type <value> expression add_expr mul_expr
%token <value> NUMBER

%%
input:
   expression
 | input expression

expression:
   expression '\n' { printf("%ld\n", $1); }
 | expression '+' add_expr { $$ = $1 + $3; }
 | expression '-' add_expr { $$ = $1 - $3; }
 | add_expr { $$ = $1; }

add_expr:
   add_expr '*' mul_expr { $$ = $1 * $3; }
 | add_expr '/' mul_expr { $$ = $1 / $3; }
 | mul_expr { $$ = $1; }

mul_expr:
   NUMBER { $$ = $1; }
```

This parser illustrates one way in which you can handle precedence and associativity without explicitly using the `%prec` declarations to guide yacc in resolving conflicts. If looser-binding operators take as operands expressions using tighter-binding operators, precedence is handled automatically. This is how, for instance, the C standard defines its formal grammar.

Note that, when a newline is seen after an expression, the value of the expression is simply printed. No long-term storage is required, no variables are modified. This works fine for a quick desktop calculator.

Note also the use of left recursion. Yacc prefers left-recursion. You can write the rule for "input" just as correctly as:

```
input: expression
| expression input
```

However, if you do this, each expression you write will cause another layer of recursion in the parser; as written originally, each expression is reduced to an *input* object before the next one needs to be parsed. Left-recursion is more efficient and, for large input sets, may be the only viable option.

Concrete example: Parsing e-mail

One likely task is reading a file containing e-mail messages and extracting their headers and contents. This is an interesting example, because the rules for reading headers are complicated enough to make it more practical to use start states for some of them. The lexer actually does some of the work of figuring out where in a message it is, but the parser still ties everything together.

The sample program can read individual messages, or the Berkeley "mbox" format. It uses a line starting with "From " as a message separator; this could be adapted, but it works well enough.

The support code for this is fairly straightforward. The "sz" type is a code convenience, a string library that hides all the work of reallocating strings when they grow. The structures are fairly minimalist and not very well generalized; this is to make the code smaller and easier to read. Real code would be a bit more general.

The grammar is mostly fairly simple. A file consists of one or more messages. Left recursion is used so that the rule can be reduced after each message. A message consists of a header, an empty line, and a body. The body is easy enough to read: a bunch of lines. You may note that the body has no explicit terminator. Rather, the first thing that isn't part of the body of a message must match the beginnings of the next possible object; a bit of reading back and forth reveals that it must be part of a header. Since only `LINE` tokens can occur in a body, the next other token that shows up will be part of a header -- or a syntax error. As a special case (implicit in yacc, rather than stated in the grammar), a 0 token indicates that there are no more tokens, and ends parsing. If the start rule (in this case, `file`) has successfully reduced, the parse is considered successful.

The header grammar is fairly complicated, although not as complicated as it could be. A header may start with a `headerline`, or with a `FROMLINE` token. The `headerline` non-terminal, in turn, requires a header name, a header body, and optional continuations. Once again, the continuation lines are handled using left recursion to keep things efficient. Note that, in theory, this grammar would accept a header with multiple "From " lines in it; this could be fixed, but there's no obvious reason to worry about it.

The lexer for this example is more complicated than the one for the calculator; quite a bit so. It uses start states -- a feature allowing the lexer to match some rules only some of the time. The two states it uses are named `BODY` and `HEADER`. The `HEADER` state is used when parsing the value part of a given name/value pair, not for parsing the entire message header. The reason that there are `HEADER` and `BODY` rules for matching the `LINE` pattern is to make sure that the lexer identifies

header names and continuations before it starts just handing back full lines that the parser would then have to analyze to decide what to do with them.

This does mean that the lexer and the parser both have to know that an empty line separates a header from a body. Similarly, the lexer has to know about the structure of continuations; the parser only knows that it sometimes gets additional text to add on to a header.

In this program, the parser uses global variables to track the list of messages parsed. This is how the test program gets access to the list of messages; the call to `yyparse()` returns only a success or failure indication, not any objects it may have created. (The return value is zero for a successful parse, and non-zero otherwise.)

Troubleshooting common problems

Lex and yacc are very good at parsing reasonably simple file formats. The biggest limitation is that yacc is limited to one token of lookahead. This means that if yacc would have to read two tokens before it knew which action to take, it would be unable to parse this grammar.

Listing 3. Uncle Shelby's ABZ, as a yacc grammar

```
a_f:
   'a' 'b' 'c' 'd' 'e' 'f' { printf("alphabet\n"); }
 | 'a' 'b' 'z' 'd' 'e' 'f' { printf("Silverstein\n"); }
```

In this case, yacc can tell, by the time it has to take an action, which alphabet it's parsing -- the regular one, or Uncle Shelby's ABZ. On the other hand, if you wrote the rules like this, it wouldn't work:

Listing 4. An unparsable grammar

```
a_f:
   'a' { printf("alphabet\n"); } 'b' 'c' 'd' 'e' 'f'
 | 'a' { printf("Silverstein\n"); } 'b' 'z' 'd' 'e' 'f'
```

In this case, the only token yacc would have available to decide which of these two actions to take would be the letter "b," which is the same in both rules. This grammar is ambiguous; yacc can't figure out what's going on.

Yacc can look ahead one character, though. So:

Listing 5. A parsable grammar

```
a_f:
   'a' 'b' { printf("alphabet\n"); } 'c' 'd' 'e' 'f'
 | 'a' 'b' { printf("Silverstein\n"); } 'z' 'd' 'e' 'f'
```

This version is fine. When yacc gets the "c" or "z," it knows enough to determine which rule to use. This is normally enough flexibility to handle reasonably simple file formats.

Conflict resolution

Most people eventually run into either a shift/reduce conflict or a reduce/reduce conflict. The first is less problematic, the second generally more severe. The most well-known example of a shift/reduce conflict is the "dangling else" ambiguity in C.

Imagine a yacc grammar something like this:

Listing 6. Dangling else

```
statement   if '(' condition ')' statement
          | if '(' condition ')' statement else statement
  | printf '(' string ')' ';'
```

This is fairly similar to the actual C grammar, if perhaps a bit simplified. Now, consider what happens to this in the case where a conditional statement is nested within another conditional statement:

Listing 7. Nested conditional

```
if (a)
if (b)
printf("a and b\n");
else
printf("???\n");
```

Does the `else` token get attached to the inner `if` (`if (b)`) or the outer one? Both are possible. Either way, there's one instance of the if/else construct, and one plain if. This is called a shift/reduce conflict. When yacc sees the `else` token, it can either *reduce* the sequence (from `if (b)` through the semicolon) into a statement (already having reduced the whole `printf` line into a statement), or it can *shift* the `else` token onto it, making it into the first half of a longer pattern. In fact, yacc's interpretation will be this one:

Listing 8. The dangling else, resolved

```
if (a)
 if (b)
  printf("a and b\n");
 else
  printf("a and not-b\n");
```

The default behavior is to prefer a shift over a reduction, and this is most often what you want. This behavior can be enforced by setting the precedence of the different patterns. Another option is to design your grammar to exclude this ambiguity: for instance, in perl, the braces around the body if an `if` statement are not optional, making it easy to tell whether the `else` is part of the inner `if` statement or the outer one.

Lexers can be problematic too. A common problem is having a rule match too much text; worse, since lex prefers the longest match it can find, this can result in a totally inappropriate rule being matched. Start states can help with this a lot. A more powerful tool is exclusive states, in which

only rules matching the exclusive state can be matched (this is not supported in all versions of lex, but should be available in anything modern).

If your version of lex lacks exclusive states, you can qualify every rule with states, and switch between them. Use a `%{..%}` block to `BEGIN` the state you want to use as a default state, and it'll work as though all other states were exclusive.

Multiple parsers and lexers

It used to be that using multiple parsers within a single program required a fair amount of careful tuning of the files generated by lex and yacc. Conveniently, this has been corrected: modern versions of both lex and yacc allow you to specify a prefix to use on the names of symbols in generated code. Just do that -- it's easier. In flex, the option is `-Pprefix`; in modern yacc, it's most often `-p prefix`. You can almost always arrange to use flex or bison, if the default lex and yacc available to you don't support this.

Going further

One of the best things you can do to learn more about lex and yacc is to write a few toy programs. Sample programs are also easy to find on the Internet: a bit of searching will find all sorts of cool stuff. A few of the links at the end of the article might help get you started.

Whenever possible, try to develop separate tests for your lexer and your parser. When you have a problem, knowing what went wrong is crucial; knowing whether it's the wrong token or an error in your grammar is a good starting point. A bit of work put into writing a good `yyerror` routine will help a lot. As a grammar gets more complicated, it gets more important to give clear and meaningful diagnostics of the errors encountered.

Related topics

- Part 1 of this series introduces lex, yacc, flex, and bison; check it out before reading this installment.
- The Lex & Yacc Page, or "The asteroid to kill this dinosaur is still in orbit," has a number of interesting historical references as well as very good lex and yacc documentation.
- The GNU versions of lex and yacc are flex and bison. As with all things GNU, they have excellent documentation including complete user manuals in a variety of formats.
- Bork bork bork! Chef/Jive/ValSpeak/Pig is a translator that speaks Swedish Chef, Valley Girl, Jive, and Pig Latin: written in pure lex code. Put in "lex and yacc are fun" and it comes out "lex und yecc ere-a foon" (The Muppet Show's Swedish Chef is the default).
- ANSI C yacc grammar is a reasonably complete C89 grammar, done in yacc. Complete with matching lexer. Write your own C compiler! Just needs a little work. :)