Yacc basics

Handout written by Maggie Johnson and revised by me.

Yacc is to parsers as lex is to scanners. You provide the input of a grammar specification and yacc generates an LALR(1) parser to recognize sentences in that grammar. Yacc stands for "yet another compiler compiler" and it is probably the most common of the LALR tools out there. Our programming projects are actually configured to use bison, a close relative of the yak, but you will likely only use the yacc-specific feature set. We'll tell you a little bit about yacc now, and on the "Other materials" section of our class web site, you'll find links to various documentation and resources for coming up to speed on yacc that will come in pretty handy for pp2.

A complete yacc specification

A yacc input file mostly consists of a listing of rules. Each rule represents a production in the grammar. It has a non-terminal on the left-hand side and one or more right-hand side expansions. You can associate an action with each rule, which allows you to do whatever processing is needed upon reducing that production. Let's dive right in and look over a full yacc specification:

```
%token  T_Int
%%

S : E             {printf("%d\n", $1);}
;
E : T             {$$ = $1;}
  | E A T         {switch ($2) {
                     case '+': $$ = $1 + $3; break;
                     case '-': $$ = $1 - $3; break;
                     }
                   }
;
T : F             {$$ = $1;}
  | T M F         {switch ($2) {
                      case '*': $$ = $1 * $3; break;
                      case '/': $$ = $1 / $3; break;
                     }
                   }
;
F : '(' E ')'     {$$ = $2;}
  | T_Int         {$$ = $1;}
;
A : '+'           {$$ = '+';}
  | '-'           {$$ = '-';}
;
M : '*'           {$$ = '*';}
  | '/'           {$$ = '/';}
;
%%

main()
{
    if (yyparse() == 0)
      printf("Valid.\n");
    else
      printf("Invalid.\n");
}
```

A few things worth pointing out in the above example:

- Just like lex, there are three sections to a yacc input file: declarations, rules, and user subroutines. The (required) first `%%` marks the end of declarations and beginning of rules, the (optional) second `%%` marks the end of the rules and beginning of user subroutines.
- In the declarations section, you declare tokens and types, add precedence and associativity options, and so on. `%` precedes special symbols in yacc such as token or type.
- All token types returned from the lex must be declared using `%token` in the declarations section. This establishes the token codes that will be used by the scanner to tell the parser what the next token is. In addition, the global variable `yylval` is used to store additional attribute information about the lexeme itself.
- For each rule, a colon is used in place of the arrow, a vertical bar separates the various productions, and a semicolon terminates the rule. Unlike lex, yacc pays no attention to line boundaries in the rules section which means you are free to use lots of whitespace to make the grammar easier to read.
- Within the braces for the action associated with a production is just ordinary C code. If no action is present, the parser will take no action upon reducing that production.
- The first rule in the file is assumed to identify the start symbol for the grammar.
- `yyparse()` is the function generated by yacc. It reads input from stdin, attempting to work its way back from the input to a valid reduction back to the start symbol. The return code from the function is 0 if the parse was successful and 1 otherwise. If it encounters an error (i.e. the next token in the input stream cannot be shifted), it calls the routine `yyerror(),` which by default prints the generic “parse error” message and causes parsing to halt.

The scanner to go with

In order to try out our parser, we need to create the scanner for it. Here is the lex file for this spec:

```
%{
  #include "y.tab.h"
  extern int yylval;
%}
%%
[0-9]+            { yylval = atoi(yytext); return T_Int;}
.                 { return yytext[0];}
[ \t\n]*          { /* ignore */ }
```

Given the above specification, `yylex()` will return the ASCII representation of most chars it sees, ignore whitespace, and recognize integers. When it assembles a series of digits into an integer, it converts to a numeric value and stores in `yylval` (the global reserved for passing attributes from the scanner to the parser) The token type T_Int is returned. Both lex and yacc need to be using the same numbers to distinguish the token types. Yacc will map the name T_Int (and any other tokens declared in the spec) to some integer constant > 256 and lists those #defines in the generated y.tab.h file that is included in the scanner.

The makefile to build it

In order to tie this all together, we first run yacc on the grammar specification to generate the y.tab.c and y.tab.h files, and then run lex on the scanner specification to generate the lex.yy.c file. Compile the two .c files and link them together, and voila— a calculator is born! Here’s the makefile:

```
express:    lex.yy.o y.tab.o
            gcc -o express lex.yy.o y.tab.o  -ll -ly
```

```
lex.yy.c:   express.l
            lex express.l

y.tab.c:    express.y
            yacc -vd express.y
```

Conflict resolution

What happens when you feed yacc a grammar that is not LALR(1)? Yacc reports any conflicts when trying to fill in the table, but rather than just throwing up its hands, it has automatic rules for resolving the conflicts and building a table anyway. For a shift/reduce conflict, yacc will choose the shift. In a reduce/reduce conflict, it will reduce using the rule declared first in the file (rarely used). These heuristics can change the language that is accepted and may not be what you want. Better than letting yacc pick for you, you can control what happens by explicitly declaring precedence and associativity for your operators.

For example, ask yacc to generate a parser for this ambiguous expression grammar.

```
%token T_int
%%

E :   E '+' E
      | E '*' E
      | E '=' E
      | '(' E ')'
      | T_int;
```

In the y.output file genereated from yacc, it tells you some facts about the table:

```
...
8/127 terminals, 1/600 nonterminals
6/300 grammar rules, 12/1000 states
9 shift/reduce, 0 reduce/reduce conflicts reported
...
```

If you look through the y.output file, you will see it contains the family of configurating sets and the actions for each input token. See, understanding all that LR(1) construction just might be useful after all! Rather than re-writing the grammar to implicitly control the precedence and associativity with a bunch of intermediate non-terminals, we can directly indicate the precedence so that yacc will know how to break ties. In the declarations section, we can add any number of precedence levels, one per line, from lowest to highest, and indicate the associativity (either left, right, or nonassoc):

```
%token T_int
%right '='
%left '+'
%left '*'
%%

E :   E '+' E
      | E '*' E
      | E '=' E
      | '(' E ')'
      | T_int;
```

The above file says that assignment has the lowest precedence and it associates right to left. Addition is next highest, multiplication even higher, and both of those associate left to right. Yacc finds no conflicts now, they were disambiguated by the precedence and associativity constraints. Where it encounters a shift/reduce conflict, if the precedence of the token to be shifted is higher than that of the rule to reduce, it will prefer the shift and vice versa. The precedence of a rule is

determined by the precedence of the rightmost terminal on the right-hand side (or can be explicitly set with the `%prec` directive). Thus if a 4 + 5 is on the stack and * is coming up, the * has higher precedence than the 4 + 5, so it shifts. If 4 * 5 is on the stack and + is coming up, it reduces. If 4 +5 is on the stack and + is coming up, the associativity breaks the tie, a left-to-right associativity would reduce the rule and then go on, a right-to-left would shift and postpone the reduce.

Even though it doesn't seem like a precedence problem, the dangling else ambiguity can be resolved using precedence rules. By insuring that the precedence of the else token is higher than the alternate production that can be reduced at that point, you can indicate you want yacc to choose the shift.

Error Handling

The special `error` token allows you to attempt some form of error recovery on an unsuccessful parse. An error production marks a context in which erroneous tokens can be deleted until something "familiar" is seen that allows the parser to continue. After reporting a syntax error via `yyerror()`, the parser will discard any partially parsed rules (i.e. pop stacks from the parse stack) until it finds one in which it can shift an error token. It then reads and discards input tokens until it finds one that can follow the error token in that production. For example, in a language like C, the rule for a simple statement might include an error token followed by a semicolon, which allows it to recover from a malformed statement by quickly scanning ahead to the next semicolon. In our expression grammar, any failed expression will bomb out to the error state below and pick back up at the next token in the follow of E.

```
%token T_Int
%right '='
%left '+'
%left '*'
%%

E :   E '+' E
      | E '*' E
      | E '=' E
      | '(' E ')'
      | T_Int
      | error
;
%%

void yyerror(char *msg)
{
      printf("Line %d: ", yylineno);
      printf("%s\n", msg);
      printf("Last token was %s\n", yytext);
}
```

Where should one put error productions? "Places where errors are expected..." It's actually more of black art than a science. As a first attempt, adding error tokens fairly-high up and using punctuation as the synchronizing tokens is a reasonable way to start. However, the parser's knowledge is just too limited to really handle this for a compiler—we'll see the addition of semantic information can be crucial in developing a coherent error recovery strategy.

Bibliography

J. Levine, T. Mason, and D. Brown, Lex and Yacc. Sebastopol, CA: O'Reilly & Associates, 1992.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.