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 complete yacc input file for a simple postfix calculator:

```
%{
  #include <stdio.h>
  #include <assert.h>

  static int Pop();
  static int Top();
  static void Push(int val);
%}

%token T_Int

%%

S   :  S E '\n' { printf("= %d\n", Top()); }
    |
    ;


E   :  E E '+' { Push(Pop() + Pop()); }
    |  E E '-' { int op2 = Pop(); Push(Pop() - op2); }
    |  E E '*' { Push(Pop() * Pop()); }
    |  E E '/' { int op2 = Pop(); Push(Pop() / op2); }
    |  E '!'   { Push(-Pop()); }
    |  T_Int   { Push(yylval); }
    ;

%%

static int stack[100], count = 0;

static int Pop() {
    assert(count > 0);
    return stack[--count];
}
static int Top() {
```

```
    assert(count > 0);
    return stack[count-1];
}
static void Push(int val) {
    assert(count < sizeof(stack)/sizeof(*stack));
    stack[count++] = val;
}

main() {
   return yyparse();
}
```

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;}
[-+*/!\n]         { return yytext[0];}
.                 { /* ignore everything else */ }
```

Given the above specification, `yylex()` will return the ASCII representation of the calculator operators, recognize integers , and ignore all other characters. 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:

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

lex.yy.c:   calc.l y.tab.c
            lex calc.l

y.tab.c:    calc.y
            yacc -vd calc.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. However, it is not recommended to let 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 (the output from bison is slightly different):

```
...
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 default behavior when a yacc parser encounters an error (i.e. the next input token cannot be shifted given the sequence of things so far on the stack), is to call the routine `yyerror()` to print a generic "parse error" message and halt parsing. However, quitting at sight of the very first error is not a desirable behavior for a compiler!

Yacc supports a form of error re-synchronization that allows you to define where in the stream to give up on an unsuccessful parse and how far to scan ahead and try to cleanup to allow the parse to restart. The special `error` token can be used in the right side of a production to mark a context in which to attempt error recovery. The usual use is to add the error token possibly followed by a sequence of one or more synchronizing tokens which tell the parser to discard any tokens until it sees that "familiar" sequence that allows the parser to continue. A simple and useful strategy might be simply to skip the rest of the current input line or current statement when an error is detected.

When an error is encountered (and reported 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:

```
Var  :  Modifiers Type IdentifierList ';'
     |  error ';'
```

The second production allows for handling error encountered when trying to recognize `Var` by accepting the alternate sequence `error` followed by a semicolon. What happens if the parser in the in the middle of processing the `IdentifierList` when it encounters an error? The error recovery rule, interpreted strictly, applies to the precise sequence of an `error` and a semicolon. If an error occurs in the middle of an `IdentifierList`, there are `Modfiers` and a `Type` and what not on the stack, which doesn't seem to fit the pattern. However, yacc can force the situation to fit the rule, by discarding the partially processed rules (i.e. popping states from the stack) until it gets back to a

state in which the `error` token is acceptable (i.e. all the way back to the state at which it started before matching the `Modifiers`). At this point the `error` token can be shifted. Then, if the look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, yacc reads and discards input until the next semi-colon so that the error rule can apply. It then reduces that to `Var`, and continues on from there. This basically allowed for a mangled variable declaration to be ignored up to the terminating semicolon, which was a reasonable attempt at getting the parser back on track. Note that if a specific token follows the error symbol it will discard until it finds that token, otherwise it will discard until it finds any token in the lookahead set for the nonterminal (for example, anything that can follow `Var` in the example above).

In our postfix calculator from the beginning of the handout, we can add a error production so that any malformed expression will be handled by the error state below and discard the rest of the line up to the newline and allow the next line to be parsed normally:

```
S   :  S E '\n' { printf("= %d\n", Top());}
    |  error '\n' {printf("Error!\n");}
    |
    ;
```

Like any other yacc rule, one that contains an error can have an associated action. It would be typical at this point to clean up after the error and other necessary housekeeping so that the parse can resume.

Where should one put error productions? It's more of a black art than a science. Putting error tokens fairly high up tends to protect you from all sorts of errors by ensuring there is always a rule to which the parser can recover. On the other hand, you usually want to discard as little input as possible before recovering, and error tokens at lower level rules can help minimize the number of partially matched rules that are discarded during recovery.

One reasonable strategy is to add error tokens fairly-high up and use punctuation as the synchronizing tokens. For example, in a programming language expecting a list of declarations, it might make sense to allow error as one of the alternatives for a list entry. If punctuation separates elements of a list, you can use that punctuation in error rules to help find synchronization points—skipping ahead to the next parameter or the next statement in a list. Trying to add error actions at too low a level (say in expression handling) tends to be more difficult because of the lack of structure that allows you to determine when the expression ends and where to pick back up. Usually you focus on trying to recover from the common error situations (forgetting a semi-colon, mistyping a keyword), but don't usually put a lot of effort into dealing with more wacky inputs. Error recovery is largely a trial and error process.

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.