Parsing Tools

CS152

Chris Pollett

Sep. 29, 2008.

Outline

- Lex/Yacc

Lex/Yacc

- Lex is a tool for writing scanners; Yacc is a tool for writing parsers. Both originated at Bell Labs in 1970s.
- Flex/Bison are their roughly equivalent GNU counterparts.
- Both are C pre-processors. That is, we write C code with Lex/Yacc directives in them, run the Lex/Yacc preprocessor and get a pure C program which can then be compiled.
- The basic structure of both a Lex and a Yacc program is as follows:

```
%{// C code to insert verbatim at start of program
%}
/* Lex/Yacc Definitions */
%%
//Lex/Yacc code
%%
// more C code.
```

A Simple Lex Example

```
%{
#include <stdio.h>
int wordCount = 0;
%}
word [^ \t\n]+ /* make an abbreviation word for the expr [^ \t\n]+ */
%%
[\t\n ]+ {printf("I see whitespace\n");} //what to do if see pattern
{word} {wordCount++;}
%%
int main()
{
  yylex(); //call the lexer. Gets input from command line until ^D
  printf("word count: %d", wordCount); return 0;
}
```

- To compile:

  ```
  lex lextest.l -o lextest.c #default output is lex.yy.c
  gcc  lextest.c -o lextest -ll  #-ll not needed if use flex.
  ```

A Yacc Example

```
%{
#include <stdio.h>
%}
%token ARTICLE NORMAL_NOUN PROPER_NOUN
%%
noun_phrase : PROPER_NOUN { printf(“Proper Noun\n”); }
    | ARTICLE NORMAL_NOUN {printf(“Usual Noun\n”); }
%%
int main(int argc, char **argv)
{
   extern FILE *yyin;
   yyin = fopen(argv[1], "r"); //sets up lexer to use this file as input
  yyparse();
  fclose(yyin);
}
```

More on Yacc Example

- To compile the above you could use the line:

  yacc -d yacctest.y
- This will produce two files: y.tab.c and y.tab.h. If you use bison you'd get yacctest.tab.c and yacctest.tab.h
- The .h file contains #defines for the tokens ARTICLE, NORMAL_NOUN, etc.
- You would then need to write a lex program which includes y.tab.h.
- It might have a rule like:

  AlalThelthe {return ARTICLE;}

Still More on Yacc Example

- Once you have run yacc and lex on the above grammar and its corresponding scanner. To compile the whole thing you would type:

```
gcc -o yacctest y.tab.c lex.yy.c -ly -ll
```

Yacc $ variables

- Yacc refers to parts of a rule using variables which begin with a dollar sign:

```
expression : expression ‘+’ expression {$$ = $1 + $3;}
        | expression ‘-’ expression {$$ = $1 - $3;}
        | NUMBER {$$ =$1;}
        ;
```

- $$ refers to the left hand side of the rule value. $*n* refers to the *n*th item on the right hand side.

Typing Tokens

- Lex uses a few built-in global variables when it scans its input: yytext, yylval.
- The first is a char pointer to the string matching the current token. The second is used to store the value of the $ variable for the given token returned.
- yylval has type YYSTYPE which is a union that you can set up in your grammar.

More on Typed Tokens

- To set up YYSTYPE in your grammar (will appear in y.tab.h file after yacc'ing):

```
%{
//stuff
%}
%union {
     double dval; // in this case we have two possibilities
     int ival; // could have more. In real world possibilities
                    // would include a struct for a syntax tree.
}
%token <ival> INTEGER
%token <dval> DOUBLE
%type <dval> expression /*notice can say type of
   nonterminal */
```

Typed Tokens and the Lexer

- The lexer may then have rules like:

  [0-9]+ {yylval.ival = atoi(yytext); return INTEGER;}
- So if you had a rule in your grammar like:

  integer_expr : INTEGER {$$ = (double)$1;}

  //$1 would have been an int
- Typically, you use the typing mechanism so that you can build up a syntax tree for the input as you are parsing it.

Error Handling in Your Grammar

- If you Yacc encounter an error while parsing it will call the function yyerror to handle.
- If you like, you can rewrite this function to do whatever you want:

```
int yyerror( char *s)
 {
    fprintf(stderr, “You caused the error: %s , bozo.\n”, s);
    return 1;
 }
```