CS24210: Using lex and yacc

Edel Sherratt

4 November 1999

1 A single digit calculator

1.1 Compiling and running the calculator

Download the file `calc.y` from the CS24210 web page. This file containe a yacc specification taken from *Aho, Sethi, Ullman , p. 259*) Here are the commands you'll need to preprocess and compile `calc.y`.

```
yacc calc.y
gcc -o calc y.tab.c -ly
calc
```

And here is an execution script

```
vapid% calc
7+2*9
25
^D
```

Notice the CTRL-D for end of input.

First `calc.y` is preprocessed using `yacc`. The resulting C program, `y.tab.c`, is then compiled and run.

Try running `calc.y` for yourself.

It is very crude. It can only handle single digit numbers, and it can't deal with spaces. Also, it only handles a single line of input, and it only knows about multiplication and addition.

1.2 The yacc specification `calc.y`

A yacc specification has three parts:

```
declarations
%%
translation rules
%%
C functions
```



```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :       expr'\n'        {printf("%d\n", $1); }
        ;
expr    :       expr '+' term   { $$ = $1 + $3; }
        |       term
        ;
term    :       term '*' factor { $$ = $1 * $3; }
        |       factor
        ;
factor  :       '(' expr ')'    { $$ = $2; }
        |       DIGIT
        ;

%%
int yylex (void) {
        int c;
        c = getchar();
        if (isdigit(c)) {
                yylval = c-'0';
                return DIGIT;
        }
        return c;
}
```

Figure 1: The yacc specification, calc.y

The yacc specification, `calc.y`, is shown in Figure 1.2.

The declarations part of `calc.y` imports standard header file `ctype.h`

```
%{
#include <ctype.h>
%}
```

`ctype.h` contains lots of declarations, but, in particular, it contains a declaration for `isdigit`, which is used in the function `yylex` in the supporting C functions part of `calc.y`.

In general, the brackets `%{ ...%}` at the start of a yacc specification contain ordinary C declarations.

The declarations part of `calc.y` also declares DIGIT to be a token that can be used later in the yacc specification.

```
%token DIGIT
```

The end of the declarations part of the yacc specification is marked by

```
%%
```

The rules part of the yacc specification contains ordinary grammar rules with associated actions.

You'll recognise the grammar rules in `calc.y` as the LR rules for describing arithmetic expressions that were used in lectures. The notation is a bit different – instead of

```
expr ::= expr + term | term
```

yacc expects

```
expr    :       expr '+' term
        |       term
        ;
```

The actions are written in C. Here is the print action which prints out the answer when the rule `line : expr` is applied.

```
line    :       expr '\n'        {printf("%d\n", $1); }
        ;
```

Here is an assignment which is executed when the rule `expr : expr '+' term` is applied.

```
expr    :       expr '+' term   { $$ = $1 + $3; }
```

The identifier

```
 $$
```

refers to a stackable value associated with the left side of the rule, while

```
 $1, $2, ...
```

refer to the stackable values associated with the symbols on the right side of the rule.

Some rules (like `expr : term`) don't have an explicit action in the yacc specification. When a rule is applied and there is no explicit action stated for the rule, the default action is taken:

```
$$ = $1
```

The third and final part of the yacc specification consists of C functions. In `calc.y`, there is only one supporting function, the scanner yylex, which must always be provided. yylex returns tokens, like DIGIT, to the parser, and it sets the value of yylval, a variable defined by yacc.

In this example, yylex is very crude. It treats each character as a lexeme. If the character is a digit, it returns the token DIGIT and sets yylval to the value of the digit. Otherwise, it returns the character itself.

Try modifying `calc.y` so that it also deals with subtraction and division.

What warnings do you see? Why do they matter? Think about expressions like 5-2-1).

2 A better calculator

2.1 Compiling and running the new calculator

The files `int_calc.y` and `int_expr.l` together specify a more sophisticated calculator. Download these from the CS24210 web page.

Here is an execution script showing how they are preprocessed, compiled and executed.

```
vapid% lex int_expr.l
vapid% yacc int_calc.y
vapid% gcc -o int_calc y.tab.c -ly -ll
vapid% int_calc
72 * 3 + 4
= 220

50 * (1+2)
= 150
^D
vapid%
```

First lex is used to generate `lex.yy.c`. Then yacc is called to generate `y.tab.c`. Then `y.tab.c` is compiled giving `int_calc`. Notice how the libraries `ly` and `ll` are also incorporated in the final program. It is important to include them in the correct order, because they each contain a `main`, and we want the `main` from `ly` to be used.

Try this for yourself. Don't forget the CTRL-D for end of input.

2.2 The lex specification `int_expr.l`

Here is the lex specification `int_expr.l`

```
%%
[ \t]   {/* whitespace, do nothing */}
[0-9]+  { sscanf(yytext, "%d", &yylval);
          return(NUMBER);
        }
[+]     return(ADDOP);
[*]     return(MULOP);
[(]     return(LPAR);
[)]     return(RPAR);
\n      return('\n');

%%
```

This specification consists of some regular expressions, with associated actions that return the tokens `NUMBER, ADDOP, MULOP, LPAR, RPAR` and that set the value of yylval.

Notice the 'do nothing' action (just a comment) associated with spaces and tabs, and the action to return a newline when one is encountered.

2.3 The yacc specification `int_calc.y`

Now look at the yacc specification `int_calc.y` (Figure 2.3).

This specification declares the tokens `NUMBER, ADDOP, MULOP, LPAR, RPAR`.

It also contains the rule alternatives

```
lines   :     lines expr'\n'      {printf("= %d\n", $2); }
        |     lines '\n'
        |     /* empty */
        |     error '\n'          { yyerror("Please reenter last line:")
;
                                    yyerrok;
                                  }
```

which allow for multiple lines of input, for blank lines, and for errors.

The third part of the specification, the supporting C functions part, now includes the file `lex.yy.c`. This file contains the function yylex(). `lex.yy.c` is generated by lex.

3 A double precision calculator

3.1 Compiling and running the double precision calculator

Download `double_expr.l` and `double_calc.y`.

Try running the double precision calculator. Here are the commands you'll need.

```
lex double_expr.l
yacc double_calc.y
gcc -o double_calc y.tab.c -ly -ll
```

And here is an execution script

```
vapid% double_calc
7.7*(1.5*2.2)
= 25.41


(8 + 2) * (3+1.6)
= 46
^D
```



```
%{
#include <ctype.h>
%}

%token NUMBER
%token ADDOP
%token MULOP
%token LPAR
%token RPAR

%%
lines   :     lines expr'\n'      {printf("= %d\n", $2); }
        |     lines '\n'
        |     /* empty */
        |     error '\n'          { yyerror("Please reenter last line:")
        ;
                                    yyerrok;
                                  }
        ;
expr    :     expr ADDOP term     { $$ = $1 + $3; }
        |     term
        ;
term    :     term MULOP factor   { $$ = $1 * $3; }
        |     factor
        ;
factor  :     LPAR expr RPAR      { $$ = $2; }
        |     NUMBER
        ;

%%
#include "lex.yy.c"
```

Figure 2: The yacc specification `int_calc.y`

3.2 The lex specification `double_expr.l`

Here is the lex specification `double_expr.l`.

```
number  [0-9]+\.?|[0-9]*\.[0-9]+
whitespace      [ \t]

%%
{whitespace}    {/* no action, no return */}
{number}        {sscanf(yytext, "%le", &yylval); return(NUMBER); }
[+]             return(ADDOP);
[*]             return(MULOP);
[(]             return(LPAR);
[)]             return(RPAR);
\n              return('\n');

%%
```

In general, a lex specification has three parts:

```
character class definitions
%%
regular expressions with associated actions
%%
C functions
```

This specification defines the character classes `whitespace` and `number` in its first section.

Notice that the numbers now allow decimal points. If you like, you can extend this specification so that numbers expressed using 'e' notation are also recognised. (Just change the definition of `number`

3.3 The yacc specification, `double_calc.y`

The yacc specification, `double_calc.y` is very nearly the same as `int_calc.y`.

This first difference is that `double_calc.y` defines the value of `YYSTYPE`, the type of values stored on the stack, to be `double`. (The default value for `YYSTYPE` is `int`).

```
%{
#include <ctype.h>
#define YYSTYPE double
%}
```

This means that we can now do arithmetic with double precision expressions.

The second (and final) difference is that the print action associated with the rule `lines : lines expr` now prints floating point values as well as integers.

Otherwise, the specification is identical to `int_calc.y`

Try modifying `double_calc.y` and `double_expr.l` so that they handle subtraction and division.

4 Using yacc to generate code

This final example shows how you can use yacc to construct a syntax tree for an arithmetic expression, and then generate code to calculate the expression.

Remember that the postfix code for an expression like

```
(23 + 45) * (1 + 2)
```

is

```
23 45 + 1 2 + *
```

This might be rendered as a sequence of instructions like

```
push 23 push 45 add push 1 push 2 add mpy
```

for a zero-address computer.

4.1 Compiling and running the postfix generator

To run the postfix generator download copies of

```
tree_expr.l
postfix.y
postorder.h
postorder.o
```

`postorder.h` contains definitions of the tree building functions `mknode` and `mkleaf`, as well as a postorder tree traversal which applies a function to each node visited.

`postorder.o` contains object code for these functions.

If you'd like to see the C sources for `postorder.o`, download `postorder.c`.

`tree_expr.l` and `postfix.y` are the lex and yacc specifications for the postfix generator.

Here is an execution script for the generator.

```
vapid% lex tree_expr.l
vapid% yacc postfix.y
vapid% gcc -o postfix y.tab.c -ly -ll postorder.o
vapid% postfix
(9.2 + 3 ) * (5.1 + 2 + 4)
push 9.2
push 3
add
push 5.1
push 2
add
push 4
add
mul
^D
```

lex and yacc have been used as before. However the compilation step – the call on `gcc` – includes the local object file `postorder.o` as well as ly and ll. Notice how no '-' sign is needed when including the local object file `postorder.o`.

Try running the postfix generator for yourself.

When you are satisfied, take a closer look at the lex and yacc specifications.

4.2 The lex specification `tree_expr.l`

Here is the lex specification `tree_expr.l`.

```
number  [0-9]+\.?|[0-9]*\.[0-9]+
whitespace      [ \t]

%%

{whitespace}    { /* whitespace, no action, no return */ }

{number}        { yylval.text = push_command(); return(NUMBER); }
[+]             { yylval.text = op_table[plus]; return(ADDOP); }
[*]             { yylval.text = op_table[times]; return(MULOP); }
[(]             return(LPAR);
[)]             return(RPAR);

\n              return('\n');
%%

char *push_command(void) {
        char *text = malloc((5+yyleng+1)*sizeof(char));
        strncpy(text,"push ",5);
        strncat(text,yytext,yyleng);
        return text;
}
```

This specification contains all three parts that can be found in a lex specification: character class definitions, regular expressions with associated actions, and a supporting C function.

The C function delivers a string consisting of the word 'push' followed by the number that has just been recognised. You don't need to worry too much about the interals of this function; C will be covered in CS23710!

Notice the references to `yylval.text` in the actions associated with some of the regular expressions. This means that 'yylval' is a structure, and 'text' is a component of that structure.

YYSTYPE is set to the type of this structure in the yacc specification `postfix.y`.

`postfix.y` also defines the table `op_table`, which contains the strings 'add' and 'mpy', and the integer indices 'plus' and 'times'.

4.3 The yacc specification `postfix.y`

Here is the declarations part of the yacc specification `postfix.y`.

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

#define TREE_LABEL_T char *
#include "postorder.h"

int visitor(char *label) {printf("%s\n",label); return 0;}

typedef struct {
        char *text;
        tree_t *treeptr;
} stack_t;

#define YYSTYPE stack_t

#define plus 0
#define times 2

static char op_table[5][4] = {"add\0","sub\0","mul\0","div\0","neg\0"};

char *push_command(void);

%}

%token NUMBER
%token ADDOP
%token MULOP
%token LPAR
%token RPAR

%%
```

It defines `TREE_LABEL_T` to be of type string (char * in C), and it then includes the header file `postorder.h`

It also defines a visitor that will be used by 'postorder' to print out the label of each node in the syntax tree.

It defines the type YYSTYPE to be a structure with fields 'text' and 'treeptr'. This means the the stack contains pointers to structures like those described in lectures.

It defines the indices 'plus' and 'times' used to index `op_table` in the lex specification, `tree_expr.l`, and it declares and initialises `op_table` to contain strings used for generating code for the operations (actually, only two of these entries are used).

It also includes a function prototype – a kind of declaration – for the C function 'push_command' used in `tree_expr.l`.

The middle section of the yacc specification – the part that contains the grammar rules and associated semantic actions – looks like this.

```
%%
lines   :     lines expr'\n'      { postorder($2.treeptr, visitor); }
        |     lines '\n'
        |     /* empty */
        |     error '\n'          { yyerror("Please reenter last line:")
;
                                    yyerrok;
                                  }
        ;
expr    :     expr ADDOP term     { $$.treeptr =
                      mknode( $2.text, $1.treeptr, $3.treeptr );
                                  }
        |     term
        ;
term    :     term MULOP factor   { $$.treeptr =
                      mknode( $2.text, $1.treeptr, $3.treeptr );
                                  }
        |     factor
        ;
factor  :     LPAR expr RPAR      { $$ = $2; }
        |     NUMBER              { $$.treeptr = mkleaf($1.text); }
        ;

%%
```

This section builds a syntax tree for the expression, labelling the tree with 'push' commands where numbers are encountered, and with 'add' or 'mpy' where '+' or '*' signs are encountered. The technique is exactly as shown in lectures.

Notice how the default action

```
$$ = $1
```

is used when rules like `term : factor` are applied.

When the tree is complete – when the rule `lines : lines expr` is applied – it is traversed in postorder, and the function 'visitor' prints the label of each node visited.

Try to to extend the postfix generator so that it also deals with subtraction and division. (Notice that the grammar rules need not be modified, since subtraction can be treated as an ADDOP, and division as a MULOP. You need only change the lex specification and the declarations part of the yacc specification).