An example using YACC and LEX together

The purpose of LEX is to build a DFA (as a C program) to correspond to anything expressible in LEX's input syntax — often a specification of legal identifiers for a programming language. YACC (**Y**et **A**another `C`ompiler `C`ompiler — parser-generators are sometimes called 'compiler compilers') builds a DPDA for parsing the more macroscopic structure of a program including procedures, nested blocks etc. The input to YACC is a set of grammar rules specified in a syntax similar to that of BNF together with 'compiling actions' to be activated whenever certain non-terminal symbols in the grammar are successfully recognised. YACC can cope with any set of grammar rules that can be transformed to the LALR(1) subset of Chomsky Type 2 (see separate 'parsing' handout for more details on this).

YACC and LEX were just two of the many software tools put out with the original UNIX operating system, from Bell Laboratories, when it was first released to Universities in the mid- to late- 1970s. They live on within the many commercial versions of UNIX and also in GNU public-domain form for Linux. The way that they communicate with one another is via a crude but effective use of global variables and global definitions inside a jointly-constructed C program.

To illustrate the stages of building a 'compiler' let us take a toy example of a compiler that will recognise arithmetic expressions in which the numbers must be expressed in roman numeral form. The compiler translates the roman numerals to arabic form, performs the computation and outputs the answer. The files shown below are in `roman1.l` and `roman1.y` respectively, within the distributed zip file. `/cs/documents/teaching/G51MAL` You can, of course, extract copies of these into your own directory for experimentation. Feel free to substitute your own regular expressions for the Roman Numbers To run this software you should do the following (where '`$`' is the UNIX shell prompt):

```
$ lex roman1.l
$ yacc roman1.y
$ cc y.tab.c -o myprog -ll
```

(or, if you are on a Linux system this will probably have to be:)

```
$ gcc y.tab.c -o myprog -lfl
$ myprog
```

Notice that the `#include` directive in the C program part of the YACC specification has the effect of textually substituting the entire file of C code which LEX has generated for the lexical analyser. This stratagem enables YACC and LEX to share the same 'code numbers' for tokens such as `NUMBER`.

After typing `myprog` the parser will expect you to type in expressions such as:

```
ix + lxix
```

and it will then tell you what the answer is (in Arabic notation). Note that there *must* be spaces around the + sign.

Here is `roman1.l`:

```
     char op;
     int sum = 0;
%%
(cm|CM) {ECHO; sum = sum + 900;}
(cd|CD) {ECHO; sum = sum + 400;}
(xc|XC) {ECHO; sum = sum + 90;};}
(xl|XL) {ECHO; sum = sum + 40;}
```

```
(ix|IX)  {ECHO;sum = sum + 9;}
(iv|IV)  {ECHO;sum = sum +4;}
(m|M)     {ECHO;sum = sum + 1000;}
(d|D)     {ECHO;sum = sum + 500;}
(c|C)     {ECHO;sum = sum + 100;}
(l|L)     {ECHO;sum = sum + 50;}
(x|X)     {ECHO;sum = sum + 10;}
(v|V)     {ECHO;sum = sum + 5;}
(i|I)     {ECHO;sum = sum + 1;}
"+"   {op = '+';}
"&"    {return('&');/*end of input*/}
@      {return('\n');}
\n    { if (sum != 0)
                { yylval=sum;
               printf ("    converts to %d\n", sum);
               sum =0;
               /* need to prepare value of previous roman number when
                * \n encountered.  But push back \n as @ onto input
                *  queue using 'unput'  so that it gets re-read and
                *  recognised. This then returns \n to YACC as marker
                *  for end of statement.
                */
               unput('@');
               return(NUMBER); /* tell YACC we've found a NUMBER*/
                 }
          else return (op);
           }
[ \t]        { /* prepare value of roman no. just encountered when space or
           * tab is recognised. If sum = 0 then an operator has been seen
           * so return it to YACC as its own ASCII code.
           */
          if (sum != 0)
                { yylval=sum;
               printf ("    converts to %d\n", sum);
               sum =0;
               return(NUMBER);
                 }
          else return (op);
           }
%%
```

This next one is roman1.y :

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

%start list

%token NUMBER

%%
list : /*empty*/
```

```
     |list stat '\n' /*accepts list of statements' separated by \n /*
     |list error '\n' /* error exit */
          { yyerror("Oh dear  -- syntax error. Baling out now"); exit(1);}
     ;


stat :     '&' /* & marks end of input */
               {printf("End of input-- Bye! ");exit(0);}
     |     NUMBER '+' NUMBER
               {printf(" Answer is %d\n", $1 + $3);}
     ;
%%
#include "lex.yy.c"

main() {  /* This is the main program. All it does is to call the parser
        * which yacc has created. You tinker with this at your peril!
        */

     return (yyparse() );

     }

yyerror(s)
     char *s;

/* routine called when an error occurs.
 * Prints out the string that you supply
 */

{
     fprintf (stderr, "%s0, s);

}
```

Notes

Distinguish carefully between the *code* that is returned for a token (`NUMBER` in this case) and the *value* associated with the token (which is returned in the global variable `yylval`). It is the value in `yylval` which becomes associated with the corresponding pseudo-variables such as `$1` and `$2` in YACC.

In YACC one can return a value for the non-terminal at the left-hand side of a rule in the pseudo-variable called `$$`. In the example given the computed value of the sum could be returned to a higher point in the parse by something like:

```
$$ = $1 + $3
```

This `$$` will be substituted into the RHS of other rules as and when it is needed (but note that this value will become some value such as `$1` or `$3` in the higher rule.