# Comparison of C vs Lex/Yacc Versions of Enhanced Chapter 1 Interpreter

## Introduction

This document describes the changes involved in rewriting the C version of the enhanced chapter 1 interpreter (https://github.com/ddawson631/chap1_enhanced_c) so it uses a parser and lexer generated by lex and yacc.
The new lex/yacc version is also written in C but I'll continue to refer to it as the lex/yacc version and to the version that I changed as the C version.

My goal with this project was to see how closely I could mimic the behavior of the C version with the lex/yacc version.
To that end, the grammar and instruction syntax did not change.
A few minor differences in behavior are documented in the unit test (UT) result comparison in Part 3 below.
The other differences listed there are due to the changes in error message format that are described in Table 2 below.

This document consists of the following 3 parts.

Part 1. Program Organization (includes Tables 1 & 2)
Part 2. Grammar, Yacc & Lex Specs (includes Table 3)
Part 3. Unit Test (UT) Result Comparison (includes Table 4)

## Part 1. Program Organization

In the C version, the main function uses a Read-Eval-Print-Loop (REPL) that reads user input, parses it, evaluates it, prints its value then returns to the beginning of the loop to read the next input. If a syntax or semantic error occurs then the errmsg() function displays error info and calls longjmp() to return to the main program and restart the REPL. The source code is in a single file (*chap1.c*) that is divided into the following 9 sections: DECLARATIONS, DATA STRUCTURE OPS, NAME MANAGEMENT, INPUT, NEW PARSING ROUTINES, ENVIRONMENTS, NUMBERS, EVALUATION, MAIN.

In the lex/yacc version, sections INPUT and NEW PARSING ROUTINES are deleted and replaced by the generated parser, *yyparse(),* and lexer, *yylex()*. MAIN (the main function) is moved into the parser specification (*chap1.y*). The other 6 sections are moved into the file *ch1_info.c.* The new main function now calls *yyparse()* which implements the REPL and calls *yylex()* to read, identify and return tokens from the input stream. When yyparse() detects a syntax error, it calls yyerror() to display the error message. So all syntax error messages were removed from the errmsg() function which is now used to only report semantic errors detected by application logic.

The following table compares the old vs. new program organization.

**Table 1** Program Organization Comparison

| | C Version | Lex / Yacc Version |
|---|---|---|
| **source file comparison** | 1 source file: **chap1.c**<br>**chap1.c** - contains following 9 sections<br>DECLARATIONS<br>DATA STRUCTURE OPS<br>NAME MANAGEMENT<br>INPUT<br>NEW PARSING ROUTINES<br>ENVIRONMENTS<br>NUMBERS<br>EVALUATION<br>MAIN - implements REPL | 4 source files: **ch1_info.h, ch1_info.c, chap1.y, chap1.l**<br>**ch1_info.h** - includes C headers and declares enums, structs, extern variables & function prototypes that are shared between the other 3 source files below. It is included in each of them.<br>**ch1_info.c** - contains following 6 sections<br>DECLARATIONS<br>DATA STRUCTURE OPS<br>NAME MANAGEMENT<br>ENVIRONMENTS<br>NUMBERS<br>EVALUATION<br>**chap1.y** - yacc parser (yyparse) grammar rules & actions<br>- contains MAIN - calls yyparse which implements the REPL<br>**chap1.l** - lexer (yylex) rules & actions |

| | | |
|---|---|---|
| **main function comparison** | //------------------------------------------------------------<br>// MAIN    READ-EVAL-PRINT LOOP (REPL)<br>//------------------------------------------------------------<br>int main (int argc, char **argv)<br>{<br>   short int error_no=0;<br><br>   initNames();<br>   initCmds();<br>   initParse();<br><br>   globalEnv = emptyEnv();<br>   quittingtime = false;<br><br>**// Set the jump location for longjmp.**<br>**// After an error, errmsg() displays the message then**<br>**// jumps here to continue with REPL below.**<br>   if (setjmp(JL99))<br>      ;<br><br>**// read, eval, print until "quit$" is entered**<br>   while (!quittingtime)<br>   {<br>      reader();<br>      getToken();<br><br>      if (toksy == quitsy)<br>         quittingtime = true;<br>      else if (toksy == funsy)<br>      {<br>         **//parse fundef & print its name**<br>         prName(parseDef());<br>         printf("\n\n");<br>      }<br>      else<br>      {<br>         **//parse & eval expr & print its value**<br>         currentExp = parseExpr();<br>         prValue(eval(currentExp, emptyEnv()));<br>         printf("\n\n");<br>      }<br>   } // while<br><br>   exit (EXIT_SUCCESS);<br>} | //-----------<br>// MAIN<br>//-----------<br>int main()<br>{<br><br>   initNames();<br><br>   globalEnv = emptyEnv();<br><br>**// Indicate to the lexer whether the program is run in**<br>**// interactive or batch mode. When lexer sees EOF,**<br>**// interactive mode will prompt for next input**<br>**// but batch mode will exit program.**<br>   if (isatty(fileno(stdin))) {<br>      printf("Input from terminal (interactive mode)\n");<br>      interactive = true;<br>   } else {<br>      printf("Input from pipe/file (batch mode)\n");<br>      interactive = false;<br>   }<br><br>   printf(PROMPT); **// display initial prompt**<br>   fflush(stdout);<br>   yyparse();  **// read, eval, print until "quit" is entered**<br>} |

Error Handling Changes

When a syntax or semantic error occurs in the C version, errmsg() is called to display error info and then to call longjmp() to return to a label in main() which restarts the REPL for the next input.

In the lex/yacc version, errmsg() is no longer called to report syntax errors. Instead, the parser spec (*chap1.y*) calls yyerror() to report syntax errors then transfers control to the following error recovery rule.

```
          |    error '$' {
                           yyclearin; // discard lookahead
                           yyerrok;   // clear the error state
                           setError(false, "stmt_list error"); //reset error flag false
                           if (readfile)
                               printf("\n"); //blank line after an error for readability in output
                           else
                               printf(PROMPT); //prompt for next input if not reading a file
                        }
          ;
```

This rule contains the parser's builtin error token, followed by a $ (end of input marker) which is the synchronizing token, followed by C action code in braces. When the parser encounters a syntax error, it calls yyerror() to display error info then transfers control to the above error rule. While processing this rule, the parser discards any parse info for the erroneous statement and skips any remaining chars related to it in the input stream by reading until it sees the next $. Then it performs the C action code and resumes normal reading/parsing of the next input.

This process of calling yyerror() and resynchronizing based on the error rule results in a different error message format than that used in the C version. Below are a few examples of the old vs. new error format taken from the UT results.

**Table 2 Old vs. new error message format**

| **C version** | **lex/yacc version** |
|---|---|
| Syntax error example<br>!<> cannot be a function name since < and > are delimiters. | Syntax error example<br>!<> cannot be a function name since < and > are delimiters. |
| fun <> (x,y):= not(x=y) nuf$<br>*****Error parsing function name.  Found < where funid or nameid is expected. | fun <<br>**yyerror:  Line 24: syntax error, unexpected '<', expecting NAME at '<'**<br>> (x, y):= not(x=y) nuf$  **//resynchronizing** |
| **The C version reads and echoes input chars until the $ is read. Then it begins parsing and detects the invalid < character where a name is expected. The errmsg() function displays the error info then jumps to the top of the REPL in main() function in order to read the next input.** | **The lexer reads and echoes input chars until a token is identified then returns it to the parser. After seeing the FUN token, a NAME token is expected by the parser. When < is returned instead, the parser calls yyerror() which displays the error message then passes control to the error rule. Then the parser resynchronizes by ignoring the remaining tokens in the erroneous input until the next $ is read. After reading the $, it is ready to resume normal reading and parsing the next input.**<br>The errmsg() function is not called in the above case or for handling syntax errors in general. |

## Part 2. Grammar, Lex & Yacc Specs

The productions for the grammar in the C and lex/yacc versions are as follows.

userinput → input $
input → quit | cmdline | fundef | expr
cmdline → ) command
    note: the right parenthesis must be first char of input line
command → load filename | sload filename
filename → any valid filename for the operating system

fundef → fun name ( paramlist ) := expr nuf
paramlist → null | name [ , name ]*

expr → ifex | whileex | seqex | exp1
ifex → if expr then expr else expr fi
whileex → while expr do expr od
seqex → seq explist qes
explist → expr [ ; expr ]*
exp1 → exp2 [ := exp1 ]*
exp2 → [ prtop ] exp3
exp3 → exp4 [ relop exp4 ]*
exp4 → exp5 [ addop exp5 ]*
exp5 → [ addop ] exp6 [ mulop exp6 ]*
exp6 → name | number | funcall | ( expr )

funcall → name ( arglist )
arglist → null | expr [ , expr ]*
prtop → print
relop → = | < | >
addop → + | -
mulop → * | /

number → sequence of digits
name → any sequence of characters not a number and not containing a delimiter

delimiter → '(', ')', ';', '+', '-', '*', '/', ':', '=', '<', '>', ',', '$', '!' or space
[ and ] enclose optional parts of a production.
[ ] indicates the option may occur once
[ ]* indicates the option may occur more than once
The exclamation point begins a comment.

Table 3 below shows how the above productions are expressed as rules in the yacc parser spec (*chap1.y*).
It omits the C action code in braces in order to focus on the rule structure.
The productions in purple text do not have corresponding yacc rules because they are handled by rules/actions in the lexer (*chap1.l*).

**Table 3 Grammar productions expressed as Yacc rules**

| **Grammar Productions** | **Corresponding Yacc Parser Rule in *chap1.y***<br>- nonterminals are in lowercase<br>- terminals (tokens) are in uppercase or quotes |
|---|---|
| input → **quit** \| **cmdline** \| expr \| fundef<br>**cmdline → ) command**<br>**note: the right parenthesis must be first char of input line**<br>**command → load filename \| sload filename**<br>**filename → any valid filename for the operating system** | `stmt_list:    /* empty */`<br>`         \|    stmt_list expr '$'`<br>`         \|    stmt_list fundef '$'` |
| fundef → fun name ( paramlist ) := expr nuf<br>paramlist → null \| name [ , name ]* | `fundef: FUN NAME '(' param_list ')' ASSIGN expr NUF`<br><br>`param_list: /* empty */`<br>`          \| NAME param_list_tail` |

expr → ifex | whileex | seqex | exp1

ifex → if expr then expr else expr fi

whileex → while expr do  expr od

seqex → seq explist qes

explist → expr [ ; expr ]*

In the C version, the recursive descent style of the following expression rules enforced operator precedence so that priority increased from exp1 (assignment lowest) through exp6  (funcall and parentheses highest). Assignment is the only right associative operator which means that in a series of adjacent assignments (e.g x:=y:=z), the rightmost assignment must occur first. This was accomplished by having the optional part of the exp1 production call itself.

exp1 → exp2 [ :=  exp1 ]*
exp2 → [ prtop ] exp3
exp3 → exp4 [ relop exp4 ]*
exp4 → exp5 [ addop exp5 ]*
exp5 → [ addop ] exp6 [ mulop exp6 ]*
exp6 → name | number | funcall | ( expr )

prtop → print
relop → = | < | >
addop → + | -
mulop → * | /
funcall → name ( arglist )

arglist → null | expr [ , expr ]*

**number → sequence of digits**
**name → any sequence of chars not a number and not containing a delimiter**

```
param_list_tail: /* empty */
               | ',' NAME param_list_tail


expr:    ifex | whileex | seqex |  exp

ifex:   IF expr THEN expr ELSE expr FI

whileex: WHILE expr DO expr OD

seqex:  SEQ expr_list QES

expr_list: expr
         | expr ';' expr_list
```

In yacc, the appropriate precedence and associativity is declared for each operator as we'll see in the full yacc spec below. This allows all rules for binary & unary operations to be listed as as alternatives in a single exp rule as follows.

```
exp:    exp '+' exp
   |    exp '-' exp
   |    exp '*' exp
   |    exp '/' exp
   |    exp '<' exp
   |    exp '=' exp
   |    exp '>' exp
   |    PRINT exp
   |    '-' exp %prec UMINUS
   |    '+' exp %prec UPLUS
   |    NUMBER
   |    NAME ASSIGN exp
   |    NAME '(' arg_list ')'
   |    NAME
   |    '(' exp ')'

arg_list: /* empty */
        | expr arg_list_tail
arg_list_tail: /* empty */
             | ',' expr arg_list_tail
```

**yacc spec *chap1.y***
Below is the full yacc spec (*chap1.y*) with comments highlighted in red text.
Abbreviations used: LHS = lefthand side, RHS = righthand side, AST = Abstract Syntax Tree

```
~/lexyacc/chap1$ cat chap1.y
%{
//Generate a parser to perform the actions associated with the grammar rules described below.
#include "ch1_info.h"
EXPLIST nilEL=0;
NAMELIST nilNL=0;
NUMTYP n;
BOOLEAN interactive;

int yylex();
void yyerror(const char *s);
%}

//Declare the types used for symbols in yacc rules below.
//They are specified in angle brackets in %token and %type directives that follow.
//The values and data structures that these types represent are used to build an AST
```

```
//that represents the fundef or expr that was input.
%union
{
    NAMEINDEX nameIndex;    //an index in the printNames array
    NUMTYP num;             //a NUMBER token (see %token below)
    EXP e;                  //a VALEXP, VAREXP or APEXP structure
    EXPLIST eL;             //a structure for a list of EXPs
    FUNDEF fd;              //a function definition structure
    NAMELIST nl;            //a structure for a list of nameIndexes
}

%define lr.type ielr       //Bison manual indicates that type=ielr is best when error=detailed.
%define parse.lac full     //Bison manual says lac=full avoids inaccurate info when error=detailed.
%define parse.error detailed   //Allows yyerror messages to report the token it expected.

//Define tokens that are made of multiple characters.
//The patterns that they represent are defined in the lexer spec. See chap1.l below.
//The ERROR token is an exception. It has no pattern defined in the lexer.
//And since it is not a part of any grammar rule defined below, returning it from the lexer causes
//a syntax error which triggers a call to yyerror() and a transfer of control to the error rule
//defined below. When lexer action code detects certain semantic errors (err_range, err_name_len,
//err_max_names), it returns this ERROR token to trigger yyerror() and the error rule.
//For these 3 semantic errors, you will see both a message from errmsg() and yyerror().
%token <nameIndex> NAME
%token <num>  NUMBER
%token ASSIGN IF THEN ELSE FI WHILE DO OD SEQ QES FUN NUF ERROR

//Declare Associativity and Precedence. The order of the following lines determine precedence
//from lowest to highest. So ';' and ',' have lowest precedence.
//'*' and '/' have highest precedence among the binary operators.
//UPLUS & UMINUS tags are used to give unary '+' and '-' the highest precedence of all.
%left ';' ','
%right ASSIGN
%left PRINT
%left '<' '=' '>'
%left '-' '+'
%left '*' '/'
%nonassoc UPLUS
%nonassoc UMINUS
//Note that the precedence of parentheses was not declared.
//Per perplexity.ai, the parentheses rule, under exp rules below, effectively gets the
//highest precedence "by its nature" since it causes the parser to reduce the enclosed expression
//before applying other operations outside the parentheses.

//Declare types of all LHS symbols in yacc rules.
%type <e> expr exp ifex whileex seqex;
%type <eL> expr_list arg_list arg_list_tail;
%type <nameIndex> fundef;
%type <nl> param_list param_list_tail;

//Below are the yacc rules cited in the above table along with their actions (C code in braces).
//The actions are executed by the parser when the input matches a rule.
//Regarding rule symbols, nonterminals are in lowercase, terminals are in uppercase or single quotes.
//The actions build an AST for the expr or fundef that was input.
//The start rule evaluates that AST by passing it to eval() then displays its result.
//The actions refer to symbols $$, $1, $2, $3, etc.
//$$ represents the symbol on the LHS of the colon.
//$1, $2, $3, ... respectively represent the symbols on the RHS of the colon.

%%
//input → expr | fundef
//This is the start rule for our parser.
//It expects an input line to be either an expr or fundef terminated by a $.
//$ is the end of input marker.
//The recursive use of stmt_list on the RHS allows multiple input lines to be entered.
stmt_list:    /* empty */
         |    stmt_list expr '$'   { //if no error then evaluate the expr that was input
                                     // and display its value
                                      if (!error)
                                          n = eval($2, emptyEnv());  //eval $2 (expr AST)

                                      if (!error)
                                      {
                                          prValue(n);       //display expr value
                                          printf("\n\n");
                                      }
                                      setError(false, "stmt_list stmt"); //reset global error flag false
                                      if (!readfile) printf(PROMPT); //display prompt if not reading file
                                   }
//The result of a fundef is to display the function name.
//It's logic is not evaluated until a function call occurs in an expr.
         |    stmt_list fundef '$' { //if no error then display function name of fundef
                                      if (!error)
                                      {
                                          prName($2);      //display function name
                                          printf("\n\n");
                                      }
                                      setError(false, "stmt_list stmt");
                                      if (!readfile) printf(PROMPT);
                                   }

//When the parser encounters a syntax error, yyerror is called and control returns to the
//action for this special error token. The $ (end of input marker) is the synchronizing token.
//The parser will skip chars in the input stream until it reads the next $ then it will resume
//nomal parsing of the next input.
         |    error '$' {
                           yyclearin; // discard lookahead
                           yyerrok;   // clear the error state
                           setError(false, "stmt_list error"); //reset error flag false
                           if (readfile)
                               printf("\n"); //blank line after an error for readability in output
                           else
                               printf(PROMPT); //prompt for next input if not reading a file
                        }
         ;

//fundef → fun name ( param_list ) := expr nuf
//When a function def is evaluated, the result is to displays the name of the defined function.
//So the line $$ = $2 in the action assigns value of NAME to the fundef symbol on the LHS.
fundef: FUN NAME '(' param_list ')' ASSIGN expr NUF {
                                                      newDef($2, $4, $7); //create a new function def
                                                                          //from $2 (NAME),
                                                                          //$4 (param_list),
                                                                          //$7 (expr) which represents
                                                                          //the body of the function.
                                                      $$ = $2;  //fundef = NAME
                                                    }
      ;


//The param_list rule defines a list of NAMES.
//It can be empty, a single NAME or a list of NAMEs separated by commas.
//It disallows a list that ends with a comma due to how param_list_tail is structured.
//When param_list_tail is empty, the list ends.
param_list: /* empty */ { $$ = nilNL; }
          | NAME param_list_tail { $$ = mkNamelist($1, $2); }
          ;

param_list_tail: /* empty */ { $$ = nilNL; }
               | ',' NAME param_list_tail { $$ = mkNamelist($2, $3); }
               ;

//The arg_list rule defines a list expressions.
//It can be empty, a single expression or a list of expressions separated by commas.
//It disallows a list that ends with a comma due to how arg_list_tail is structured.
//When arg_list_tail is empty, the list ends.
arg_list: /* empty */ { $$ = nilEL; }
        | expr arg_list_tail { $$ = mkExplist($1, $2); }
        ;

arg_list_tail: /* empty */ { $$ = nilEL; }
             | ',' expr arg_list_tail { $$ = mkExplist($2, $3); }
             ;


//ifex → if e1 then e2 else e3 fi
```

```
ifex:   IF expr THEN expr ELSE expr FI
        { $$ = mkAPEXP(ifsy_index, mkExplist($2, mkExplist($4, mkExplist($6,nilEL)))); }
    ;

//whileex → while e1 do e2 od
whileex: WHILE expr DO expr OD
         { $$ = mkAPEXP(whilesy_index, mkExplist($2, mkExplist($4,nilEL))); }
       ;

//seqex → seq expr_list qes
seqex:  SEQ expr_list QES { $$ = mkAPEXP(seqsy_index, $2); }
     ;

//expr_list → expr [ ; expr ]*
expr_list: expr  { $$ = mkExplist($1, nilEL); }
         | expr ';' expr_list { $$ = mkExplist($1, $3); }
         ;

//expr → ifex | whileex | seqex | exp
expr:    ifex              { $$ = $1; }
    |    whileex           { $$ = $1; }
    |    seqex             { $$ = $1; }
    |    exp               { $$ = $1; }
    ;
//For a binary op, we make an APEXP that applies it to the left ($1) and right ($3) operands.
exp:    exp '+' exp { $$ = mkAPEXP(addsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '-' exp { $$ = mkAPEXP(subsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '*' exp { $$ = mkAPEXP(mulsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '/' exp { $$ = mkAPEXP(divsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '<' exp { $$ = mkAPEXP( lssy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '=' exp { $$ = mkAPEXP( eqsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    exp '>' exp { $$ = mkAPEXP( gtsy_index, mkExplist($1, mkExplist($3,nilEL))); }
   |    PRINT exp   { $$ = mkAPEXP(printsy_index, mkExplist($2, nilEL)); }
//For unary minus, we make an APEXP that multiplies the exp by -1
   |    '-' exp %prec UMINUS { $$ = mkAPEXP(mulsy_index, mkExplist(mkVALEXP(-1), mkExplist($2,nilEL))); }
   |    '+' exp %prec UPLUS  { $$ = $2; }
   |    NUMBER        { $$ = mkVALEXP($1); }
//exp1 → exp2 [ :=  exp1 ]*
//When the assignment rule is matched, the action makes an APEXP for the assignment operator.
//When evaluated, the APEXP assigns the value of the exp ($3) to a VAREXP for the NAME ($1).
   |    NAME ASSIGN exp   { $$ = mkAPEXP(assignsy_index, mkExplist(mkVAREXP($1), mkExplist($3,nilEL))); }
//funcall → name ( arglist )
//make an APEXP that applies the function name ($1) to an Explist ($3) containing the arguments.
   |    NAME '(' arg_list ')' { $$ = mkAPEXP($1, $3); }
//variable name - make a VAREXP for the NAME. When evaluated, it returns value assigned to the variable.
   |    NAME          { $$ = mkVAREXP($1); }
//parentheses - assign the value of the parenthesized exp on RHS to the LHS.
//This has the effect of evaluating the operations inside parentheses before those outside.
   |    '(' exp ')'   { $$ = $2; }
  ;
%%
int main()
{
    initNames();
    globalEnv = emptyEnv();

//Indicate to the lexer whether the program is run in interactive or batch mode.
//When lexer sees EOF, interactive mode will prompt for next input but batch mode will exit program.
    if (isatty(fileno(stdin))) {
        printf("Input from terminal (interactive mode)\n");
        interactive = true;
    } else {
        printf("Input from pipe/file (batch mode)\n");
        interactive = false;
    }

    printf(PROMPT);
    fflush(stdout);
    yyparse();  //perform the REPL until quit is input or EOF in batch mode
}
```

**lex spec *chap1.l***

Below is the full lex spec with comments highlighted in red text.

```
~/lexyacc/chap1$ cat chap1.l
%{
/*
  Generate a lexer to read the input stream and perform the associated
  actions for the patterns described below.
  Some actions only return a token to the parser.
  Others run C code before returning, e.g. to install a name in the
  symbol table before returning the NAME token.
  Others don't return to the parser. E.g. for \n, lineNo is incremented and
  a prompt displayed before the lexer reads the next input.
  Or for a load command, a file is opened and the lexer begins reading from it.
*/

#include "ch1_info.h"
#include "chap1.tab.h"

/*
  if not reading stdin and echo is true then echo all chars read.
  echo is set true for a load file command and reset false at EOF.
*/
#define YY_USER_ACTION \
        if (yyin != stdin && echo) { \
            printf("%s", yytext); \
        }

char *filename, *endptr;
FILE *infile;
int lineNo = 0;
BOOLEAN echo = false;
char cwd[PATH_MAX];
extern BOOLEAN interactive;

/*
  When lex detects an error, it calls yyerror() which returns control to
  the error rule in the parser. A few actions below return an ERROR token which
  is not a valid token in the grammar. This causes the parser to call yyerror()
  and give control to the error rule. My own version of yyerror() is defined
  at bottom of this file. It adds display of lineNo and
*/
void yyerror(const char *);
%}

/* prevent call to yywrap() to read another file at EOF */
%option noyywrap

/* exclusive state for matching a file name */
%x fname

%%
"quit"             { printf("\nquitting\n"); return 0; }

^")user"           { prUserList(); }

^")load "          { BEGIN(fname); echo = true; }
^")sload "         { BEGIN(fname); echo = false; }

 /*
  Match a filename as part of a load or sload command.
  Open the file and set up the lexer to begin reading from it.
 */
<fname>[^\t\n]+    {    // match filename - any chars except tab & newline
                       BEGIN(INITIAL);  // reset INITIAL, unset fname state
                       filename = strdup(yytext);

                       if (!readfile) // if not already reading a file
                       {
                           // open file and prepare to read from it
                           infile = fopen(filename,"r");
                           if (!infile)
```

```
                               errmsg(err_open, filename, null_int);
                           else
                           {
                               // create buffer for it and switch to it
                               yyin = infile;
                               yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));
                               readfile = true;
                               lineNo = 0;
                               if (getcwd(cwd, sizeof(cwd)) != NULL)
                                   printf("\nCurrent Directory is: %s\n\n", cwd);
                               else
                                   errmsg(err_cwd, null_str, null_int);
                               printf(" Loading file : %s\n\n", filename);
                           }
                       }
                       else //file already open - nested load cmds not allowed
                       {
                           errmsg(err_nested_load, filename, null_int);
                           fclose(yyin);
                           return 0; // quit program
                       }

                   }

"if"    { return IF;    }
"then"  { return THEN;  }
"else"  { return ELSE;  }
"fi"    { return FI;    }
"while" { return WHILE; }
"do"    { return DO;    }
"od"    { return OD;    }
"seq"   { return SEQ;   }
"qes"   { return QES;   }
"fun"   { return FUN;   }
"nuf"   { return NUF;   }
"print" { return PRINT; }

!.*     ;  // ignore comments

"$".*\n   { // match $ (the end of input marker) and ignore any chars that follow it.
            lineNo++;
            return '$';
          }

\n      { //at end of line, increment lineNo, display prompt if not reading file
          lineNo++;
          if (!readfile)
          {
              if (error)
              {
                  printf(PROMPT);  // restart input at initial prompt after error
                  setError(false, "lex action for newline");
              }
              else
                  printf(PROMPT2); // continuation prompt
          }
        }

[ \t]   ;  // ignore white space

 /*
  Match a number as a sequence of digits followed by a delimiter.
  Digits followed by a nondelimiter will be matched as a name in next pattern below.
  The trailing context lists the delimiters.
  After adding tab, newline, carriage return to the list, I decided to include the
  full range of ASCII control chars plus the delete char.
 */
[0-9]+/[();+\-*/:=<>,$! \t\n\r\x00-\x1F\x7F] {
                                               errno = 0;
                                               yylval.num = strtoll(yytext, &endptr, 10);
                                               if ((yylval.num == LONG_MAX || yylval.num == LONG_MIN)
                                                    && errno == ERANGE)
                                               {
                                                   errmsg(err_range, yytext, null_int);
                                                   return ERROR; //trigger yyerror for err_range
                                               }
                                               else
                                                   return NUMBER;
                                             }

 /*
  Match a name as any sequence of chars that is not a number (matched above) and
  does not contain a delimiter. This includes a sequence of digits that contain nondelimiters.
 */
[^();+\-*/:=<>,$! \t\n\r\x00-\x1F\x7F]+  {
                                           if (yyleng > NAMELENG)
                                           {
                                               errmsg(err_name_len, yytext, NAMELENG);
                                               return ERROR; //trigger yyerror for err_name_len
                                           }
                                           else
                                           {
                                               if (yylval.nameIndex = install(yytext)) // install, obtain
index

                                                   return NAME;
                                               else
                                                   return ERROR; //trigger yyerror for err_max_names
                                           }
                                         }

":="    { return ASSIGN; }

.       { return yytext[0]; }

 /*
   Handle EOF as follows:
    - An input of ctrl-D from stdin triggers EOF and exits.
    - EOF from a file:
        close file, display number of lines processed
        if interactive mode (e.g. user issued a load command)
            prompt for next input
        else batch mode (e.g. input file was redirected to the program)
            return 0 to exit
 */
<<EOF>> {
          if (yyin == stdin)
          {
              clearerr(yyin); // prevent infinite loop when ctrl-D is input
              return 0;
          }
          else // delete buffer, close file, restart stdin
          {
              readfile = false;
              echo = false;
              yy_delete_buffer(YY_CURRENT_BUFFER); // reclaim buffer space
              fclose(yyin);
              printf("   %d lines processed from %s\n", lineNo, filename);

              if (interactive)
              {
                  yyrestart(stdin);
                  printf(PROMPT);
              }
              else
                  return 0; // batch mode only processes the redirected file then quits
          }
        }
%%

void yyerror(const char *msg)
{
    printf("\nyyerror:  Line %d: %s at '%s'\n", lineNo, msg, yytext);
    setError(true, "yyerror");   // flag the error
}
```

Part 3. Unit Test (UT) Result Comparison

Table 4 below provides a side by side of the old vs new UT result file (*chap1_ut.rs0*).

The UT cases for the enhanced C version were reused to test the lex/yacc version.
Most of the differences are due to the new error message format that I described above in Table 2.
A few test cases have new behavior and I document the reason for this.

The UT script (*chap1_ut.sh*) runs the interpreter four times to process the following four input files.

chap1_ut.input1 - tests cases for many valid and invalid instructions
chap1_ut.input2 - error cases that exceed maximum sizes
chap1_ut.input3 - error case for a load command inside a file being loaded - aborts program
chap1_ut.input4 - error case for a sload (silent load - noecho) command inside a file being loaded - aborts program

A load command for each file is redirected into the interpreter as a Here String like the following.

```
./chap1 <<< ")load chap1_ut.input1"
```

Table 4 below has four sections, each beginning with the Here String for its corresponding input file.
Section 1 for *chap1_ut.input1* is the longest.

Results from C version are in column 1.
Results from lex/yacc version are in column 2.
For easier comparison, only cases with differences are listed from the C version in Column 1.
So white space in Column 1 means that the results were identical to those to the right in Column 2.

My comments are in bold red text and are prefixed by my initials and a colon. Search for “DD:” to read them.

**Table 4 UT Results Comparison**

**C version -** difference sections only

```
~/c/proglang/chap1$ cat chap1_ut.rs0
./chap1 <<< ")load chap1_ut.input1"

->
Current Directory is: /home/dawsond/c/proglang/chap1
 Loading file : chap1_ut.input1

!Redo tests in section 1.1.3 of Kamin's text using Pascal syntax
```

**lex/yacc version**

```
~/lexyacc/chap1$ cat chap1_ut.rs0
./chap1 <<< ")load chap1_ut.input1"
Input from pipe/file (batch mode)
->
Current Directory is: /home/dawsond/lexyacc/chap1
 Loading file : chap1_ut.input1

!Redo tests in section 1.1.3 of Kamin's text using Pascal syntax

3$
3

4+7$
11

x:=4$
4

x+x$
8

print x$
4
4

y:=5$
5

seq print x; print y; x*y qes$
4
5
20

if y>0 then 5 else 10 fi$
5

while y>0 do
  seq x:=x+x; y:=y-1 qes
od$
0

x$
128

fun #1 (x) := x + 1 nuf$
#1

#1(4)$
5

fun double(x):=x+x nuf$
```

| | |
|---|---|
| | double<br><br>double(4)$<br>8<br><br>x$<br>128<br><br>fun setx(x,y):= seq x:=x+y; x qes nuf$<br>setx<br><br>setx(x,1)$<br>129<br><br>x$<br>128<br><br>fun not(boolval):= if boolval then 0 else 1 fi nuf$<br>not<br><br>!<> cannot be a function name since < and > are delimiters. |
| fun <> (x,y):= not(x=y) nuf$<br>*****Error parsing function name.  Found < where funid or nameid is expected. | fun <<br>yyerror: Line 24: syntax error, unexpected '<', expecting NAME at '<'<br>> (x, y):= not(x=y) nuf$  **//resynchronizing**<br>**DD:The above test case was described earlier in Table 2.** |
| | !# is not a delimiter and can be used in a name.<br>fun ## (x,y):= not(x=y) nuf$<br>##<br><br>fun mod(m,n):=m-n*(m/n)nuf$<br>mod<br><br>fun gcd(m,n):=<br> seq<br>  r:=mod(m,n);<br>  while ##(r,0) do<br>   seq<br>    m:=n;<br>    n:=r;<br>    r:=mod(m,n)<br>   qes<br>  od;<br>  n<br> qes<br>nuf$<br>gcd<br><br>gcd(6,15)$<br>3<br><br><br>fun gcd(m,n):=<br>  if n=0 then m else gcd(n,mod(m,n)) fi nuf$<br>gcd |

```
!Unary plus and minus
```

```
gcd(6,15)$
3


!Normal precedence and associativity are implemented.
5*3+7$
22

5+3*7$
26

14-7-3$
4

48/12/2$
2

10/0$
***** applyValueOp: divide by zero


!relational operators
5<10$
1

5>10$
0

5=5$
1

10<5$
0

10<5>-1$
1

10<5>-1=1$
1

5*3>2+2$
1

5*(3>2)+2$
7

!Unary plus and minus
10--5$
15

10++5$
15

10+-5$
5
```

**DD: The C version cannot handle 2 or more unary plus/minus signs in a row. But the lex/yac version handles them correctly.**

```
10---5$
*****Error parsing exp6.  Found - where nameid, funid, "(", or a
number is expected.

10+++5$
*****Error parsing exp6.  Found + where nameid, funid, "(", or a
number is expected.

10-+--+5$
*****Error parsing exp6.  Found - where nameid, funid, "(", or a
number is expected.
```

```
!min negative number

min:=-9223372036854775808$
-9223372036854775808
```

```
!Max digits is 19. Following value has 20 digits.
d:=99999999999999999999$
*****parseVal: Max digits allowed in 64 bit signed long is 19
```

**DD:The C version will allow 19 digit values that are out of range to be entered but they will wrap around.**

```
10-+5$
5

10---5$
5

10+++5$
15

10-+--+5$
5
```

```
!Valid number range
!max positive number
max:=+9223372036854775807$
9223372036854775807

!The min negative number below cannot be entered directly because
!numbers are first interpreted as positive then negated.
min:=-9223372036854775808
***** 9223372036854775808 is out of range.
It must be between -9223372036854775808 and 9223372036854775807,
inclusive.

yyerror:  Line 78: syntax error, unexpected ERROR at
'9223372036854775808'
$     //resynchronizing
```

**DD: In my comments about the ERROR token (see chap1.y above), I explained why both errmsg() and yyerror() are called for the err_range error id. The above error is an example of this. Purple text is from errmsg() and bold black text from yyerror().**

```
!Negate max positive number then subtract 1 to get min negative
number.
min:=-9223372036854775807-1$
-9223372036854775808

max+min$
-1

!Numbers that are input directly must be in range LLONG_MIN to LLONG_MAX.
d:=99999999999999999999
***** 99999999999999999999 is out of range.
It must be between -9223372036854775808 and 9223372036854775807,
inclusive.

yyerror:  Line 83: syntax error, unexpected ERROR at
'99999999999999999999'
$     //resynchronizing
```

```
!Keywords cannot be redefined
fun if (x) := x+5 nuf$
*****Error parsing function name.  Found if where funid or nameid is
expected.


if := 20$
*****Error parsing expr.  Found := where one of the following is
expected:
"if", "while", "seq", "print", nameid, funid, number, or "("


!A string of digits is not a valid name.
fun 222  (x) := x+222 nuf$
*****Error parsing function name.  Found 222 where funid or nameid is
expected.


!Inserting a delimiter in a name causes erroneous results.
a(b:=25$
*****Undefined variable: a
```

```
!Numbers that go outside of that range in calculations will wrap
!around.
d:=max+1$
-9223372036854775808

d:=min-1$
9223372036854775807


!Keywords cannot be redefined
fun if
yyerror: Line 83: syntax error, unexpected IF, expecting NAME at 'if'
 (x) := x+5 nuf$     //resynchronizing


if :=
yyerror:  Line 84: syntax error, unexpected ASSIGN at ':='
 20$     //resynchronizing


!Names may contain any char that is not a delimiter and must
!not contain only digits.
!Delimiters = ' ','(',')','+','-','*','/',':','=','<','>',';',',','$','!'
~12#ab:=25$
25

~12#ab$
25

x:=15-~12#ab+7$
-3


!A string of digits is not a valid name.
fun 222
yyerror: Line 94: syntax error, unexpected NUMBER, expecting NAME
at '222'
  (x) := x+222 nuf$     //resynchronizing


!Inserting a non-delimiter into a string of digits makes a valid name
fun 222# (x) := x+222 nuf$
222#

222#(3)$
225

x:=100-222#(3)-50$
-175


!Inserting a delimiter in a name causes erroneous results.
a(b:=25$
yyerror:  Line 103: syntax error, unexpected '$' at '$
```

```
!Function name may not be reused as a variable name.
DD:This limitation is removed in the lex/yacc version
fun inc10 (x) := x+10 nuf$
inc10

inc10:=25$
*****Error in match. Found :=  where ( is expected.



!ERROR MESSAGE TESTS



fun david(x,+,z):= x+1 nuf$
*****Error parsing arglist.  Found + where ")" or nameid is expected.


fun +++(x):= x+1 nuf$
*****Error parsing function name.  Found + where funid or nameid is
expected.
```

```
'


!A name may be used for a variable and a function at the same time.
DD: Define function first then the variable
fun inc10 (x) := x+10 nuf$
inc10

inc10:=25$
25

inc10$
25

inc10(88)$
98

DD:Define variable first then the function
sum:=25$
25

fun sum(x,y):= x+y nuf$
sum

sum$
25

sum(33,44)$
77


!Multiple assignment
i:=j:=k:=25$
25

i$
25

j$
25

k$
25


!ERROR MESSAGE TESTS

!Syntax Errors

fun david(x,+
yyerror:  Line 125: syntax error, unexpected '+', expecting NAME at '+'
,z):= x+1 nuf$     //resynchronizing

fun +
yyerror:  Line 126: syntax error, unexpected '+', expecting NAME at '+'
++(x):= x+1 nuf$     //resynchronizing
```

| | |
|---|---|
| abc:=)25$<br>*****Error parsing exp6.  Found ) where nameid, funid, "(", or a number is expected. | abc:=)<br>yyerror:  Line 127: syntax error, unexpected ')' at ')'<br>25$     **//resynchronizing** |
| !Max NAMELENG=20 but the var and function names below have 21 chars.<br>print abcdefghijklmnopqrstu$<br>*****Name exceeds 20 chars, begins: abcdefghijklmnopqrst | !Max NAMELENG=20 but the var and function names below have 21 chars.<br>abcdefghijklmnopqrstu<br>***** **Name exceeds 20 chars, begins: abcdefghijklmnopqrstu**<br><br>**yyerror:  Line 130: syntax error, unexpected ERROR at 'abcdefghijklmnopqrstu'**<br>+77-10/0+33-abc*8$     **//resynchronizing**<br><br>**DD: In my comments about the ERROR token (see chap1.y above), I explained why both errmsg() and yyerror() are called for the err_name_len error id. The above error is an example of this. Purple text is from errmsg() and bold black text from yyerror().** |
| fun abcdefghijklmnopqrstu(x):=x+10 nuf$<br>*****Name exceeds 20 chars, begins: abcdefghijklmnopqrst | fun abcdefghijklmnopqrstu<br>***** **Name exceeds 20 chars, begins: abcdefghijklmnopqrstu**<br><br>**yyerror:  Line 131: syntax error, unexpected ERROR, expecting NAME at 'abcdefghijklmnopqrstu'**<br>(x):=x+10 nuf$     **//resynchronizing** |
| 22:=4$<br>*****parseExp1: left hand side of assignment must be a variable | !lhs of an assignment must be a name<br>22:=<br>yyerror:  Line 134: syntax error, unexpected ASSIGN at ':='<br>4$     **//resynchronizing** |
| | !Multiple syntax errors in a single input.<br>!First is reported, others skipped as parser resynchronizes<br>!with next $. Value of ijk is not changed.<br>ijk:=33$<br>33<br><br>ijk:=50-sum(10,*<br>**yyerror:  Line 140: syntax error, unexpected '*' at '*'**<br>)+25-4*3+sum(*,15)-48/4+)load bad_data.txt$     **//resynchronizing**<br><br>ijk$<br>33 |
| !Semantic Errors<br><br>**DD: The following error messages are basically the same as those in the lex/yacc version. The difference is that the values for c$, a$ and b$ are not set to zero as in the lex/yacc version.** | !Semantic Errors<br>!<br>!Since these are not syntax errors, the parser does not call<br>!yyerror(). Such errors are detected in the application logic and<br>!cause a value of zero to be assigned to the erroneous expression.<br>!So the values of c$, a$ and b$ below return 0 due to such errors. |
| !Wrong number of arguments to mod() | !Wrong number of arguments to mod() |

```
c:=17-10-mod(100)+25*3$
*****Wrong number of arguments to: mod

c$
*****Undefined variable: c


!Division by zero.
10/0$
*****applyValueOp: attempted to divide by zero


a:=17-10/0+25*3$
*****applyValueOp: attempted to divide by zero

a$
*****Undefined variable: a


!Undefined function
b:=33$
33

b:=25-10+square(3)-5$
*****Undefined variable: square
```

**DD: The C version is imprecise and identifies square as a variable name.**

```
b$
33
```

**DD: The C version's errmsg() function calls longjmp() at the end to return to the REPL in main. So below, the errmsg() function exits after the first error message is reported. Only one semantic error can be reporter per input.**

```
!Three semantic errors (1 undef var & 2 div by 0) in a single input.
!Only first is reported.
c:=xyz+25-10/0-3*5+10/0-8*3$

*****Undefined variable: xyz









c$
*****Undefined variable: c
```

```
c:=17-10-mod(100)+25*3$
***** Wrong number of arguments to: mod

c$
0


!Division by zero.
10/0$
***** applyValueOp: divide by zero


a:=17-10/0+25*3$
***** applyValueOp: divide by zero

a$
0


!Undefined function
b:=33$
33

b:=25-10+square(3)-5$
***** Undefined function: square
```

**DD: The lex/yacc version is more accurate and identifies square as a function name. This is because the parser's function call rule is matched and an APEXP is built that applies square() to argument 3. During eval(), the function def for square() is looked up and found to be undefined.**

```
b$
0
```

**DD: The lex/yacc version's errmsg() function does not call longjmp() at the end. So it does not exit the application logic after the first error message as done in the C verison. Instead, errmsg() is called to report each semantic error in the input.**

```
!Three semantic errors (1 undef var & 2 div by 0) in a single input
!
c:=xyz+25-10/0-3*5+10/0-8*3$

***** Undefined variable: xyz


***** applyValueOp: divide by zero


***** applyValueOp: divide by zero


c$
0


!Syntax & Semantic errors in same input
```

```
c:=xyz+25-10/0-3*5+10/0-8*3-sum(10,*)$
*****Undefined variable: xyz



quit$



./chap1 <<< ")load chap1_ut.input2"

->
Current Directory is: /home/dawsond/c/proglang/chap1
 Loading file : chap1_ut.input2


!Test Cases for exceeding MAXINPUT and MAXNAMES
!Following lines exceed MAXNAMES=28. Builtin names = 26.
!So third name below is 29th name and triggers the error.












a:=25$
25

b:=100$
100

c:=200$
*****No more room for names
```

```
!
!Note that a syntax error prevents an expression from being
!evaluated. This is demonstrated below by inserting a syntax error
!in the above expression with three semantic errors.
!Since undef var & div by 0 errors are detected during eval() and
!eval() is not called, they are not reported.
!Only the syntax error is reported.
!
c:=xyz+25-10/0-3*5+10/0-8*3-sum(10,*
yyerror:  Line 178: syntax error, unexpected '*' at '*'
)$     //resynchronizing


quit
quitting


./chap1 <<< ")load chap1_ut.input2"
Input from pipe/file (batch mode)
->
Current Directory is: /home/dawsond/lexyacc/chap1
 Loading file : chap1_ut.input2

!
!MAXNAMES Test Case
!
!Generates error due to exceeding MAXNAMES allowed in printNames array.
!
!To run this test, build chap1 with the command "make testsize".
!This will define the TESTSIZE macro and set MAXNAMES=28 in the
!conditional compilation logic in ch1_info.h.
!
!The program begins with 26 built-in names in printNames.
!The two assignments below will create the 27th and 28th names.
!
!The function definition tries to create the 29th name, not2, which
!triggers the "no more room for names" error.
!The two references to the name boolValue also trigger the same error
!The expression a+b then evaluates without error.
a:=25$
25

b:=100$
100

fun not2
***** No more room for names

yyerror:  Line 17: syntax error, unexpected ERROR, expecting NAME at
'not2'
(boolValue                    //resynchronizing
***** No more room for names
):= if boolValue              //resynchronizing
***** No more room for names
 then 0 else 1 fi nuf$      //resynchronizing
```

**The following test case for exceeding the size of the userinput array was removed from the lex/yacc version since the input buffer size is now defined and handled inside the generated lexer.**

```
!Following function definition exceeds MAXINPUT=50. This error causes
!the program to skip the rest of the file and exit. So the function
!definition for not2 below is skipped and no error message will
!appear for it in the result file.

fun not(boolVal):=
if boolVal then
print 0
els*****Input exceeds 50 chars. Last char read = s

Skipping rest of input and quitting.
```

```
./chap1 <<< ")load chap1_ut.input3"
```

```
->
Current Directory is: /home/dawsond/c/proglang/chap1
 Loading file : chap1_ut.input3
```

```
a+b$
125
```

**DD: Note that after the first "No more room for names" error above, we get two more such errors while the parser is resyncronizing. This is because whenever the lexer matches a NAME token, it tries to install it in the symbol table before returning it to the parser.**

**So the lexer matches the name not2, tries to install it and gets the first "No more room" error. The lexer then returns the ERROR token to the parser and this triggers a call to yyerror() which displays the purple text. Then control passes to the parser's error rule so it can resynchronize which means it will not act upon tokens returned by the lexer until it sees the next $ token.**

**The lexer returns the ( token, then matches the name boolval, tries to insall it and gets the second "No more room" error. Again the lexer returns the ERROR token. But this time it doesn't trigger yyerror since the parser is in resynch mode and is ignoring tokens until it sees a $.**

**The lexer continues by returning tokens ), ASSIGN, IF, then it matches the second name boolval, tries to install it and gets the third "No more room" error. Again the lexer returns the ERROR token then it returns each of the other tokens on the last resynchronizing line which ends with the $.**

**The $ token ends the resynchronization and the parser is ready to proceed with normal reading/parsing of the next input.**

```
quit
quitting
```

```
./chap1 <<< ")load chap1_ut.input3"
```

```
Input from pipe/file (batch mode)
->
Current Directory is: /home/dawsond/lexyacc/chap1
 Loading file : chap1_ut.input3

!
```

**DD:Same result as lex/yacc version**

**./chap1 <<< ")sload chap1_ut.input4"**

```
->
Current Directory is: /home/dawsond/c/proglang/chap1
 Loading file : chap1_ut.input4
```

**DD:Same result as lex/yacc version**

```
!
!Nested load Command Test
!
!Generate error due to detecting a load command inside a file
!that is being loaded.
!
!First the following not function is defined and called successfully.
!Then a load command is issued which fails because it cannot be
!issued from inside another file.
!An error message is printed and the program aborts.
!
fun not(boolval):=
  if boolval then
  0
  else
  1
  fi
nuf$
not

not(3)$
0

not(0)$
1


)load gcd_only.txt
***** Load commands cannot occur inside a file being loaded.
Remove the load command for file gcd_only.txt
```

**./chap1 <<< ")load chap1_ut.input4"**

```
Input from pipe/file (batch mode)
->
Current Directory is: /home/dawsond/lexyacc/chap1
 Loading file : chap1_ut.input4
!
!
!Nested sload Command Test
!
!Generate error due to detecting a sload (silent load - noecho)
!command inside a file that is being loaded.
!
!First the following not function is defined and called successfully.
!Then a sload command is issued which fails because it cannot be
!issued from inside another file.
!An error message is printed and the program aborts.
!Note that the filename in the sload command is not echoed in the
!result file due to the global variable echo being set to false
!after lex reads ")sload".
!
fun not(boolval):=
  if boolval then
  0
  else
  1
  fi
nuf$
not

not(3)$
0

not(0)$
1

)sload
***** Load commands cannot occur inside a file being loaded.
Remove the load command for file gcd_only.txt
```

**DD: Test an open file error**
**This test had to be performed from the terminal since issuing a load command from inside a file will trigger the above nested load errors for chap1_ut.input3 and chap1_ut.input4.**

```
./chap1 <<< ")load does_not_exist.txt"
Input from pipe/file (batch mode)
-> fopen() error: No such file or directory

***** filename= does_not_exist.txt
```