CSCI131

Introduction to Computer Systems

Task Group 5: Compilers

Complete the exercises, and try to get them marked, before attempting the assignment!

There are several exercise tasks in this task group as detailed below. At the end of this document there is a link to the assignment. You should try to complete all the exercise tasks before attempting the assignment task.

The exercises focus on the construction of compilers using scanner and parser generators.

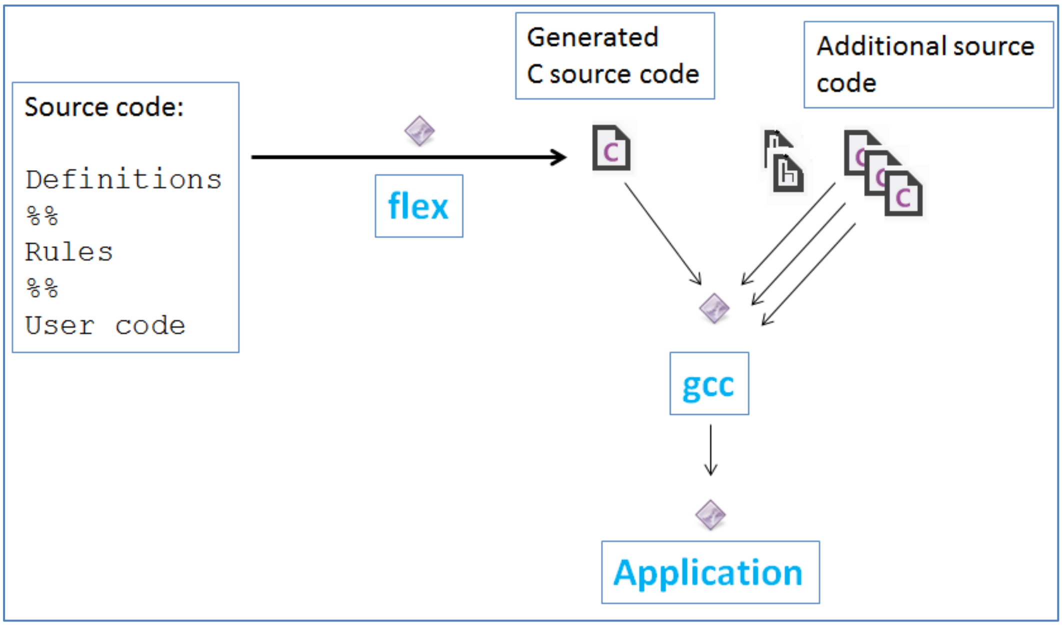
Task 1: Flex – generating lexical scanners: 3 marks

This task introduces flex – the GNU/Linux implementation of Unix's "lex" generator of lexical scanners.

The primary use for a lexical scanner generator is the creation of the scanner for a compiler. But sometimes, lexical scanners have uses in other kinds of application. This example involves the generation of a scanner that gets used in a program that performs a simple word count analysis of text documents.

Setting up a flex project – using a supplied prototype NetBeans project

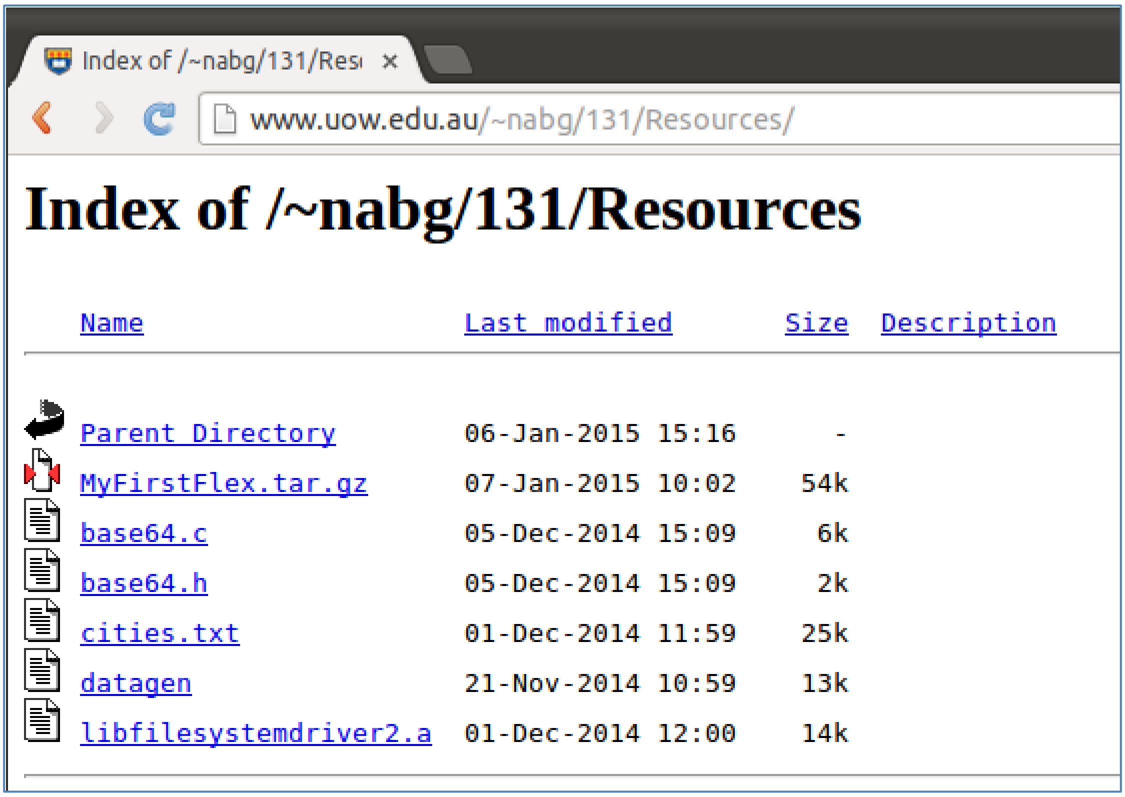
Projects that use a flex generated scanner require a more complex build process:



The scanner definition, written as input to flex, must be compiled by flex to generate the corresponding C code. This generated code, along with additional source code as required by the specific application, must then be compiled and linked using gcc to create the application.

Because the compilation chain is more elaborate, it's not handled as a standard NetBeans C application project. A special script has to be defined for NetBeans. It's easiest just to start with an existing prototype flex NetBeans project and duplicate this.

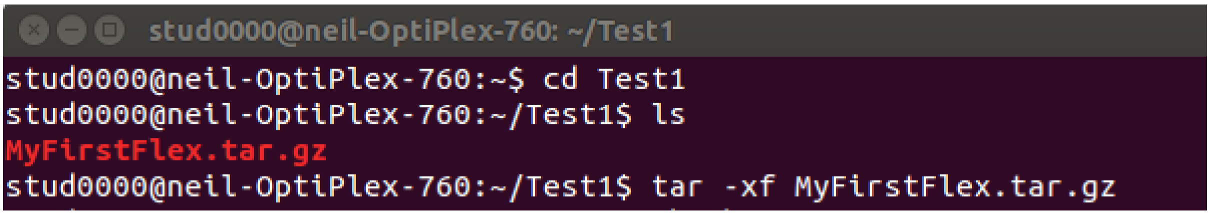
A NetBeans flex prototype project is provided in the "Resources" part of the subject web site; the project is packed as a tar.gz archive.



Index of /~nabg/131/Resources

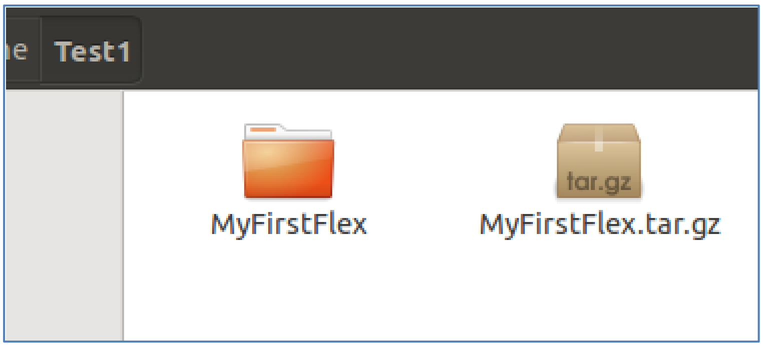
```
     Name                      Last modified       Size  Description

     Parent Directory          06-Jan-2015 15:16     -
     MyFirstFlex.tar.gz        07-Jan-2015 10:02    54k
     base64.c                  05-Dec-2014 15:09     6k
     base64.h                  05-Dec-2014 15:09     2k
     cities.txt                01-Dec-2014 11:59    25k
     datagen                   21-Nov-2014 10:59    13k
     libfilesystemdriver2.a    01-Dec-2014 12:00    14k
```

Fetch the MyFirstFlex.tar.gz file (the Chrome browser will put this file in your "Downloads" directory). Move the file to a newly created directory and unpack using the Linux tar command.

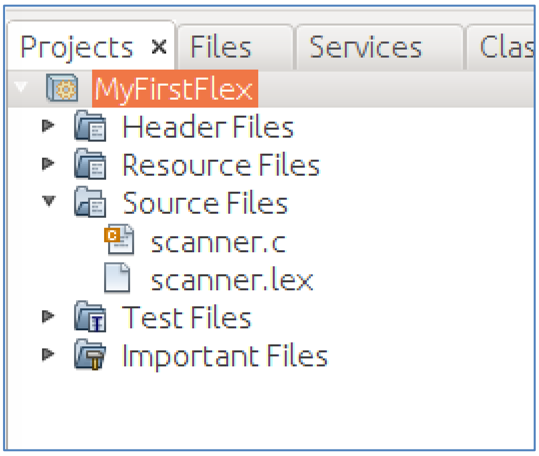


```
stud0000@neil-OptiPlex-760:~$ cd Test1
stud0000@neil-OptiPlex-760:~/Test1$ ls
MyFirstFlex.tar.gz
stud0000@neil-OptiPlex-760:~/Test1$ tar -xf MyFirstFlex.tar.gz
```

The tar program should extract the files re-creating a folder with the NetBeans project:

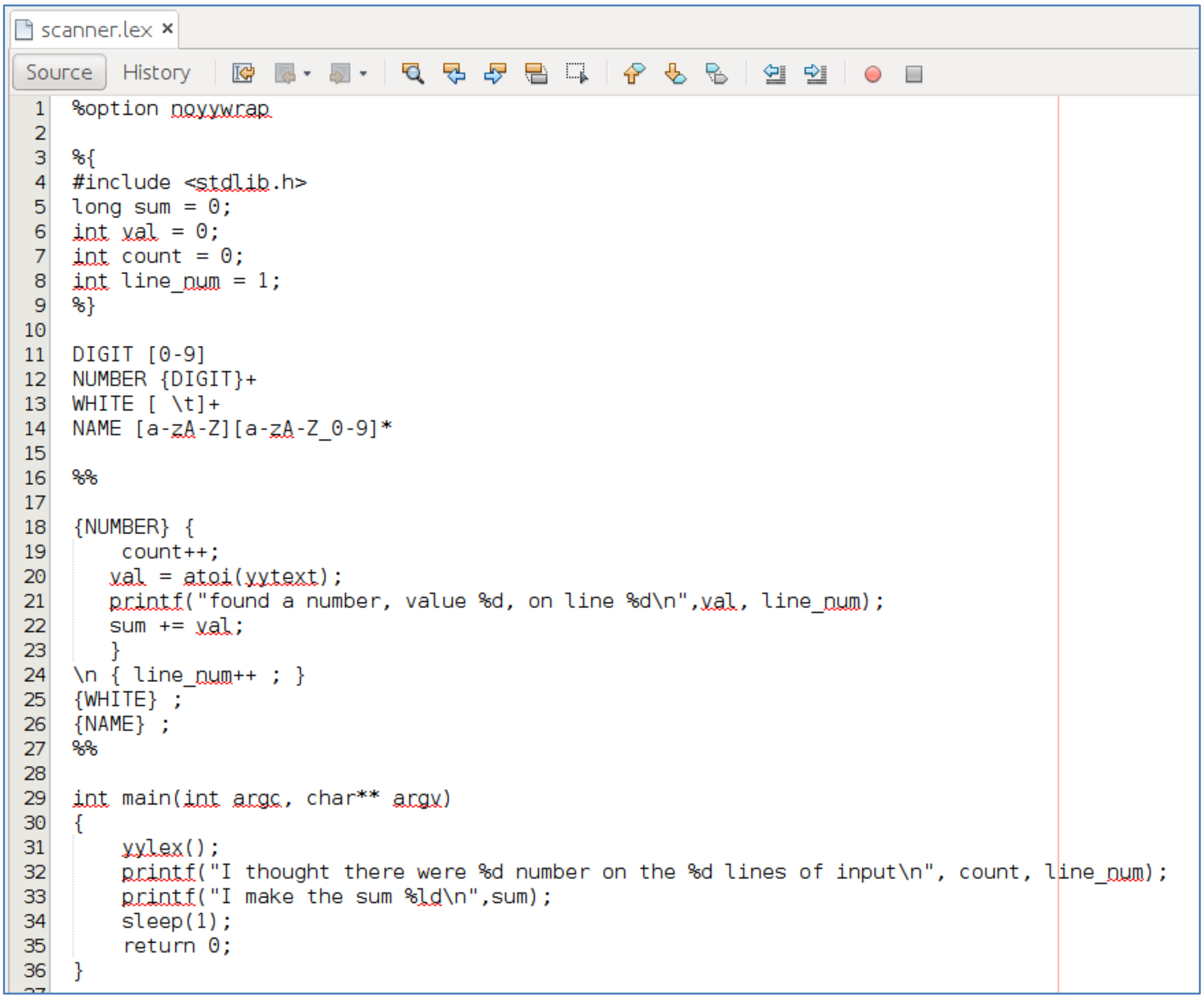


This prototype project has a “scanner.lex” file with flex code (the scanner.c file is the C code generated by compiling scanner.lex with flex).



The supplied scanner.lex file contains code for a very simple flex application. Compiling the supplied code, with flex and then gcc, results in a program that looks for numbers in an input stream.

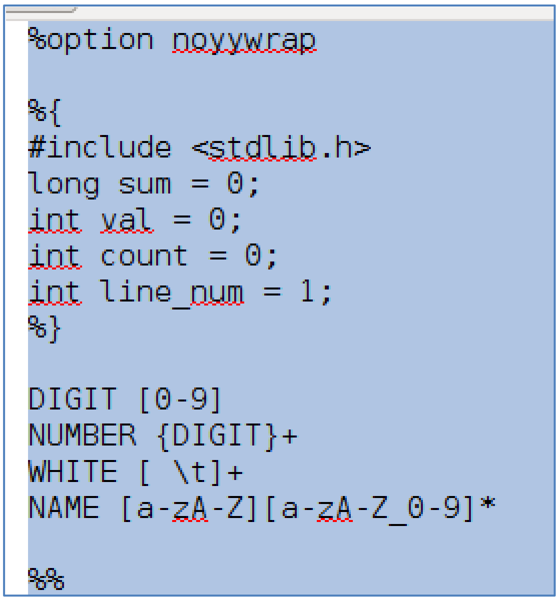
A flex program is comprised of three parts – definitions, rules, and user code.



```
%option noyywrap

%{
#include <stdlib.h>
long sum = 0;
int val = 0;
int count = 0;
int line_num = 1;
%}

DIGIT [0-9]
NUMBER {DIGIT}+
WHITE [ \t]+
NAME [a-zA-Z][a-zA-Z_0-9]*

%%

{NUMBER} {
    count++;
   val = atoi(yytext);
   printf("found a number, value %d, on line %d\n",val, line_num);
   sum += val;
   }
\n { line_num++ ; }
{WHITE} ;
{NAME} ;
%%

int main(int argc, char** argv)
{
    yylex();
    printf("I thought there were %d number on the %d lines of input\n", count, line_num);
    printf("I make the sum %ld\n",sum);
    sleep(1);
    return 0;
}
```

Flex definitions

The definitions section starts with any directives for the flex compiler (%option directives) and then will typically contain text (C code and comments) that is to be placed at the start of the generated C code file. There will also typically be some regular-expression definitions of patterns that the generated scanner is to look for when processing text input.

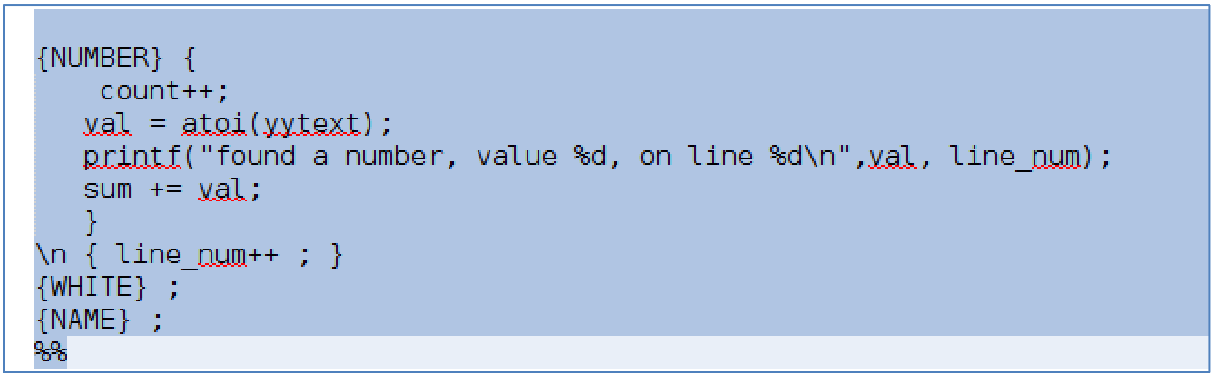


```
%option noyywrap

%{
#include <stdlib.h>
long sum = 0;
int val = 0;
int count = 0;
int line_num = 1;
%}

DIGIT [0-9]
NUMBER {DIGIT}+
WHITE [ \t]+
NAME [a-zA-Z][a-zA-Z_0-9]*

%%
```

In this example, the copied text (between %{ and %}) has an include statement and declarations of some global variables. The pattern definitions identify digits, numbers (a sequence of one or more digits), whitespace (space and tab characters), and "name" – any sequence comprising just letters. The definitions sections is terminated with %%.

Flex rules

Each flex rule consists of a pattern and an action. The pattern will be composed of literal character sequences, regular-expressions, or elements from the definitions section. The action part of a rule specifies the code that the generated scanner is to execute when it finds input text that matches a pattern.



```
{NUMBER} {
    count++;
   val = atoi(yytext);
   printf("found a number, value %d, on line %d\n",val, line_num);
   sum += val;
   }
\n { line_num++ ; }
{WHITE} ;
{NAME} ;
%%
```

The rules here state:

- If the scanner finds some input text that matches with "NUMBER" (i.e. a sequence of digits) then it is to:

  - Increment a count.
  - Convert the matched sequence of characters (available via the char* pointer yytext) into an integer value.
  - Print a message.
  - Add the integer value to a sum.

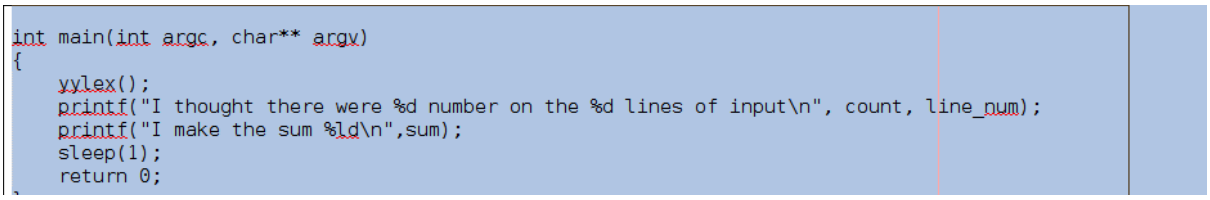
- If the scanner finds a newline, it is to increment the line_num counter;
- If the scanner finds some white space, it is to ignore it.
- If the scanner finds alphabetic strings, these are also ignored.
- There is an implicit rule that any other input that the scanner finds is to be echoed to stdout.

(The action part of a rule may be a single C statement – possibly just the empty statement ";" – or a C block.)

The pattern-action rules section is again terminated with %%.

Flex user code

The "user code" section would typically be empty in a flex file that contains code for the scanner that is to be used in a compiler. When flex is being used to build a scanner for a stand-alone application, the "user code" section will typically include the main() driver function that invokes the parser code (along with other application specific functions).



```
int main(int argc, char** argv)
{
    yylex();
    printf("I thought there were %d number on the %d lines of input\n", count, line_num);
    printf("I make the sum %ld\n",sum);
    sleep(1);
    return 0;
```

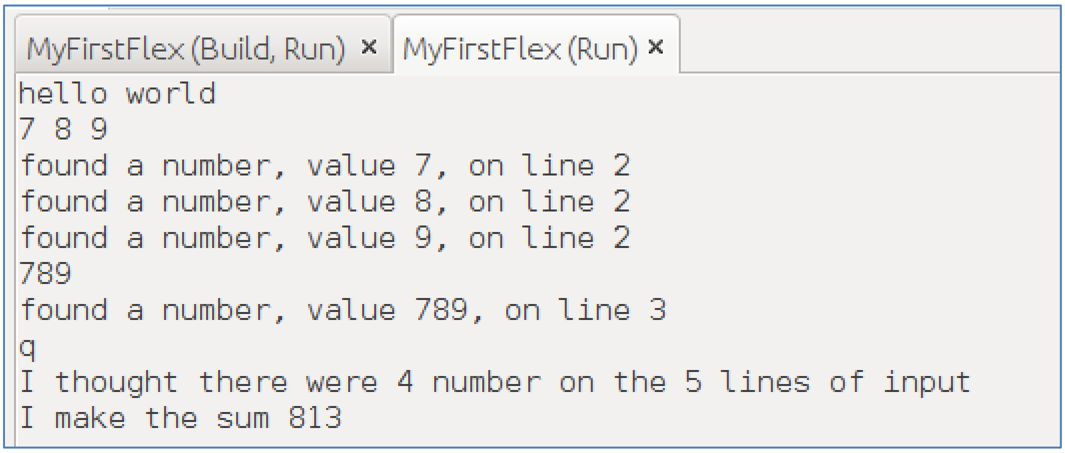
Here, main() immediately invokes yylex() – the driver routine that will run the generated scanner code. By default, the code invoked from yylex() will read from stdin.

Input lines are read and processed in accord with the code generated from the rules. When end-of-input is encountered, yylex() returns and any remaining statements in main() are executed. In this example, summary statistics are printed.

Build and run the protype

Just invoke build and then run.

Type in some input and check the resulting output. The program should recognize numbers. (If you enter characters like "+=(&%$#" these will simply be echoed – using the implicit default flex rule that one echoes any data that one doesn't understand.)



```
hello world
7 8 9
found a number, value 7, on line 2
found a number, value 8, on line 2
found a number, value 9, on line 2
789
found a number, value 789, on line 3
q
I thought there were 4 number on the 5 lines of input
I make the sum 813
```

Word-count application

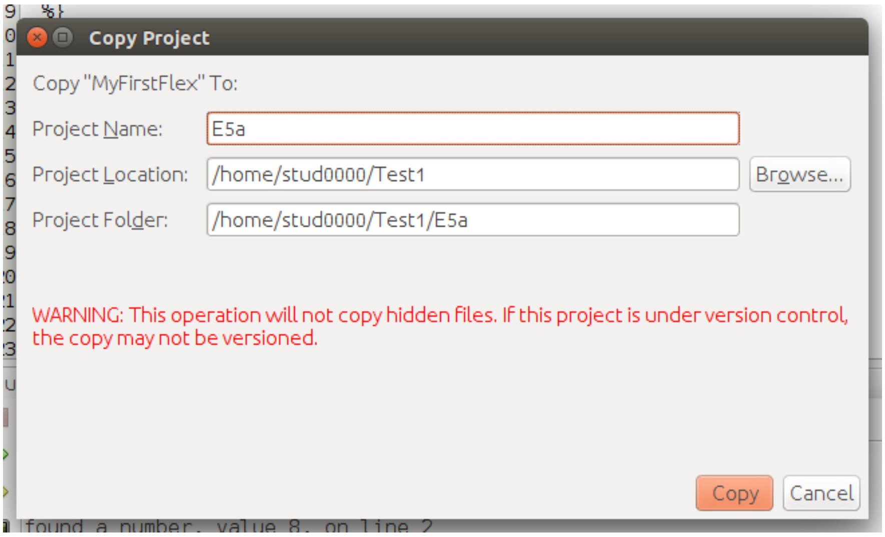
An application is required that will process a text file to determine all the distinct words that it contains along with their number of occurrences. When the input has all been consumed, the application is to print details of the words and counts; the words are to be printed in alphabetic order. (Words are all to be converted to solely lower case; “The” and “the” count as the same word.)

The text file must be read. Each line must be tokenized to finds words – sequences of letters. The program has to maintain a collection of distinct words; each word being stored along with a count. As each word is found in the current input line it should be “added” to the collection – if there is already an entry for that word its count is incremented, otherwise a new entry is to be created.

Spend 5 minutes thinking of how you might implement this in C++. You will probably get a bit bogged down in the code to tokenize input strings etc. It would take you quite a bit of time to implement a pure C++ version.

Then build the application in a small fraction of that time by exploiting flex to generate an input scanner that will resolve all issues of input handling for you.

Duplicate the prototype project



Copy "MyFirstFlex" To:

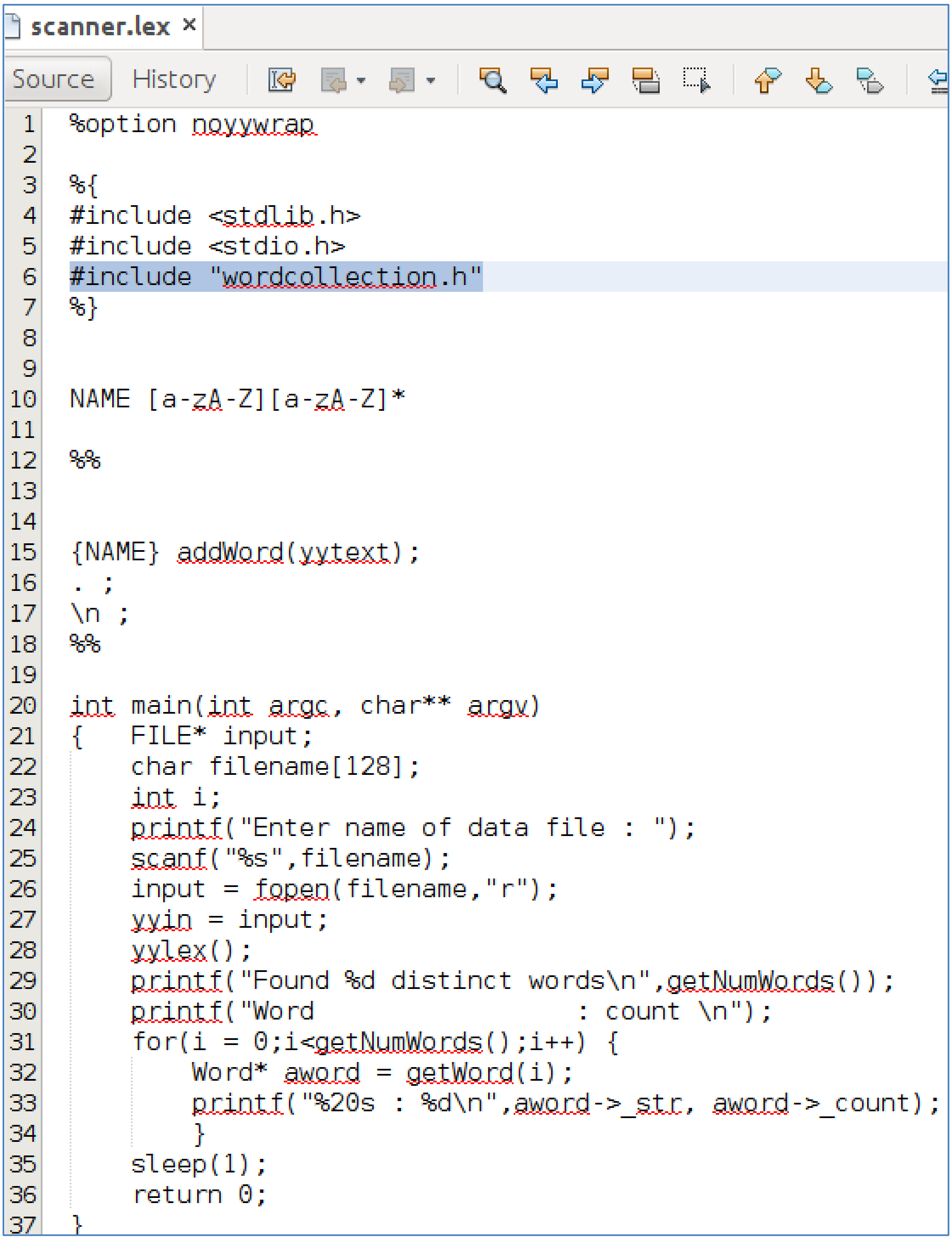
Project Name: E5a

Project Location: /home/stud0000/Test1

Project Folder: /home/stud0000/Test1/E5a

WARNING: This operation will not copy hidden files. If this project is under version control, the copy may not be versioned.

Replace the content of the scanner.lex file with the new flex program:



```
%option noyywrap

%{
#include <stdlib.h>
#include <stdio.h>
#include "wordcollection.h"
%}


NAME [a-zA-Z][a-zA-Z]*

%%


{NAME} addWord(yytext);
. ;
\n ;
%%

int main(int argc, char** argv)
{   FILE* input;
    char filename[128];
    int i;
    printf("Enter name of data file : ");
    scanf("%s",filename);
    input = fopen(filename,"r");
    yyin = input;
    yylex();
    printf("Found %d distinct words\n",getNumWords());
    printf("Word                 : count \n");
    for(i = 0;i<getNumWords();i++) {
        Word* aword = getWord(i);
        printf("%20s : %d\n",aword->_str, aword->_count);
        }
    sleep(1);
    return 0;
}
```

The definitions section contains code to be copied into the generated C file and a definition of “NAME” – a sequence of letters. The copied code here is just a set of #include statements. The code for maintaining a collection of words is going to go in separate files – wordcollection.h and wordcollection.c.

The rules section says

The rules here state:

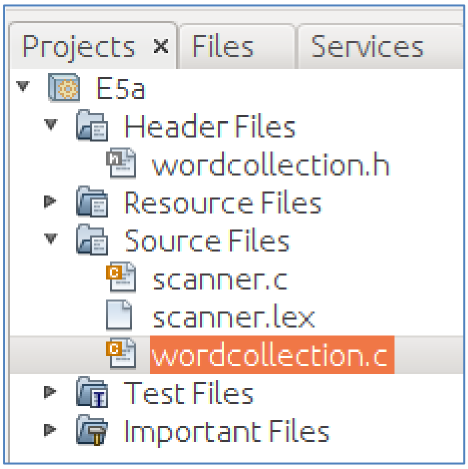
- If the scanner finds some input text that matches with "NAME" (i.e. a sequence of letters) then it is to invoke a function named "addWord()" (this will be defined in wordcollection.c)
- If the scanner finds a newline, it is to do nothing.
- If the scanner finds any other character in the input, it is to do nothing (digits and punctuation are not processed, but we don't want these characters echoed to stdout).

(Newlines are treated as distinct from other characters, so an explicit rule is necessary for these.)

The user code section has the definition of main().

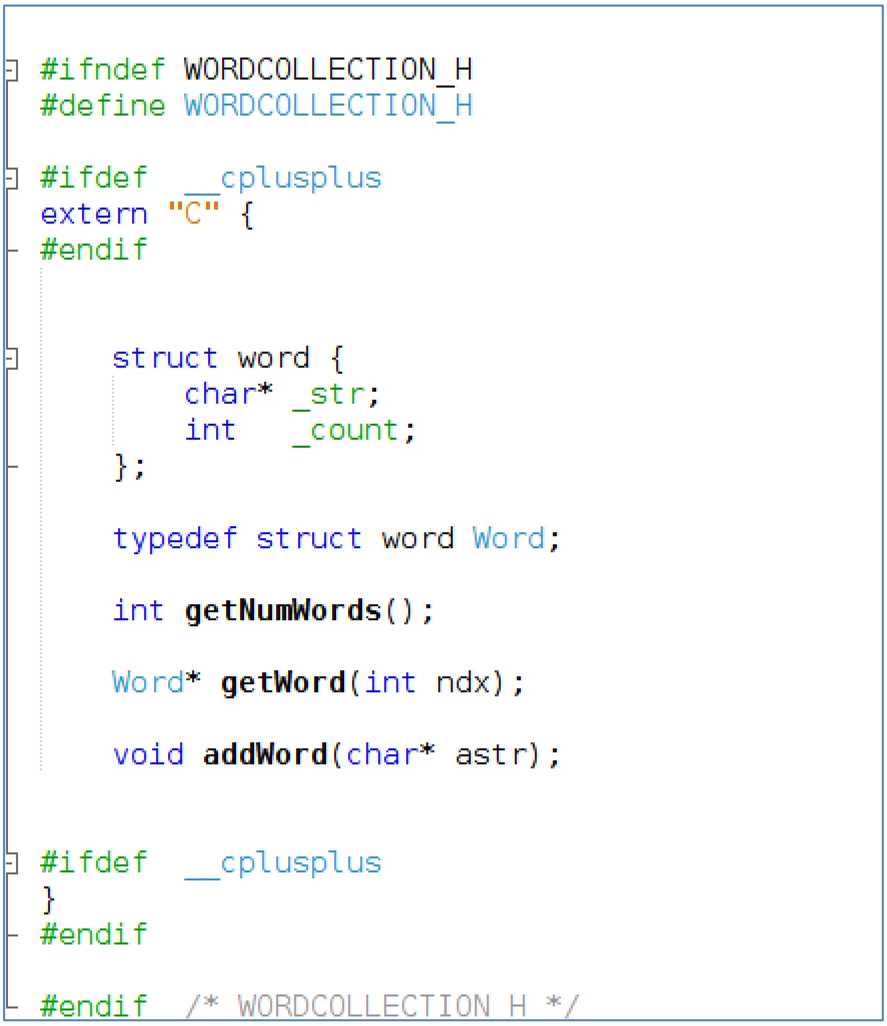
- As the program is to process data from a file (rather than text typed on stdin), it starts by prompting for the filename.
- The specified file is opened (no checks for failures!) as a FILE* stream input.
- Setting yyin = input; makes the scanner read from this stream rather than stdin.
- The scanner is invoked via the call to yylex(). The scanner will process the input text in accord with the supplied rules.
- When the end-of-file is reached, yylex returns; the remaining statements in main() list the words and their counts.
- (The sleep(1) statement is there to prevent NetBeans from cutting off the end of output from the program.)

Apart from scanner.lex (and the generated scanner.c file), this project must also define the code for the wordcollection.



Create the files wordcollection.h and wordcollection.c as part of the project.

wordcollection.h

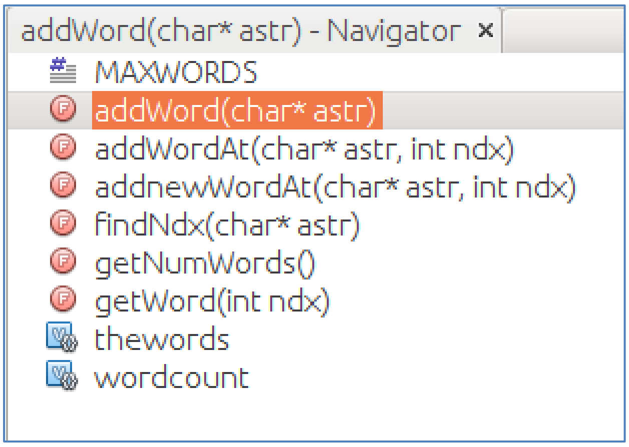


```
#ifndef WORDCOLLECTION_H
#define WORDCOLLECTION_H

#ifdef  __cplusplus
extern "C" {
#endif


    struct word {
        char*  _str;
        int    _count;
    };

    typedef struct word Word;

    int getNumWords();

    Word* getWord(int ndx);

    void addWord(char* astr);


#ifdef  __cplusplus
}
#endif

#endif  /* WORDCOLLECTION_H */
```

A Word (a typedefed type) is a struct with a char* pointer to a string allocated in the heap, and an integer count.

The word collection is manipulated through the functions addWord(), getWord(), and getNumWords().

wordcollection.c



addWord(char* astr) - Navigator ×

MAXWORDS

addWord(char* astr)

addWordAt(char* astr, int ndx)

addnewWordAt(char* astr, int ndx)

findNdx(char* astr)

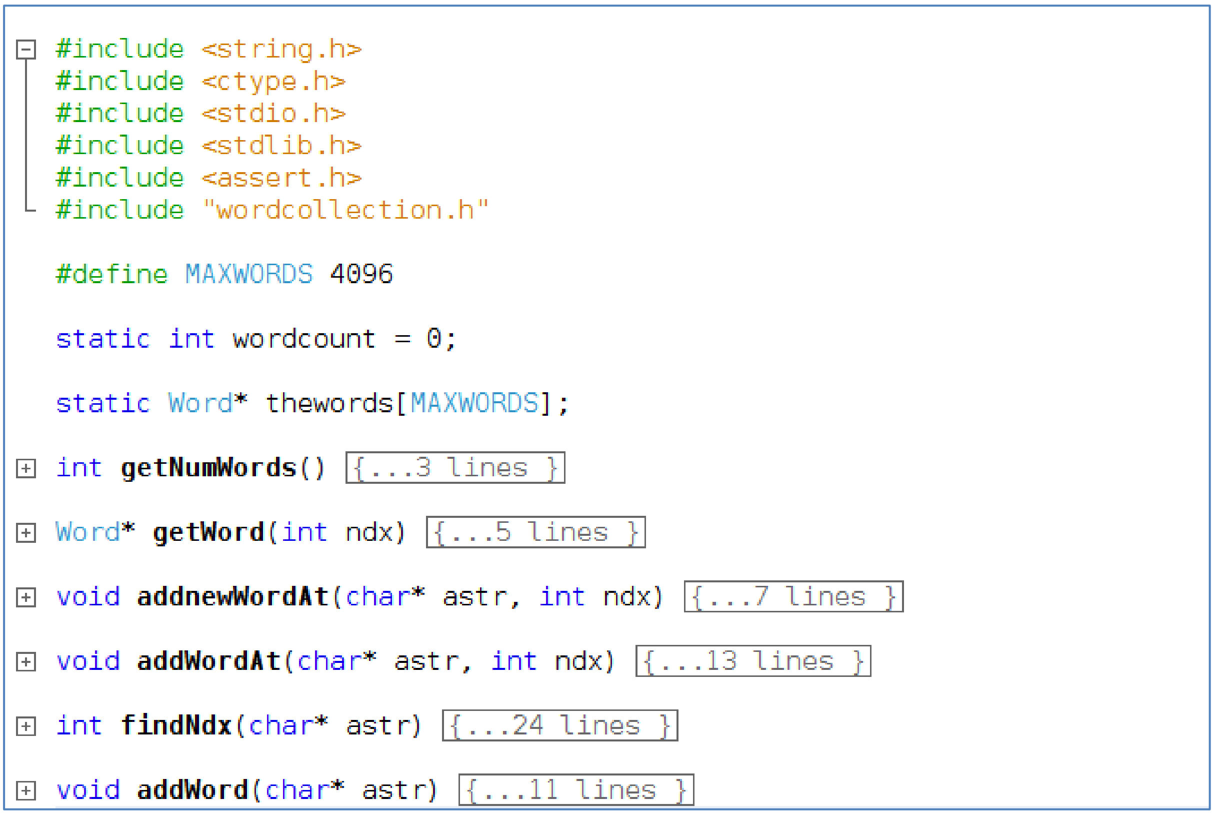
getNumWords()

getWord(int ndx)

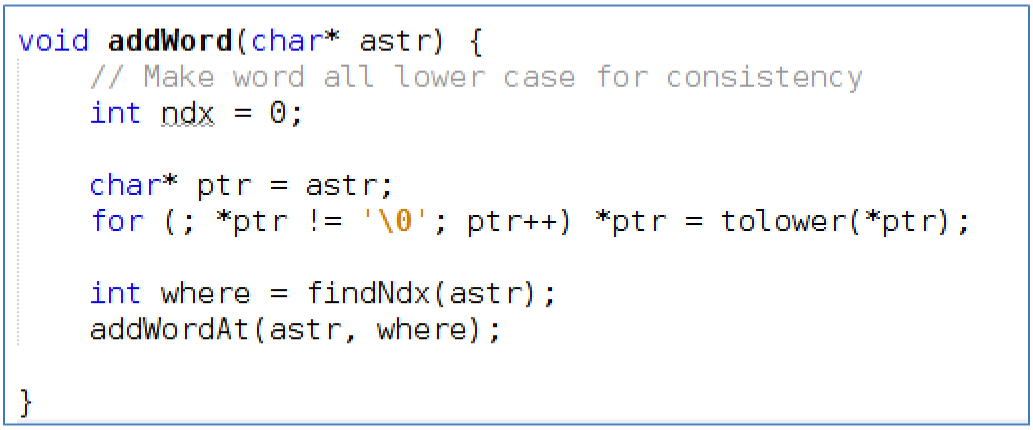
thewords

wordcount

The implementation of the word collection is quite naïve. It uses an array of Word* pointers. The array entries are kept in alphabetic order. When a new word is to be inserted, the appropriate position is found and alphabetically later entries are simply moved up one spot so as to allow the pointer to the word to be set appropriately. Since the array is ordered, a binary search can be used when seeking an existing entry or trying to find the spot for a new entry.



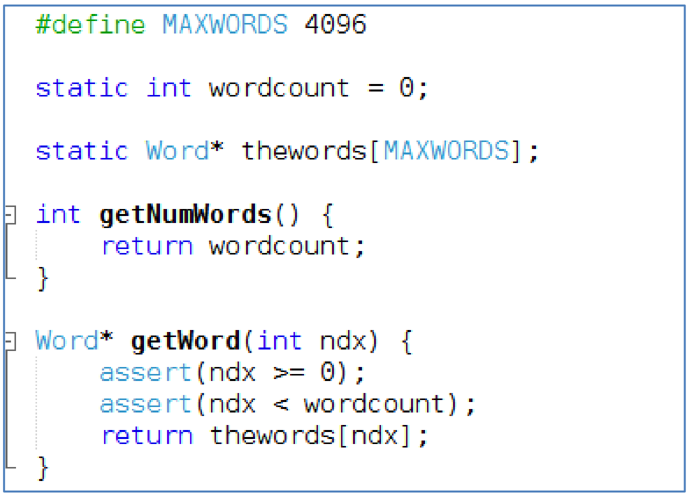
```
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "wordcollection.h"

#define MAXWORDS 4096

static int wordcount = 0;

static Word* thewords[MAXWORDS];

int getNumWords() {...3 lines }

Word* getWord(int ndx) {...5 lines }

void addnewWordAt(char* astr, int ndx) {...7 lines }

void addWordAt(char* astr, int ndx) {...13 lines }

int findNdx(char* astr) {...24 lines }

void addWord(char* astr) {...11 lines }
```



```
void addWord(char* astr) {
    // Make word all lower case for consistency
    int ndx = 0;

    char* ptr = astr;
    for (; *ptr != '\0'; ptr++) *ptr = tolower(*ptr);

    int where = findNdx(astr);
    addWordAt(astr, where);

}
```

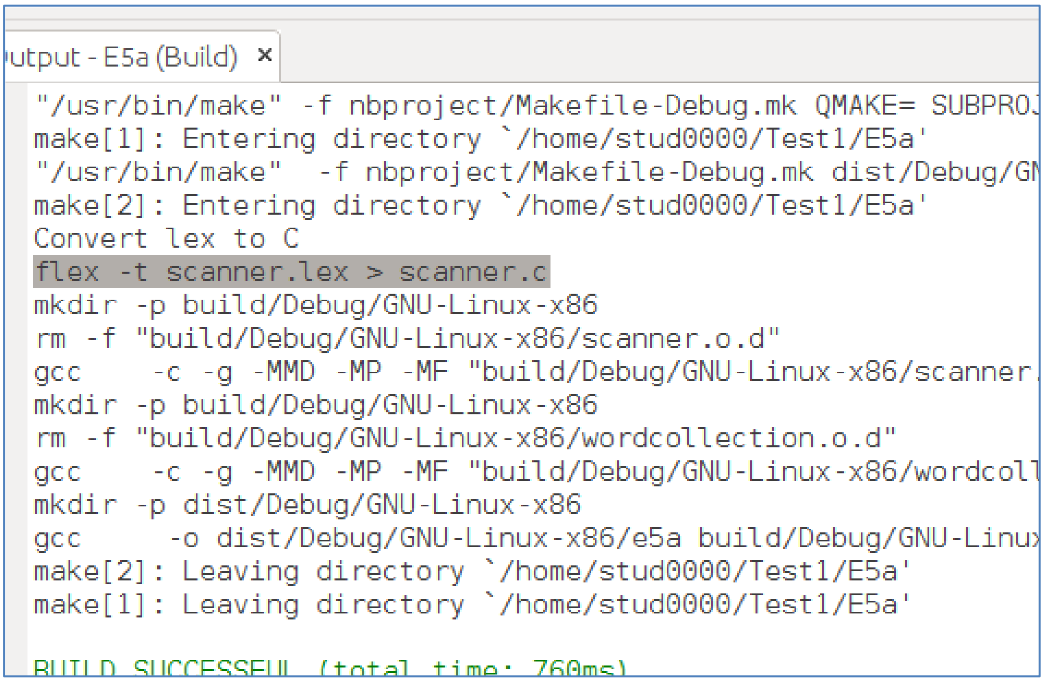
```c
int findNdx(char* astr) {
    // Ordered array
    // Binary search
    // Find index such that all earlier entries have string that
    // sorts less than 'astr'
    int min = 0;
    int max = wordcount - 1;
    if (wordcount == 0) return 0;

    while (min <= max) {
        int pos = (min + max) / 2;
        Word* aword = thewords[pos];
        int cmp = strcmp(aword->_str, astr);
        if (cmp < 0) {
            min = pos + 1;
        } else
            if (cmp == 0) return pos;
        else {
            max = pos - 1;
        }
    }

    return min;
}
```

```c
void addnewWordAt(char* astr, int ndx) {
    Word* newword = malloc(sizeof (Word));
    newword->_count = 1;
    newword->_str = strdup(astr);
    thewords[ndx] = newword;
    wordcount++;
}

void addWordAt(char* astr, int ndx) {
    if (ndx == wordcount) {
        addnewWordAt(astr, ndx);
    } else
        if (0 == strcmp(astr, thewords[ndx]->_str)) {
        thewords[ndx]->_count++;
    } else {
        int i;
        for (i = wordcount; i > ndx; i--)
            thewords[i] = thewords[i - 1];
        addnewWordAt(astr, ndx);
    }
}
```



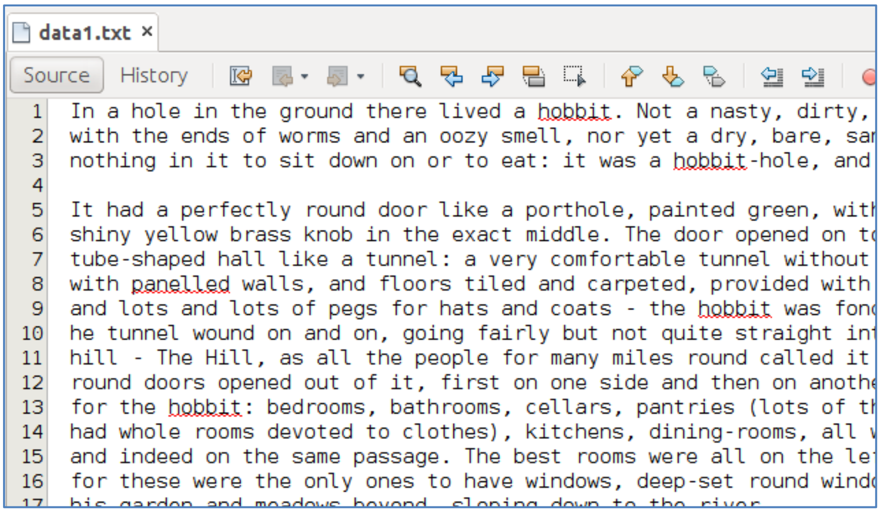
```
#define MAXWORDS 4096

static int wordcount = 0;

static Word* thewords[MAXWORDS];

int getNumWords() {
    return wordcount;
}

Word* getWord(int ndx) {
    assert(ndx >= 0);
    assert(ndx < wordcount);
    return thewords[ndx];
}
```

Build the application



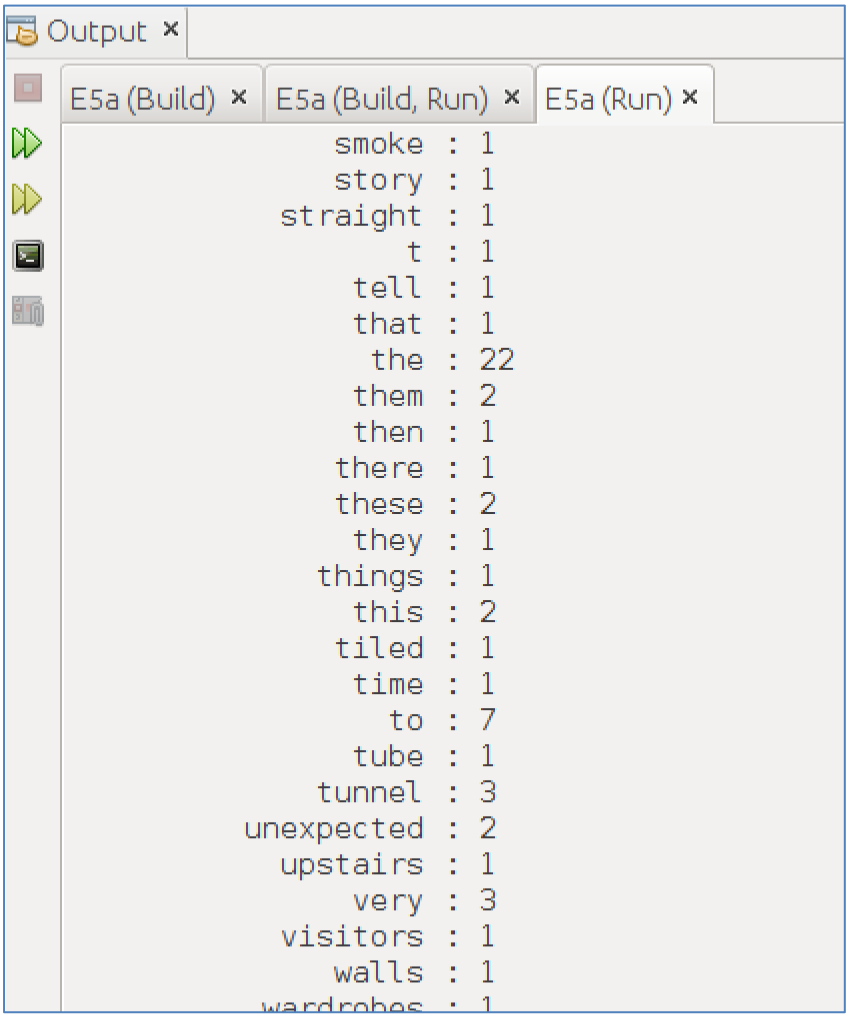
```
"/usr/bin/make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJ
make[1]: Entering directory `/home/stud0000/Test1/E5a'
"/usr/bin/make"  -f nbproject/Makefile-Debug.mk dist/Debug/GN
make[2]: Entering directory `/home/stud0000/Test1/E5a'
Convert lex to C
flex -t scanner.lex > scanner.c
mkdir -p build/Debug/GNU-Linux-x86
rm -f "build/Debug/GNU-Linux-x86/scanner.o.d"
gcc    -c -g -MMD -MP -MF "build/Debug/GNU-Linux-x86/scanner.
mkdir -p build/Debug/GNU-Linux-x86
rm -f "build/Debug/GNU-Linux-x86/wordcollection.o.d"
gcc    -c -g -MMD -MP -MF "build/Debug/GNU-Linux-x86/wordcoll
mkdir -p dist/Debug/GNU-Linux-x86
gcc     -o dist/Debug/GNU-Linux-x86/e5a build/Debug/GNU-Linux
make[2]: Leaving directory `/home/stud0000/Test1/E5a'
make[1]: Leaving directory `/home/stud0000/Test1/E5a'

BUILD SUCCESSFUL (total time: 760ms)
```

Grab a text file from the Internet to use as input



```
In a hole in the ground there lived a hobbit. Not a nasty, dirty,
with the ends of worms and an oozy smell, nor yet a dry, bare, sa
nothing in it to sit down on or to eat: it was a hobbit-hole, and

It had a perfectly round door like a porthole, painted green, wit
shiny yellow brass knob in the exact middle. The door opened on t
tube-shaped hall like a tunnel: a very comfortable tunnel without
with panelled walls, and floors tiled and carpeted, provided with
and lots and lots of pegs for hats and coats - the hobbit was fon
he tunnel wound on and on, going fairly but not quite straight in
hill - The Hill, as all the people for many miles round called it
round doors opened out of it, first on one side and then on anoth
for the hobbit: bedrooms, bathrooms, cellars, pantries (lots of t
had whole rooms devoted to clothes), kitchens, dining-rooms, all 
and indeed on the same passage. The best rooms were all on the le
for these were the only ones to have windows, deep-set round wind
his garden and meadows beyond, sloping down to the river.
```

Run the application

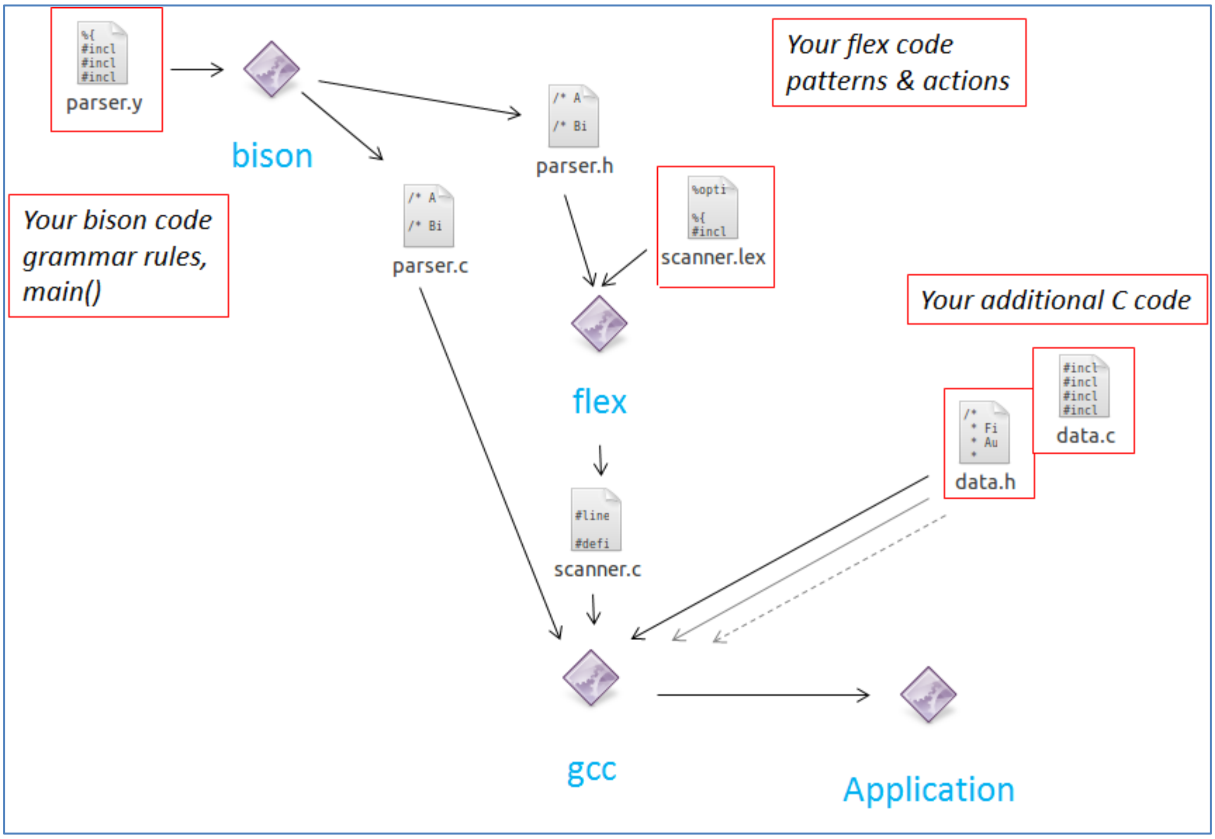


```
     smoke : 1
     story : 1
  straight : 1
         t : 1
      tell : 1
      that : 1
       the : 22
      them : 2
      then : 1
     there : 1
     these : 2
      they : 1
    things : 1
      this : 2
     tiled : 1
      time : 1
        to : 7
      tube : 1
    tunnel : 3
unexpected : 2
  upstairs : 1
      very : 3
  visitors : 1
     walls : 1
 wardrobes : 1
```

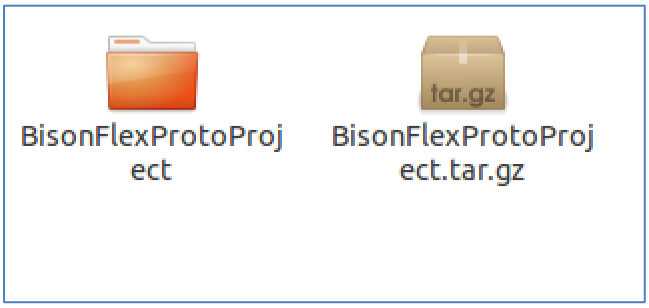
Task 2: Bison – parsing and processing: 5 marks

Parser generators are not unlike scanner generators – you give them a set of rules and actions and they generate the necessary C code. It's just that the grammar rules are a lot more complex than the simple pattern-action rules for a scanner.

The compilation process is also more complex.



A prototype bison-flex NetBeans project is supplied in the Resources part of the subject web-site; you should download BisonFlexProtoProject.tar.gz and unpack the files.

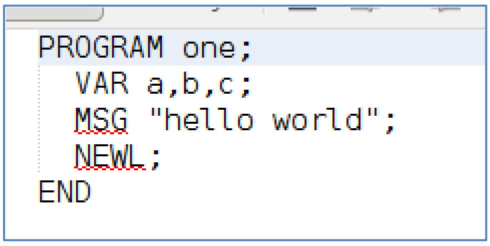


The prototype project contains a tiny part of the code of a compiler. Really, all that it does is read the source code for a single function (in a simplified quasi Basic syntax) and handle variable declarations. It has some example grammar rules in "parser.y", and flex code for generating a scanner to recognise elements in "scanner.lex". A more realistic compiler can be created by replacing the content of these files.

BisonFlex Proto Project

Code compiled

The programs that it is to compile (actually, interpret) take a form like the following:



```
PROGRAM one;
   VAR a,b,c;
   MSG "hello world";
   NEWL;
END
```

A program has a “PROGRAM” statement with a name; a declarations line that declares some variables (implicitly all are short integer type); any number of “MSG” statements that print quoted strings when the compiled code is run; and “NEWL” statements that would print newlines. The program must terminate with an END statement.

It’s not much of a programming language. (You can declare variables, but you cannot use them.) It is not intended to be a practical language. It is just a minimal example that serves to illustrate the definition of the bison grammar rules and flex pattern-action rules.

Bison code – grammar and mainline

The structure of a bison source file is similar to that of a flex source file.

There are three sections:

- A definitions section

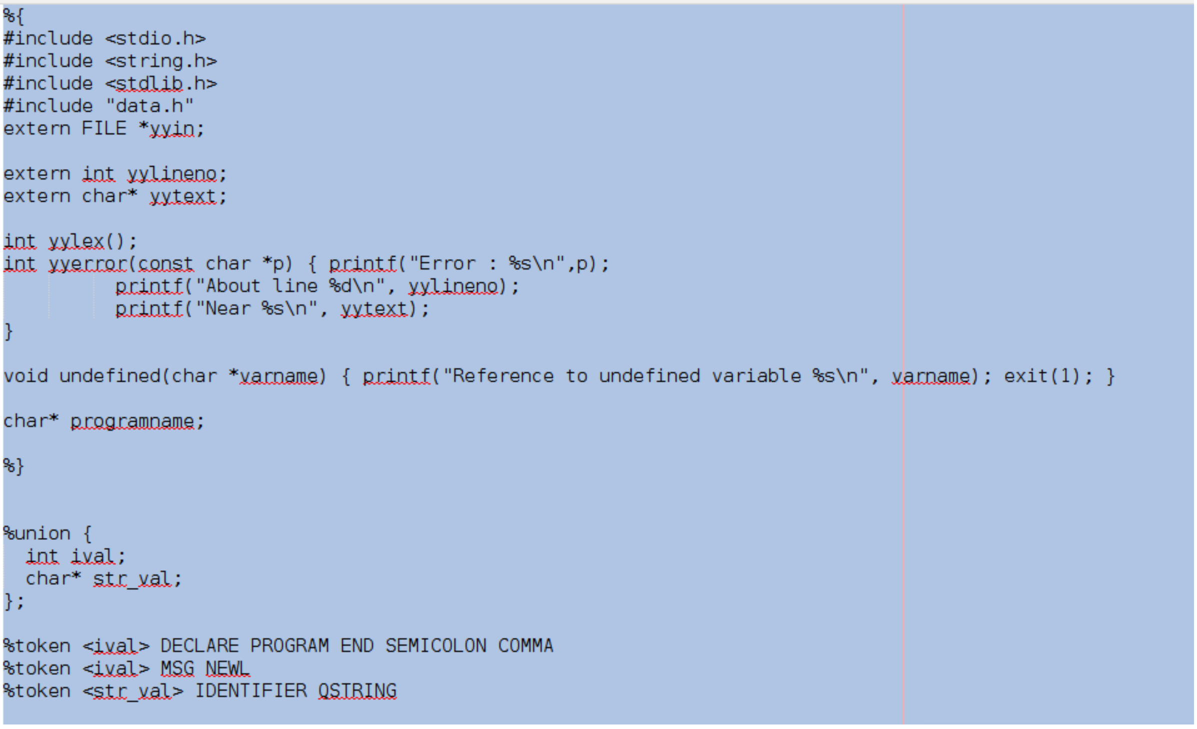
    - Code to be copied and placed at the start of a generated parser.c file; this will have #include statements, declarations of global variables and constants etc.
    - Definitions of the tokens that the scanner is to find for the parser. (These definitions get processed to generate a C enum and other structs. The flex generated parser is to identify input strings, loading the values into structures, and reporting their enum value to the parser.)
    - Specification of the “type” associated with different enums – integer, string, sometimes other types.

- The grammar rules

    - Each rule will define a construct in the language that is to be compiled, and action code that specifies how the parser is to handle that construct.

- User code – the main() that will invoke the parsing process

Prototype bison code - definitions



```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "data.h"
extern FILE *yyin;

extern int yylineno;
extern char* yytext;

int yylex();
int yyerror(const char *p) { printf("Error : %s\n",p);
          printf("About line %d\n", yylineno);
          printf("Near %s\n", yytext);
}

void undefined(char *varname) { printf("Reference to undefined variable %s\n", varname); exit(1); }

char* programname;

%}


%union {
  int ival;
  char* str_val;
};

%token <ival> DECLARE PROGRAM END SEMICOLON COMMA
%token <ival> MSG NEWL
%token <str_val> IDENTIFIER QSTRING
```

Copied code (within %{ and %} brackets) –

- Include statements
- “extern” declarations – flex will generate a yylex() function that has to be called by the code generated for the parser, and defines variables yylineno (line of input), and yytext (text matching current token) but it doesn’t generate any header file that could be included so these explicit extern declarations are needed.
- yyerror – this is a replacement for the default error handling function that the bison parser would have generated; this version tries to identify where an error was detected.
- undefined() – will be used with later more complex versions of the grammar, it will report on undefined variables.
- Global variable to hold name of program being compiled

Bison struct and (implicit) enum token definition –

- The (anonymous) union structure definition specifies how flex will return information about a current matched token. It may return an integer, or a string (you will see how later in the flex code). You could have other elements – e.g. a float if the language needed to handle floating point numbers.
- Tokens:
  What is the scanner to find?

  - A “PROGRAM” token (this will be the string PROGRAM)
  - A “DECLARE” token (this will be the string VAR)
  - An “END” token (the string END)
  - A “SEMICOLON” token (this will be a semi-colon character)

  - A “COMMA” token (a comma character)
  - A “MSG” token, and a “NEWL” token (once again, these literal strings)
  - An “IDENTIFIER” token (this will have a flex pattern specifying a sequence of lower case letters)
  - A “QSTRING” token (this will have a flex pattern specifying a doubly quoted string).

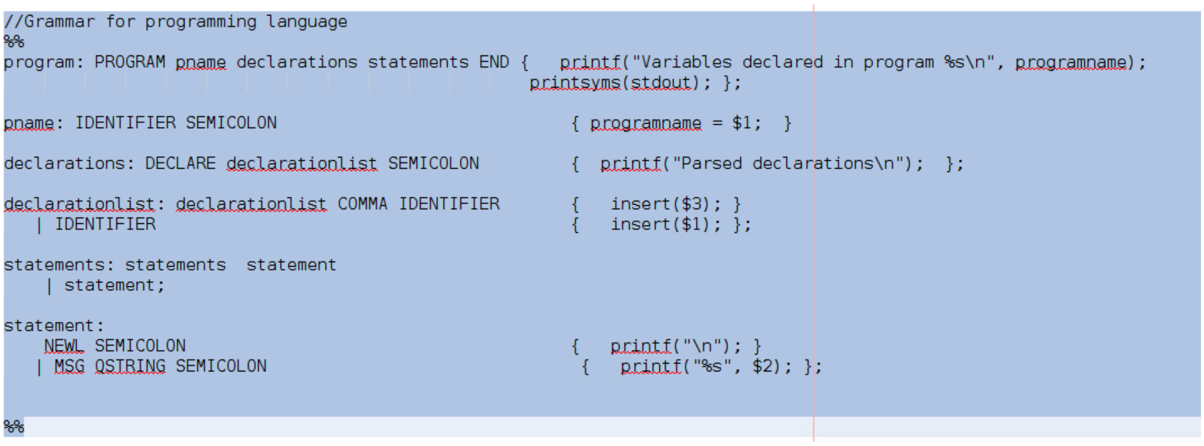
- Tokens can have values of different types

  - Most of those tokens were elements in the grammar and will simply have integer values as defined in the C enum.
  - IDENTIFIERS and QSTRINGS will have string values

- Token values will be held in instances of that (anonymous) union that was declared. When the tokens are defined, the union member name is used to specify their type. So MSG is <ival> (integer) while QSTRING is str_val (char* string).

(Tokens can be declared individually, or may be grouped. The organisation of token declarations only matters in more complex examples. In this simple case, it’s simply whatever was convenient for the programmer writing the bison code.)

Prototype bison code – grammar rules

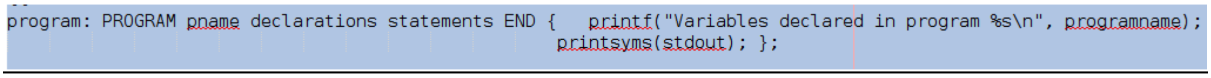


```
//Grammar for programming language
%%
program: PROGRAM pname declarations statements END {   printf("Variables declared in program %s\n", programname);
                                                   printsyms(stdout); };

pname: IDENTIFIER SEMICOLON                            { programname = $1;  }

declarations: DECLARE declarationlist SEMICOLON        {  printf("Parsed declarations\n");  };

declarationlist: declarationlist COMMA IDENTIFIER      {   insert($3); }
   | IDENTIFIER                                        {   insert($1); };

statements: statements  statement
    | statement;

statement:
    NEWL SEMICOLON                                     {   printf("\n"); }
   | MSG QSTRING SEMICOLON                              {   printf("%s", $2); };


%%
```

(There aren’t strict formatting guidelines for grammar rules. But it helps if the grammar rules are laid out in a quasi-tabular form.)

Rules for the simple programming language:

Program



```
program: PROGRAM pname declarations statements END {   printf("Variables declared in program %s\n", programname);
                                                   printsyms(stdout); };
```

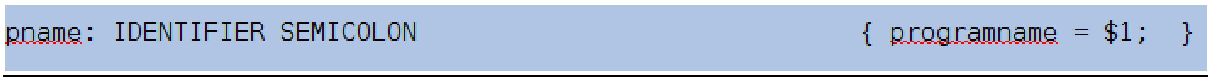
- A program consists of:

  - A PROGRAM token
  - An element that is a “pname” (non-terminal)
  - An element that is a “declarations” (non-terminal)
  - An element that is a “statements” (non-terminal)

  - An END token

- If the parser successfully matches a complete program, it is to finish by printing off the symbol table with the variables that were declared in the program.

pname



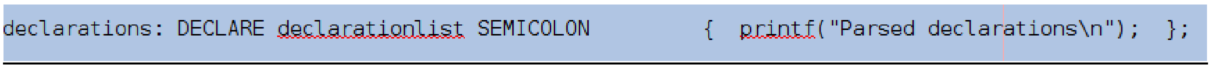
```
pname: IDENTIFIER SEMICOLON                          { programname = $1;   }
```

- A pname consists of:

  - An IDENTIFIER token
  - A SEMICOLON token

- If the parser successfully matches a pname, it is to save the name in the global variable programname.

declarations



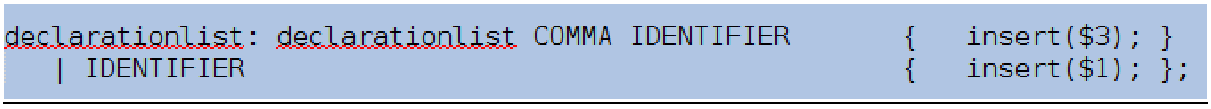
```
declarations: DECLARE declarationlist SEMICOLON          {  printf("Parsed declarations\n");  };
```

- A declarations consists of:

  - A DECLARE token
  - A declarationlist (non-terminal)
  - A SEMICOLON token

- If the parser successfully matches a “declarations” it is to announce this achievement.

declarationlist



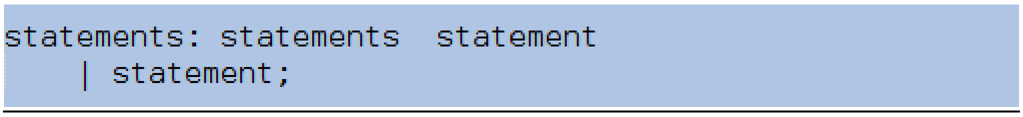
```
declarationlist: declarationlist COMMA IDENTIFIER       {   insert($3); }
    | IDENTIFIER                                        {   insert($1); };
```

- A declarationlist could consist of:

  - A” delarationlist” followed by a COMMA token and then an IDENTIFIER token
  - Or, simply an IDENTIFIER token

- If the parser matches the first pattern, then the third part of the match $3 (an IDENTIFIER) is the name of another variable that is being declared. The parser should insert the value of this third part into the symbol table (the insert function is defined in “data.c” illustrated later).
- If the parser matches the alternative pattern, then the single element $1 matched is an IDENTIFIER that must be inserted into the symbol table.
- This is first example with multiple rules. The “|” symbol separates the different rules. The action part of the final rule must have a terminating semi-colon.

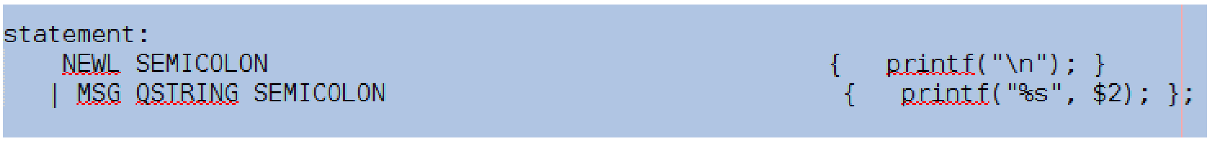
statements



```
statements: statements  statement
    | statement;
```

- A statements could consist of:
  - A “statements” followed by a statement
  - Or, just a single statement
- The parser doesn’t have to do anything when handling these; any work required will be done when handling each single statement.

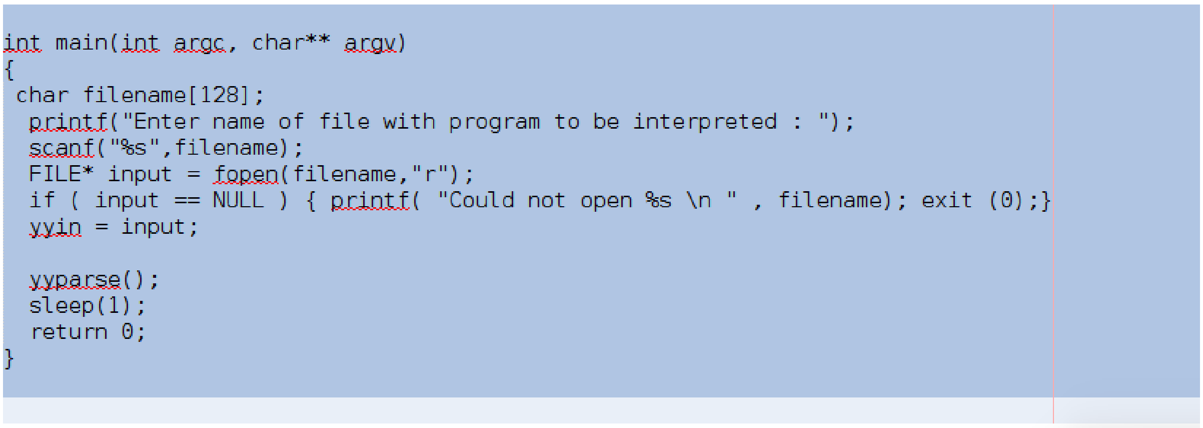
statement



```
statement:
    NEWL SEMICOLON                                          {   printf("\n"); }
   | MSG QSTRING SEMICOLON                                   {   printf("%s", $2); };
```

- A statements could consist of:
  - A NEWL token followed by a SEMICOLON token
  - Or, a MSG token followed by a QSTRING, and the a SEMICOLON
- In the first case, the parser (which is actually interpreting the program rather than compiling it) will print a newline; in the second case, the parser will print the string (the second element $2 in the MSG QSTRING SEMICOLON pattern.

Prototype bison code – main() etc

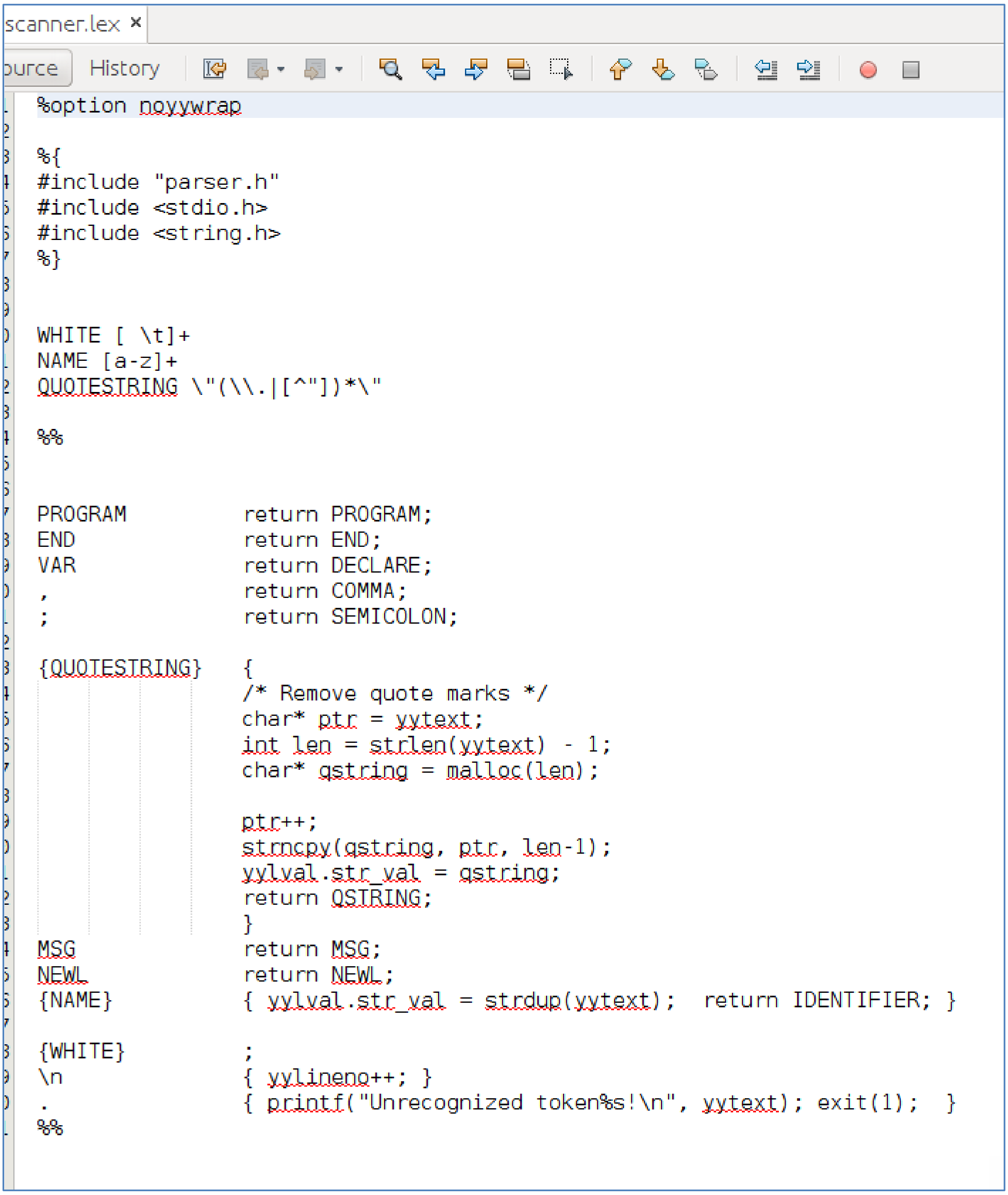


```
int main(int argc, char** argv)
{
 char filename[128];
  printf("Enter name of file with program to be interpreted : ");
  scanf("%s",filename);
  FILE* input = fopen(filename,"r");
  if ( input == NULL ) { printf( "Could not open %s \n " , filename); exit (0);}
  yyin = input;

  yyparse();
  sleep(1);
  return 0;
}
```

It is a simple driver program:

- Get the name of the input file, open a FILE*, and set this as the input for the scanner.
- Invoke the parser.
- (sleep(1) – just for NetBeans, allow all output to be printed after program termination).

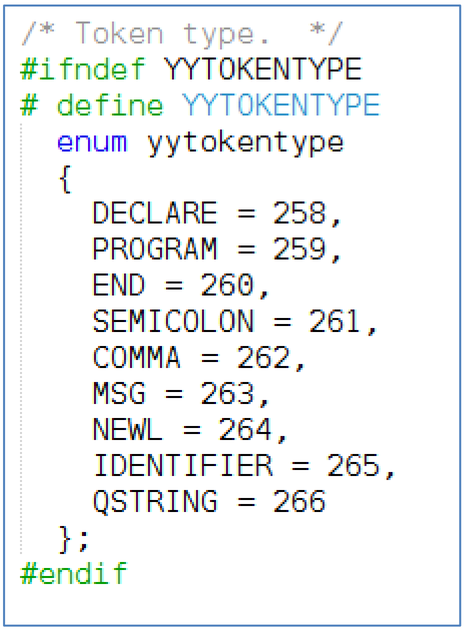
The flex code



```
%option noyywrap

%{
#include "parser.h"
#include <stdio.h>
#include <string.h>
%}


WHITE [ \t]+
NAME [a-z]+
QUOTESTRING \"(\\.|[^"])*\"

%%


PROGRAM         return PROGRAM;
END             return END;
VAR             return DECLARE;
,               return COMMA;
;               return SEMICOLON;

{QUOTESTRING}   {
                /* Remove quote marks */
                char* ptr = yytext;
                int len = strlen(yytext) - 1;
                char* qstring = malloc(len);

                ptr++;
                strncpy(qstring, ptr, len-1);
                yylval.str_val = qstring;
                return QSTRING;
                }
MSG             return MSG;
NEWL            return NEWL;
{NAME}          { yylval.str_val = strdup(yytext);   return IDENTIFIER; }

{WHITE}         ;
\n              { yylineno++; }
.               { printf("Unrecognized token%s!\n", yytext); exit(1);  }
%%
```

As is typical for a flex program used to generate the scanner for a compiler, this flex program has only two parts – definitions and pattern-action rules.

Note the inclusion of parser.h. This file will be generated by bison. It will have an enum for the tokens that the scanner is to return.



```
/* Token type.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
  enum yytokentype
  {
    DECLARE = 258,
    PROGRAM = 259,
    END = 260,
    SEMICOLON = 261,
    COMMA = 262,
    MSG = 263,
    NEWL = 264,
    IDENTIFIER = 265,
    QSTRING = 266
  };
#endif
```

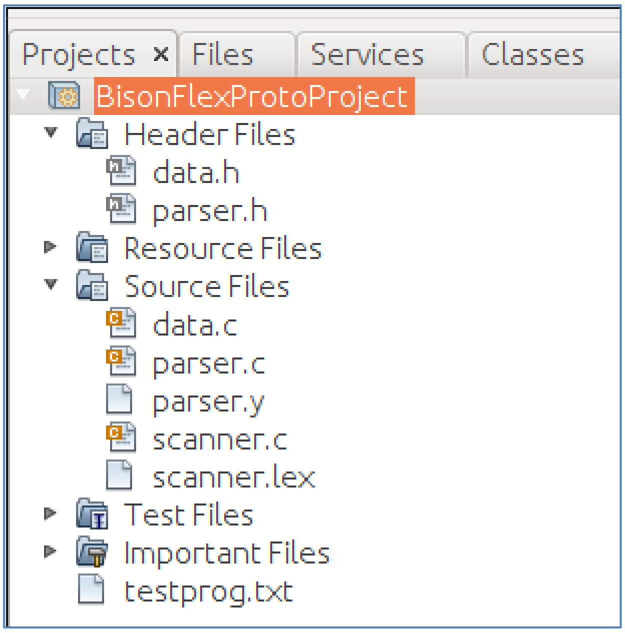
The definitions section scanner.lex defines “WHITE” as any sequence of space and tabs, “NAME” as a sequence of lower case letters, and “QUOTESTRING” as a starting double quote character, any characters, and finally a closing double quote character.

The flex rules:

- PROGRAM   return PROGRAM;
  (if match the literal string PROGRAM, return the enum token PROGRAM).
- VAR   return DECLARE;
  (if match the literal string VAR, return the enum token DECLARE; while it is common to have the enum token names the same as the literal character sequences, as with PROGRAM, this isn’t required.)
- ;   return SEMICOLON;
  (if match a semicolon character, return the enum token SEMICOLON).
- MSG return MSG; NEWL return NEWL;
  (similar to others)
- {NAME}   { yylval.str_val = …; return IDENTIFIER; }
  (if match a name pattern, i.e. a sequence of lower case letters, duplicate this string with strdup, assign it to the str_val member of the yylval structure – this yylval is an instance of that union defined in the bison code; finally return the enum token IDENTIFIER. Note that a new string gets created every time a name is found; the code defined in the parser should free these strings if it does not need them.)
- {QUOTESTRING} { *lots of code*; return QSTRING; }
  (if match a quoted string, make a copy without the quotes, store in yylval, return QSTRING. Again, a string does get allocated on heap and really should be free when no longer needed.)
- {WHITE} ;
  (Ignore extraneous white space).
- .   { printf(…); … exit(1); }
  (if find any unexpected characters in the input, just terminate.)

(Literal strings, e.g. PROGRAM, can be used directly as patterns as shown above; or they may be bracketed by quote characters – 'PROGRAM'. Quoting the strings is sometimes necessary, e.g. strings that incorporate spaces or other characters that might confuse the scanner generator.)

The complete BisonFlexProtoProject



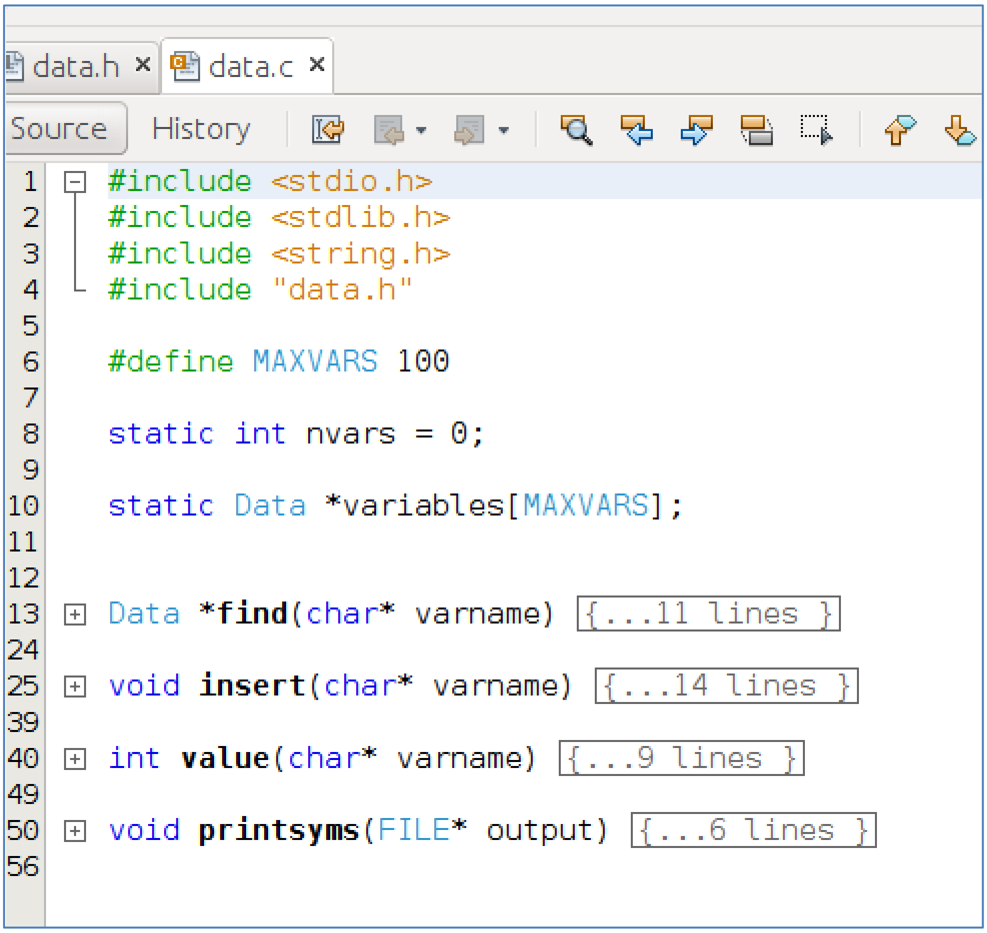
```
BisonFlexProtoProject
  ▼ Header Files
      data.h
      parser.h
  ▶ Resource Files
  ▼ Source Files
      data.c
      parser.c
      parser.y
      scanner.c
      scanner.lex
  ▶ Test Files
  ▶ Important Files
    testprog.txt
```

The project that you download has the parser.y, and scanner.lex files as already shown. It also has the files data.h, and data.c. These files contain code for a really naïve symbol table for variables used in compiled programs. This symbol table is just an unordered array of structs; each struct has a symbol name, and an int value field.

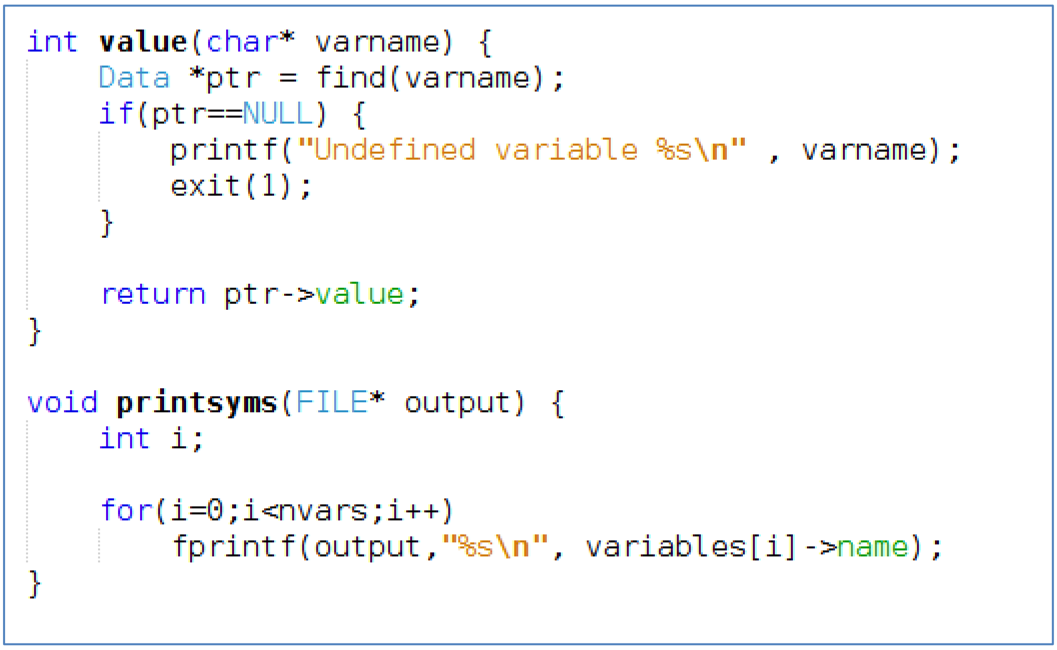
data.h

```
#ifndef DATA_H
#define DATA_H
#include <stdio.h>
struct data {
    char* name;
    int value;
};

typedef struct data Data;

Data *find(char* varname);

void insert(char* varname);

int value(char* varname);

void printsyms(FILE* output);

#endif  /* DATA_H */
```

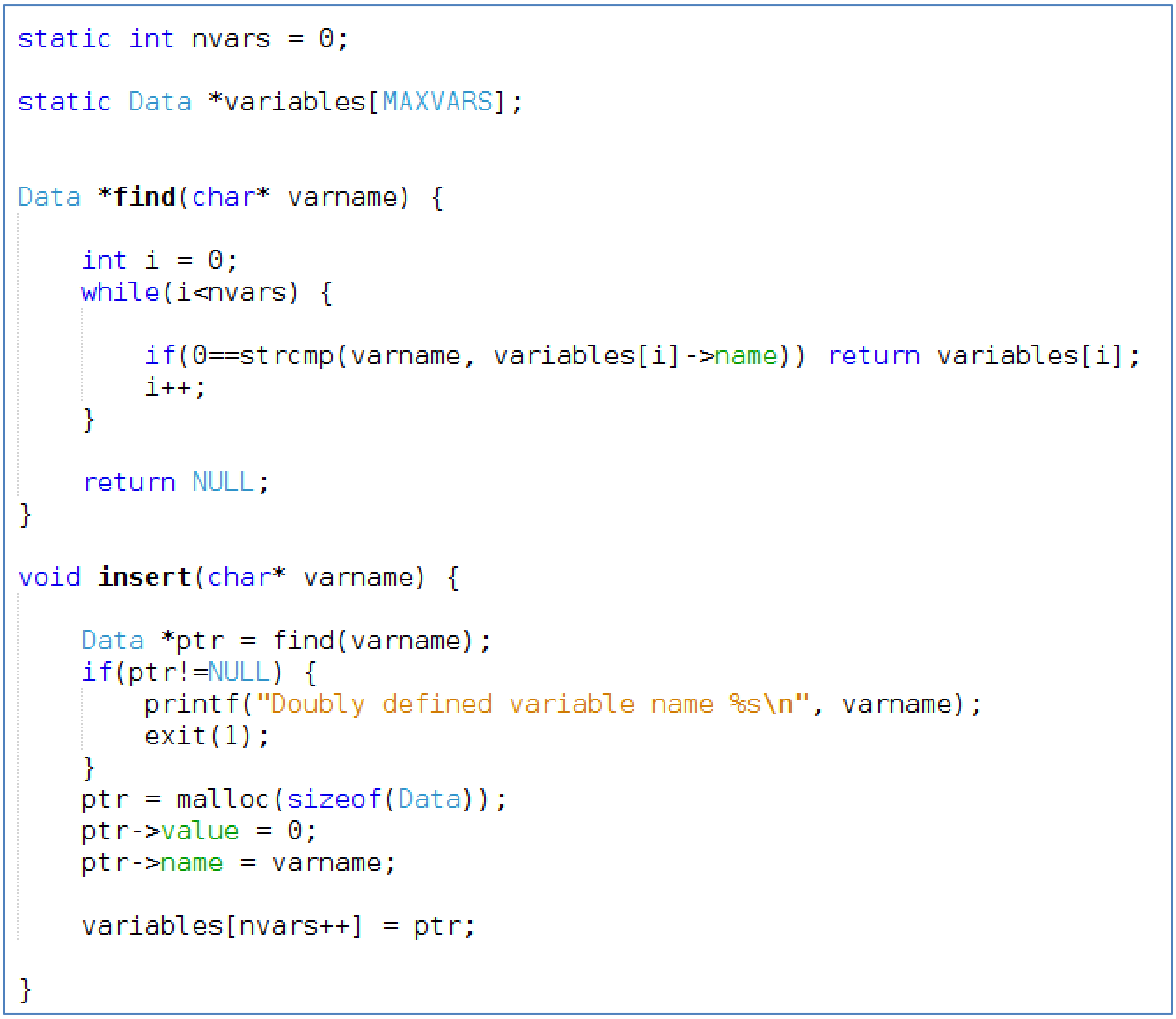
data.c



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "data.h"

#define MAXVARS 100

static int nvars = 0;

static Data *variables[MAXVARS];


Data *find(char* varname) {...11 lines }

void insert(char* varname) {...14 lines }

int value(char* varname) {...9 lines }

void printsyms(FILE* output) {...6 lines }
```



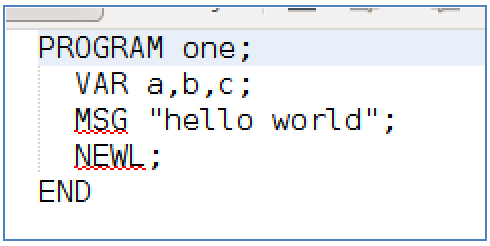
```
int value(char* varname) {
    Data *ptr = find(varname);
    if(ptr==NULL) {
        printf("Undefined variable %s\n" , varname);
        exit(1);
    }

    return ptr->value;
}

void printsyms(FILE* output) {
    int i;

    for(i=0;i<nvars;i++)
        fprintf(output,"%s\n", variables[i]->name);
}
```



```c
static int nvars = 0;

static Data *variables[MAXVARS];


Data *find(char* varname) {

    int i = 0;
    while(i<nvars) {

        if(0==strcmp(varname, variables[i]->name)) return variables[i];
        i++;
    }

    return NULL;
}

void insert(char* varname) {

    Data *ptr = find(varname);
    if(ptr!=NULL) {
        printf("Doubly defined variable name %s\n", varname);
        exit(1);
    }
    ptr = malloc(sizeof(Data));
    ptr->value = 0;
    ptr->name = varname;

    variables[nvars++] = ptr;

}
```

Build and run the prototype project

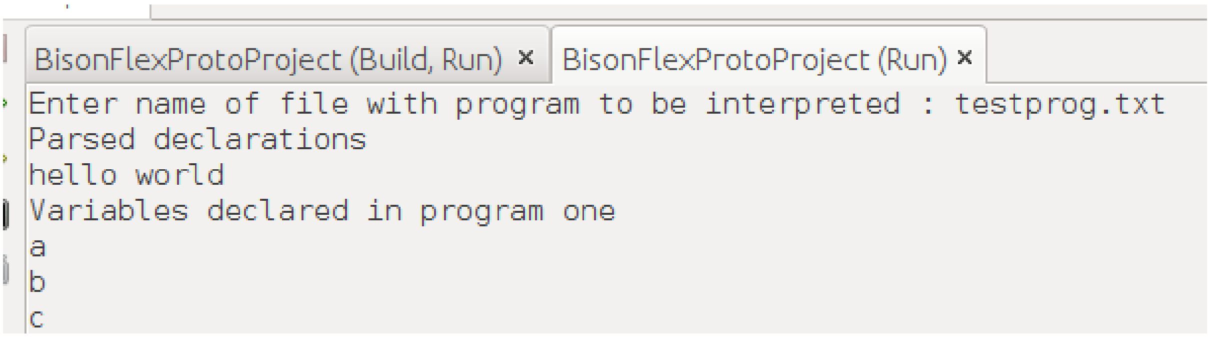
The project contains a file testprog.txt with the program text that is to be processed by the generated application.



```
PROGRAM one;
  VAR a,b,c;
  MSG "hello world";
  NEWL;
END
```

©nabg

Running the application on this file should produce the output:



BisonFlexProtoProject (Build, Run) ×
BisonFlexProtoProject (Run) ×

```
Enter name of file with program to be interpreted : testprog.txt
Parsed declarations
hello world
Variables declared in program one
a
b
c
```

The line "Parsed declarations" is printed once the parser has dealt with the input line "VAR a,b,c;". The MSG statement is executed by printing the string "hello world". The NEWL statement results in a newline. When the generated parser has dealt with the complete input file, it prints out the symbol table.

(It shouldn't have taken you long to do that part of the exercise, you just downloaded and ran some code. Spend some time on convincing yourself that flex and bison rules aren't really that hard to understand. They may be hard to write correctly, but once compsed they aren't usually hard to understand.)

The real exercise – parsing and interpreting logical expressions!

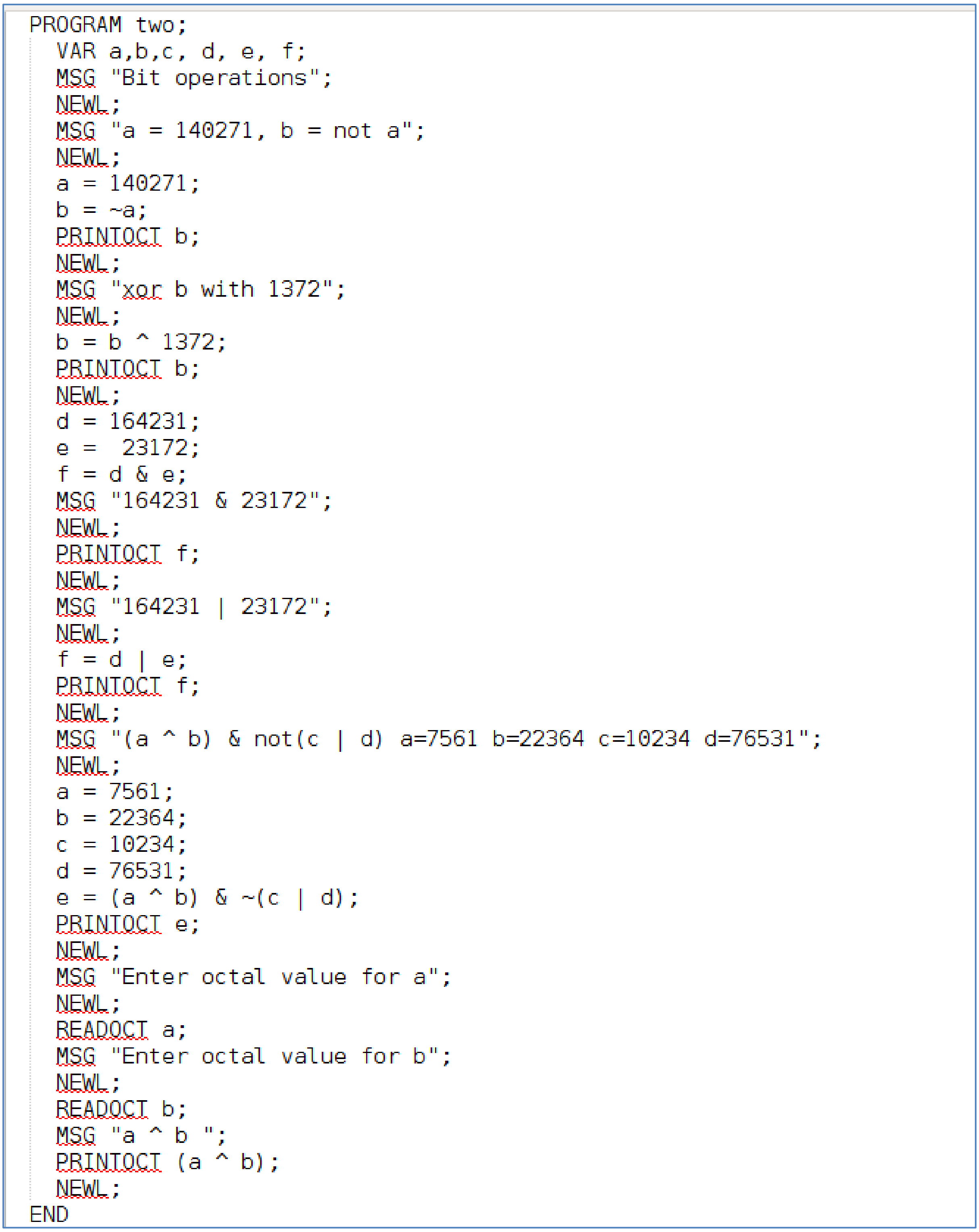
Duplicate the downloaded project creating a new E5b project.

This one is to interpret somewhat more elaborate programs that involve logical operations on values.

The trivial programming language of the last example is now extended a little. Variables declared in the VAR declaration can now be assigned values; expressions can combine values from variables and constants to generate new values; data values can be written and read.

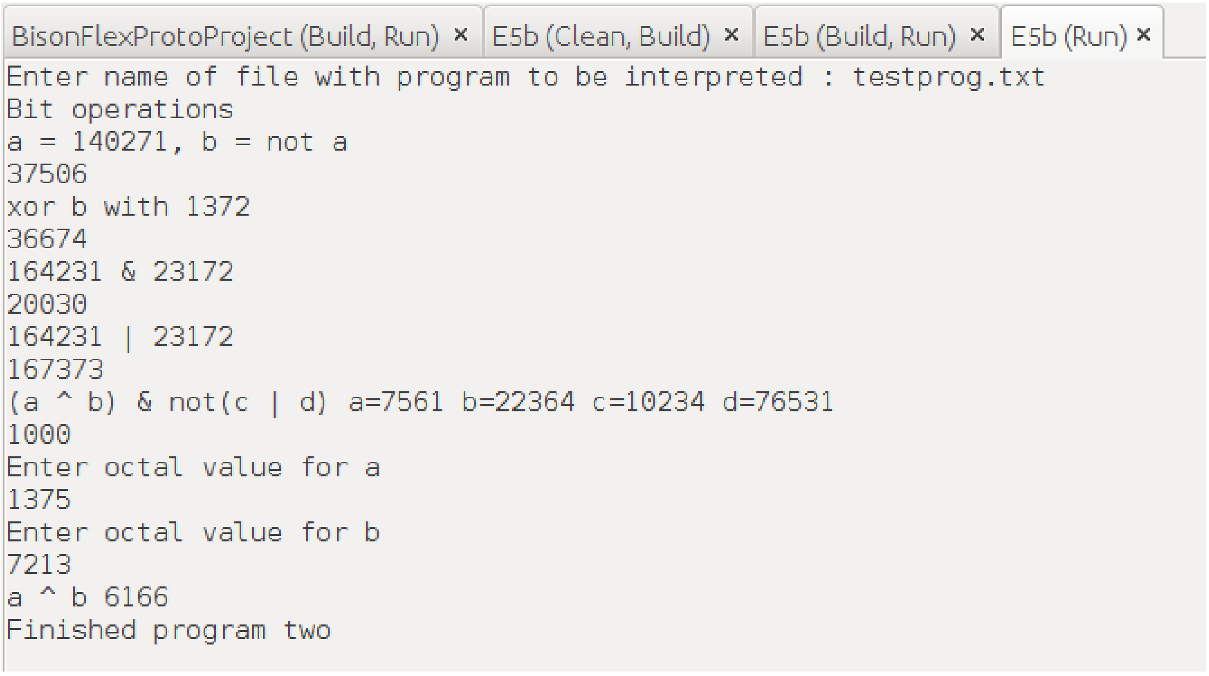
All data will be represented in octal (and the implementation code will limit values to 16-bits). Only logical operations can be used in expressions - &, |, ~ (not), and ^ (xor). (The octal data values are not displayed with a leading 0 character as is typically done in C.)

Example program



```
PROGRAM two;
  VAR a,b,c, d, e, f;
  MSG "Bit operations";
  NEWL;
  MSG "a = 140271, b = not a";
  NEWL;
  a = 140271;
  b = ~a;
  PRINTOCT b;
  NEWL;
  MSG "xor b with 1372";
  NEWL;
  b = b ^ 1372;
  PRINTOCT b;
  NEWL;
  d = 164231;
  e =  23172;
  f = d & e;
  MSG "164231 & 23172";
  NEWL;
  PRINTOCT f;
  NEWL;
  MSG "164231 | 23172";
  NEWL;
  f = d | e;
  PRINTOCT f;
  NEWL;
  MSG "(a ^ b) & not(c | d) a=7561 b=22364 c=10234 d=76531";
  NEWL;
  a = 7561;
  b = 22364;
  c = 10234;
  d = 76531;
  e = (a ^ b) & ~(c | d);
  PRINTOCT e;
  NEWL;
  MSG "Enter octal value for a";
  NEWL;
  READOCT a;
  MSG "Enter octal value for b";
  NEWL;
  READOCT b;
  MSG "a ^ b ";
  PRINTOCT (a ^ b);
  NEWL;
END
```

Output from execution

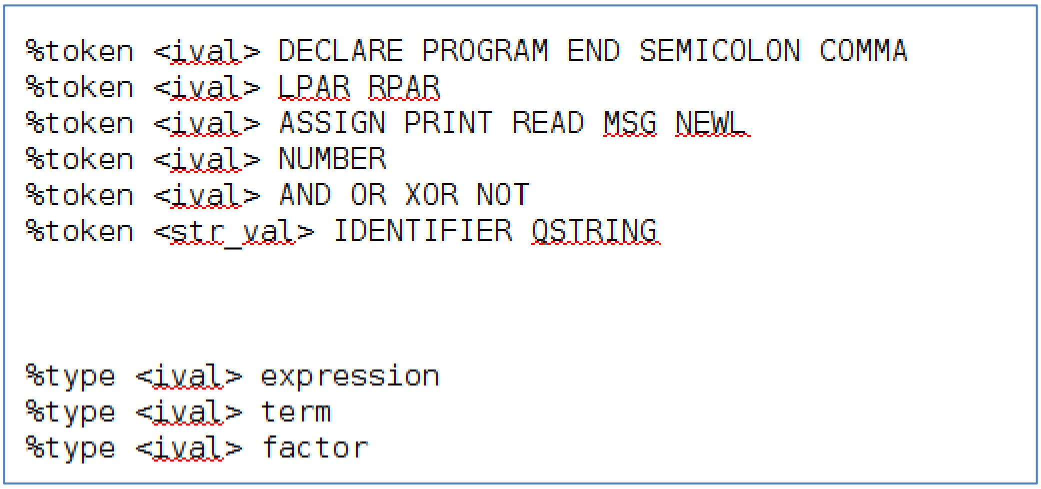


```
Enter name of file with program to be interpreted : testprog.txt
Bit operations
a = 140271, b = not a
37506
xor b with 1372
36674
164231 & 23172
20030
164231 | 23172
167373
(a ^ b) & not(c | d) a=7561 b=22364 c=10234 d=76531
1000
Enter octal value for a
1375
Enter octal value for b
7213
a ^ b 6166
Finished program two
```

A richer grammar

Some parts of the existing parser.y file can be used without change – the %{ … %} code include and the main(). But the language that is to be processed is now richer; there are more terminal symbols (tokens), and more grammar rules.

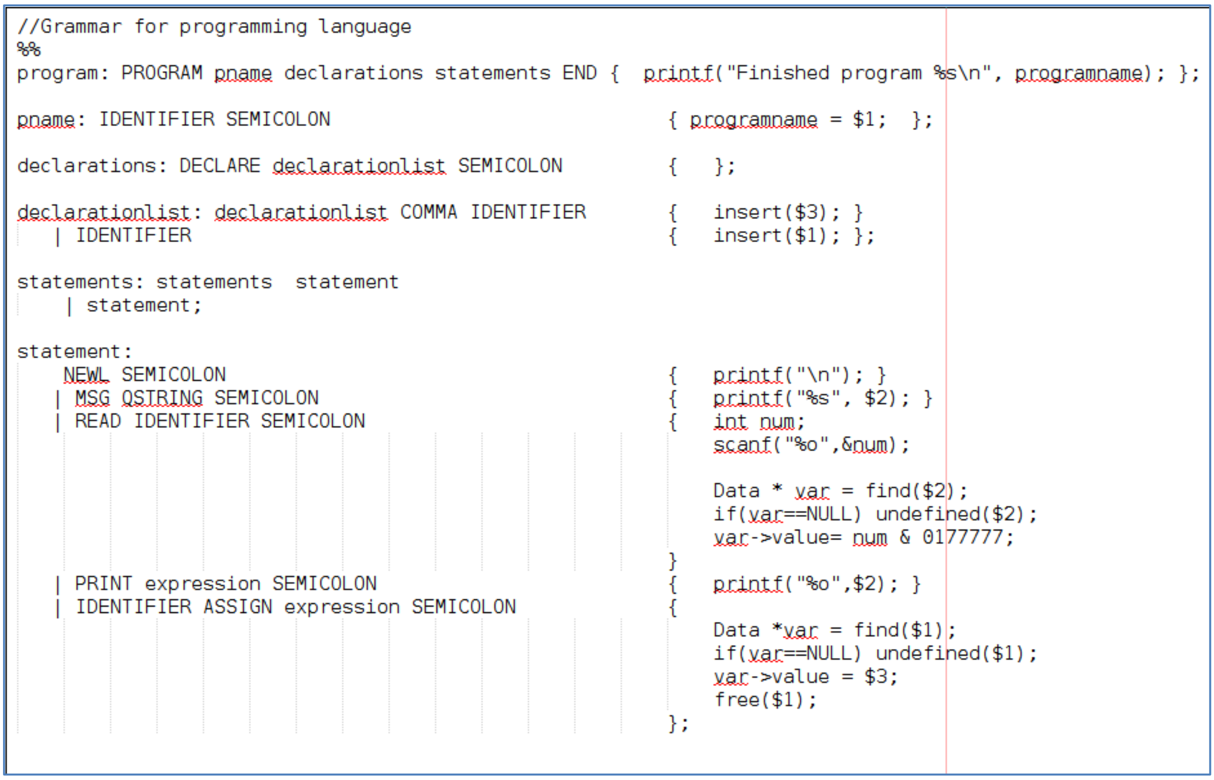
You will need to update the definition of the tokens in parser.y:



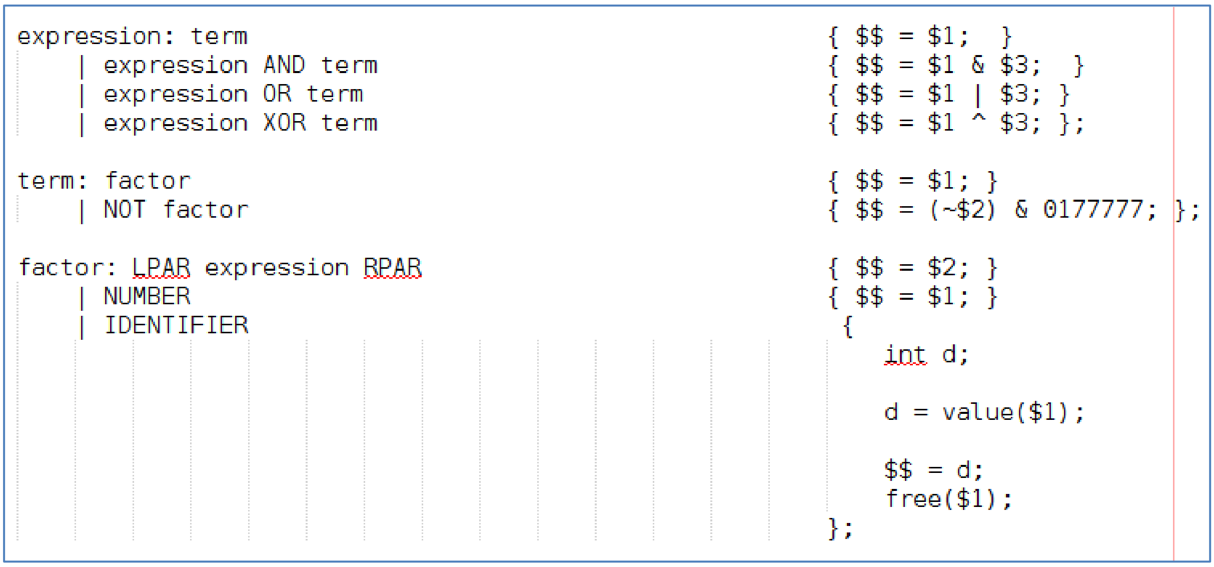
```
%token <ival> DECLARE PROGRAM END SEMICOLON COMMA
%token <ival> LPAR RPAR
%token <ival> ASSIGN PRINT READ MSG NEWL
%token <ival> NUMBER
%token <ival> AND OR XOR NOT
%token <str_val> IDENTIFIER QSTRING



%type <ival> expression
%type <ival> term
%type <ival> factor
```

Note how a new section has appeared with “expression”, “term”, and “factor”. These are non-terminals; constructs defined in grammar rules. But they will have values that are going to be manipulated by code for the generated program interpreter. For all of these, the value will be an integer type.

There are many more grammar rules:



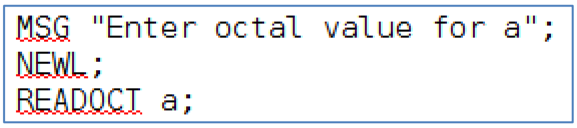
```
//Grammar for programming language
%%
program: PROGRAM pname declarations statements END {  printf("Finished program %s\n", programname); };

pname: IDENTIFIER SEMICOLON                            { programname = $1;  };

declarations: DECLARE declarationlist SEMICOLON        {   };

declarationlist: declarationlist COMMA IDENTIFIER      {   insert($3); }
     | IDENTIFIER                                      {   insert($1); };

statements: statements  statement
      | statement;

statement:
    NEWL SEMICOLON                                     {   printf("\n"); }
     | MSG QSTRING SEMICOLON                           {   printf("%s", $2); }
     | READ IDENTIFIER SEMICOLON                       {   int num;
                                                           scanf("%o",&num);

                                                           Data * var = find($2);
                                                           if(var==NULL) undefined($2);
                                                           var->value= num & 0177777;
                                                       }
     | PRINT expression SEMICOLON                      {   printf("%o",$2); }
     | IDENTIFIER ASSIGN expression SEMICOLON          {
                                                           Data *var = find($1);
                                                           if(var==NULL) undefined($1);
                                                           var->value = $3;
                                                           free($1);
                                                       };
```



```
expression: term                              { $$ = $1;  }
     | expression AND term                    { $$ = $1 & $3;  }
     | expression OR term                     { $$ = $1 | $3; }
     | expression XOR term                    { $$ = $1 ^ $3; };

term: factor                                  { $$ = $1; }
     | NOT factor                             { $$ = (~$2) & 0177777; };

factor: LPAR expression RPAR                  { $$ = $2; }
     | NUMBER                                 { $$ = $1; }
     | IDENTIFIER                              {
                                                 int d;

                                                 d = value($1);

                                                 $$ = d;
                                                 free($1);
                                              };
```

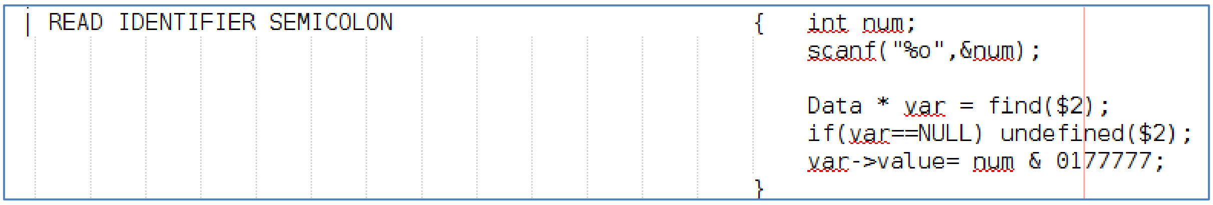
The rules for program, pname, declarations, declarationlist, statements, and statement are unchanged. But now there are a number of additional types of statement as well as the unchanged NEWL and MSG style statements previously encountered.

In this enhanced program language, we can have “READ” statements:



```
MSG "Enter octal value for a";
NEWL;
READOCT a;
```

The grammar rule handling this is:



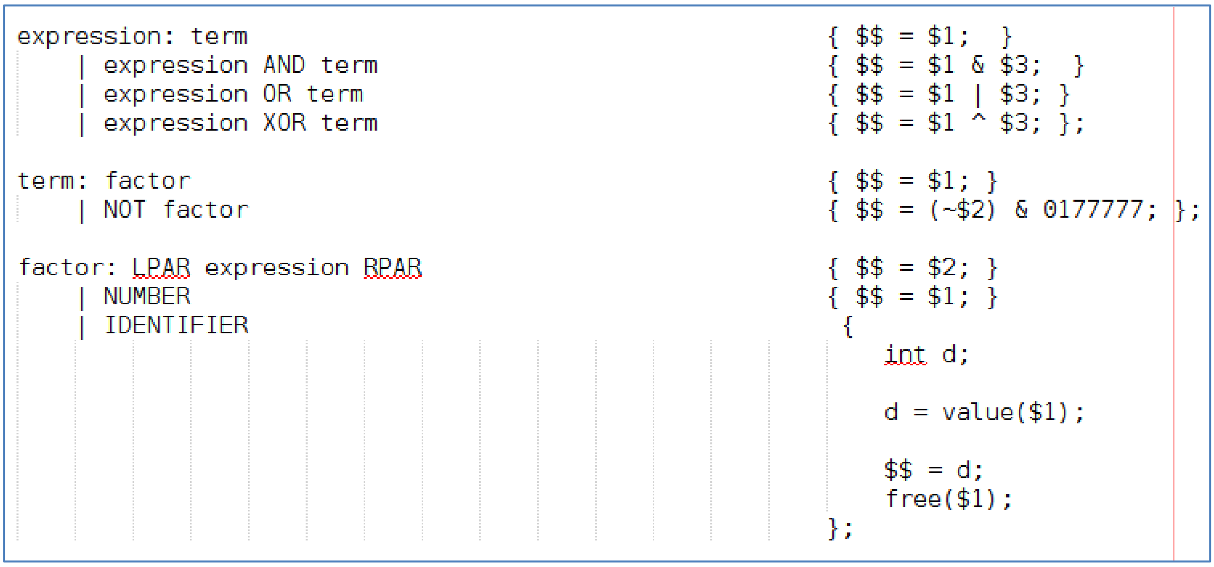
```
| READ IDENTIFIER SEMICOLON                          {   int num;
                                                         scanf("%o",&num);

                                                         Data * var = find($2);
                                                         if(var==NULL) undefined($2);
                                                         var->value= num & 0177777;
                                                     }
```

The statement has a "READ" token (literal form handled by scanner is READOCT), a variable name (returned from scanner as an IDENTIFIER token, with the struct for $2 containing the actual string name), and a SEMICOLON token. The example source string READOCT `a;` has READOCT (the READ token), 'a' (an IDENTIFIER with value "a"), and ';' (the SEMICOLON token).

The generated interpretive code is to read a value (using "%o" octal format), find the struct that would have been created for the variable 'a' (if the source statement was READOCT `z;` the find() operation would fail as 'z' hasn't been declared and so the program would terminate after calling undefined()). The value read is assigned to the value field of the struct in the symbol table. (*There is probably a memory leak around here, can you spot it?*)

There are also PRINT and ASSIGN statements. You should try to sort out the code for these statements by yourself.

The "expression", "term", and "factor" group define how to parse and interpret expressions involving Boolean operators.



```
expression: term                                      { $$ = $1;  }
    | expression AND term                             { $$ = $1 & $3;  }
    | expression OR term                              { $$ = $1 | $3; }
    | expression XOR term                             { $$ = $1 ^ $3; };

term: factor                                          { $$ = $1; }
    | NOT factor                                      { $$ = (~$2) & 0177777; };

factor: LPAR expression RPAR                          { $$ = $2; }
    | NUMBER                                          { $$ = $1; }
    | IDENTIFIER                                       {
                                                          int d;

                                                          d = value($1);

                                                          $$ = d;
                                                          free($1);
                                                      };
```

In these rules, $$ represents the value of the result, while $1, $2, and $3 will represent values for components matched in the grammar rule.

Consider and input such as PRINTOCT `(a ^ b);`

The scanner would return this as the sequence of tokens:

- PRINT

- LPAR
- IDENTIFIER (value= ‘a’)
- XOR
- IDENTIFIER (value= ‘b’)
- RPAR
- SEMICOLON

The parser would handle this sequence starting with its rule PRINT expression SEMICOLON.

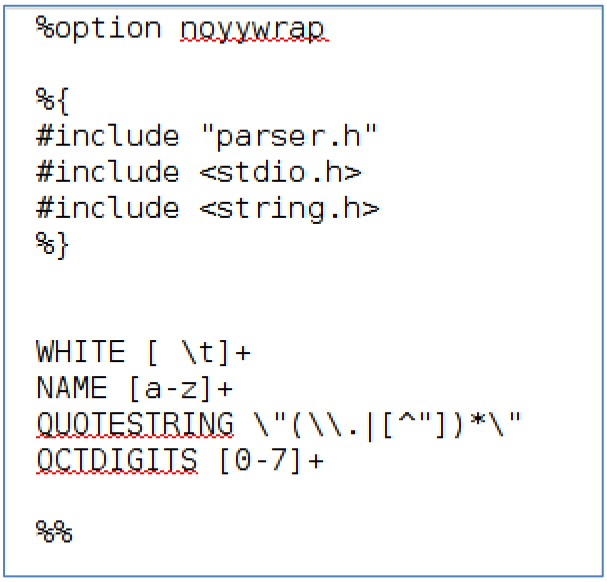
It’s an LALR parser. So it would shift and reduce symbols on its stack, recognizing the parenthesised sub-expression, then the term combining two factors which are both identifiers. It gets identifier values from the symbol table entries. It combines values in terms by using the appropriate &, |, or ^ operator.

Read the code a couple of times, it should all become clear.

A more elaborate scanner

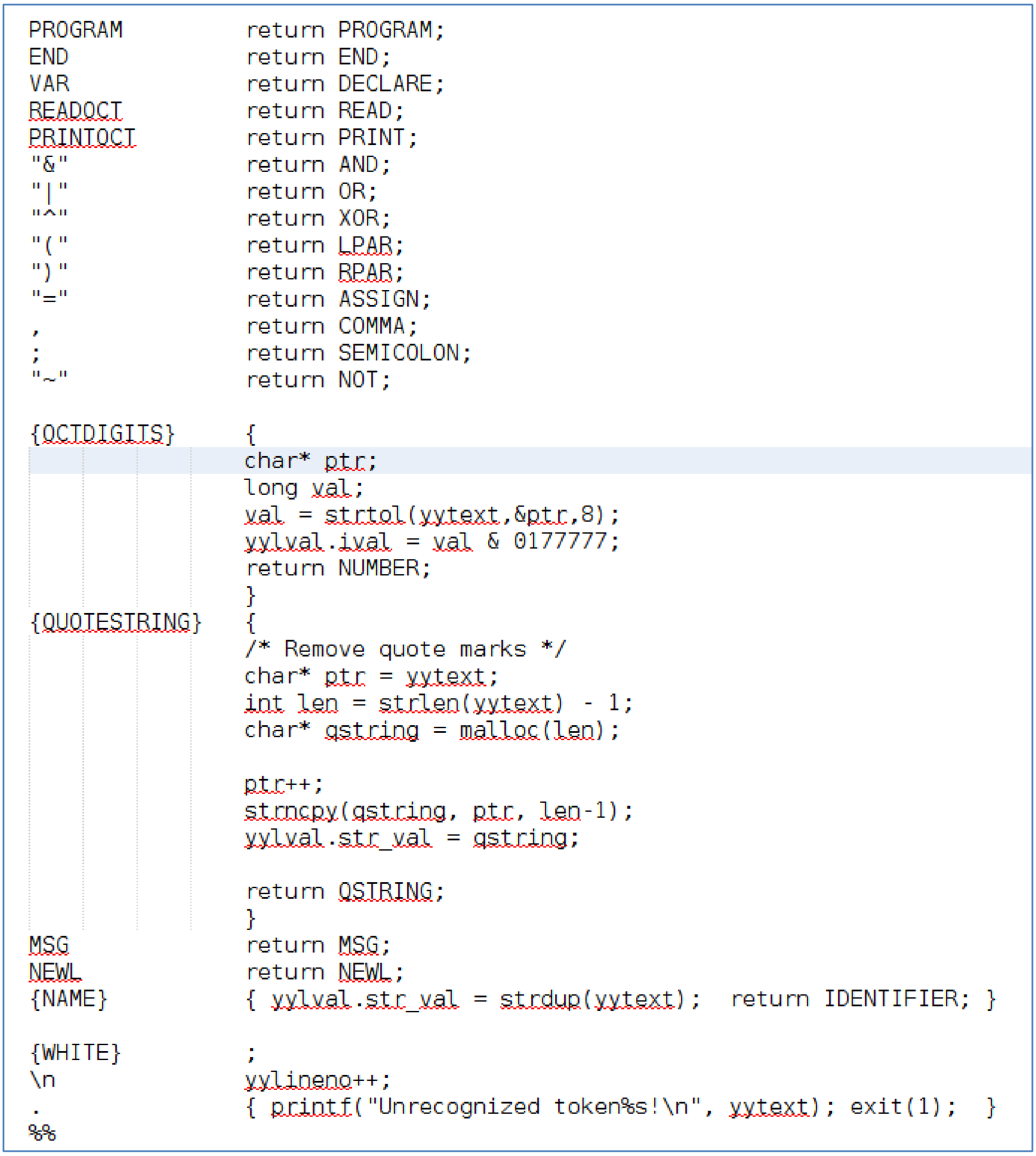
The parser.lex file needs to define additional patterns that the scanner is to recognize in the input. It’s also necessary to define what an octal number will look like.

The definitions section of parser.lex becomes:



```
%option noyywrap

%{
#include "parser.h"
#include <stdio.h>
#include <string.h>
%}


WHITE [ \t]+
NAME [a-z]+
QUOTESTRING \"(\\.|[^"])*\"
OCTDIGITS [0-7]+

%%
```

The pattern-action rules are:

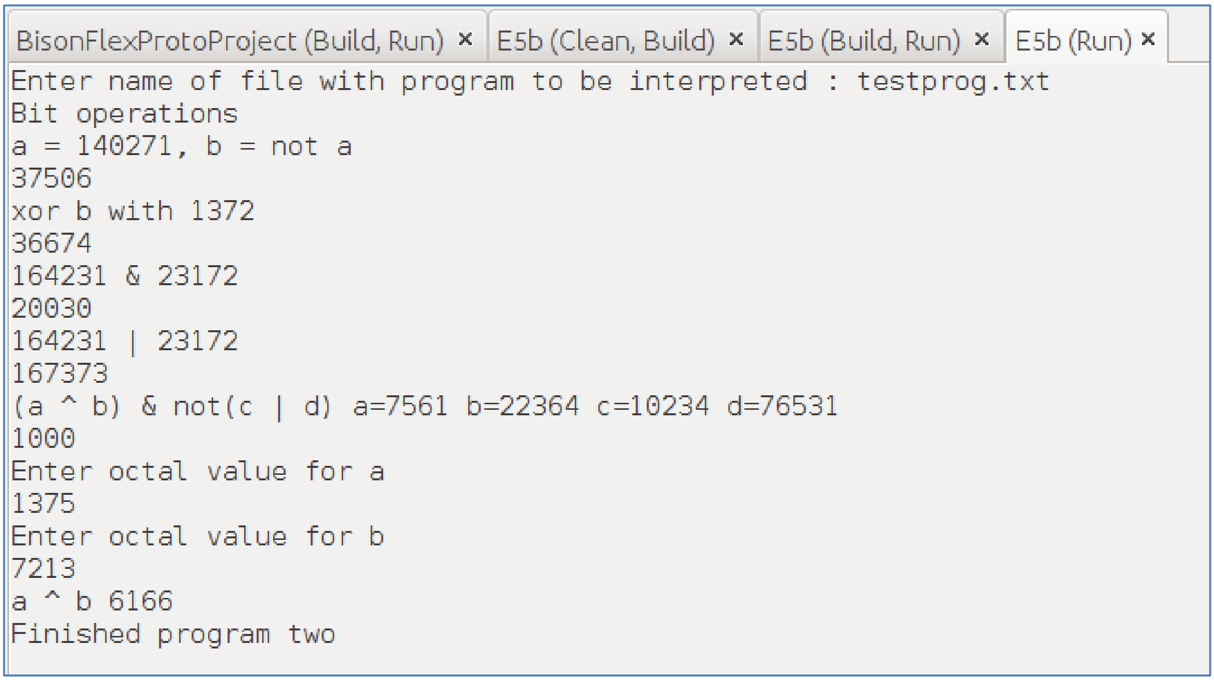


```
PROGRAM         return PROGRAM;
END             return END;
VAR             return DECLARE;
READOCT         return READ;
PRINTOCT        return PRINT;
"&"             return AND;
"|"             return OR;
"^"             return XOR;
"("             return LPAR;
")"             return RPAR;
"="             return ASSIGN;
,               return COMMA;
;               return SEMICOLON;
"~"             return NOT;

{OCTDIGITS}     {
                char* ptr;
                long val;
                val = strtol(yytext,&ptr,8);
                yylval.ival = val & 0177777;
                return NUMBER;
                }
{QUOTESTRING}   {
                /* Remove quote marks */
                char* ptr = yytext;
                int len = strlen(yytext) - 1;
                char* qstring = malloc(len);

                ptr++;
                strncpy(qstring, ptr, len-1);
                yylval.str_val = qstring;

                return QSTRING;
                }
MSG             return MSG;
NEWL            return NEWL;
{NAME}          { yylval.str_val = strdup(yytext);  return IDENTIFIER; }

{WHITE}         ;
\n              yylineno++;
.               { printf("Unrecognized token%s!\n", yytext); exit(1);  }
%%
```

There are many more simple tokens that must be recognized – READOCT, &, =, etc. A rule is needed to handle (octal) numeric constants, their value must be determined and they get returned as NUMBER tokens. The other rules are unchanged.

Edit, compile, and run

Update both parser.y and scanner.lex along the lines shown. Build the project. It should run.



```
Enter name of file with program to be interpreted : testprog.txt
Bit operations
a = 140271, b = not a
37506
xor b with 1372
36674
164231 & 23172
20030
164231 | 23172
167373
(a ^ b) & not(c | d) a=7561 b=22364 c=10234 d=76531
1000
Enter octal value for a
1375
Enter octal value for b
7213
a ^ b 6166
Finished program two
```

Exercises complete

Show the tutor that you have completed all the parts.

(That was 8 somewhat easy marks wasn't it – just copy my code. Hopefully, you learnt a little about parsers and lexical scanners.)

Assignment

The assignment: implement some part of a compiler.