

Project 1 – Lexical Analyzer using the Lex Unix Tool

No due date – project not graded

Description: In this project you will be asked to develop a scanner for a programming language called mini-C. The project (and the compiler) consists of three steps: lexical analysis, parsing and code generation will be developed throughout the semester. These three parts shall be developed in order and they have different deadlines. The compiler should be written in C using the lex and yacc Unix utility tools. ***Due to its simplicity this first project will not be graded. It's objective is to provide you with some basic understanding of simple lex specifications which will be used as part of the second project.***

The project as well as its follow-on parts shall be developed individually. You are encouraged to discuss with other students, ask teachers, search for ideas on the Internet, etc. But, ***do not copy code.*** The solutions will be tested in a UNIX environment. The compiler can be developed in many environments. But, when the solutions are delivered, it must be possible to generate the lexical analyzer and the parser from their source codes, and compile the results, etc., in UNIX. See the instruction on how to turn-in your projects. The evaluations of the solutions are based on a series of automated tests.

The language mini-C is a small C-like imperative language. A program in mini-C consists of a sequence of function definitions. Each function consists in turn of variable declarations, type declarations, function declarations, and statements. As in C, a statement is an expression (usually with) side-effects.

The type system in mini-C is very restricted. It only has an integer type (int), character type (char) record types (struct), single dimensional arrays and pointer types possibly using user defined struct types as in C. The array index value can only be simple unary expression such as an identifier, a constant or another simple array access expression. More complex indices need to be made using additional statements and temporary variables. The language also contains some arithmetic and logic operators (+, -, *, /, &&, ||, ~, .) as well as relational operators (==, <, >, !=, <=, >=). Access to structure types can be done using the “pointer dereferencing” operator (“->”) or the field access operator “.”.

In order to have a fairly simple grammar, expressions in mini-C are restricted. You may develop the compiler in such a way that expressions can only contain one operation. This means, for instance, that, if we would like to write:

```
res=x+2*f(y,z);
x = A[i+1];
```

then we must use variables to store intermediate results, as follows

```
tmp1=f(y,z);
tmp2=2*tmp1;
res=x+tmp2;
tmp3=i+1;
x = A[tmp3];
```

Comments in mini-C are written between /* and */.

More details are given in the grammar will be given in the second project. Beware that the lexer must recognize a positive integer numbers the ‘+’ prefix. Negative integer numbers, may have a single ‘-’ signal prefix. As such you need to be aware of a sequence such as “-1” and report it as a integer number rather than a sequence of two tokens, a minus operator followed by a positive integer number. Conversely, you may

assume that statements of the form “ $i = i + 1;$ ” will always have a space between the operands of the plus arithmetic operator as this simplifies the scanner.

Lexical analysis. The lexer shall be developed using the lexer generator lex or flex.

The lexer shall recognize the following keywords: else, int, void, if, else, while, return. For each one of them the lexer shall return the tokens INT, CHAR, VOID, IF, ELSE, WHILE, RETURN respectively. The lexer shall also recognize identifiers and integer numbers. An identifier is a sequence of letters and digits, starting with a letter. The underscore ‘_’ counts as a letter. An integer number is a sequence of digits, possibly starting with a + or -. For each identifier, the lexer shall return the token IDENTIFIER, and for each integer number, it shall return the token CONSTANT.

The lexer shall recognize the operators ‘->’, ‘&&’, ‘||’, ‘.’, for which it shall return the tokens PTR_OP, AND_OP, OR_OP, and DOT_OP respectively; the relation ‘==’, for which it shall return the token EQ_OP, and the other relational operators (‘!=’ for NE_OP, ‘<=’ for LE_OP, ‘>=’ for GE_OP, ‘>’ for LT_OP and ‘<’ for LT_OP); and the following list of operators and separators ‘;’, ‘{’, ‘}’, ‘,’, ‘=’, ‘(’, ‘)’, ‘&’, ‘~’, ‘-’, ‘+’, ‘*’, ‘/’, ‘[’ and ‘]’ for which it shall return the same character as token (token ‘;’ for the separator ‘;’, and so on).

Include File Command

In order to facilitate the inclusion of multiple files, your lexer is also responsible for directly handling the *include file* command. When encountering the *include* directive placing at the first column of a given line, the lexer shall open the file indicated by the file name in the directive and start processing its contents. Once the included file has been processed the lexer must return to processing the original file. An included file may also include another file and so forth. If the file names does not exist in the local directory you should simply ignore the include command and proceed with the tokens in the current file.

The syntax of the include command is as in C where the file name can be placed within the characters “ or between “<” and “>” (**Note:** you need to separate this characters in the context of the include file with the one for the relational operators). The filename in this directive may have a single file name extension as in “.txt”. For simplicity there are no path separators as in “/” so that all include files will be searched in the local working directory. You need to look on the **www** for the references to the yy_buffer_state to effectively support nested include files. This is by far the trickiest part of your project so you need to take advantage of all the resources available to you.

Comments in mini-C

Your scanner shall ignore all comments and white space. For all characters that are not specified above, simply use the default behavior of lex, which is to echo the character.

Tokens and Return Values

The tokens are defined as integer numbers and it is important for the test cases that the token are as defined below as it will be provided to you in a file called “y.tab.h”.

```
/* Tokens. */
#define IDENTIFIER 258
#define NUMBER 259
#define SIZEOF 260
#define PTR_OP 261
#define GT_OP 262
#define LT_OP 263
#define GE_OP 264

#define LE_OP 265
#define NE_OP 266
#define EQ_OP 267
#define AND_OP 268
#define OR_OP 269
#define DOT_OP 270
#define TYPEDDEF 271
#define INT 272

#define CHAR 273
#define VOID 274
#define STRUCT 275
#define IF 276
#define ELSE 277
#define WHILE 278
#define RETURN 279
```

Output of your Project

Normally, in a compiler, the parser repeatedly calls the lexer, which returns the next token to the parser. But, now you will test the lexer separately. Let the source code to the lexer be in a file called lex1.l. Add the following code to lex1.l:

```
int main() {
    int tok;
    int n;
    n = 0;
    while(1) {
        tok = yylex();
        if(tok == 0)
            break;
        printf("line:%2d token type:%3d token text:(%s)\n", yylineno, tok, yytext);
        n++;
    }
    printf("Number of tokens matched is %d\n",n);
    return 0;
}
```

The output of your scanner should contain the following elements separated by a single space:

```
line token string
```

as indicate `line` in the snippet of code above and where `line` indicates as an integer the line number where the token was detected, `token` is the integer denoting the token as described above and finally the string element presents the matched string itself. Finally as the last line of your output you should have the line stating the number of matched tokens as:

```
Number of Tokens: number
```

where `number` indicates an integer value.

When processing a new file via the include file command you should start the line numbering at zero and recall where you have left off in the original file.

How to Generate a Scanner Executable: Generate the lexer using flex:

```
flex lex1.l
```

The result is in a file with name `lex.yy.c`. Compile and link:

```
gcc -c lex.yy.c
gcc lex.yy.o -ll
```

Some test cases with the correct results can be found at the class website. Test your work against those provided sample inputs and output to make sure that your lexer works.

Turn-in Instructions: You need to mail in your work to the TA as a zip file named `proj1.tar.zip` file. Make while *untaring* it creates a folder with you student ID and that we can create an executable by invoking the command “make”. Our make shall create an `a.out` executable, which we use to automatically test your project using the command line:

```
./a.out < test1.in
```

for all `test1.in` thought `test10.in` files.

Sample Input and Output File: Below is a sample input file (left) and the corresponding output file (right):

```

/* comment */
void main(){
  int i;
  i=0;
  while(i<10)
    i=i+1;
}

line: 2 token type:275 token text:(void)
line: 2 token type:258 token text:(main)
line: 2 token type: 40 token text:(()
line: 2 token type: 41 token text:())
line: 2 token type:123 token text:({)
line: 3 token type:273 token text:(int)
line: 3 token type:258 token text:(i)
line: 3 token type: 59 token text:(;)
line: 4 token type:258 token text:(i)
line: 4 token type: 61 token text:(=)
line: 4 token type:259 token text:(0)
line: 4 token type: 59 token text:(;)
line: 5 token type:279 token text:(while)
line: 5 token type: 40 token text:(()
line: 5 token type:258 token text:(i)
line: 5 token type:264 token text:(<)
line: 5 token type:259 token text:(10)
line: 5 token type: 41 token text:())
line: 6 token type:258 token text:(i)
line: 6 token type: 61 token text:(=)
line: 6 token type:258 token text:(i)
line: 6 token type: 43 token text:(+)
line: 6 token type:259 token text:(1)
line: 6 token type: 59 token text:(;)
line: 7 token type:125 token text:({)
Number of tokens matched is 25
```