Project 1 – Lexical Analyzer using the Lex Unix Tool

Due date: April 2, 2009 at midnight

Description: In this project you will be asked to develop a scanner for a programming language miniC. 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 flex and yacc Unix utility tools.

The project 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 the Linux-i386 environment. The evaluations of the solutions are based on these tests. The compiler can be devolved 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 Linux-i386. See the instruction on how to turn-in your projects.

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

The type system in miniC is very restricted. It only has an integer type (int), character type (char) record types (struct), unions (unions) and pointer types (void, *). The statements are assignments (=), if-statements, while-loops, return and function calls. The language also contains some arithmetic and logic operators (+, -, *, /, &&, ||, ~) as well as relational operators (==, <, >, !=, <=, >=). Lastly, there are struct an union field access operators, such as "->" and ".".

As you are dealing with pointer and can manipulate addresses directly, you also have the "address-off" operator "&" and have access to a special built-in function "offset(struct,field)". This offset function, which you will learn later on in the class, determines the integer offset of a given field name in a specific structure type. The example below shows a simple translation of an allocation of a structure of type "struct A".

```
res= (char*)malloc(sizeof(struct A);
res->p = 0;
```

```
tmp1=sizeof(struct A);
tmp2 = malloc(tmp1);
res = tmp2;
tmp3 = &res+offset(A,p);
*tmp3 = 0
```

Comments in miniC 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.

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

The lexer shall recognize the following keywords: else, typedef, int, void, struct, if, else, while, return and sizeof. For each one of them the lexer shall return the tokens TYPEDEF, INT, CHAR, VOID, STRUCT, IF, ELSE, WHILE, RETURN and SIZEOF, 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 single + 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, 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 miniC
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".

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();
    <put your code to emit the output here>
    n++;
  }
  printf("#Total Number of Tokens Matched was: %d\n",n);
  return 0;
}
```

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

```
line token string
```

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:

```
#Total 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 the lexer and make sure that it works.

Turn-in Instructions: In you Unix account on the tlinux machine, create a folder in your tlinux machine accound named xfer. In this account create another folder named project1. Inside this project1 place a file named proj1.tar.zip created using the zip and tar utilities.

Make sure you can unzip and untar the file and that a makefile you have put inside your proj1.tar.zip file works properly, *i.e.,* we will make you're a.out executable using the "make" command and invoke the many test 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;
}
```

```
2 272 void
2 257 main
2 40 (
2 41 )
2 123 {
3 270 int
3 257 i
3 59 ;
4 257 i
4 61 =
4 258 0
4 59 ;
5 276 while
5 40 (
5 257 i
5 262 <
5 258 10
5 41 )
6 257 i
6 61 =
6 257 i
6 43 +
6 258 1
6 59 ;
7 125 }#Total Number of Tokens Matched was
25
```