CSE4100: Compilers **Fall 2015**

Homework 1: Lexical Analysis

Out Date:	09/15/2015
Due Date:	09/22/2015

Objectives

The purpose of this homework is to review your skills at defining regular expressions and to get you started on the compiler project of the semester. At the time being, we only use a *fake* parser to call upon the scanner. A subsequent homework will focus on the parser creation.

As before, hand-in via moodle a single zip archive with a suitably named folder (follow the same convention as for the previous homework). You will be using a Makefile and the GNU suite compilation tools. The addition is `flex`, the GNU scanner generator. The flex documentation is, of course, on line: http://flex.sourceforge.net/manual/. Both `flex` and its companion `bison` are already installed in your docker container.

1 Handout

To ease your task, there is a small handout that you should download from moodle. It contains a `Makefile` and the template for the main program. Naturally, you are expected to define the lexer and the actual driver that fakes the parser and repeatedly invokes the scanner. You need to write a little bit of C++ code to *print* the tokens that you are getting from the scanner to make sure everything works as expected.

2 Regular Expressions and Automata

Your first task is to write the regular expression that recognize any lexeme that belongs to a token class of `C--`. `C--` has the following tokens (set of strings defined by a regular language). Each entry in the list below gives the token name and a description of the set of strings (lexemes) that this token represents.

SemiColon ';' is used to separate statements.

Comma ',' is used to separate arguments in method calls and formal argument lists.

Binary operators `C--` uses the following set `+,-,*,/,<=,>=,==,!=,<,>,&&,||,=`. **Make sure that you define one token per operator** (rather than lumping all the operators in one class!)

Unary operators `C--` uses the following set `!`. Make sure that you define one token per operator (rather than lumping all the operators in one class!)

Brackets `C--` uses square brackets `[ ]`, parenthesis `( )`, curly braces { }. This defines 6 distinct tokens!

BrackPair a pairs of brackets `[]` (i.e., two brackets next to each other without any intervening blanks – they are used for array declarations –) defines another token class.

Dots `C--` uses a single dot '.' for method invocation.

Keywords `C--` uses the following keywords `while, if,else, this, class, extends, new return,int, bool, void`. once again, define one token per member of this set. You should have 11 distinct tokens.

Booleans `C--` has two Booleans written `true` and `false` (Hence two tokens)

Integers `C--` represent integers in decimal. An integer is any sequence of digit that either does not start with zero, or, if it starts with zero, it contains only 1 digit (the zero itself). For instance, `123,45,0` are valid integers but `0124` is not. Note that the lexeme of the token should contain the actual integer value.

Identifiers The `C--` identifiers start with a letter in the roman alphabet or, possibly, an underscore. The remainder of an identifier can contain any number of alpha-numeric symbol including an underscore. For instance `hello,id3,i,k_25` are all valid but `hello-you, home%5` are not. Note that the lexeme of the token should contain the actual string.

Whitespaces White spaces are used for clarity and carry no meaning at all. White spaces include any number of blank, tabulation, carriage return or line feed.

Comments `C--` support `C++` one-line style comments. A comment starts with `//` and ends with a line-feed (`\n`) with anything in between.

For each token, you are expected to write a regular expression that recognizes all lexemes that belong to that language. Write your regular expressions directly in `scanner.lex`. Two new rules are particularly important in the Makefile. Namely:

```
grammar.c grammar.h: grammar.y
    $(BISON) --debug -d $< -o grammar.c
scanner.c: scanner.lex grammar.h
    $(LEX) --bison-bridge -o $@ $<
```

Beware of the PDF. The dashes are ASCII dashes!. The first rule invokes the `flex` scanner generator to produce a c file (`scanner.c`). Note that it uses the option `--bison-bridge` to create a bison compatible file[1]. The second creates a *fake* trivial parser from a trivial (almost empty) parser generation specification. This is done with `bison`. As you can see, the specification of that grammar is in a file named `grammar.y`.

2.1 Flex

To make your life easier, please, consult the following flex example (also at http://pastie.org/1541339) file for a language based on a few very simple tokens (this is not `C--`, merely an example language!), namely:

(,) parenthesis

+,-,*,/ the four arithmetic operators

if,then,else,def,let,fun a few keywords

NUMBER floating point numbers

ID simple identifiers.

The syntax of a flex file shown in Figure 1 is fairly straightforward. It is based on several sections (surrounded by %{ and %}) to define the standard inclusions, basic regular expressions (namely, `DIGIT`,`ID` and `NUMBER` in the above example) and a final section where the tokens are defined. Note how each token definition gives a regular expression followed by a block of `C` code that is executed when hitting the final state of the DFA corresponding to that token.

The lexeme itself is stored in an array `yytext` that you can easily retrieve and store (for instance, the `NUMBER` regular expression is associated with a piece of `C` code that retrieves the lexeme – `yytext` – converts it to a floating point value and stores the result in the `val` field of the structure pointed to by `yylval`). Finally, note that some characters are *reserved* for describing regular expressions. If you wish to recognize such a character, it should be `escaped` within the regular expression (preceded by a backslash as in `\+`.

Observe also that `yytext` is a data-structure of the parser that is destructively updated as the scanning proceeds, so it is important to **copy its content** if you wish to preserve it inside a lexeme.

[1] Bison is the tool we will use for creating a parser

```
%{
   #include <iostream>
   #include <iomanip>
   #include "parser.H"
   #include "grammar.h"
%}
DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9]*
NUMBER {DIGIT}+("."{DIGIT}*)?

%%
{NUMBER} {
   yylval->val = atof(yytext);
   return NUMBER;
}

\(                  { return LRB;}
\)                  { return RRB;}
\+                  { return TADD;}
\-                  { return TSUB;}
\*                  { return TTIMES;}
\/                  { return TDIV;}
if                  { return TIF;}
then                { return TTHEN;}
else                { return TELSE;}
def                 { return TDEF;}
let                 { return TLET;}
fun                 { return TFUN;}

{ID} {
   yylval->id = strdup(yytext); // must copy the string. Can't use the constant.
   return TID;
}

[ \t\n]*   /* ignore ws */;
```

Figure 1: Flex file for a simple expression language.

2.2 Bison

The bison skeleton (Figure 2) is quite minimal. It's sole purpose is to provide the definition of the various token names that you are to encounter. While bison is not our topic quite yet, you can find its documentation here http://www.gnu.org/software/bison/manual/. **You will have to edit this file as well to list all the tokens that your scanner will support**. The changes are minimal of course and would appear in the third section of the file (with the other `%token` directives). The first section defines, once again, the inclusions. The second defines with a `%union` statement the values that can be embedded inside a token. Note the name of the fields in that union, namely `val` and `id` which you would refer to from the lexer specification. The second section continues with a block of `C` code defining a few standard function prototypes, namely `yyerror`, `yywrap` and `yylex`. The third section ends with two bison requirements to specify a pure reentrant parser (no global variables allowed) and the fact that the parser receives an argument whose type is `Parser&` and name is `parser`. This is a class that you provide and which contains the fake parser implementation which repeatedly gets tokens from the scanner and prints them out.

```
%{
#include <math.h>
#include <stdlib.h>
#include "parser.H"
#define YYERROR_VERBOSE
%}

%union {
   int         val;
   char*       id;
}

%code{
   int yyerror(Parser* p,const char* s);
   int yylex(YYSTYPE*);
}

%pure-parser
%parse-param { Parser* parser }
/*************************************************
  This is where your additions should be made
 *************************************************/
%token <val> NUMBER TRUE FALSE
%token <id> TID
%token TIF TTHEN
%%
/**** No edits beyond this point                 ****/
Top:
;
%%
```

Figure 2: Bison specification example.

3 Fake Parser

Once your scanner.lex is ready, you need to implement a three line program that simply calls the scanner to get all the tokens. This program is trivial and can be placed in the `run` method of the `Parser` class in `Parser.C`. The `main` entry point then reduces to:

```
#include <iostream>
#include <iomanip>
#include "parser.H"

using namespace std;
int main(int argc,char* argv[]) {
   Parser p;
   p.run();
}
```

4 Testing

Executing your code on the following sample input

```
125 45678 class x foo int = { } ( ) [ ] + - * / < > <=>=!===;,.if else
extends bar while return this [] = // comment to be discarded
&&||!bool int void new
```

should produce the output

```
NUMBER(125)
NUMBER(45678)
CLASS
ID(x)
ID(foo)
INT
=
{
}
(
)
[
]
+
-
*
/
<
>
<=
>=
!=
==
;
,
.
IF
ELSE
EXTENDS
ID(bar)
WHILE
RETURN
THIS
[]
=
&&
||
!
BOOL
INT
VOID
NEW
```

in which one can see that the symbol sequence was indeed properly broken down at the boundaries of lexemes. Note that this is not meant to be an exhaustive test. It is **your responsibility** to write a solid battery of tests. You should, however, stick to the nomenclature and formatting shown above to ease the

test automation. Note how tokens that carry lexeme (like INTEGER or ID) show the value of the lexeme in parenthesis.

To ease your testing, remember that you can place test inputs in text files and use I/O redirection at the command prompt. For instance, if the input above is placed in a file `testme.txt`, then doing

```
./nightfury < testme.txt
```

will redirect the standard input to read from the file `testme.txt`.

Have fun!