TD 02: Lexer & Parser

For this second exercise sessions, you're going to implement the first part of our compiler for ImpL language. This first part will be the lexical and syntactic analysis.

Some file are available on the webpage to give you base code for your compiler.

1. Lexer

Almost all the lexer code is already in the resource file. You can check if everything good even so. For instance you can change some of the operator names if you want to.

2. Parser

Now, let's write the syntactic rules for our parser in order to be able to read your source file and find the errors. To help you do this part, we propose to split it in several parts described below:

- Write rules for variable declaration
- Write rules for variable access (take attention of arrays)
- Write rules for mathematical operations
- Write rules for boolean expressions
- Write rules for loop and condition
- Write rules for function call
- Write rules for function declaration

If everything went good, you can now test your compiler to check if everything is recognized as it supposed to.

3. Lexer – second part

Next step, we need to write code in our Lexer to store all needed value. For instance, variable identifier or hard coded numbers.

In the code you can define in the Lexer, you already have some variable that you can use :

- yyleng : the length of the string read (useful to get the current column position)
- yytext : the string read (Warning, those data will be freed after the end of your portion of code so if you need to store it, don't forget to duplicate them)
- yylval : is a variable of type union which will then be stored and where you can store information that you need to persist. This union type has to be declared in your .y file in the same way than shown below :

%union { int intValue; char* strValue; } (of course, you can add other fields)

Then in your .y file, you will have to specify for each token which return something of type yylval, which union member is returned. It's done in the way below :

%token <strValue> IDENTIFIER

4. Parser – Second part

Now that you've got all the informations from the Lexer, your goal will be to create an AST (http://en.wikipedia.org/wiki/Abstract_syntax_tree) from your instructions.

This AST is an abstract way to represent your file content. For instance, a function declaration will be a node of the tree and all the instructions of the function will be a child in this root node.

First of all, you have to implement a Node class to easily handle a tree node. This class need to be able to store the children of the node and also to have the type of the node (IDENTIFIER, CONST_INT, …) since, you need to declare an enum type to handle the node's type. It's up to you to define what type of node you need. Also think to store the line and column relative to the code to be able to display good errors then.

After the implementation of this Node class, you will easily undestand that it will not be enough since we also need to store some value (like hard coded numbers or variable names). To be able to handle this nodes too, you can create some specialized node class which inherits from the base one. (For instance, IntNode, FloatNode, ...)

Once all of this class has been implemented, you can start code the parser part to create the tree from the rules of your parser.

Remarks:

In the zip file which is available on the webpage, you can find a Makefile which you can used to easily compile the code of your compiler. This Makefile only works if you are coding in C++.

The test.sh file help you to compile your code and test your compiler with the input file "sample" in the directory "sample" (in other words : sample/sample).

In order to get the compilation of your compiler without any major problem, please be aware that all of your source file (lex, yacc, c++, c …) must be under the "src" directory.