CSE4100: Compilers **Fall 2015**

Homework 4: LALR(1) Parsing (Part II)

Out Date:	10/15/2015
Due Date:	10/22/2015

Objectives

The purpose of this homework is to finish the parser for `C--`. You will be using bison again to produce the *augmented grammar* that allows you to specify the actions that build a parse tree. In addition you are expected to write the classes that represent the nodes of the parse tree.

Word of caution This is a big homework. You will not get an extension. You should start **immediately**. Pace yourself! I strongly encourage everyone to make use of the documentation (of bison). Work incrementally, testing small pieces of the grammar. Do not attempt to write everything and test all at once.

Handout

The handout is the solution of the previous homework. Feel free to use your grammar.y instead (as long as it is correct!). Since you already have a grammar file (i.e., `grammar.y`), your work revolves around one task: augment the grammar with semantic rules (actions) that compute the semantic attributes of the non-terminal defined by each rule. The objective is to *synthesize* a parse tree that would be the semantic value associated to the start symbol of the grammar. Since your starting grammar is LALR(1), the focal points of the exercise are

- Write classes for each type of AST node you an encounter. In particular, think carefully about how many and which children each node should have. This work will occur in `ast.C` and `ast.H` where you will give the class prototype and the implementation of the basic methods.
- Write the actions in the grammar file to instantiate the nodes of the AST. As discussed in class this involves using the semantic values from the other non-terminal appearing in the body of the rule to compute the new semantic value for the non-terminal on the left hand side.

To test your code, it is sufficient to implement a `print` method throughout the `ASTNode` composite to let you print out the tree. By producing suitably parenthesized expressions, this print out can easily demonstrate that your trees are correctly balanced and capture correct operators precedence and "good" associativity. The `print` methods would be specified in `ast.H` and implemented in `ast.C`.

It is your responsibility to thoroughly test all your code by writing sample `C--` programs. We will test your parser on an extensive test suite. Once you are ready to handin, archive the directory again (as a zip file) and proceed to the handin section of the website.

1 Grammar

You are using the very same grammar as for the previous assignment. The source code handout contains a solution (an LALR(1) grammar).

2 Bison

This time, when `bison` runs, it will not only report whether the parse was successful, but it will also place in the parser the root of the abstract syntax tree that was constructed by the parser. It is a *trivial* affair to invoke the print method on the root of the tree. The code fragment below illustrates what needs to be done in `main.C` to retrieve that root and print the tree.

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

using namespace std;
int main(int argc,char* argv[]) {
   const char* fn = 0;
   if (argc == 2)
      fn = argv[1];
   Parser p;
   p.run(fn);
   AST::Node* root = p.getRoot();
   if (root) {
      root->print(std::cout);
   } else std::cout << "couldn't parse" << std::endl;
}
```

As you can see, the code simply invokes the `getRoot` method of the parser object to retrieve the root of the tree. If such as root exist, the parse was successful and the tree is printed. The root of the tree is saved into the parser by the semantic action of the very first production of the grammar. Namely, with the start symbol of the grammar named `Top`, the grammar.y production section starts with:

```
%%
Top: ClassList { parser->saveRoot(new AST::Program($1));}
;
```

Namely, a C– is a list of classes defined by the non-terminal *ClassList* and the semantic action instantiate a `Program` class object, passing it the semantic value from the first symbol ($1, i.e., `ClassList`) and passes the pointer to that object as the root of the tree to a simple `saveRoot` method that stores the given pointer in an attribute of the parser class.

To implement the semantic actions for all the rules, do not forget that you need to make 3 changes each time:

1. add a semantic value and its type in the %union section
2. add a %type statement (after the %token) to specify which semantic value is produced by each non-terminal. For instance, the stanza

   ```
   %type<expr> Expr
   ```

   states that the non-terminal `Expr` has a semantic attribute named `expr` whose type should be found in %union and is defined as

   ```
   AST::Expr* expr;
   ```

   namely, `expr` is a pointer to a C++ object of type `AST::Expr`
3. add the semantic rule at the end of each production.

3 Testing

To test, you should print the parse tree without much frills. Here is an example input / output pair. With the input

```
class X {
        X(int x,int [] z) {
            int y;
            int z;
            z = 2;
            int [] t;
            t = new int[10];
            y = foo.bar(y,3,z);
            if (y==1)
                y = 2;
            else {
              y = 3;
              y = y * 2 + 1;
            }
            t[2] = y;
        }
        void bar(int x,int y,int z) {
           return x;
        }
};

class Y {

};

class Z {

};
```

The printing of the AST should produce

```
class X
{
X(int x,int [] z)
{
int y;
int z;
z = 2;
int [] t;
t = new int[10];
y = foo.bar(y,3,z);
if ((y==1))
y = 2;
else
{
y = 3;
y = ((y*2)+1);
}
t[2] = y;
}

void bar(int x,int y,int z)
```

```
{
return x;
}

}
class Y
{
}
class Z
{
}
```

Note how the input is not indented, puts a statement per line, and uses parenthetic notation on expressions. (Recall that we do not care –yet– as to the meaning of the program, only its syntax!).

Have fun!