Assignment 2 - Part 1

Submission Date: 10th February 2016

First note that there is a change in the grammar. The new grammar defines `struct` types, and variables of such types can be declared. The language also provides facilities for declaring pointer variables and provides operations on pointer variables. Unless there are serious omissions, this will be the language that we shall stay with for the rest of the lab assignments.

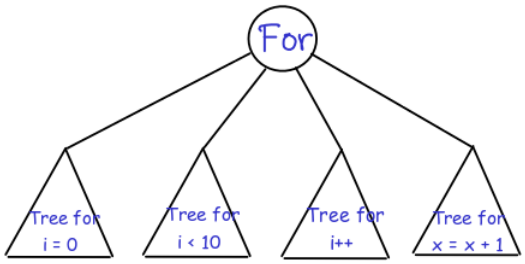
In this assignment we shall generate an abstract syntax tree (AST) for a program. An AST is a structure that represents the imperative (i.e. the non-declarative) part of the program as a tree. Here is a more complete picture:

1. As you process the program, the information contained in the declarations is stored in the *symboltable*. As we shall see in the class, this consists of the types of variables and functions, and the sizes and offsets of variables.
2. When you process the non-declarative part, then two things happen:
   (a) The information in the symboltable is used to ensure that the variables are used within the scope of their declarations and are used in a type-correct manner. If they are not, then the program is rejected.
   (b) If the program is syntactically and semantically (type and scope) correct, a tree structure called an AST is generated. This structure along with the information in the symboltable is used to generate code.

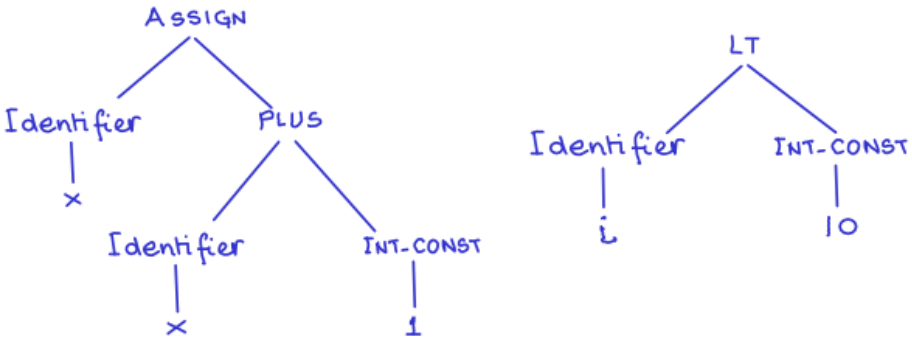
The way we shall proceed in the rest of the lab course is as follows:

1. In the first part of the second assignment, you will construct ASTs for programs without constructing the symboltable or doing semantic checks. *This means that we may construct ASTs for even wrong programs.* The AST will also be printed in a format that we shall describe below. The deadline for this part is 11th February.
2. In the second part of the second assignment, we shall construct the symboltable, do the semantic checks and then generate ASTs using the AST construction scheme that was developed in the first part. If an error is encountered, it will be reported and no AST will be generated. This part will have to be submitted by 3rd March.
3. In the third assignment, we shall generate target code using the symboltable and the AST.

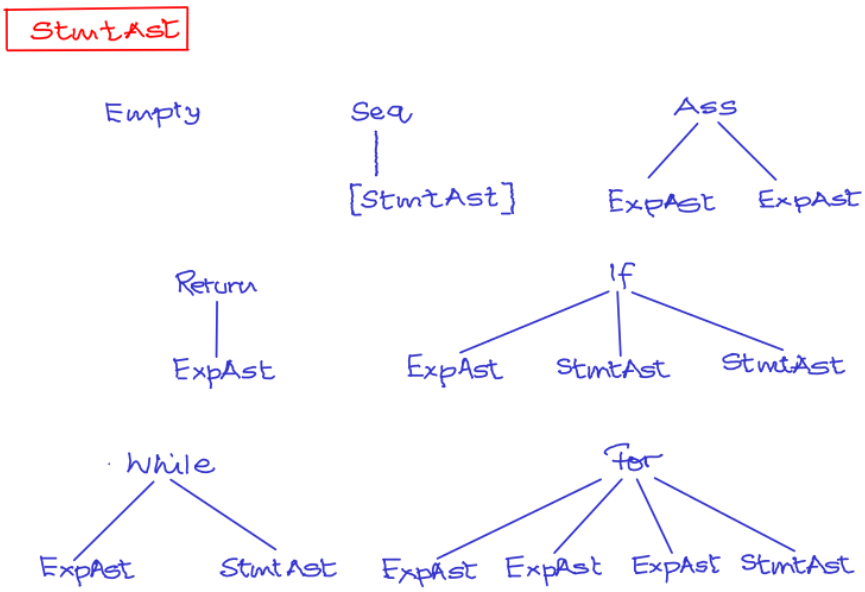
An AST represents the essential structure of the program. In an AST, we represent each construct (statement, expression) etc as a tree, consisting of root node representing an operator operating on some sub-trees. As an example, the AST for a `for` statement `for (i=0; i<10; i++) x = x + 1;` can be picturized like this.

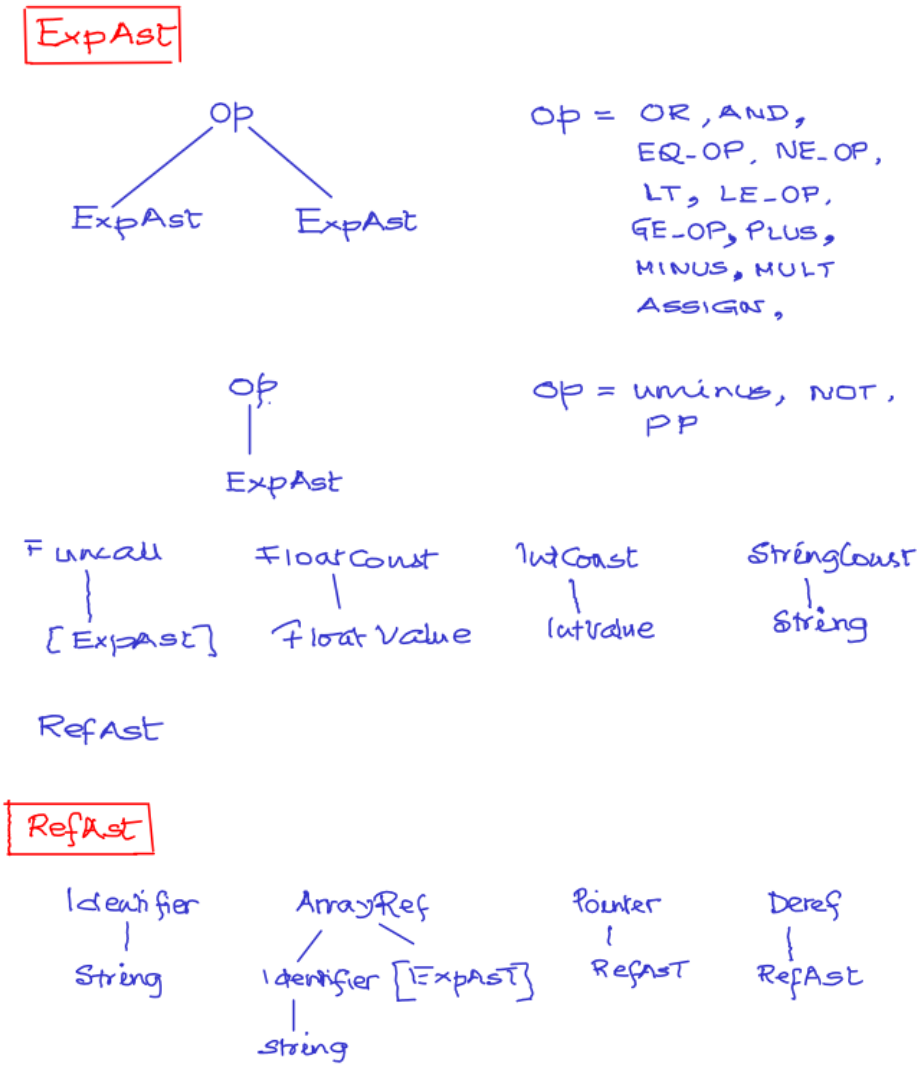


This says that the essential structure of a `for` consists of an *initilizer expression*, a *guard*, a *step expression* and a *body*. To complete the example, here are the ASTs for the guard and the step expression.



Below, I give in pictures all the different types of AST nodes would be required.





What you have to do is:

1. Design classes to represent the ASTs.
2. Add actions to the bisonc++ script to create ASTs

To start off, here is a abstract class from which you can inherit other classes that describe the AST. The only function that you would have to implement in this is `print`. The other functions will be implemented in the second part of the second assignment and the third assignment.

```
class abstract_astnode
{
 public:
```

```
  virtual void print ()  = 0;
  virtual std::string generate_code(const symbolTable&) = 0;
  virtual basic_types getType() = 0;
  virtual bool checkTypeofAST() = 0;
 protected:
  virtual void setType(basic_types) = 0;
 private:
  typeExp astnode_type;
}
```

To tie all things together, here is a program and the expected output and the end of the first part of the second assignment.

```
main()
{
  int x, y;
  for (x=0; x <10; x++);
  if (y >1) {x=x-1; y=y+1;} else ;
}
```

The expected output is:

```
(Block [(For(Assign_exp (Id ‘‘x’’) (IntConst 0))
            (LT (Id ‘‘x’’) (IntConst 10))
            (PP (Id ‘‘x’’))
            (Empty))
        (If (GT (Id ‘‘y’’) (IntConst 1))
            (Block [(Ass (Id ‘‘x’’) (Minus (Id ‘‘x’’) (IntConst 1)))
                    (Ass (Id ‘‘y’’) (Plus (Id ‘‘y’’) (IntConst 1)))])
            (Empty))])
```