Assignment 3

Submission Date: 18th Feb 2015

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:

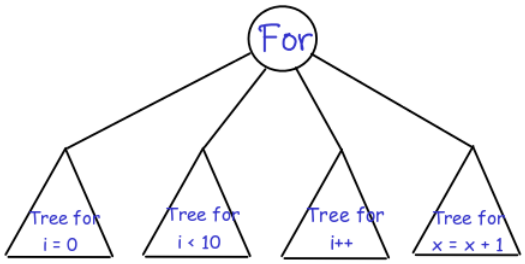
1. As you process the program, the information contained in the declarations is stored in the *symboltable*. As we shall see in the class, this consists of the types of variables and functions, and the sizes and offsets of variables.
2. When you process the non-declarative part, then two things happen:
   (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 third assignment we shall construct the AST for programs without constructing the symboltable or doing semantic checks. This means that we may construct ASTs for even wrong programs.
2. In the fourth assignment, we shall construct the symboltable, do the semantic checks and then generate ASTs using the AST construction scheme that was developed in the third assignment.
3. In the fifth assignment, we shall generate target code using the symboltable and the AST.

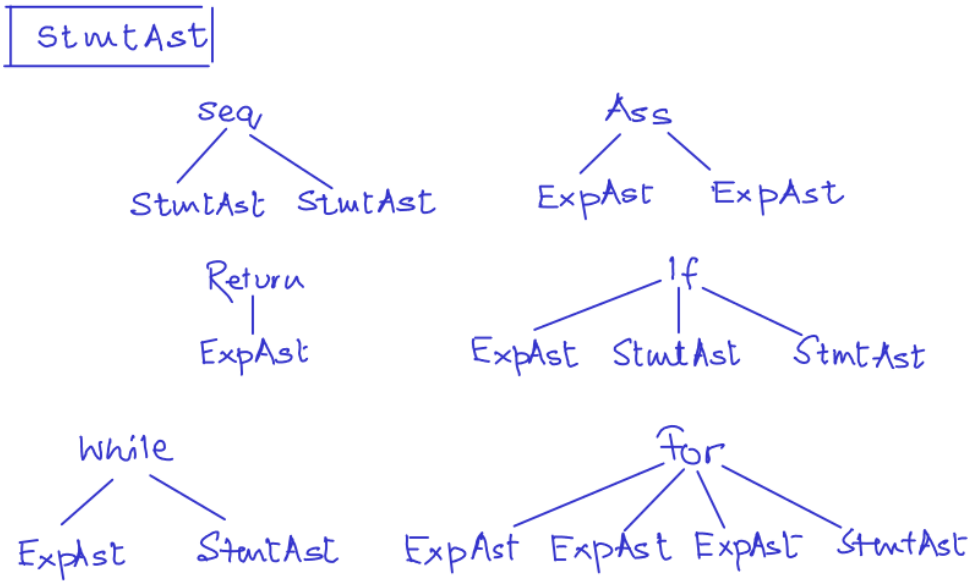
1 AST

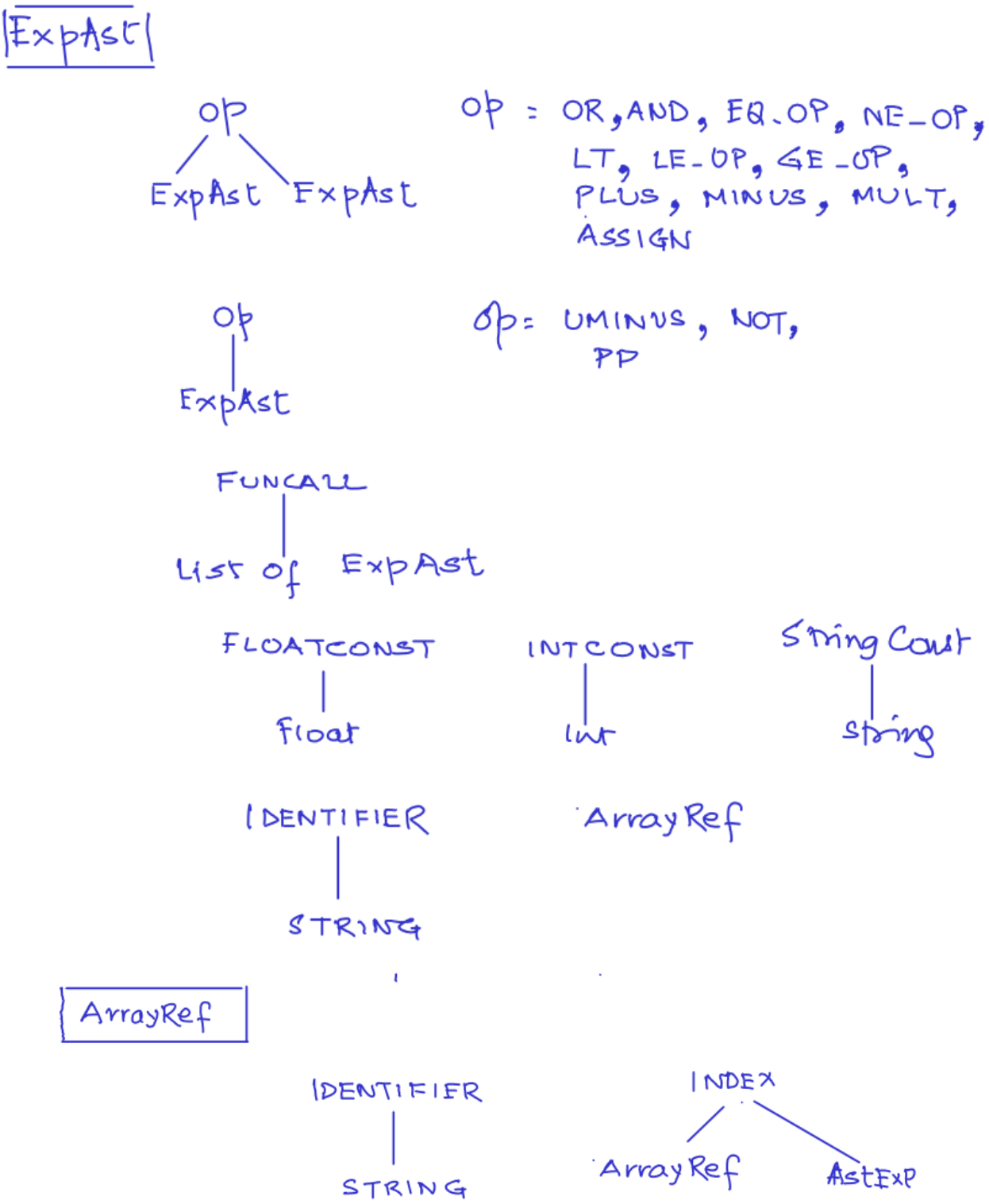
The AST represents the essential structure of the program. In an AST, we represent each construct (statement, expression) etc as a tree node representing an operator operating on some sub-trees. As an example, the AST for a `for` statement `for (i=0; i<10; i++) x = x + 1;` can be picturized like this.



This says that the essential structure of a `for` consists of an *initilizer expression*, a *guard*, a *step expression* and a *body*. Each component of the `for` has its own AST and so on.

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`.

```
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;
}
```

1.1 Continued: 11th Feb 2015

Please note the following modifications and hints:

1. There is a minor change in the grammar. The previous grammar had

   ```
   statement
          : compound_statement
               ...
   ```

   This would introduce local declarations within blocks. To avoid this complication, the modified grammar has been changed to:

   ```
   statement
          : '{' statement_list '}'
            ...
   ```

2. Notice that we also have an empty statement:

   ```
   assignment_statement
   : ';'
             ...
   ```

   This has to be modeled as a AST. If we do not, then in a statement like `for(x=0; x<10; x++);` the fourth component of the `for` would be missing. Therefore we add a empty AST to the list of ASTs.

3. Initially we suggested that `statements` in a `statement_list` should be joined to a `Seq` AST. WE are now suggesting that you make a `List` of ASTs using the STL containers. The reason for this change is that you can get the container methods for free.

4. In continuation of the earlier point, since a `statement_list` can reduce to a `statement`, introduce a `block_statement` AST.

5. For now assume that there will be a single function. So you just have to create the AST corresponding to this function. In the later stages we shall extend this to handle multiple functions.

To tie these suggestions together, here is a program and the expected output:

```
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))])
```