Compiling and Executing

There are 2 simple ways to compile the program on a Linux machine. You can either use the `ant` or `make` utility in the terminal, depending on which utility you have on your system already. I have written the `build.xml` file in order to easily compile the program with `ant`. Similarly, the `makefile` can be used if you choose to compile with `make`. The compilation process will generate an executable `jar` file.

To compile, change the terminal to the directory where `build.xml` & `makefile` are located. And type either one of these:

```
-  $ ant
-  $ make
```

To run, type either one of these:

```
-  $ make run
-  $ java -jar Robol.jar
```

And to clean, type either one of these:

```
-  $ ant clean
-  $ make clean
```

Design (see the last page for class diagram)

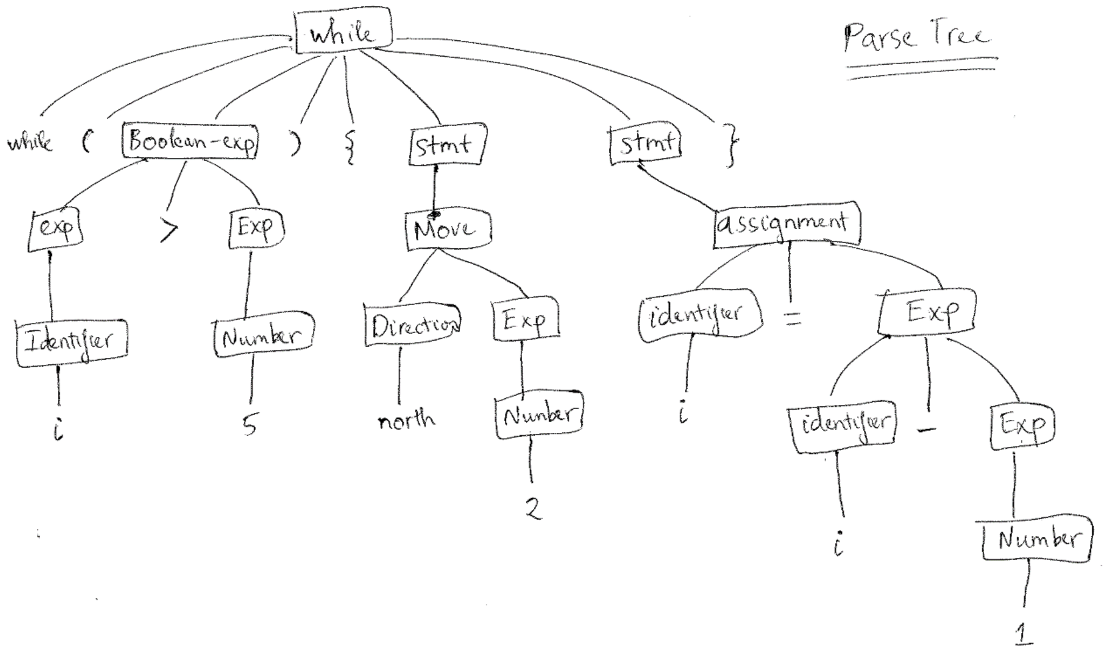
I have chosen to implement the Robol program in a very object-oriented way. All classes have private instance variables which can only be accessed/modified through public methods of the class. In general, there are 2 abstract classes (`Statement` and `Expression`) which provide a common "interface" for all objects that are either a statement or an expression. Since different types of statements must be interpreted in different ways, thus I have chosen to make the `interpret()` method inside the `Statement` class to be abstract, forcing all subclasses to implement/override this method. Similarly, the `Expression` class also has an abstract method which all subclasses must implement. However, unlike a statement, an expression can be interpreted by *evaluating* it. So instead of having a common `interpret()` method, I decided to let all classes that represent an expression to have an `evaluate()` method which must return an integer value as the value that the expression is evaluated to. The `interpret()` and `evaluate()` methods have the following signatures:

```
-  void interpret(Robot r, HashMap<String,Integer> vars);
-  int evaluate(HashMap<String,Integer> vars);
```

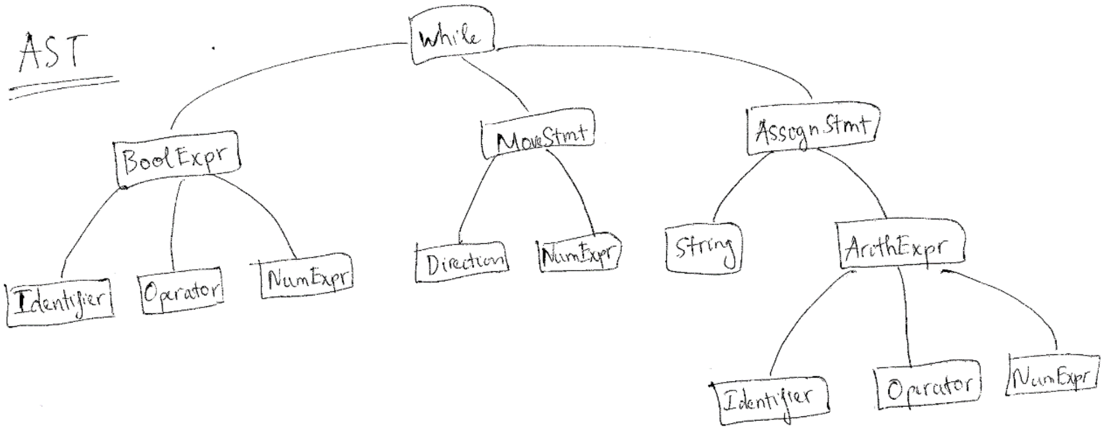
The parameter `r` represents the actual robot object, while the variable `vars` represents the global list of variables. It's obvious that not all subclasses of `Statement` need to receive both `r` and `vars` to work properly (for example, an assign-statement does not need the robot object since the only job that an assignment does is to change the value of a declared variable). And likewise, not all types of expressions need to use the list of variables (for example, a number expression doesn't have to search anything on the variables list). However, since I have declared these two methods as abstract in the two super-classes, it is impossible to avoid this.

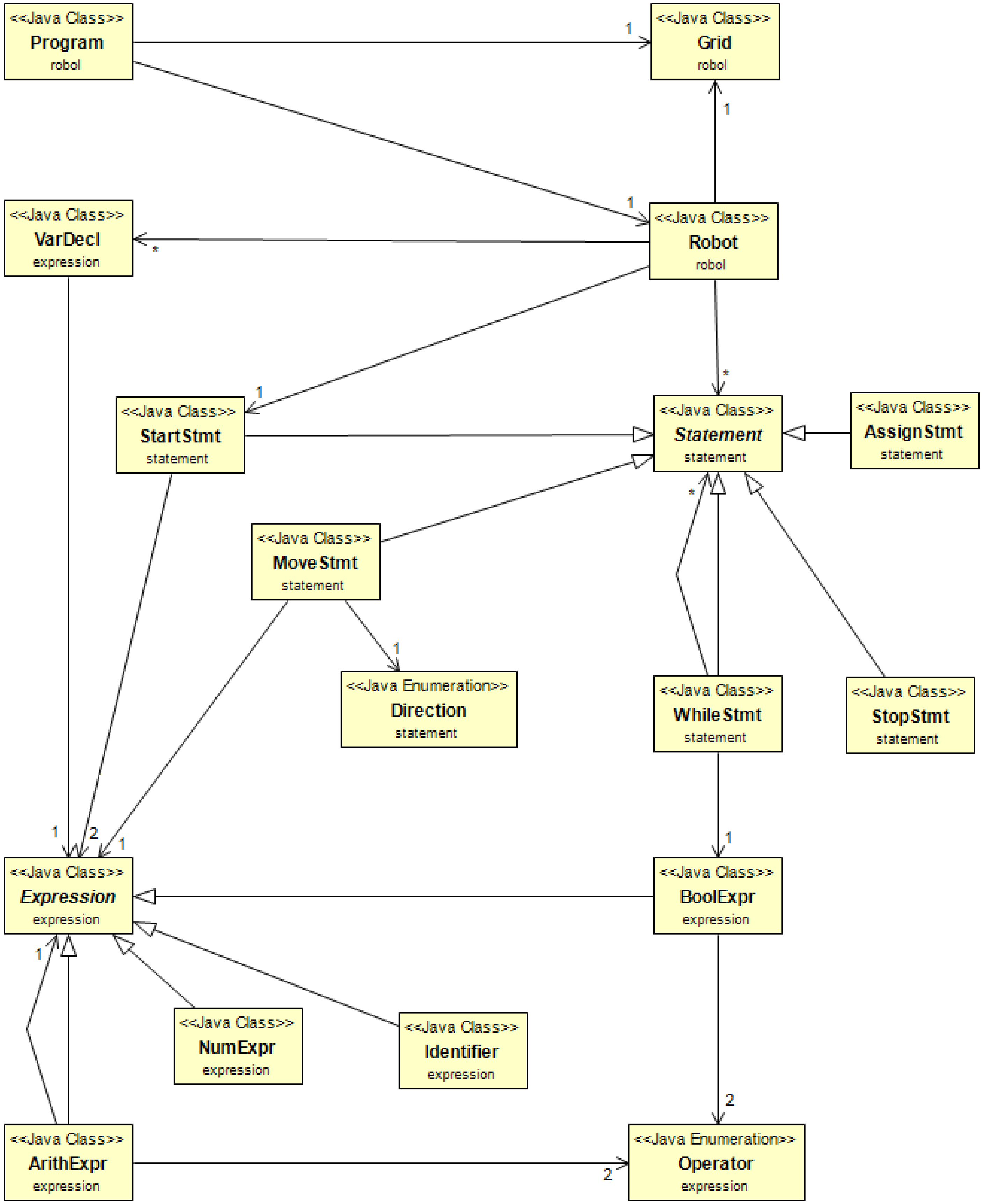
Because of the reasons mentioned above, the program that I wrote now contains 2 different `interpret()` methods with different signatures. The `interpret()` method that receives no parameters are declared inside only 2 classes (`Robot` and `Program`). This means that if I were to include a `Robol` interface with the common method `interpret()` (as done in the pre-code), then this interface would be implemented by only 2 classes. Hence, I find it unnecessary to include the `Robol` interface in my code.

The interpreter, in general, interpret a robol program by using a recursive technique, which goes from the top of the AST (abstract syntax tree) down to the bottom, and then jumps back up level by level. For example, a `Program` node in the AST would call the interpret method of the `Robot` node right below, which would then call the interpret methods of its children nodes (which are either of type `Statement` or of type `VarDecl`). Similarly, a statement can invoke the evaluate method of an expression that locates at the node right below it. So for the following code, the parse tree and the AST looks like this:



```
while (i < 5) {
   north 2
   i = i - 1
}
```




<<Java Class>>
Program
robol
<<Java Class>>
Grid
robol
1
1
1
<<Java Class>>
VarDecl
expression
*
<<Java Class>>
Robot
robol
1
*
<<Java Class>>
StartStmt
statement
<<Java Class>>
Statement
statement
<<Java Class>>
AssignStmt
statement
*
<<Java Class>>
MoveStmt
statement
1
<<Java Enumeration>>
Direction
statement
<<Java Class>>
WhileStmt
statement
<<Java Class>>
StopStmt
statement
1
2
1
1
<<Java Class>>
Expression
expression
<<Java Class>>
BoolExpr
expression
1
<<Java Class>>
NumExpr
expression
<<Java Class>>
Identifier
expression
2
<<Java Class>>
ArithExpr
expression
2
<<Java Enumeration>>
Operator
expression