PA3 -- Abstract Syntax Tree

1. Description

For this assignment, you will build the Abstract Syntax Tree(AST) that represents the program source code. You will do that by adding Abstract Syntax Tree(AST) creation logic into your parser.y from PA2.

The parser you built in PA2 contains only the reduction **rules**; You did not write any **action** for the reduction rules. The action that will be taken for each reduction(the code that will be executed for each reduction) should be the code that creates the nodes for the AST. Therefore, after your parser successfully finished all the reduction, all the nodes should be created and thus the AST will be built. In the later assignments you will be able to do the *semantic checking* and *code generation* by just traversing the AST.

2. Starter Code

Makefile	used to build the parser
utility.h / .cc	interface / implementation of various utility functions
errors.h / .cc	class that defines error reporting methods
location.h	class that defines type for token location
lexer.h	type definitions and prototype declarations for lexer
lexer.l	flex input file (use your own lexer.l from PA2, we will not provide lexer.l in PA3's starter code)
parser.h	declare parser functions and types
parser.y	**skeleton of a bison parser (You should swap in your own parser.y from PA2. The given parser.y should just serve as an example)**
main.cc	the entry point where the whole program will be started
samples	directory contains test input files
ast.h/ .cc	interface / implemention of AST node classes
ast_type.h/ .cc	interface / implemention of AST type classes
ast_decl.h/ .cc	interface / implemention of AST declaration classes
ast_expr.h/ .cc	interface / implemention of AST expression classes
ast_stmt.h/ .cc	interface / implemention of AST statement classes

When you are going through the starter code, please read the comment as well. The comments are really helpful in explaining the structure of the code.

In this assignment you only need to modify parser.y. Please go through the ast files and get yourself familiar with the ast classes. It should be straightforward to figure out what do they stand for by reading the class name.

In PA2, all the non-terminals of the parser have type integerConstant; that's why you have all those type clash warnings. For this assignment, since we have all the ast classes, we need to assign our non-terminals with the correct type with the classes in the those ast files. You can take a look at the starter code where some simple examples are given on how to assign types to the non-terminals. There are two steps you need to follow:

1. Create the non-terminal type in **%union**

   As you can see in the starter code, four non-terminal types have been created using the classes in those ast files:

   ```
   Type *varType;
   Decl *decl;
   VarDecl *varDecl;
   List<Decl*> *declList;
   ```

2. Assign the non-terminal with the correct non-terminal type using **%type** keyword

   ```
   %type <declList>  DeclList
   %type <decl>      Decl
   %type <varDecl>   single_declaration
   %type <varType>   type_specifier
   ```

3. Example

Take a look at the action for the first reduction rule. You can see the following code:

```
program->Print(0);
```

This will call the Print() method that has already been implemented for you so that the whole AST will be printed out that you can verify your implementation.

As you can see some actions have already been implemented, you should play with it and try to understand what they are doing. You can try it by doing the following:

```
%  make
% ./parser < samples/test1.java
```

And the output would be the AST that represents the source code in *test1.java*

```
%    Program:
%  1   VarDecl:
%         Type: int
%  1      Identifier: var1
%  2   VarDecl:
%         Type: boolean
```

```
%  2      Identifier: var2
```

This AST has a root node `Program`; under it with two `VarDecl` children, and each `VarDecl` has two children. You can try different inputs and see what's the output look like. It also has the source code line number printed out for each AST node.

4. Testing your work

We will provide some simple test cases and the corresponding output which you can find in the **samples** folder. You should definitely come up with your own test cases as well to test that the AST you build correctly represents the input source code.

5. Grading

You can go to any teaching staff's lab hours and ask for a check-off when you are done with your assignment. You will be asked to run your assignment against the given test cases, and we will check the output of your assignment. You also need to generally describe how did you approach the assignment to the teaching staff.