COSC 4785
Compiler Construction I
Program 5, Type Checking with Symbol Tables

1 Project

1. Having gotten your parser to run reasonably, we need to take the next step in determining the validity of the input program, type checking. This also includes determining whether identifiers are in the current scope. This means methods as well. For this project that means doing three things:

   (a) Inserting all variable and method declarations into some symbol table. This will require having some string that represents the identifier and some type information. For methods the type is more complex and must identify the types (and order) of all the parameters as well as the return type.

   (b) Maintaining information on all classes so that they can be used as types. This means putting them into a "global" or "root" or "type" symbol table that is primogenitor of all the other symbol tables.

   (c) Validating the input. As we have discussed in class we are not going to stick with the C/C++ and Java (for variables) version of type checking where everything has to be declared before use (that means physically occurs before it is used as a type/variable). Instead we will simplify this by saying that scope will be the defining factor on type correctness. The exception being that all the classes used as types must exist by the time all the code has been processed. Specifically that some class, A, does not have to be declared before being used as a type in some other class, B. Of course this means that we really cannot do type checking "on-the-fly" but that should not really be a problem.

2. The specific implementation is entirely up to you. The symbol tables should support three basic operations:

   (a) A constructor. This creates/initializes an empty symbol table. It should take as an argument a pointer/reference to a *parent* symbol table. Note that there should be some way to determine if the creation was successful. That is just smart programming.

   (b) A **lookup()** method/function. This takes an identifier string and determines whether that identifier has been declared in the current scope. If so it returns a value that can be used to validate the type of the operation the identifier is being used in. Otherwise it returns some indication of failure. If you do not like that name, fine but something like it would be reasonable.

       Think, before programming, about what this operation is going to return. For instance when you trying to type check

```
x = obj.f(y);
```

you are going to have to make sure the function call is correct. That means the number, order and types of the arguments match. Then you will have make sure that the assignment is correct. Can `x` be an *l-value* and is the return value of the function the same type. Be nice if you did not have to look the function up twice.

(c) An **insert()** method/function. This takes an identifier and type information and inserts it into the current symbol table. It should check to see if that string is already declared in the local scope and return an error if so. Otherwise it returns a value indicating success. Again, the name of the function is not really relevant as long as reasonable. But what really are you passing it and did you generate that information.

2 Format of output

1. A set of error messages. These are same error messages you should have lovingly crafted in the previous two versions of the compiler. If the program has no errors, nothing should be printed for this part.

2. After having processed the entire input, you should NOT PRINT the information from the previous two assignments. I do not want to see that. You should instead begin type checking. If there are type errors, please try to print out as much information as possible about the error and where is occurred. Preferable some like

   invalid l-value, line 23: 5 = x + 6; type mismatch in expression, line 28; x / y

   I know that line numbers and the original input are a problem at this point, just do the best you can.

3. Once the type checking is done, your program should *dump* the symbol tables. That is, output information that shows what the contents of the tables are and their relationship. We will do this as simply as possible. Basically, process any global table, printing out its entries, one per line. Then do an in order traversal of the rest of the symbol tables printing their content as you go. This means that each block (scope) will be printed, then the blocks contained in it, and so forth. Each scope should be offset two spaces from the parent scope, something like:

```
foo  class_type
  x int
  y int
  foo  method_type  null <- null
  something method_type int <- int x int
    a int
    b int
```

```
    c int

Bill class_type
  x int
  .
  .
  .
and so on
```

The two sections of the output should be distinct. That is the error output should occur first (as the input is processed), type errors next, and the dump should be last. The output from the previous assignments should be eliminated **except** for the errors with their corresponding line and column number.

3 Typing Information

Initialization. All the variables should be be initialized. The integers are initialized to zero (0) and reference variables are initialized to `null`. Most useful for the next assignment.

Name Spaces. There are three name spaces in Decaf, class name space, method name space, and block name space. Names and methods may have the same name as previously declared local variable in another method or as that of a method or instance variable in any other class.

Scope of a Local Variable. A local variable's scope is in force throughout the block in which it is declared. If an identifier of the same name was previously declared as an instance (class) variable or a method in the enclosing class, then the other declaration is hidden for the remainder of the block. Outside the block, the other declaration maintains its current status. To access the other declaration within the block, preface the other declaration with **this** (e.g., `this.x`). If a local variable in a block masks a local variable in an enclosing block, that masked variable is **not** accessible.

Type Scope. Like C++ but unlike Java, an identifier with the same name as a previously defined class does **not** hide the class name. Hence, the previously defined class name can still be used as a type name within the block. In other words, a type name always remains in force.

4 What to turn in

1. Attach the source files for you current version of the compiler. Submit them on the WyoCourses assignment page.

2. Your source code should have comments (like your name, date, course at the beginning). These should identify the file and explain any non-obvious operations. Feel free to comment as necessary to help yourself but do not comment every line.

   Including a "readme" of some type is not necessary but if there things you think I **need** to know about your code, feel free. I prefer txt files for this.

   Include a VALID Makefile. The name of the executable should be **program5**.

   DO NOT *tar* DIRECTORIES. I only want files when I extract the archive.

5 Grammar

1. *Program*	$\rightarrow$	*ClassDeclaration*$^+$
2. *ClassDeclaration*	$\rightarrow$	**class identifier** *ClassBody*
3. *ClassBody*	$\rightarrow$	{ *VarDeclaration** *ConstructorDeclaration** *MethodDeclaration** }
4. *VarDeclaration*	$\rightarrow$	*Type* **identifier** ;
5. *Type*	$\rightarrow$	*SimpleType*
	$\mid$	*Type* []
6. *SimpleType*	$\rightarrow$	**int**
	$\mid$	**identifier**
7. *ConstructorDeclaration*	$\rightarrow$	**identifier** (*ParameterList*) *Block*
8. *MethodDeclaration*	$\rightarrow$	*ResultType* **identifier** (*ParameterList*) *Block*
9. *ResultType*	$\rightarrow$	*Type*
	$\mid$	**void**
10. *ParameterList*	$\rightarrow$	ϵ
	$\mid$	$Parameter\ < ,\ Parameter\ >^*$
11. *Parameter*	$\rightarrow$	*Type* **identifier**
12. *Block*	$\rightarrow$	{ *LocalVarDeclaration** *Statement** }
13. *LocalVarDeclaration*	$\rightarrow$	*Type* **identifier** ;

14. *Statement*	$\rightarrow$	;
	$\mid$	*Name* = *Expression* ;
	$\mid$	*Name* (*Arglist*) ;
	$\mid$	**print** (*Arglist*) ;
	$\mid$	*ConditionalStatement*
	$\mid$	**while** (*Expression*) *Statement*
	$\mid$	**return** *OptionalExpression* ;
	$\mid$	*Block*
15. *Name*	$\rightarrow$	**this**
	$\mid$	**identifier**
	$\mid$	*Name* . **identifier**
	$\mid$	*Name* [*Expression*]
16. *Arglist*	$\rightarrow$	ϵ
	$\mid$	$\mathit{Expression}\ <\ ,\ \mathit{Expression}\ >^*$
17. *ConditionalStatement*	$\rightarrow$	**if** (*Expression*) *Statement*
	$\rightarrow$	**if** (*Expression*) *Statement* **else** *Statement*
18. *OptionalExpression*	$\rightarrow$	ϵ
	$\mid$	*Expression*
19. *Expression*	$\rightarrow$	*Name*
	$\mid$	**number**
	$\mid$	**null**
	$\mid$	*Name* (*ArgList*)
	$\mid$	**read** ( )
	$\mid$	*NewExpression*
	$\mid$	*UnaryOp Expression*
	$\mid$	*Expression RelationOp Expression*
	$\mid$	*Expression SumOp Expression*
	$\mid$	*Expression ProductOp Expression*
	$\mid$	(*Expression*)
20. *NewExpression*	$\rightarrow$	**new** *identifier* (*Arglist*)
	$\mid$	**new** $\mathit{SimpleType}\ <\ [\ \mathit{Expression}\ ]\ >^*\ <\ []\ >^*$

21. *UnaryOp* $\rightarrow$ $+$
$\quad\mid$ $-$
$\quad\mid$ $!$

22. *RelationOp* $\rightarrow$ $==$
$\quad\mid$ $!=$
$\quad\mid$ $<=$
$\quad\mid$ $>=$
$\quad\mid$ $<$
$\quad\mid$ $>$

23. *SumOp* $\rightarrow$ $+$
$\quad\mid$ $-$
$\quad\mid$ $\|$

24. *ProductOp* $\rightarrow$ $*$
$\quad\mid$ $/$
$\quad\mid$ $\%$
$\quad\mid$ $\&\&$

5.1 Operator Precedence

Operator precedence is as follows (from lowest to highest):

1. RelationOp
2. SumOp
3. ProductOp
4. Unaryop

Within each group of operators, each operator has the same precedence.

5.2 Operator Associativity

All operators are left associative. For example, **a + b + c** should be interpreted as **(a + b) + c**.

6 Lexical conventions

1. Identifiers are unlimited sequences of letters, numbers and the underscore character. They *must* begin with a letter or underscore. They may contain upper and lower case letters in

any combination.

2. The *number* is an unsigned sequence of digits. It may be preceded with a unary minus operator to construct a negative number.

3. The decaf language has a number of reserved **keywords**. These are words that cannot be used as identifiers. As the language is also case sensitive, versions of the keywords containing uppercase letters can be used as identifiers. It is a compile time error to use a keyword as an identifier. Keywords are separated from **identifiers** by the use of whitespace or punctuation. The keywords are as follows:

int	**void**	**class**	**new**
print	**read**	**return**	**while**
if	**else**	**this**	

4. The following set of symbols are operators in the decaf language:

<table>
<tr><td>[</td><td>]</td><td>{</td><td>}</td></tr>
<tr><td>!=</td><td>==</td><td><</td><td>></td></tr>
<tr><td><=</td><td>>=</td><td>&&</td><td>||</td></tr>
<tr><td>!</td><td>+</td><td>-</td><td>*</td></tr>
<tr><td>/</td><td>%</td><td>;</td><td>,</td></tr>
<tr><td>(</td><td>)</td><td>=</td><td>//</td></tr>
<tr><td>.</td><td></td><td></td><td></td></tr>
</table>

5. Whitespace is space, tab and newline. Other than using it to delimit keywords (and the newline for a
comment) it is not required. It will **not** be passed to the parser.