CS 536 — Spring 2015

Programming Assignment 4
CSX Type Checker

Due: Tuesday, April 14, 2015

Not accepted after Tuesday, April 21, 2015

You are to write member functions and classes that implement a **type checker** for CSX programs. Your main program will call your CSX parser. If the parse is successful, it will call the type checker. The CSX source program to be compiled is named on the compiler's command line or entered through a GUI. Error messages are written to standard output. An Eclipse archive containing a complete type checker for CSX-lite may be found at `www.cs.wisc.edu/~fischer/cs536.s15/course/proj4/startup/eclipse/`.

The Type Checker

The type checker will be implemented as a visitor class (`TypeChecking.java`) with member functions operating on the abstract-syntax tree built by the CSX parser. The type checker should produce an error message for each scoping and type error in the program represented by the AST, and should return a boolean value indicating whether the AST had any type or scoping errors.

The scope rules of CSX are similar to those of C++ and Java. A program consists of a single named class. All members within the class (fields and methods) are static (in the Java sense). Further, all fields must have distinct names; methods may be overloaded. Local declarations (within a method or statement block) override any global declaration, but any one identifier may be declared only once in any particular local scope. Parameters of a method are considered local declarations within that method's body.

All identifiers, whether class members, or local declarations, must be declared before they are used. The last member declaration must be a `void` method named `main` with no arguments. As in Java, execution will commence with this method.

You must print an error message if a use of an undeclared identifier is found or if an identifier is illegally redeclared within the same class, method body, or statement block. (The class name is external to all other scopes; it *never* conflicts with any other declaration.)

An identifier may denote class name, a label, a field (either a variable or a constant), a method, a parameter of a method, or a local variable or a local constant. Local variables and constants, fields, functions (methods that return a value) and parameters may be of type `int`, `bool`, or `char`. Variables (fields or locals) and parameters may be arrays of `int`, `bool`, or `char` values.

The type and scope rules of the CSX language require the following:

- Arithmetic operators may be applied to `int` or `char` values; the result is of type `int`.
- Only variables (including parameters) of type `int` or `char` may be incremented (`++` operator) or decremented (`--` operator).

- Logical operators (&&, ||, and !) may be applied only to bool values; the result is of type bool.
- Relational operators (==, <, >, !=, <=, >=) may be applied only to a pair of arithmetic values (int or char) or to a pair of bool values; the result is of type bool.
- Relational operators *can* be applied to bool values; false is less than true.
- The scope of a field declared in the CSX class comprises all fields and methods that follow it; forward references to fields not yet declared are not allowed.
- The scope of a method comprises its own body and all methods that follow it. Recursive calls are allowed, but calls to methods not yet declared are not allowed.
- The scope of a local variable or constant declared in a method or block comprises all fields and statements that follow it in the method or block; forward references to locals not yet declared are not allowed.
- A formal parameter of a method is considered local to the body of the method.
- Except for method names, an identifier may only be declared once within a class, method or block. However, an identifier already declared outside a method or block may be redefined locally.
- Method names may be overloaded. All method definitions sharing the same name must return the same type (possibly void) and must differ in the number, type or kind (scalar or array) of their parameters.
- The type of a constant is the type of the expression that defines the constant's value.
- The type of a control expression (in an if or while construct) must be bool.
- Int, bool and char values, char arrays and string literals may be printed.
- Only int and char values may be read.
- The types of an assignment statement's left- and right-hand sides must be identical. Entire arrays may be assigned if they are have the same size and component type. A string literal may be assigned to a character array if both contain the same number of characters.
- The size of an array parameter is not known at compile-time. Hence all size restrictions involving the assignment of array parameters are enforced at run-time.
- The types of an actual parameter and its corresponding formal parameter must be identical.
- Arrays are passed as reference parameters. (This is an code generation issue.)
- Assignment to constant identifiers (fields or locals) is illegal. Constant identifiers may not be incremented or decremented.
- Only identifiers denoting *procedures* (methods with a void result type) may be called in statements.
- Only identifiers denoting *functions* (methods with a non-void result type) may be called in expressions. The type of a function call is the result type of the function.
- Return statements with an expression may only appear in functions. The expression returned by a return statement must have the same type as the function within which it appears.
- Return statements without an expression may only appear in procedures (void result type).
- If necessary, an implicit return statement is assumed at the end of a method.

- Expressions (including variables, constants and literals) of type `int`, `char` or `bool` may be type-cast to an `int`, `char` or `bool` value. Only these type casts are allowed.
- An identifier that labels a `while` statement is considered to be a local declaration in the scope immediately containing the `while` statement. No other declaration of the identifier in the same scope is allowed.
- An identifier referenced in a `break` or `continue` statement must denote a label (on a `while` statement). Moreover, the `break` or `continue` statement must appear within the body of the `while` statement that is selected by the label.
- A `void` method of no arguments named `main` must be the last method declared in the class that constitutes a CSX program.
- The size of an array (in a declaration) must be greater than zero.
- Only expressions of type `int` or `char` may be used to index arrays.

To prevent one type error from causing multiple error messages, you should assume that the result of an arithmetic operation is always `int`, and that the result of a logical or relational operation is always `bool`, even when an operand is type-incorrect. For example, the following expression should produce only one error message:

```
(true + 3) + 4
```

Use the line and column numbers contained in AST nodes to improve the specificity of your error messages; try to make them as informative as possible.

How to Proceed

The type checker will need a block-structured symbol table. You may reuse the symbol table methods and classes you implemented in project 1 or you may use the symbol table implementation we supply. Walk the AST recursively, executing member functions in class `TypeChecking`. You start at the root of the AST (a `csxLiteNode` or a `classNode`). When you encounter identifiers in declarations, you'll create symbol table entries for them. When you encounter uses of identifiers you'll look them up in the symbol table. In this way all uses of an identifier `id` will access the declaration corresponding to `id`, even though that declaration may be far removed from the uses.

The Eclipse archive at `www.cs.wisc.edu/~fischer/cs536.s15/course/proj4/startup/eclipse/` contains a complete type checker for CSX-lite, extended to include print statements. Look over class `TypeChecking` to see how the methods that implement type checking are organized. Note that a method corresponding to a particular AST node simply enforces the scope and type rules that pertain to the construct the AST node represents.

Corresponding to possible root nodes of a CSX-lite or CSX AST (a `csxLiteNode` or `classNode`) class `checkTypes` contains two special boolean-valued member functions named `isTypeCorrect`(). These functions call their corresponding type checking method, beginning a recursive walk of the entire AST. After type checking completes, `isTypeCorrect` returns a boolean value indicating whether any scoping or type errors have been discovered.

If an AST node has subtrees, those subtrees will usually need to be recursively type checked as part of type checking a parent node. For nodes that represent constructs that are expected to have a type, (expressions, identifiers, literals, etc.) it is convenient to add **`type`** and **`kind`** fields to the node. (These are defined in the `ast` class).

Possible values for `type` include `Integer` (`int`), `Boolean` (`bool`), `Character` (`char`), `String`, `Void`, `Error` and `Unknown`. `Void` is used to represent objects that have no declared type (e.g., a label or procedure). `Error` is used to represent objects that should have a type, but don't (because of type errors). Unknown is used as an initial value, before the type of an object is determined.

Possible values for kind include Var (a local variable or field that may be assigned to), Value (a value that may be read but not changed), Array, ScalarParm (a by-value scalar parameter), ArrayParm (a by-reference array parameter), Method (a procedure or function) and Label.

Most combinations of type and kind represent something in CSX. Hence type==Boolean and kind==Value is a bool constant or expression. type==Void and kind==Method is a procedure (a method that returns no value).

Type checking procedure and function declarations and calls requires some care. When a method is declared, you should build a list of (type,kind) pairs, one for each declared parameter. When a call is type checked you should build a second list of (type,kind) pairs for the actual parameters of the call. You compare the lengths of the list of formal and actual parameters to check that the correct number of parameters have been passed. You then compare corresponding formal and actual parameter pairs to check if each individual actual parameter correctly matches its corresponding formal parameter.

For example, if we had the declaration

```
p(int a, bool b[]){ ... }
```

and the call

```
p(1,false);
```

we'd create the parameter list (Integer, ScalarParm), (Boolean, ArrayParm) for p's declaration and the parameter list (Integer, Value),(Boolean, Value) for p's call. Since a Value can't match an ArrayParm, we can determine that the second parameter in p's call is incorrect.

If a method name is overloaded, you must check the parameter list of each definition. Exactly one definition must correctly match the call being processed.

What to hand in

As was the case for Project 3, your program should expect a text file on the command line (if no file name is found, a GUI will prompt you to enter one). This file is first parsed, then the resulting abstract syntax tree is type checked.

Create a folder (directory) and name it using your first and last name (e.g., CharlesFischer). Copy into this folder a README file, a build.xml file and all source files necessary to build an executable version of your program (.java source files, a csx.jlex file and a csx.cup file). Do not hand in any .class files. Name the class that contains your main P4.java. Upload this handin folder to the cs 536 project 4 Dropbox folder in learn@uw (https://learnuw.wisc.edu). You may compress your handin folder into a single file using zip if you wish.

Test your type checker using the test programs in www.cs.wisc.edu/~fischer/cs536.s15/course/proj4/tests/. These programs are named test0.csx, test1.csx,.... Hand in the output produced by your type checker in a file TestResults. We'll also run your type checker on a variety of our own test programs.

If you wish to claim extra credit, *clearly* state (in the README file) what you've added and include examples of its operation.