CS 536 — Spring 2014

Programming Assignment 4
CSX Type Checker

Write member functions and classes that implement a **type checker** for CSX programs. Your main program calls your CSX parser. If the parse succeeds, it calls the type checker. The CSX source program to be compiled is named on the compiler's command line. Your program writes error messages to standard output. A skeleton for the type checker module may be found in the directory `~raphael/-courses/cs541/public/proj4/startup.`

The Type Checker

The type checker is an AST member function, operating on the abstract syntax tree built by the CSX parser. The type checker produces an error message for each scoping and type error in the program represented by the AST and returns a Boolean value indicating whether the AST has 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). All members must have distinct names. Local declarations (within a method or statement block) override any global declaration, but any one identifier may be declared only once in any particular scope. Parameters of a method are 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 parameters. As in Java, execution commences with this method.

Your type checker must print an error message if the CSX program uses an undeclared identifier or if it redeclares an identifier 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 a 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`**.
- 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; by definition, **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.

• 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.

• 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 actual parameters in `print` statements.

• Only **int** and **char** values may be actual parameters in `read` statements.

• 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 **char** 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 may be passed as parameters.

• Assignment to constant identifiers (fields or locals) is invalid.

• 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 procedure (but not a function).

• Any expression (including variables, constants and literals) of type **int**, **char** or **bool** may be type-cast to an **int**, **char** or **bool** value. These are the only type casts allowed.

• An identifier that labels a **while** statement is a local declaration in the scope immediately containing the **while** statement. No other declaration of the identifier in the same scope is allowed. Labels on **while** statements are an extra-credit option.

• An identifier referenced in a **break** or **continue** statement (which you may implement for extra credit) 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 parameters 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 may 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. For instance, the following messages are fairly informative:

```
Error (line 69): Can only read into a variable, not a constant.
Error (line 76): Cannot apply subscript to non-array aa.
```

How to Proceed

To implement the type checker, you may consider the following a set of hints, not requirements. Use the block-structured symbol table classes you implemented in project 1. Walk the AST recursively, executing the member function `checkTypes`(). When you encounter identifiers in declarations, create symbol-table entries for them. When you encounter uses of identifiers, look them up in the symbol table. In this way, all uses of an identifier `b` access the declaration corresponding to `b`, even though that declaration may be far removed from the uses.

The skeleton in `~raphael/courses/cs541/public/proj4/startup` contains a complete type checker for CSX-lite, extended to include variable declarations and print statements. Look over `ast.java` to see how it organizes type checking. Note that `checkTypes`() for each particular AST node simply enforces the scope and type rules that pertain to the construct the AST node represents. The coding practices in this type checker are not up to standard; you should make use a a style checker to improve the coding.

The root node of an AST (a `csxLiteNode` or `classNode`) contains a special boolean-valued member function `isTypeCorrect`(). This function calls its own `checkTypes()` function, which recursively walks the entire AST. After `checkTypes()` completes, `isTypeCorrect()` checks to see if any scoping or type errors have been discovered and returns a corresponding boolean value.

If an AST node has subtrees, those subtrees are usually 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), it is convenient to add `type` and `kind` fields to the node.

Possible values for `type` include `Integer` (`int`), `Boolean` (`bool`), `Character` (`char`), `String`, `Void`, `Error` and `Unknown`. `Void` represents objects that have no declared type (a label or procedure). `Error` represents objects that should have a type but don't (because of type errors). `Unknown` is an initial value before the type of an object is determined. You might want to use a Java enumeration instead of integers for types.

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`. Again, you may prefer to use a Java enumeration.

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 might build a list of (`type,kind`) pairs, one for each formal parameter. When a call is type-checked, you could build a second linked list of (`type,kind`) pairs for the actual parameters of the call. Compare the lengths of the lists of formal and actual parameters to check that the correct number of parameters have been passed. Then compare corresponding formal and actual parameter pairs to check that each individual actual parameter matches its corresponding formal parameter.

For example, if we have the declaration

```
p(int a, bool b[]){… }
```

and the call

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

we create the parameter list `(Integer, ScalarParm), (Boolean, Array-Parm)` 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 determine that the second parameter in `p`'s call is invalid.

What to hand in

As was the case for Project 3, your program should take the name of a text file on the command line. Your program first parses this file and then type-checks the resulting abstract syntax tree.

Test your type checker using the test programs in `~raphael/-courses/cs541/public/proj4/startup/tests`. These programs are named `test-00.csx, test-01.csx,`.... You should also run your own tests, as always. Hand in the output produced by your type checker in a file `TestResults`.