CS 541 — Fall 2021

Programming Assignment 4
CSX_go Semantics Checker

Write member functions and classes that implement a **semantics checker** for CSX_go programs. Your main program calls your CSX_go parser. If the parse succeeds, it calls the semantics checker. The CSX_go 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 semantics checker module may be found in the directory `~raphael/-courses/cs541/public/proj4/startup.`

The Semantics Checker

The semantics checker is an AST member function, operating on the abstract syntax tree built by the CSX_go parser. The semantics 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_go are similar to those of C++, Java, and Go. A program consists of a single named package. All declarations within the package (variables and functions) are static (in the Java sense). All members must have distinct names. Local declarations (within a function or statement block) override any global declaration, but any one identifier may be declared only once in any particular scope. Parameters of a function are local declarations within that function's body.

Functions and variables must be declared before they are used. The last function must be named `main` with no parameters and no return type. As in Java, execution commences with this function.

Your type checker must print an error message if the CSX_go program uses an undeclared identifier or if it redeclares an identifier within the same scope (top-level, function body, or statement block). The package name is external to all other scopes; it *never* conflicts with any other declaration.

An identifier may denote a package name, a label, a variable (which can be declared **`const`**), a function, a parameter of a function, or a local variable. Local variables, global variables, value-returning functions, and parameters may be of type **`int`**, **`bool`**, or **`char`**. Except for value-returning functions, they may also be may be arrays of **`int`**, **`bool`**, or **`char`** .

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

1. Arithmetic operators may be applied to **`int`** or **`char`** values; the result is of type **`int`**.
2. Logical operators (`&&`, `||`, and `!`) may be applied only to **`bool`** values; the result is of type **`bool`**.
3. 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`**.
4. Relational operators *may* be applied to **`bool`** values; by definition, **`false`** is less than

true.

5. The scope of a global variable comprises all variables and functions that follow it.
6. The scope of a function comprises its own body and all functions that follow it. Recursive calls are allowed, but calls to functions not yet declared are not allowed.
7. The scope of a local variable declared in a function or block comprises all variables and statements that follow it in the function or block; forward references to locals not yet declared are not allowed.
8. A formal parameter of a function is considered local to the body of the function.
9. An identifier may only be declared once within a scope. However, an identifier already declared outside a scope may be hidden by a local declaration.
10. The type of a variable declared **const** is the type of the expression that defines the variable's value.
11. The type of a control expression (in an **if** or **for** construct) must be **bool**.
12. **int**, **bool** and **char** values, **char** arrays, and string literals may be actual parameters in **print** statements.
13. Only **int** and **char** values may be actual parameters in **read** statements.
14. The types of the left- and right-hand sides of assignment statements and initialized variables 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 only if both contain the same number of characters.
15. 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.
16. The types of an actual parameter and its corresponding formal parameter must be identical.
17. Arrays may be passed as parameters.
18. Assignment to constant identifiers (global or local) is invalid.
19. Only identifiers denoting *procedures* (**func**s without a result type) may be called in statements.
20. Only identifiers denoting true *functions* (**func**s with a result type) may be called in expressions. The type of a function call is the result type of the function.
21. **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.
22. **return** statements without an expression may only appear in procedures (functions without a result type).
23. If necessary, an implicit **return** statement is assumed at the end of a procedure (but not a function).
24. Any expression (including variables 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.
25. An identifier that labels a **for** statement is a local declaration in the scope immediately containing the **for** statement. No other declaration of the identifier in the same scope is allowed.
26. An identifier referenced in a **break** or **continue** statement must denote a label (on a **for** statement). Moreover, the **break** or **continue** statement must appear within the body of the **for** statement that is selected by the label.
27. A function of no parameters and no return type named `main` must be the last function declared in the CSX_go program.
28. The size of an array (in a declaration) must be greater than zero.
29. Only expressions of type **int** or **char** may be used to index arrays.

30. Declarations are permitted in blocks that form the body of **`if`** and **`for`** statements; they obey usual block structure.
31. It is invalid to use the value of a local variable if there is no earlier assignment to that variable.

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 `checkSemantics()`. 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 **`display`** statements. Look over `ast.java` to see how it organizes type checking. Note that `checkSemantics()` 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 `isSemanticsCorrect()`. This function calls its own `checkSemantics()` function, which recursively walks the entire AST. After `checkSemantics()` completes, `isSemanticsCorrect()` 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 **`func`**). `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 or global variable that may be assigned to), `Constant` (a value that may be read but not changed), `Array`, `ScalarParm` (a by-value scalar parameter), `ArrayParm` (a by-reference array parameter), `Function`, and `Label`.

Again, you may prefer to use a Java enumeration.

Most combinations of `type` and `kind` represent something in CSX_go. Hence `type==Boolean && kind==Value` represents a **`bool`** expression. `type==Void` and `kind==Function` represents a function that does not return a value.

Type-checking procedure and function declarations and calls requires some care. When a function 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, 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 determine that the second parameter in `p`'s call is invalid.

What to hand in

As before, your program should take the name of a CSX_go file on the command line. Your program first parses this file and then, if the parse succeeds, type-checks the resulting abstract syntax tree.

Test your type checker using the test programs in `~raphael/courses/cs541/public/proj4/tests`. These programs are named `test-00.csx_go, test-01.csx_go,`.... You should also run your own tests, as always. In addition to your code and documentation, turn in the output produced by your type checker in a file `TestResults`.