CS 536 — Spring 2016

Programming Assignment 1
Identifier Cross-Reference Analysis

Due: Tuesday, February 9, 2016

Not accepted after Tuesday, February 16, 2016

Introduction

A compiler performs many analyses on the structure of a program. Some analyses check for correctness. Others look for optimization opportunities. Still others translate a source program into executable form. In this project you will implement an **identifier cross-reference** analysis. This analysis locates each variable declaration and links the declared variable with all its uses. Consider the following simple program in **CSX Lite.** (CSX Lite is a simple subset of **CSX**, the programming language this class will compile).

```
{ int a;
  bool b;
  if (b)
    a = a+1;
}
```

A cross-reference analysis would generate this

```
1: a(int): 4(2)
2: b(bool): 3
```

This says that on line 1, `a`, an `int`, was declared. It was used twice on line 4.
On line 2, `b`, a `bool`, was declared. It was used on line 3.

A cross-reference analysis can be very useful in developing and debugging a program. For example, Eclipse supports a program restructuring technique called **refactoring**. Refactoring allows a declaration and all of its uses to be systematically updated. A common use of refactoring is to rename the declaration of an identifier along with all of its uses. Clearly, a cross-reference analysis, which links a declaration with all of its uses, makes renaming a simple task.

The Structure of CSX Lite

CSX Lite is a very simple programming language. It is too small to program anything useful. Still, it contains declarations, statements and expressions. Thus it is representative of real programming languages.

A CSX Lite program begins with a "`{`" and ends with a "`}`". The body of the program consists

of zero or more declarations followed by zero or more statements. Thus
`{int i;i=0;}` is a valid program. So is `{}`.

Declarations are simple variable declarations of the form `int` *id* or `bool` *id*, where *id* is any valid identifier. In CSX Lite identifiers are a letter followed by zero or more letters and digits. Examples include `i`, `abc`, `GoBadgers` and `CVN76`. Case is insignificant (`AA` and `aa` are the same identifier).

There are three kinds of statements: assignments, conditionals and blocks. An assignment is of the form

id = *expr*;

where *id* is any valid identifier and *expr* is any valid expression. Expressions may be an identifier, an unsigned integer literal (e.g., `123`) or a binary expression. Valid binary operators are `+`, `-`, `==` and `!=`. Parentheses may be used to group operands in an expression. Thus `abc`, `123`, `a+1` and `a-(b-c)` are all valid expressions in CSX Lite.

Conditionals are if statements with no else part, just as in Java or C++. Examples include

`if (b) c=1;` and `if (a==1) if (b==2) c=3;`

Blocks have the same form as a CSX Lite program: a left brace, zero or more declarations, zero or more statements and a closing right brace. For example,

```
if (b==c) {int d; d=b+c;};
```

Just as in C, C++ and Java, a block opens a new **name scope**. That is, a new declaration of an already declared identifier is allowed. A new declaration, in the local scope, overrides or "hides" an earlier declaration in a containing scope.
Thus the following is legal

```
{int i; bool b; if (i==10) {int b; b=i-5;}}
```

Single line comments that begin with "`//`" and terminate at the end of the current line are allowed.

Abstract Syntax Trees

From the programmer's point of view, a program is just a sequence of characters, split into lines to give it a two dimensional structure. Tools like Eclipse enforce formatting rules, adding indentation and line breaks to enhance readability. Still, programs are treated as character files and program development consists of adding, deleting and changing selected character sequences.

From a compiler's point of view, a program is represented by a much more structured data object — an **abstract syntax tree** (**AST**). Nodes in the tree represent important program components. The root of the AST always represents a complete program. Subtrees represent individual declarations, statements and expressions. Thus an AST for a CSX Lite program is rooted by a node with two subtrees: one containing all declarations and the other containing all statements. Similarly, the AST for an if statement is rooted by a node corresponding to the structure of an if statement. It has a subtree corresponding to the control expression, and a subtree corresponding to the then statement (recall there are no else parts in CSX Lite).

ASTs are *syntax trees* because their tree structure reflects the syntactic structures found in programs. They are *abstract* because certain source program components, like parentheses,

commas and semicolons are absent in the AST. This is because these components exist primarily to enhance program readability for humans. The tree structure in an AST contains only information necessary to analyze and translate program structures.

CSX Lite Abstract Syntax Tree Nodes

The AST nodes used by CSX Lite are listed in Table 1. All ASTs are rooted by a `csxLiteNode`. This node has two references to subtrees, `progDecls` and `progStmts`. The type of `progDecls` is `fieldDeclsOption` and the type of `progStmts` is `stmtsOption`. Many subtree references are declared to be some sort of "option." Option types (like `fieldDeclsOption` or `stmtsOption`) are **abstract classes**. That is, they are never actually allocated via a call to `new`. Rather, they are placeholders for one of their subclasses. Possible subclasses for each abstract class used in the CSX Lite AST are listed in Table 2.

For type `fieldDeclsOption` we see that `fieldDeclsNode` and `nullFieldDeclsNode` are possible subclasses. This means that a subtree declared to be a `fieldDeclsOption` will really be rooted by either a `fieldDeclsNode` or `nullFieldDeclsNode`. We use "option" nodes when a subtree can either have real structure or may be empty (null).

There are a variety of nodes whose name begins with "null" (for example `nullFieldDeclsNode`). In ASTs we **never** use a null-valued reference to indicate an null subtree. Instead an actual null-node (like `nullFieldDeclsNode`) is allocated. This guarantees that we never need worry about null reference errors — subtrees, whether empty or not, are always allocated. As a convenience in processing CSX Lite ASTs, all tree nodes contain a Boolean-valued method named `isNull`. If you want to know if a node or subtree is null, you just ask! For example, if `root` references a CSX Lite AST, then `root.progDecls.isNull()` will tell you whether any declarations are present at the top (global) level.

Java class	Fields Used	Type of Fields	Represents
`csxLiteNode`	`progDecls` `progStmts`	`fieldDeclsOption` `stmtsOption`	CSX Lite program
`fieldDeclsNode`	`thisField` `moreFields`	`declNode` `fieldDeclsOption`	variable declaration sequence

`varDeclNode`	`varName`
	`varType`
	`initValue`
		`identNode`
		`typeNode`
		`exprOption`
			one variable declaration

`intTypeNode`			int type
`boolTypeNode`			bool type
`stmtsNode`	`thisStmt` `moreStmts`	`stmtNode` `stmtsOption`	statement sequence
`asgNode`	`target` `source`	`identNode` `exprNode`	assignment statement

Table 1. **Classes Used to Define AST Nodes in CSX Lite**

Java class	Fields Used	Type of Fields	Represents

`ifThenNode`	`condition`
	`thenPart`
	`elsePart`
		`exprNode`
		`stmtNode`
		`stmtOption`
			if statement

`blockNode`	`decls` `stmts`	`fieldDeclsOption` `stmtsOption`	block statement

`binaryOpNode`	`leftOperand`
	`rightOperand`
	`operatorCode`
		`exprNode`
		`exprNode`
		`int`
			binary operator expression

`identNode`	`idname`	`String`	identifier
`intLitNode`	`intval`	`int`	integer literal
`null nodes` `(many kinds)`	`none`		null program component

Table 1. Classes Used to Define AST Nodes in CSX Lite

Abstract AST Node	Subclasses	Abstract AST Node	Subclasses
`declNode`	`varDeclNode`	`exprOption`	`exprNode` `nullExprNode`

`exprNode`	`binaryOpNode`
	`identNode`
	`intLitNode`
		`fieldDeclsOption`	`fieldDeclsNode`
			`nullFieldDeclsNode`

`typeNode`	`intTypeNode`
	`boolTypeNode`
	`nullTypeNode`
		`stmtOption`	`stmtNode`
			`nullStmtNode`

`stmtNode`	`asgNode`
	`ifThenNode`
	`blockNode`
		`stmtsOption`	`stmtsNode`
			`nullStmtsNode`

Table 2 Abstract Classes Used in AST Nodes and Their Subclasses

Block-Structured Symbol Tables

In solving this assignment you will need to implement a **block-structured symbol table**. This is a data structure designed to support identifier declaration and lookup in block-structured languages. In most programming languages, identifiers can be declared globally (at the top level) or locally (in a limited scope). Scopes may nest. All declarations are placed in the nearest (innermost) scope containing the declaration. Access to declared identifiers is stepwise. First the innermost scope is considered. If a declaration is found, that is the one that is used. Otherwise, the next containing scope is examined, then the next, until the outermost (global) scope is examined. If no declaration is found in any of the containing scopes the identifier is considered undeclared and hence illegal. To simplify this assignment, you need not worry about undeclared identifiers or identifiers illegally redeclared in the same scope.

A block-structured symbol table is typically implemented by a sequence (or list) of symbol tables, one for each scope. In Java, a symbol table (for one scope) can be readily implemented using class `Hashtable` (see `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Hashtable.html`). (If you are more familiar with class `HashMap`, it also may be used.)

The original definition of Hashtable assumed values stored in a table were of type Object, which required explicit casting into the correct class. Since Java 1.5, a more convenient definition of Hashtable is available. Hashtable<K,V> defines a hash table in which all keys have class K and all table entries have class V. Explicit casting is not needed.

In your project solution, you will create a hash table (for global declarations) when the AST root node (a `csxLiteNode`) is first visited. Whenever a block is processed (a `blockNode`) a new hash table is allocated and linked to the head of a list of hash tables, one for each nested scope. When the AST for the block is fully processed, the hash table for its scope is removed (since the block's local declarations are no longer visible).

Declarations are always placed in the head hash table, representing the innermost scope. Method `put(key,value)` places a new entry into a `Hashtable`. When an identifier is used, we use method `get(key)` to see if the identifier is declared in the innermost scope. If it isn't, we do a `get` in the next hash table in the list. This is repeated until the identifier's declaration is found. Since we handle only valid programs, a declaration must be found in one of the hash tables.

What You Must Do

We will give you a working scanner and parser for CSX Lite. These components will build a valid AST. You will implement a method called `buildCrossReferences` in the various CSX Lite AST nodes. When called in the AST root node (a `csxLiteNode`) the method will traverse the AST gathering information on the declaration and use of all identifiers. This information will be stored in a data structure you design and implement. This data structure will implement a `toString` method. This method will display the cross-reference information you have gathered in string form. The string will be subdivided into lines, one for each declaration. Each line will be of the form

`linenumber: identifier(type): use`$_1$`,use`$_2$`,...`

`linenumber` is the source line number of the declaration. Note that each AST node has a field `linenum`, set by the scanner and parser, that shows where the information in the AST node was found. `identifier` is the name of the declared identifier. `type` is the declared type (either `int` or `bool`). `use`$_1$ is the source line number of the first use of this identifier. `use`$_2$ is the second use, etc.

Lines should be in ascending line number order. This will be easy to do if AST nodes are visited in a normal in-order traversal. An identifier may be used more than once on a source line. This may be represented as (e.g.) ...`,5,5,5`... or as ...`,5(3),`... (your choice). It is valid for a declared identifier to not be used anywhere.

Getting Started

To get you started, we will give you a complete working solution to a simpler CSX Lite analysis, declaration and use counts. For each scope (including the main program) identifier declarations and uses are counted. Thus for

```
{ int a;
  bool b;
  { int c;
    if (b)
      c = a+1;}
  a=a+1;
}
```

the analysis informs us

```
Scope 1 (at line 1): 2 declaration(s), 2 identifier use(s)
Scope 2 (at line 3): 1 declaration(s), 3 identifier use(s)
```

This analysis is inaccurate because it assumes (incorrectly) that all identifiers used in a scope are declared in that scope. Your analysis will use a block-structured symbol table to correctly match identifier uses with their proper declarations.

This program is available in two forms: as an Eclipse archive file (`cs536project1.zip`) and as a simple folder containing Java source files (in sub-folder `src`), libraries, an `ant` build file (see below) and a few test files.

The following Java source files are provided:

- **`ast.java`**
  This contains class definitions for all the CSX Lite AST nodes. You will need to examine all definitions of `countDeclsandUses` and update them with code to implement `buildCrossReferences`.
- **`ScopeInfo.java`**
  This contains class definitions for the data structures and utility routines used to gather declaration and use counts. You will need to update it to process and store identifier cross reference information.
- **`P1.java`**
  This contains the main program that scans and parses CSX Lite programs into AST form. It then calls `countDeclsandUses` in the AST root to begin the analysis. All you'll need to change here is the call at the root to begin cross reference analysis.
- **`scanner.java`**
  This is the scanner component, automatically generated by the **JLex** scanner generator. Don't change anything in it!
- **`parser.java`**

This is the parser component, automatically generated by the **Java CUP** parser generator. Don't change anything in it!

- **`sym.java`**
  This is a set of token definitions generated by the parser generator for use by the scanner generator. Don't change anything in it!

- **`Unparsing.java`**
  This class implements an **unparser** — a routine that walks an AST and generates a human-readable representation of the original source program. It is used to verify the correctness of the AST built by the scanner and parser. Don't change anything in it!

- **`Visitor.java`**
  This class allows the CSX Lite unparsing code to be organized using the **visitor pattern**. Using visitors, methods that conceptually walk the AST can be organized in different classes. This is important if we wish to avoid cluttering AST classes with large numbers of unrelated methods. Don't change anything in it!

Ant and Build Files

The software we have distributed to get you started includes a special file `build.xml` that is used by the `ant` program building tool. `Ant` is a Java-oriented version of the widely-used `make` utility found in Unix. Build files specify the steps needed to build a complete project. For your initial assignment such an elaborate tool isn't really needed — you can simply compile all the Java source files (into "`.class`" files) and then execute the `main` method in class `P1`. Still, you must be sure to get the details (like classpaths) right. More importantly, future projects will use tools other than the Java compiler (the JLex scanner generator, the JavaCUP parser generator and the Jasmin JVM assembler). Ant will be very useful in integrating these tools into the program build process.

At the command line level the command

```
ant
```

will recompile classes as needed after any changes you make. The command

```
ant compile
```

will do the same thing (`compile` is the default "target" in `build.xml`).

```
ant test
```

will do necessary recompilations and then test the program by running `P1.main` with two small test programs, `test.lite` and `biggertest.lite`.

```
ant clean
```

will remove all classfiles created by the compiler. It is useful when you want to force a "clean" and complete recompilation.

One of the best things about `ant` is that it is nicely integrated in Eclipse. If your project directory in Eclipse contains the special file `build.xml`, Eclipse will use it. You can activate ant by right-clicking on the build file. Select the menu item "Run as" and then the submenu item "Ant Build..." (be sure to include the "..."). A window appears, allowing you to select the build target you want

and then activate the build. You can also configure the "External tools" button (on the top command line of the Eclipse window).

Extra Credit

You should not consider extra credit work until your project solution is running correctly. Still, you may be bothered by the simplifying assumption that a program contains no declaration errors. Two common programming errors are illegally redeclaring an identifier in the same scope and using an identifier that is undeclared. Both are fairly easy to handle.

An identifier that is illegally redeclared has a name and location. But its type is neither `int` nor `bool`. Rather we can consider it to be a special type, "illegal." The redeclaration is not placed in the symbol table and it is never used (because it is "hidden" by the earlier valid declaration). Still, it is useful to remind the programmer that there is an error and show where it is.

Undeclared identifiers are also common — some programmers write code first and add declarations later! When an undeclared identifier is found, it can be placed in a separate table. There is no declared type or declaration site, but uses can be chained and later presented to the programmer. When the needed declaration is later added, it is easy to inspect places where it is used and verify that the declaration is appropriate.

What To Hand In

Create a folder (directory) and name it using your first and last name (e.g., `CharlesFischer`). Copy into this folder all the Java source files you changed or added. If you changed the `build.xml` we provided, include your version (so that we can properly build and test your solution). You should include a `README` file to hold external documentation. We'll run your program on a variety of our own test programs. Do not hand in any class files; we'll create them as needed using your source files and `build.xml`. Upload this handin folder to the `cs 536 project 1` Dropbox folder in `learn@uw` (`https://learnuw.wisc.edu`). Partners should submit only *one* solution. The other partner should submit only a README file identifying the partnership.

When your program begins execution it should print out your full name and student ID number. It should also print out the name of the file being analyzed. We will grade your program on the basis of the correct operation of your cross-reference generator. The quality of your documentation is also important. Make sure that you provide both external documentation (in the `README` file) and internal documentation (in the source files). It should be easy for the grader to understand the organization and structure of your program. We may exact significant penalties if we find your program poorly documented or difficult to understand.