To Load Proj #1 Startup:

- Down Load Proj 1 zip from Web Page

In Eclipse:

File/Import

General/Existing Proj

Select Archive File

ANT –

A PROGRAM BUILD TOOL

BUILD.XML CONTAINS RULES ON HOW TO BUILD A PROGRAM.

ECLIPSE CAN USE BUILD FILES

CAN ALSO CALL ANT EXTERNALLY:

ANT COMPILE OR
ANT TEST

RIGHT-CLICK ON BUILD.XML

RUN AS / ANT BUILD ...

SELECT & ORDER TARGETS

RUN

CAN USE TOP LINE "SUITCASE" ICON

TO GET STARTED WE GIVE YOU A WORKING SOLUTION TO A SIMPLER PROBLEM:

COUNT DECLS AND USES PER SCOPE (W/O SCOPING RULES!)

```
{ INT a; INT b;
  ...
  { INT c;
    a = b+c;}}
```

SCOPE 1 (LINE 1): 2 DECLS, 0 USES
SCOPE 2 (LINE 3): 1 DECL, 3 USES

↑ IGNORES SCOPING!

ABSTRACT CLASSES

NEVER ALLOCATED (VIA NEW)

PLACE HOLDER FOR SET
OF SUBCLASSES

NULL AST NODES
USED TO AVOID
NULL REFERENCES
IN TREES

```
 1 import java.io.*;
 3 // CS536 Spring 2015, project 1 (count identifier definitions and uses on
 4 //  a per scope basis for CSX Lite programs).
 5 // A scope is the entire program, or a block (delimited by "{" and "}").
 6 // Scopes may nest.
 7
 8 // In your solution of this project you will need to replace the call
 9 //  to countDeclsAndUses (at the bottom of the program) with a call
10 //  to the method you write to do a cross-reference analysis
11
12 public class P1 {
13   public static void
14   main(String args[]) throws java.io.IOException {
15
16     // Test that a  program name appears on the command line.
17     if (args.length != 1) {
18             System.out.println(
19             "Error: Input file must be named on command line." );
20         System.exit(-1);
21     }
22
23
24     java.io.FileInputStream yyin = null;
25
26         // Open the file named on the command line. yyin will reference
   it.
27     try {
28         yyin = new java.io.FileInputStream(args[0]);
29     } catch (FileNotFoundException notFound){
30             System.out.println ("Error: unable to open input file.");
31         System.exit(-1);
32     }
33
34     Scanner.init(yyin); // Initialize Scanner class that will read and
   scan yyin
35
36     //Create a parser that will parse the tokens returned by the scanner
37     parser csxParser = new parser();
38
39     Symbol root=null;
40
41     // Call the parser. If the parse is successful, root will point
```

```
42      //  to the root of the AST (abstract syntax tree) the papser builds
43      try {
44      root = csxParser.parse(); // do the parse
45      System.out.println ("CSX Lite program parsed correctly.");
46
47      } catch (Exception e) {
48      System.out.println ("Compilation terminated due to syntax errors.");
49      System.exit(0);
50      }
51
52      // Print out a listing of the progeam just parsed. This is done using
  an
53      //  "unparser". A unparser reverses the parsing process, transforming
  an AST
54      //  for a program into a textual representation of the program. This
  mechanism
55      //  is useful in verifying that the AST for a program is correct.
56      // The unparsing is done using the visitor mechanism, which we will
  discuss in class.
57      System.out.println ("Here is its unparsing:");
58      Unparsing unparse = new Unparsing();
59       unparse.visit((csxLiteNode) root.value,0);
60
61      //To do the identifier declaration and use analysis, we call
  countDeclsAndUses
62      // in the root node of the AST. This method creates the necessary
  data structures
63      // and walks the AST, calling countDeclsAndUses as necessary in
  various subtrees.
64      // When the entire AST is traversed (and analyzed), the data
  structures built are
65      // converted to textual form and returned to the caller.
66
67      System.out.println ("\n\nHere is an analysis of identifier
  declarations and uses for "+
68          args[0]+ ":");
69      System.out.println (((csxLiteNode)root.value).countDeclsAndUses());//
  Change needed here
70
71
72      return;
73      }
```

ScopeInfo.java

```
1 //import java.io.*;
2
3 // ScopeInfo is the primary data structure used to hold information on
  identifier
4 //  declarations and uses. Each ScopeInfo node contains information for
  one scope.
5 // ScopeInfo nodes are linked together, in order of appearance of scopes
  in the CSX Lite
6 //  program. The entire list of nodes contains count information for the
  entire program.
7
8 // Note that ScopeInfo is NOT a block-structured symbol table because:
9 //   (1) It stores information on a per-scope rather than per-identifier
  basis
10 //   (2) It incorrectly assumes that all identifier uses in a scope
  belong to that scope.
11 //       In a modern programming languages the use of an identifier
  belongs to the scope
12 //        containing its declaration. That is why you will need to extend
  (or replace)
13 //        this data structure with on that uses a block-structured symbol
  table.
14
15 public class ScopeInfo {
16     int number; // sequence number of this scope (starting at 1)
17     int line;   // Source line this scope begins at
18     int declsCount; // Number of declarations in this scope
19     int usesCount;  // Number of identifier uses in this scope
20     ScopeInfo next; // Next ScopeInfo node (in list of all scopes found
  and processed)
21
22     // A useful constructor
23     ScopeInfo(int num, int l){
24         number=num;
25         line=l;
26         declsCount=0;
27         usesCount=0;
28         next=null;
29     }
30
31     // A useful constructor
32     ScopeInfo(int l){
```

```
33          number=0;
34          line=l;
35          declsCount=0;
36          usesCount=0;
37          next=null;
38      }
39
40      // This method converts a list of ScopeInfo nodes into string form.
41      // It controls what the caller sees as the result of the analysis
  after
42      //  the ScopeInfo list is built.
43      public String toString() {
44          String thisLine="Scope "+number+ " (at line "+line+"):
  "+declsCount+" declaration(s), "+
45                              usesCount+" identifier use(s)"+"\n";
46          if (next == null)
47              return thisLine;
48          else return thisLine+next.toString();
49         }
50
51      // Method append follows list to its end. Then it appends newNode as
  the new end of list.
52      // It also sets number in newNode to be one more than number in the
  previous
53      //  end of list node. Thus append numbers list nodes in sequence as
54      //  the list is built.
55      public static void append(ScopeInfo list, ScopeInfo newNode){
56          while (list.next != null){
57              list=list.next;
58          }
59          list.next=newNode;
60          newNode.number=list.number+1;
61
62      }
63
64  //  This is used only to test this class (during development or
  modification).
65      public static void  main(String args[]) {
66          ScopeInfo test = new ScopeInfo(1,1);
67          System.out.println("Begin test of ScopeInfo");
68          append(test,new ScopeInfo(2));
69          append(test,new ScopeInfo(3));
```

```
70          System.out.println(test);
71          System.out.println("End test of ScopeInfo");
72      }
73
74 }
75
```

```
 1 /*************************************************************
 2  *
 3  * After scanning  and parsing, the structure and content of a program
   is represented as an
 4  * Abstract Syntax Tree (AST).
 5  * The root of the AST represents the entire program. Subtrees represent
   various
 6  * components, like declarations and statements.
 7  * Program translation and analysis is done by recursively walking the
   AST, starting
 8  * at the root.
 9  * CSX will use a variety of AST nodes since it contains a variety of
   structures (declarations,
10  * methods, statements, expressions, etc.).
11  * The AST nodes defined here represent CSX Lite, a small subset of CSX.
   Hence many fewer
12  * nodes are needed.
13  *
14  * The analysis implemented here counts the number of identifier
   declarations and uses
15  * on a per scope basis. The entire program is one scope. A block
   (rooted by a blockNode)
16  * is a local scope (delimited in the source program by a "{" and "}").
17  * The method countDeclsAndUses implements this analysis. Each AST node
   has a definition of this
18  * method. It may be an explicit definition intended especially for one
   particular node.
19  * If an AST node has no local definition of countDeclsAndUses it
   inherits a definition from its
20  * parent class. The class ASTNode (which is the ancestor of all AST
   nodes) has a default definition
21  * of countDeclsAndUses. The definition is null (does nothing).
22  *
23  */
24 // abstract superclass; only subclasses are actually created
25 abstract class ASTNode {
26
27     public final int    linenum;
28     public final intcolnum;
29
30     ASTNode(){linenum=-1;colnum=-1;}
31     ASTNode(int l,int c){linenum=l;colnum=c;}
```

```
32    boolean   isNull(){return false;}; // Is this node null?
33
34    abstract void accept(Visitor v, int indent);// Will be defined in
  sub-classes
35
36    // default action on an AST node is to record no declarations and no
  identifier uses
37     void countDeclsAndUses(ScopeInfo currentScope){
38        return;
39    }
40
41 };
42
43
44
45 // This node is used to root only CSXlite programs
46 class csxLiteNode extends ASTNode {
47
48    public final fieldDeclsOption   progDecls;
49    public final stmtsOption    progStmts;
50    private ScopeInfo        scopeList;
51
52    csxLiteNode(fieldDeclsOption decls, stmtsOption stmts, int line, int
  col){
53        super(line,col);
54        progDecls=decls;
55        progStmts=stmts;
56        scopeList=null;
57    };
58
59
60    void accept(Visitor u, int indent){ u.visit(this,indent); }
61
62    // This method begins the count declarations and uses analysis.
63    //  It first creates a ScopeInfo node for the entire program.
64    //  It then passes this ScopeInfo node to the declarations subtree
  and then
65    //   the statements subtree. Visiting these two subtrees causes all
  identifier uses and
66    //     declarations to be recognized and recorded in the list rooted
  by the ScopeInfo node.
67    //  Finally, the information stored in the ScopeInfo list is
```

```
  converted to string form
68      //   and returned to the caller of the analysis.
69
70       String countDeclsAndUses(){
71           scopeList = new ScopeInfo(1,linenum);
72           progDecls.countDeclsAndUses(scopeList);
73           progStmts.countDeclsAndUses(scopeList);
74           return scopeList.toString();
75       }
76
77  };
78
79  abstract class fieldDeclsOption extends ASTNode{
80      fieldDeclsOption(int line,int column){
81          super(line,column);
82      }
83      fieldDeclsOption(){ super(); }
84  };
85
86  class fieldDeclsNode extends fieldDeclsOption {
87
88      public final declNode       thisField;
89      public final fieldDeclsOption   moreFields;
90
91      fieldDeclsNode(declNode d, fieldDeclsOption f, int line, int col){
92          super(line,col);
93          thisField=d;
94          moreFields=f;
95      }
96
97      static nullFieldDeclsNode NULL = new nullFieldDeclsNode();
98
99      void accept(Visitor u, int indent){ u.visit(this,indent);}
100
101     void countDeclsAndUses(ScopeInfo currentScope){
102         thisField.countDeclsAndUses(currentScope);
103         moreFields.countDeclsAndUses(currentScope);
104         return;
105     }
106 };
107
108 class nullFieldDeclsNode extends fieldDeclsOption {
```

```
109
110     nullFieldDeclsNode(){};
111
112     boolean   isNull(){return true;};
113
114     void accept(Visitor u, int indent){ u.visit(this,indent);}
115
116     void countDeclsAndUses(ScopeInfo currentScope){
117             return;
118         }
119 };
120
121 // abstract superclass; only subclasses are actually created
122 abstract class declNode extends ASTNode {
123     declNode(){super();};
124     declNode(int l,int c){super(l,c);};
125 };
126
127
128 class varDeclNode extends declNode {
129
130     public finalidentNode   varName;
131     public finaltypeNode    varType;
132     public finalexprOption  initValue;
133
134     varDeclNode(identNode id, typeNode t, exprOption e,
135             int line, int col){
136         super(line,col);
137         varName=id;
138         varType=t;
139         initValue=e;
140     }
141
142     void accept(Visitor u, int indent){ u.visit(this,indent);}
143
144     // This node represents a variable declaration, so we increment the
    declarations
145     //  count by 1
146     void countDeclsAndUses(ScopeInfo currentScope){
147         currentScope.declsCount+=1;
148     }
149
```

```
150 };
151
152 abstract class typeNode extends ASTNode {
153 // abstract superclass; only subclasses are actually created
154     typeNode(){super();};
155     typeNode(int l,int c){super(l,c);};
156     static nullTypeNode NULL = new nullTypeNode();
157 };
158
159 class nullTypeNode extends typeNode {
160
161     nullTypeNode(){};
162
163     boolean   isNull(){return true;};
164
165     void accept(Visitor u, int indent){ u.visit(this,indent); }
166 };
167
168
169 class intTypeNode extends typeNode {
170     intTypeNode(int line, int col){
171         super(line,col);
172     }
173
174     void accept(Visitor u, int indent){ u.visit(this,indent); }
175 };
176
177
178 class boolTypeNode extends typeNode {
179     boolTypeNode(int line, int col){
180         super(line,col);
181     }
182
183     void accept(Visitor u, int indent){ u.visit(this,indent); }
184 };
185
186 //abstract superclass; only subclasses are actually created
187 abstract class stmtOption extends ASTNode {
188     stmtOption(){super();};
189     stmtOption(int l,int c){super(l,c);};
190     //static nullStmtNode NULL = new nullStmtNode();
191 };
```

```
192
193 // abstract superclass; only subclasses are actually created
194 abstract class stmtNode extends stmtOption {
195     stmtNode(){super();};
196     stmtNode(int l,int c){super(l,c);};
197     static nullStmtNode NULL = new nullStmtNode();
198 };
199
200 class nullStmtNode extends stmtOption {
201     nullStmtNode(){};
202     boolean   isNull(){return true;};
203     void accept(Visitor u, int indent){ u.visit(this,indent);}
204     void countDeclsAndUses(ScopeInfo currentScope){return;}
205 };
206
207 abstract class stmtsOption extends ASTNode{
208     stmtsOption(int line,int column){
209         super(line,column);
210     }
211     stmtsOption(){ super(); }
212 };
213
214 class stmtsNode extends stmtsOption {
215     public final stmtNode       thisStmt;
216     public final stmtsOption    moreStmts;
217
218     stmtsNode(stmtNode stmt, stmtsOption stmts, int line, int col){
219         super(line,col);
220         thisStmt=stmt;
221         moreStmts=stmts;
222     };
223
224     static nullStmtsNode NULL = new nullStmtsNode();
225
226     void accept(Visitor u, int indent){ u.visit(this,indent);}
227
228     void countDeclsAndUses(ScopeInfo currentScope){
229      // Count decls and uses in both subtrees:
230          thisStmt.countDeclsAndUses(currentScope);
231          moreStmts.countDeclsAndUses(currentScope);
232         }
233 };
```

```
234
235
236 class nullStmtsNode extends stmtsOption {
237     nullStmtsNode(){};
238     boolean   isNull(){return true;};
239
240     void accept(Visitor u, int indent){ u.visit(this,indent);}
241
242     void countDeclsAndUses(ScopeInfo currentScope){return;}
243
244 };
245
246 class asgNode extends stmtNode {
247
248     public final identNode  target;
249     public final exprNode   source;
250
251     asgNode(identNode n, exprNode e, int line, int col){
252         super(line,col);
253         target=n;
254         source=e;
255     };
256
257     void accept(Visitor u, int indent){ u.visit(this,indent);}
258
259     void countDeclsAndUses(ScopeInfo currentScope){
260         // The target of the assign counts as 1 use
261         currentScope.usesCount +=1;
262         // Visit the source expression to include the identifiers in it
263         source.countDeclsAndUses(currentScope);
264         }
265 };
266
267
268 class ifThenNode extends stmtNode {
269
270     public final exprNode       condition;
271     public final stmtNode       thenPart;
272     public final stmtOption elsePart;
273
274     ifThenNode(exprNode e, stmtNode s1, stmtOption s2, int line, int
    col){
```

```
275         super(line,col);
276         condition=e;
277         thenPart=s1;
278         elsePart=s2;
279     };
280
281     void accept(Visitor u, int indent){ u.visit(this,indent);}
282
283     void countDeclsAndUses(ScopeInfo currentScope){
284         // Count identifier uses in control expression and then
    statement.
285         // In CSX Lite the else statement is always null
286         condition.countDeclsAndUses(currentScope);
287         thenPart.countDeclsAndUses(currentScope);
288         }
289 };
290
291
292 class blockNode extends stmtNode {
293
294     public final fieldDeclsOption   decls;
295     public final stmtsOption    stmts;
296
297     blockNode(fieldDeclsOption f, stmtsOption s, int line, int col){
298         super(line,col);
299         decls=f;
300         stmts=s;
301     }
302
303      void accept(Visitor u, int indent){ u.visit(this,indent);}
304
305      void countDeclsAndUses(ScopeInfo currentScope){
306         /* A block opens a new scope, so a new ScopeInfo node is
    created.
307            It is appended to the end of the ScopeInfo list.
308            The new scope is used to record local declarations and uses
    in the block
309         */
310          ScopeInfo  localScope = new ScopeInfo(linenum);
311          ScopeInfo.append(currentScope,localScope);
312          decls.countDeclsAndUses(localScope);
313          stmts.countDeclsAndUses(localScope);
```

NEW SCOPE!

```
314     }
315 };
316
317
318 //abstract superclass; only subclasses are actually created
319 abstract class exprOption extends ASTNode {
320     exprOption(){super();};
321     exprOption(int l,int c){super(l,c);};
322     //static nullStmtNode NULL = new nullStmtNode();
323 };
324
325 // abstract superclass; only subclasses are actually created
326 abstract class exprNode extends exprOption {
327     exprNode(){super();};
328     exprNode(int l,int c){super(l,c);};
329     static nullExprNode NULL = new nullExprNode();
330 };
331
332 class nullExprNode extends exprOption {
333     nullExprNode(){super();};
334     boolean   isNull(){return true;};
335     void accept(Visitor u, int indent){}
336 };
337
338 class binaryOpNode extends exprNode {
339
340     public final exprNode   leftOperand;
341     public final exprNode   rightOperand;
342     public final intoperatorCode; // Token code of the operator
343
344     binaryOpNode(exprNode e1, int op, exprNode e2, int line, int col){
345         super(line,col);
346         operatorCode=op;
347         leftOperand=e1;
348         rightOperand=e2;
349     };
350
351     void accept(Visitor u, int indent){ u.visit(this,indent);}
352
353     // Count identifier uses in left and right operands
354     void countDeclsAndUses(ScopeInfo currentScope){
355             leftOperand.countDeclsAndUses(currentScope);
```

```
356             rightOperand.countDeclsAndUses(currentScope);
357         }
358 };
359
360 class identNode extends exprNode {
361
362     public final String idname;
363
364     identNode(String identname, int line, int col){
365         super(line,col);
366         idname   = identname;
367     };
368
369     void accept(Visitor u, int indent){ u.visit(this,indent);}
370
371     //One identifier used here:
372     void countDeclsAndUses(ScopeInfo currentScope){
373             currentScope.usesCount+=1;
374         }
375 };
376
377
378 class intLitNode extends exprNode {
379     public final int    intval;
380     intLitNode(int val, int line, int col){
381         super(line,col);
382         intval=val;
383     }
384
385     void accept(Visitor u, int indent){ u.visit(this,indent);}
386 };
387
```