10

Semantic Analysis: Control Structures and Subroutines

10.1 Semantic Analysis for Control Structures

Control structures are an essential component of all programming languages. With control structures a programmer can combine individual statements and expressions to form an unlimited number of specialized program constructs.

Some languages constructs, like the if, switch and case statement, provide for conditional execution of selected statements. Others, like while, do and for loops, provide for iteration (or repetition) of the body of a looping construct. Still other statements, like the break, continue, throw, return and goto statements force program execution to depart from normal sequential execution of statements.

The exact form of control structures differs in various programming languages. For example, in C, C++ and Java, if statements don't use a `then` keyword; in Pascal, ML Ada, and many other languages, `then` is required.

Minor syntactic differences are unimportant when semantic analysis is performed by a compiler. We can ignore syntactic differences by analyzing an **abstract syntax tree**. Recall that an abstract syntax tree (AST) represents the essential structure of a construct while hiding minor variations in source level representation. Using an AST, we can discuss the semantic analysis of fundamental constructs like conditional and looping statements without any concern about exactly how they are represented at the source level.

An important issue in analyzing Java control structures is **reachability**. Java requires that unreachable statements be detected during semantic analysis, with suitable error messages generated. For example, in the statement sequence

```
...; return; a=a+1; ...
```

the assignment statement must be marked as **unreachable** during semantic analysis.

Reachability analysis is **conservative**. In general, determining whether a given statement can be reached is very difficult. In fact it is impossible! Theoretical computer scientists have proven that it is **undecidable** whether a given statement is ever executed, even when we know in advance all the data a program will access (reachability is a variant of the famous halting problem first discussed in [Turing 1936]).

Because our analyses will be conservative, we will not detect all occurrences of unreachable statements. However, the statements we recognize as unreachable will definitely be erroneous, so our analysis will certainly be useful. In fact, even in languages like C and C++, which don't require reachability analysis, we can still produce useful warnings about unreachable statements that may well be erroneous.

To detect unreachable statements during semantic analysis, we'll add two boolean-valued fields to the ASTs that represent statements and statement lists. The first, isReachable, marks whether a statement or statement list is considered reachable. We'll issue an error message for any non-null statement or statement list for which isReachable is false.

The second field, terminatesNormally, marks whether a given statement or statement list is expected to terminate normally. A statement that terminates normally will continue execution "normally" with the next statement that follows. Some statements (like a break, continue or return) may force execution to proceed to a statement other than the normal successor statement. These statements are marked with terminatesNormally = false. Similarly, a loop may never terminate its iteration (e.g., `for(;;) {a=a+1;}`). Loops that don't terminate (using a conservative analysis) also have their terminatesNormally flag set to false.

The isReachable and terminatesNormally fields are set according to the following rules:

- If isReachable is true for a statement list, then it is also true for the first statement in the list.
- If terminatesNormally is false for the last statement in a statement list, then it is also false for the whole statement list.
- The statement list that comprises the body of a method, constructor, or static initializer is always considered reachable (its isReachable value is true).
- A local variable declaration or an expression statement (assignment, method call, heap allocation, variable increment or decrement) always has terminatesNormally set to true (even if the statement has isReachable set to false). (This is done so that all the statements following an unreachable statement don't generate error messages).

- A null statement or statement list never generates an error message if its isReachable field is false. Rather, the isReachable value is propagated to the statement's (or statement list's) successor.
- If a statement has a predecessor (it is not the first statement in a list), then its isReachable value is equal to its predecessor's terminatesNormally value. That is, a statement is reachable if and only if its predecessor terminates normally.

As an example, consider the following method body

```
void example() {
        int a; a++; return; ; a=10; a=20; }
```

The method body is considered reachable, and thus so is the declaration of `a`. This declaration and the increment of `a` complete normally, but the return does not (see Section 10.1.4). The null statement following the return is unreachable, and propagates this fact to the assignment of 10, which receives an error message. This assignment terminates normally, so its successor is considered reachable.

In the succeeding sections we will study the semantic analysis of the control structures of Java. Included in this analysis will be whether the statement in question terminates normally. Thus we will set the terminatesNormally value for each kind of control statement, and from this successor statements will set their isReachable field.

Interestingly, although we expect most statements and statement lists to terminate normally, in functions (methods that return a non-void value) we **require** the method body to terminate abnormally. Because a function must return a value, it cannot return by "falling though" to the end of the function. It must execute a return of some value or throw an exception. These both terminate abnormally, so the method body must also terminate abnormally. After the body of a function is analyzed, the terminatesNormally value of the statement list comprising the body is checked; if it is not false, an error message ("Function body must exist with a return or throw statement") is generated.

In analyzing expressions and statements, we must be aware of the fact that the constructs may throw an exception rather than terminate normally. Java requires that all checked exceptions be accounted for. That is, if a checked exception is thrown (see Section 10.1.6), then it **must** either be caught an a catch block or listed in a method's throw list.

To enforce this rule, we'll accumulate the checked exceptions that can be generated by a given construct. Each AST node that contains an expression or statement will have a throwsSet field. This field will reference a linked list of throwItem nodes. Each throwItem has fields type (the exception type that is thrown), and next, a link to the next throwItem. This list, maintained as a set (so that duplicate exceptions are removed), will be propagated as ASTs are analyzed. It will be used when catch blocks and methods and constructors are analyzed.

10.1.1 If Statements

The AST corresponding to an if statement is shown in Figure 10.1. An IfNode has three subtrees, corresponding to the condition controlling the if, the then statements and the else statements. The semantic rules for an if statement are simple—the condition must be a valid boolean-valued expression and the then and else statements must be semantically valid. An if statement terminates normally if either its then part or its else part terminates normally. Since null statements terminate trivially, an if-then statement (with a null else part) always terminates normally.

The Semantics method for an if is shown in Figure 10.2.

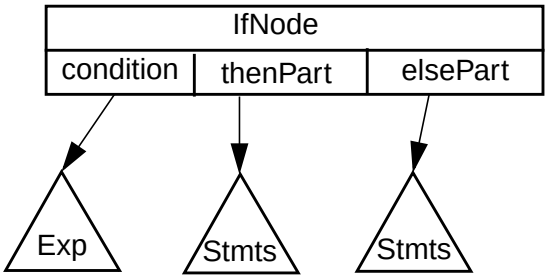

Figure 10.1 Abstract Syntax Tree for an if statement

Since condition must be an expression, we first call ExprSemantics to determine the expression's type. We expect a boolean type. It may happen that the condition itself contains a semantic error (e.g., an undeclared identifier or an invalid expression). In this case, ExprSemantics returns the special type errorType, which indicates that further analysis of condition is unnecessary (since we've already marked the condition as erroneous). Any type other than boolean or errorType causes an "Illegal conditional expression" error message to be produced.

Besides the condition subtree, the thenPart and elsePart ASTs must be checked. This is done in lines 4 to 8 of the algorithm. The if statement is marked as completing normally if either the then or else part completes normally. The set of exceptions that an if statement can throw is the set union of throws potentially generated in the condition expression and the then and else statements. Recall that if an if statement has no else part, the elsePart AST is a NullNode, which is trivially correct.

As an example, consider the following statement

```
if (b) a=1; else a=2;
```

We first check the condition expression, `b`, which must produce a boolean value. Then the then and else parts are checked; these must be valid statements. Since assignment statements always complete normally, so does the if statement.

The modularity of our AST formulation is apparent here. The checks applied to control expression `b` are the same used for *all* expressions. Similarly, the checks applied to the then and else statements are those applied to *all* statements. Note

```
IfNode.Semantics( )
1.  ConditionType ← condition.ExprSemantics()
2.  if ConditionType ≠ boolean and ConditionType ≠ errorType
3.      then GenerateErrorMessage("Illegal conditional expression")
4.  thenPart.isReachable ← elsePart.isReachable ← true
5.  thenPart.Semantics()
6.  elsePart.Semantics()
7.  terminatesNormally ←
            thenPart.terminatesNormally or elsePart.terminatesNormally
8.  throwsSet ← union(condition.throwsSet, thenPart.throwsSet, elsePart.throwsSet)
```

Figure 10.2 Semantic Checks for an If Statement

too that nested if statements case no difficulties—the same checks are applied each time an IfNode is encountered.

10.1.2 While, Do and Repeat Loops

The AST corresponding to a while statement is shown in Figure 10.3. A WhileNode has two subtrees, corresponding to the condition controlling the loop and the loop body. The semantic rules for a while statement are simple. The condition must be a valid boolean-valued expression and the loop body must be semantically valid.

Reachability analysis must consider the special case that the control expression is a constant. If the control expression is false, then the statement list comprising the loop body is marked as unreachable. If the control expression is true, the while loop is marked as abnormally terminating (because of an infinite loop). It may be that the loop body contains a reachable break statement. If this is the case, semantic processing of the break will reset the loop's terminatesNormally field to true. If the control expression is non-constant, the loop is marked as terminating normally. The exceptions potentially thrown are those generated by the loop control condition and the loop body.

The method EvaluateConstExpr traverses an expression AST that is semantically valid to determine whether it represents a constant-valued expression. If the AST is a constant expression, EvaluateConstExpr returns its value; otherwise it returns null. The Semantics method for a while loop is shown in Figure 10.4.

As an example, consider

```
while (i >= 0) {
                    a[i--] = 0; }
```

The control expression, `i >= 0`, is first checked to see if it is a valid boolean-valued expression. Since this expression is non-constant, the loop body is assumed reachable and the loop is marked as terminating normally. Then the loop body is checked for possible semantic errors.

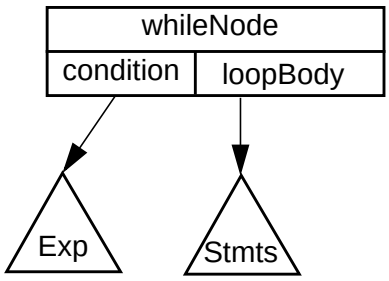

Figure 10.3 Abstract Syntax Tree for a While Statement

```
WhileNode.Semantics( )
 1.   ConditionType ← condition.ExprSemantics()
 2.   terminatesNormally ← true
 3.   loopBody.isReachable ← true
 4.   if ConditionType = boolean
 5.        then ConditionValue ← condition.EvaluateConstExpr()
 6.             if ConditionValue = true
 7.                  then terminatesNormally ← false
 8.             elsif ConditionValue = false
 9.                  then loopBody.isReachable ← false
10.   elsif ConditionType ≠ errorType
11.        then GenerateErrorMessage("Illegal conditional expression")
12.   loopBody.Semantics()
13.   throwsSet ← union(condition.throwsSet, loopBody.throwsSet)
```

Figure 10.4 Semantic Checks a for While Statement

Do While and Repeat Loops. Java, C and C++ contain a variant of the while loop—the do while loop. A do while loop is just a while loop that evaluates and tests its termination condition *after* executing the loop body rather than before. The semantic rules for a do while are almost identical to those of a while. Since the loop body always executes at least once, the special case of a false control expression can be ignored. For non-constant loop control expressions, a do while loop terminates normally if the loop body does. Assuming the same AST structure as a while loop, the semantic processing appropriate for a do while loop is shown in Figure 10.5.

A number of languages, including Pascal and Modula 3, contain a repeat until loop. This is essentially a do while loop except for the fact that the loop is terminated when the control condition becomes false rather than true. The semantic rules of a repeat until loop are almost identical to those of the do while loop. The only change is that the special case of a non-terminating loop occurs when the control expression is false rather than true.

```
DoWhileNode.Semantics( )
 1.   loopBody.isReachable ← true
 2.   loopBody.Semantics()
 3.   ConditionType ← condition.ExprSemantics()
 4.   if ConditionType = boolean
 5.        then ConditionValue ← condition.EvaluateConstExpr()
 6.             if ConditionValue = true
 7.                  then terminatesNormally ← false
 8.                  else terminatesNormally ← loopBody.terminatesNormally
 9.   elsif ConditionType ≠ errorType
10.        then GenerateErrorMessage("Illegal conditional expression")
11.   throwsSet ← union(condition.throwsSet, loopBody.throwsSet)
```

Figure 10.5 Semantic Checks a for Do While Statement

10.1.3 For Loops

For loops are normally used to step an index variable through a range of values. However, for loops in C, C++ and Java, are really a generalization of while loops. Consider the AST for a for loop, as shown in Figure 10.6.

As was the case the while loops, the for loop's AST contains subtrees corresponding to the loop's termination condition (condition) and its body (loopBody). In addition it contains ASTs corresponding to the loop's initialization (initializer) and its end-of-loop increment (increment).

There are a few differences that must be properly handled. Unlike the while loop, the for loop's termination condition is optional. (This allows "do forever" loops of the form `for (;;) {...}`). In C++ and Java, an index local to the for loop may be declared, so a new symbol table name scope must be opened, and then closed, during semantic analysis.

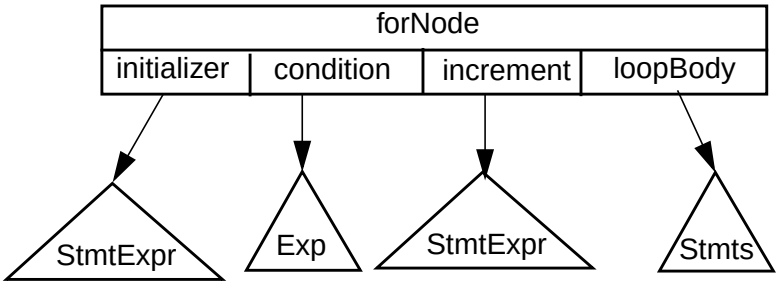

Figure 10.6 Abstract Syntax Tree for a For Loop

Reachability analysis is similar to that performed for while loops. The loop initializer is a declaration or statement expression; the loop increment is a statement expression; the termination condition is an expression. All of these are marked as terminating normally.

A null termination condition, or a constant expression equal to true, represent a non-terminating loop. The for loop is marked as not terminating normally (though a break within the loop body may change this when it is analyzed). A termination condition that is a constant expression equal to false causes the loop body to be marked as unreachable. If the control expression is non-null and non-constant, the loop is marked as terminating normally.

The Semantics method for a for loop is shown in Figure 10.7.

A new name scope is opened in case a loop index is declared in the initializer AST. The initializer is checked, allowing local index declarations to be processed. The increment is checked next. If condition is not a NullNode, it is type-checked, with a boolean (or errorType) required. The special cases of a null condition and a constant-valued condition are considered, with normal termination and reachability of the loop body updated appropriately. The loopBody AST is then checked. Finally, the name scope associated with for loop is closed and the loop's throwsSet is computed.

```
forNode.Semantics( )
 1.  OpenNewScope()
 2.  initializer.Semantics()
 3.  increment.Semantics()
 4.  terminatesNormally ← true
 5.  loopBody.isReachable ← true
 6.  if condition ≠ NullNode
 7.      then ConditionType ← condition.ExprSemantics()
 8.           if ConditionType = boolean
 9.               then ConditionValue ← condition.EvaluateConstExpr()
10.                    if ConditionValue = true
11.                        then terminatesNormally ← false
12.                    elsif ConditionValue = false
13.                        then loopBody.isReachable ← false
14.           elsif ConditionType ≠ errorType
15.               then GenerateErrorMessage("Illegal termination expression")
16.      else terminatesNormally ← false
17.  loopBody.Semantics()
18.  Close Scope()
19.  throwsSet ← union(initializer.throwsSet, condition.throwsSet,
                        increment.throwsSet, loopBody.throwsSet)
```

Figure 10.7 Semantic Checks a for For Loop

As an example, consider the following for loop

```
for (int i=0; i < 10; i++)
     a[i] = 0;
```

First a new name scope is created for the loop. When the declaration of `i` in the initializer AST is processed, it is placed in this new scope (since all new declarations are placed in the innermost open scope). Thus the references to `i` in the con-

dition, increment and loopBody ASTs properly reference the newly declared loop index `i`. Since the loop termination condition is boolean-valued and con-constant, the loop is marked as terminating normally, and the loop body is considered reachable. At the end of semantic checking, the scope containing `i` is closed, guaranteeing that not subsequent references to the loop index will be allowed.

A number of languages, including Fortran, Pascal, Ada, Modula 2, and Modula 3, contain a more restrictive form of for loop. Typically, a variable is identified as the "loop index." Initial and final index values are defined, and sometimes an increment value is specified. For example, in Pascal a for loop is of the form

```
for id := intialVal to finalVal do
     loopBody
```

The loop index, `id`, must already be declared and must be a scalar type (integer or enumeration). The `initialVal` and `finalVal` expressions must be semantically valid and have the same type as the loop index. Finally, the loop index may not be changed within the `loopBody`. This can be enforced by marking `id`'s declaration as "constant" or "read only" while `loopBody` is being analyzed.

10.1.4 Break, Continue, Return and Goto Statements

Java contains no goto statement. It does, however, include break and continue statements which are restricted forms of a goto, as well as a return statement. We'll consider the continue statement first.

Continue Statements. Like the continue statement found in C and C++, Java's continue statement attempts to "continue with" the next iteration of a while, do or for loop. That is, it transfers control to the bottom of a loop where the loop index is iterated (in a for loop) and the termination condition is evaluated and tested.

A continue may only appear within a loop; this must be verified during semantic analysis. Unlike C and C++ a loop label may be specified in a continue statement. An unlabeled continue references the innermost for, while or do loop in which it is contained. A labeled continue references the enclosing loop that has the corresponding label. Again, semantic analysis must verify that an enclosing loop with the proper label exists.

Any statement in Java may be labeled. As shown in Figure 10.8 we'll assume an AST node labeledStmt that contains a string-valued field stmtLabel. If the statement is labeled, stmtLabel contains the label in string form. If the statement is unlabeled, stmtLabel is null. labeledStmt also contains a field stmt that is the AST node representing the labeled statement.

In Java, C and C++ (and most other programming languages) labels are placed in a different name space than other identifiers. This just means that an identifier used as a label may also be used for other purposes (a variable name, a type name, a method name, etc.) without confusion. This is because labels are

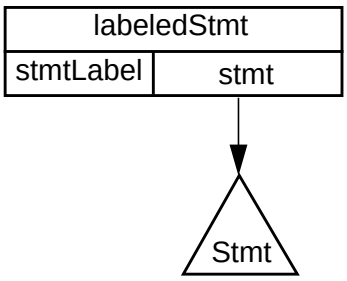

Figure 10.8 Abstract Syntax Tree for a Labeled Statement

used in very limited contexts (in continues, breaks and perhaps gotos). Labels can't be assigned to variables, returned by functions, read from files, etc.

We'll represent labels that are currently visible using a labelList variable. This variable is set to null when we begin the analysis of a method or constructor body, or a static initializer.

A labelListNode contains four fields: label, a string that contains the name of the label, kind (one of iterative, switch or other) that indicates the kind of statement that is labeled, AST, a link to the AST of the labeled statement and next, which is a link to the next labelListNode on the list.

Looking at a labelList, we can determine the statements that enclose a break or continue, as well as all the labels currently visible to the break or continue.

The semantic analysis necessary for a labeledStmt is shown in Figure 10.9. The labelList is extended by adding an entry for the current labeledStmt node, using its label (which may be null), and its kind (determined by a call to an auxiliary method, getKind(stmt)). The stmt AST is analyzed using the extended labelList. After analysis, labelList is returned to its original state, by removing its first element.

```
labeledStmt.Semantics( )
1.  labelList ← labelListNode(stmtLabel, getKind(stmt), stmt, labelList)
2.  stmt.isReachable ← labeledStmt.isReachable
3.  stmt.Semantics()
4.  terminatesNormally ← stmt.terminatesNormally
5.  throwsSet ← stmt.throwsSet
6.  labelList ← labelList.next
```

Figure 10.9 Semantic Analysis for Labeled Statements

A continue statement without a label references the innermost iterative statement (while, do or for) within which it is nested. This is easily checked by looking for a node on the labelList with kind = iterative (ignoring the value of the label field).

A continue statement that references a label L (stored in AST field stmtLabel) must be enclosed by an iterative statement whole label is L. If more than one con-

taining statement is labeled with L, the nearest (innermost) is used. The details of this analysis are shown in Figure 10.10.

```
ContinueNode.Semantics( )
 1.   terminatesNormally ← false
 2.   throwsSet ← null
 3.   currentPos ← labelList
 4.   if stmtLabel = null
 5.        then while currentPos ≠ null
 6.                   do if currentPos.kind = iterative
 7.                            then return
 8.                        currentPos = currentPos.next
 9.             GenerateErrorMessage("Continue not inside iterative statement")
10.        else while currentPos ≠ null
11.                   do if currentPos.label = stmtLabel and
                                 currentPos.kind = iterative
12.                           then return
13.                       currentPos = currentPos.next
14.             GenerateErrorMessage("Continue label doesn't match an
                                          iterative statement")
```

Figure 10.10 Semantic Analysis for Continue Statements

As an example, consider the following code fragment

```
L1: while (p != null) {
     if (p.val < 0)
       continue
     else ... }
```

The labelList in use when the continue is analyzed is shown in Figure 10.11. Since the list contains a node with kind = iterative, the continue is correct.

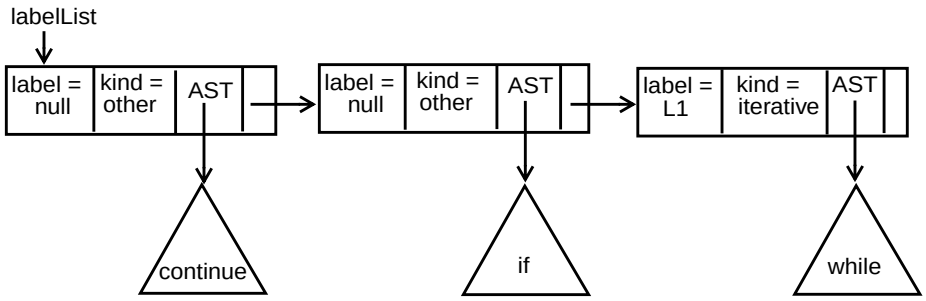

Figure 10.11 Example of a LabelList in a Continue Statement

In C and C++ semantic analysis is even simpler. Continues don't use labels so the innermost iterative statement is always selected. This mean we need only lines 1 to 9 of Figure 10.10.

Break Statements. In Java an unlabeled break statement has the same meaning as the break statement found in C and C++. The innermost while, do, for or switch statement is exited, and execution continues with the statement immediately following the exited statement. Thus a reachable break forces the statement it references to terminate normally.

A labeled break exits the enclosing statement with a matching label (not necessarily a while, do, for or switch statement), and continues execution with that statement's successor (again, if reachable, it forces normal termination of the labeled statement). For both labeled and unlabeled breaks, semantic analysis must verify that a suitable target statement for the break exists.

We'll again use the labelList introduced in the last section. For unlabeled breaks, we'll need to find a node with kind = iterative or switch. For labeled breaks, we'll need to find a node with a matching label (its kind doesn't matter). In either case, the terminatesNormally field of the referenced statement will be set to true if the break is marked as reachable.

This analysis is detailed in Figure 10.12.

```
BreakNode.Semantics( )
1.  terminatesNormally ← false
2.  throwsSet ← null
3.  currentPos ← labelList
4.  if stmtLabel = null
5.      then while currentPos ≠ null
6.                do if currentPos.kind = iterative
                        or currentPos.kind = switch
7.                   then if isReachable
8.                           then currentPos.AST.terminatesNormally ← true
9.                        return
10.               currentPos = currentPos.next
11.           GenerateErrorMessage("Break not inside iterative or
                                    switch statement")
12.     else while currentPos ≠ null
13.               do if currentPos.label = stmtLabel
14.                  then if isReachable
15.                           then currentPos.AST.terminatesNormally ← true
16.                       return
17.               currentPos = currentPos.next
18.           GenerateErrorMessage("Continue label doesn't match any
                                    statement label")
```

Figure 10.12 Semantic Analysis for Break Statements

As an example, consider the following code fragment

```
L1: for (i=0; i < 100; i++)
       for (j=0; j < 100; j++)
```

```
if (a[i][j] == 0)
   break L1;
else ...
```

The labelList in use when the break is analyzed is shown in Figure 10.13. Since the list contains a node with label = L1, the break is correct. The for loop labeled with L1 is marked as terminating normally.

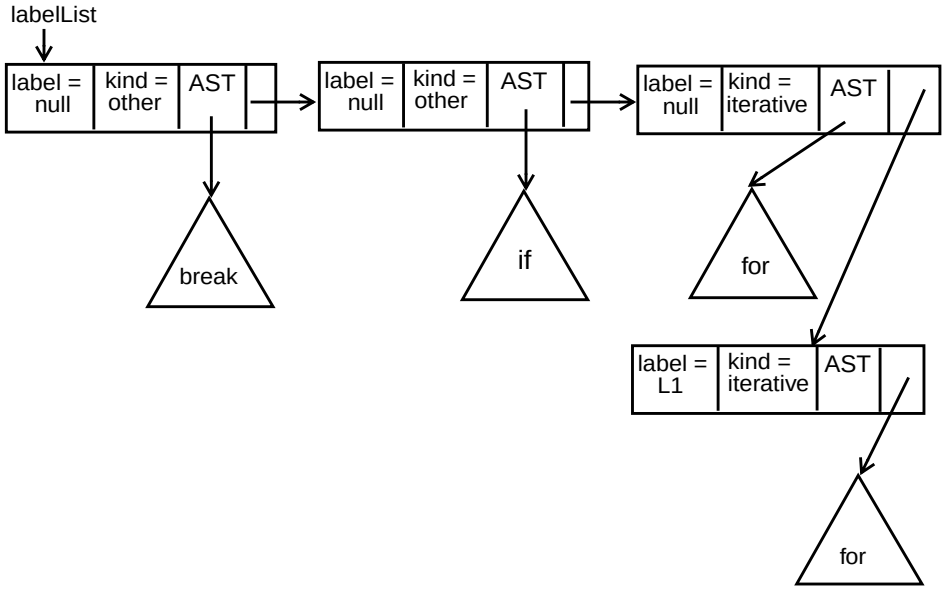

Figure 10.13 Example of a LabelList in a Break Statement

Return Statements. An AST rooted by a returnNode, as shown in Figure 10.14, represents a return statement. The field returnVal is null if no value is returned; otherwise it is an AST representing an expression to be evaluated and returned.

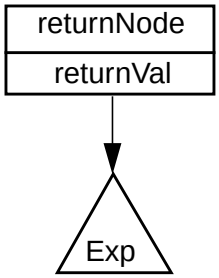

Figure 10.14 Abstract Syntax Tree for a Return Statement

The semantic rules governing a return statement depend on where the statement appears. A return statement without an expression value may only appear in a void method (a subroutine) or in a constructor. A return statement with a return value may only appear in a method whose type may be assigned the return type (this excludes void methods and constructors). A return (of either form) may not appear in a static constructor.

A method declared to return a value (a function) **must** exit via a return of a value or by throwing an exception. This requirement can be enforced by verifying that the statement list that comprises a function's body has its terminatesNormally value set to false.

To determine the validity of a return we will check the kind of construct (method, constructor or static initializer) within which it appears. But AST links all point downward; hence looking "upward" is difficult. To assist our analysis, we'll assume three global pointers, currentMethod, currentConstructor and currentStaticInitializer, set during semantic analysis.

If we are checking an AST node contained within the body of a method, currentMethod will tell us which one. If we are analyzing an AST node not within a method, then currentMethod is null. The same is true for currentConstructor and currentStaticInitializer. We can determine which kind of construct we are in (and details of its declaration) by using the pointer that is non-null.

The details of semantic analysis for return statements appears in Figure 10.15. For methods we assume currentMethod.returnType is the type returned by the method (possibly equal to void). The auxiliary method is assignable(T1,T2) tests whether type T2 is assignable to type T1 (using the assignability rules of the languages being compiled).

```
return.Semantics( )
1.   terminatesNormally ← false
2.   if  currentStaticInitializer ≠ null
3.        then  GenerateErrorMessage("A return may not appear within a
                                                  static initializer")
4.   elsif  returnVal ≠ null
5.        then  returnVal.ExprSemantics()
6.              throwsSet ← returnVal.throwsSet
7.              if  currentMethod = null
8.                   then  GenerateErrorMessage("A value may not be returned
                                  from a constructor")
9.              elsif  not assignable(currentMethod.returnType,returnValue.type)
10.                  then  GenerateErrorMessage("Illegal return type")
11.         else  if  currentMethod ≠ null  and  currentMethod.returnType ≠ void
12.                  then  GenerateErrorMessage("A value must be returned")
13.             throwsSet ← null
```

Figure 10.15 Semantic Analysis for Return Statements

C++ has semantic rules very similar to those of Java. A value may only be returned from a non-void function, and the value returned must be assignable to

the function's return type. In C a return without a value is allowed in a non-void function (with undefined behavior).

Goto Statements. Java contains no goto statement, but many other languages, including C and C++, do. C, C++ and most other languages that allow gotos restrict them to be **intraprocedural**. That is, a label and all gotos that reference it must be in the same procedure or function.

As noted earlier, identifiers used as labels are usually considered distinct from identifiers used for other purposes. Thus in C and C++, the statement

```
a: a+1;
```

is legal. Labels may be kept in a separate symbol table, distinct from the main symbol table used to store ordinary declarations.

Labels need not be defined before they are used; "forward gotos" are allowed. Semantic checking must guarantee that all labels used in gotos are in fact defined somewhere in the current procedure.

Because of potential forward references, it is a good idea to check labels and gotos in two steps. First, the AST that represents the entire body of a subprogram is traversed, gathering all label declarations into a declaredLabels table stored as part of the current subprogram's symbol table information. Duplicate labels are detected as declaredLabels is built.

During normal semantic processing of the body of a subprogram (after declaredLabels has been built), an AST for a goto can access declaredLabels (through the current subprogram's symbol table). Checking for valid label references (whether forward or not) is easy.

A few languages, like Pascal, allow **non-local gotos**. A non-local goto transfers control to a label in a scope that contains the current procedure. Non-local gotos can be checked by maintaining a stack (or list) of declaredLabels tables, one for each nested procedure. A goto is valid if its target appears in any of the declaredLabels tables.

Finally, some programming languages forbid gotos into a conditional or iterative statement from outside. That is, even if the scope of a label is an entire subprogram, a goto into a loop or from a then part to an else part is forbidden. Such restrictions can be enforced by marking each label in declaredLabels as either "active" or "inactive." Gotos are allowed only to active labels, and a a label within a conditional or iterative statement is active only while the AST that contains the label is being processed. Thus a label L within a while loop becomes active when the loop body's AST is checked, and is inactive when statements outside the loop body are checked.

10.1.5 Switch and Case Statements

Java, C and C++ contain a **switch** statement that allows the selection of one of a number of statements based on the value of a control expression. Pascal, Ada and

Modula 3 contain a **case** statement that is equivalent. We shall focus on translating switch statements, but our discussion applies equally to case statements.

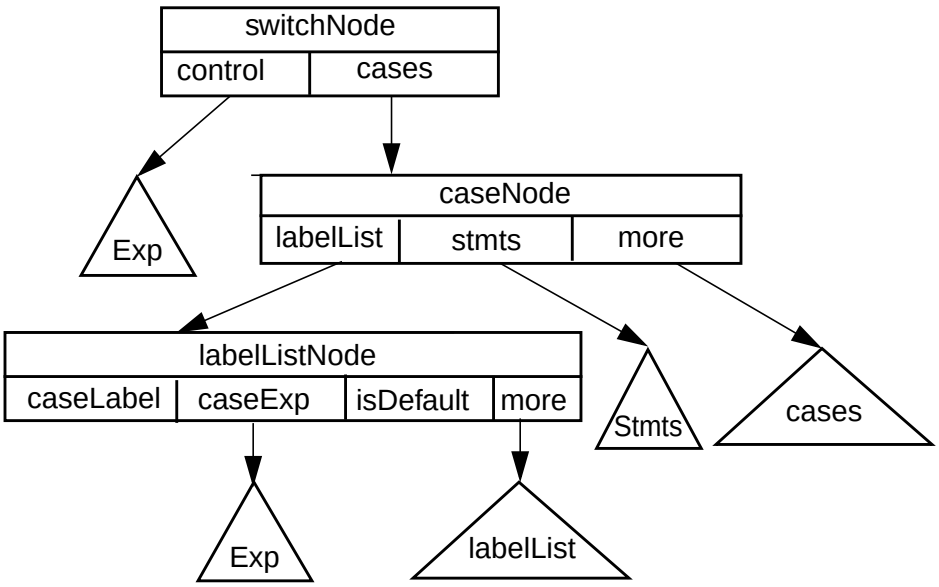

Figure 10.16 Abstract Syntax Tree for a Switch Statement

The AST for a switch statement, rooted at a switchNode is shown in Figure 10.16 (fields not needed for semantic analysis are omitted for clarity). In the AST control represents an integer-valued expression; cases is a caseNode, representing the cases in the switch. Each caseNode has three fields. labelList is a labelListNode that represents one or more case labels. stmts is an AST node representing the statements following a case constant in the switch. more is either null or another caseListNode, representing the remaining cases in the switch.

A labelListNode contains an integer field caseLabel, an AST caseExp, a boolean field isDefault (representing the default case label) and more, a field that is either null or another labelListNode, representing the remainder of the list. The caseExp AST represents a constant expression that labels a case within the switch; when it is evaluated, caseLabel will hold its value.

A number of steps are needed to check the semantic correctness of a switch statement. The control expression and all the statements in the case body must be checked. The control expression must be an integer type (32 bits or less in size). Each case label must be a constant expression assignable to the control expression. No two case labels may have the same value. At most one default label may appear within the switch body.

A switch statement can terminate normally in a number of ways. An empty switch body (uncommon, but legal) trivially terminates normally. If the last switch group (case labels followed by a statement list) terminates normally, so does the

switch (since execution "falls through" to the succeeding statement. If any of the statements within a switch body contain a reachable break statement, then the entire switch can terminate normally.

We'll need a number of utility routines in our semantic analysis. To process case labels, we will build a linked list of node objects. Each node will contain a value field (an integer) and a next field (a reference to a node).

The routine buildLabels (Figure 10.17) will traverse a labelListNode (including its successors), checking the validity of case labels and building a list of node objects representing the case labels that are encountered. The related routine buildLabelList (Figure 10.17) will traverse a caseNode (including its successors). It checks the validity of statements within each caseNode and builds a list of node objects representing the case labels that are found within the caseNode.

```
buildLabels( labelList )
1.   if   labelList = null
2.        then   return null
3.   elsif  isDefault
4.        then   return buildLabels(more)
5.         else   caseExp.ExprSemantics()
6.              if  caseExp.type = errorType
7.                   then   return buildLabels(more)
8.              elsif  not assignable(CurrentSwitchType, CaseExp.type)
9.                   then   GenerateErrorMessage("Invalid Case Label Type")
10.                       return buildLabels(more)
11.                  else   caseLabel ← caseExp.EvaluateConstExpr()
12.                       if  caseLabel = null
13.                            then   GenerateErrorMessage("Case Label Must be
                                          a Constant Expression")
14.                                 return buildLabels(more)
15.                           else   return node(caseLabel, buildLabels(more))

buildLabelList ( cases )
1.   stmts.isReachable ← true
2.   stmts.Semantics()
3.   if   more = null
4.        then   terminatesNormally ← stmts.terminatesNormally
5.             throwsSet ← stmts.throwsSet
6.             return  buildLabels(labelList)
7.         else   restOfLabels ← buildLabelList(more)
8.             terminatesNormally ← more.terminatesNormally
9.             throwsSet ← union(stmts.throwsSet, more.throwsSet)
10.            return append(buildLabels(labelList), restOfLabels)
```

Figure 10.17 Utility Semantic Routines for Switch Statements (Part 1)

The routine checkForDuplicates (Figure 10.18) takes a sorted list of node objects, representing all the labels in the switch statement, and checks for duplicates by comparing adjacent values. Routines countDefaults and countDefaultLabels (Figure 10.18) count the total number of default cases that appear within a switchNode.

```
checkForDuplicates( node )
1.    if  node ≠ null and node.next ≠ null
2.        then  if  node.value = node.next.value
3.                  then  GenerateErrorMessage("Duplicate case label:",
                                node.value)
4.              checkForDuplicates(node.next)

countDefaults( cases )
1.    if  cases = null
2.        then  return 0
3.         else    return countDefaultLabels(labelList) + countDefaults(more)

countDefaultLabels( labelList )
1.    if  labelList = null
2.        then  return 0
3.              if  isDefault
4.                  then  return 1 + countDefaultLabels(more)
5.                   else  return countDefaultLabels(more)
```

Figure 10.18 Utility Semantic Routines for Switch Statements (Part 2)

We can now complete the definition of semantic processing for switch statements, as shown in Figure 10.19. We first mark the whole switch as not terminating normally. This will be updated to true if cases is null, or if a reachable break is encountered while checking the switch body or if the stmts AST of the last caseNode in the AST is marked as terminating normally. The switch's control expression is checked. It must be a legal expression and assignable to type `int`. A list of all the case label values is built by traversing the cases AST. As the label list is built, case statements and case labels are checked. After the label list is built, it is sorted. Duplicates are found by comparing adjacent values in the sorted list. Finally, the number of default cases is checked by traversing the cases AST again.

As an example, consider the following switch statement

```
switch(p) {
 case 2:
 case 3:
 case 5:
 case 7:isPrime = true; break;
 case 4:
 case 6:
```

```
switchNode.Semantics( )
1.   terminatesNormally ← false
2.   control.Semantics()
3.   if control.type ≠ errorType and not assignable(int, control.type)
4.        then GenerateErrorMessage("Illegal Type for Control Expression")
5.   CurrentSwitchType ← control.type
6.   if cases = null
7.        then terminatesNormally ← true
8.             throwsSet ← control.throwsSet
9.        else labelList ← buildLabelList(cases)
10.            terminatesNormally ← terminatesNormally
                        or cases.terminatesNormally
11.            throwsSet ← union(control.throwsSet, cases.throwsSet)
12.            labelList ← sort(labelList)
13.            checkForDuplicates(labelList)
14.            if countDefaults(cases) > 1
15.                 then GenerateErrorMessage("More than One Default
                                Case Label")
```

Figure 10.19 Semantic Analysis for Switch Statements

```
    case 8:
    case 9:isPrime = false; break;
    default:isPrime = checkIfPrime(p);
  }
```

Assume that p is declared as an integer variable. We check p and find it a valid control expression. The label list is built by examining each caseNode and labelListNode in the AST. In doing so, we verify that each case label is a valid constant expression that is assignable to p. The case statements are checked and found valid. Since the last statement in the switch (the default) terminates normally, so does the entire switch statement. The value of labelList is {2,3,5,7,4,6,8,9}. After sorting we have {2,3,4,5,6,7,8,9}. No adjacent elements in the sorted list are equal. Finally, we count the number of default labels; a count of 1 is valid.

The semantic rules for switch statements in C and C++ are almost identical to those of Java.

Other languages include a case statement that is similar in structure to the switch statement. Pascal allows enumerations as well as integers in case statements. It has no default label and considers the semantics of an unmatched case value to be undefined. Some Pascal compilers check for complete case coverage by comparing the sorted label list against of range of values possible in the control expression (Pascal includes subrange types that limit the range of possible values a variable may hold).

Ada goes farther than Pascal in requiring that all possible control values must be covered within a case statement (though it does allow a default case). Again, comparing sorted case labels against possible control expression values is required.

Ada also generalizes a case label to a **range** of case values (e.g., in Java notation, `case 1..10`, which denotes 10 distinct case values). Semantic checks that look for duplicate case values and check for complete coverage of possible control values must be generalized to handle ranges rather than singleton values.

10.1.6 Exception Handling

Java, like most other modern programming languages, provides an **exception handling** mechanism. During execution, an exception may be **thrown**, either explicitly (via a throw statement) or implicitly (due to an execution error). Thrown exceptions may be **caught** by an **exception handler.**

Exceptions form a clean and general mechanism for identifying and handling unexpected or erroneous situations. They are clearer and more efficient than using error flags or gotos. Though we will focus on Java's exception handling mechanism, most recent language designs, including C++, Ada and ML, include an exception handling mechanism similar to that of Java.

Java exceptions are **typed**. An exception throws an object that is an instance of class `Throwable` or one of is subclasses. The object thrown may contain fields that characterize the precise nature of the problem the exception represents, or the class may be empty (with its type signifying all necessary information).

Java exceptions are classified as either **checked** or **unchecked**. A checked exception thrown in a statement must be caught in an enclosing try statement or listed in the throws list of the enclosing method or constructor. Thus it must be handled or listed as a possible result of executing a method or constructor.

An unchecked exception (defined as an object assignable to either class `RuntimeException` or class `Error`) may be handled in a try statement, but need not be. If uncaught, unchecked exceptions will terminate execution. Unchecked exceptions represent errors that may appear almost anywhere (like accessing a null reference or using an illegal array index). These exceptions usually force termination, so explicit handlers may clutter a program without adding any benefit (termination is the default for uncaught exceptions).

We'll first consider the semantic checking needed for a try statement, whose AST is shown in Figure 10.20.

To begin we will check the optional finally clause (referenced by final). The correctness of statements within it is independent of the contents of the try block and the catch clauses.

Next, the catch clauses must be checked. Each catch clause is represented by a catchNode, as shown in Figure 10.21.

Catch clauses require careful checking. Each clause introduces a new identifier, the parameter of the clause. This identifier must be declared as an exception (of class `Throwable` or a subclass of it). The parameter is local to the body of the catch clause, and must not hide an already visible local variable or parameter.

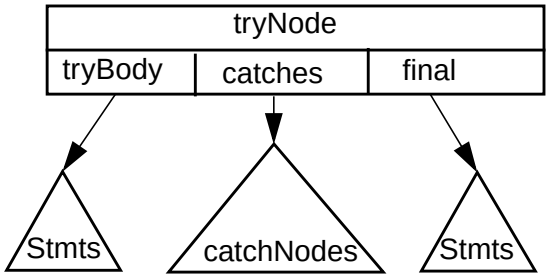

Figure 10.20 Abstract Syntax Tree for a Try Statement

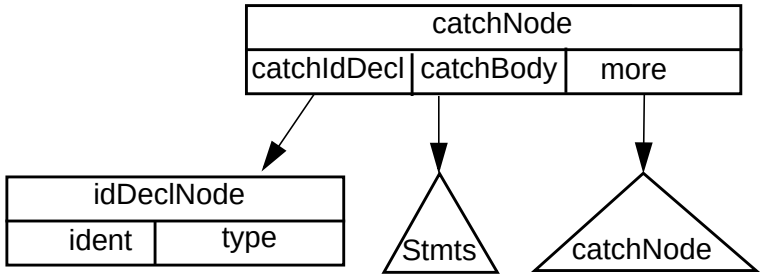

Figure 10.21 Abstract Syntax Tree for a Catch Block

Moreover, within the catch body, the catch parameter may not be hidden by a local declaration.

We compute the exceptions that are caught by the catch clauses (localCatches) and verify (using method isSubsumedBy, defined in Figure 10.22) that no catch clause is "hidden" by earlier catches. This is a reachability issue—some exception type must be able to reach, and activate, each of the catch clauses.

Finally, the body of the try block is checked. Statements within the try body must know what exceptions will be caught by the try statement. We will create a data structure called catchList that is a list of throwItem nodes. Each throwItem on the list represents one exception type named in a catch block of some enclosing try statement.

Before we begin the semantic analysis of the current tryNode, the catchList contains nodes for all the exceptions caught by try statements that enclose the current try statement (initially this list is null).

We compute the exceptions handled by the current catch clauses, building a list called localCatches. We extend the current catchList by prepending localCatches. The extended list is used when the try statements are checked, guaranteeing that the definitions of local exception handlers are made visible within the try body.

After the try body is analyzed, we compute the throwsSet for the tryNode. This is the set of exceptions that "escape" from the try statement. It is the set of exceptions thrown in the finally clause, or any catch clause, plus exceptions generated in the try body, but not caught in an of the catch clauses. The method filterOutThrows, defined in Figure 10.22, details this process.

The complete semantic processing for a tryNode is detailed in Figure 10.23.

```
isSubsumedBy( catchList, exceptionType )
1.   while catchList ≠ null
2.         do if assignable(catchList.type, exceptionType)
3.                  then return true
4.                  else currentList ← currentList.next
5.   return false

filterOutThrows( throwsList, exceptionType )
1.   if throwsList = null
2.        then return null
3.   elsif assignable(exceptionType, throwsList.type)
4.        then return filterOutThrows(throwsList.next, exceptionType)
5.        else return throwItem(throwsList.type,
                                filterOutThrows(throwsList.next, exceptionType))
```

Figure 10.22 Utility Routines for Catch Clauses.

The AST for a throw statement is shown in Figure 10.24. The type of value thrown in a throw statement must, of course, be an exception (a type assignable to `Throwable`). If the exception thrown is checked, then semantic analysis must also verify that an enclosing try block can catch the exception or that the method or constructor that contains the throw has included the exception in its throws list.

The set of exceptions generated by the throws statement (its throwsSet) is the exception computed by the thrownVal AST **plus** any exceptions that might be thrown by the expression that computes the exception value.

In processing statements we built a throwsSet that represents all exception types that may be thrown by a statement. We will assume that when the header of a method or constructor is semantically analyzed, we create a declaredThrowsList (composed of throwItem nodes) that represents all the exception types (if any) mentioned in the throws list of the method or constructor. (A static initializer will always have a null declaredThrowsList). Comparing the throwsSet and declaredThrowsList, we can readily check whether all checked exceptions are properly handled.

Semantic analysis of a throw statement is detailed in Figure 10.25.

As an example, consider the following Java code fragment

```
class ExitComputation extends Exception{};
try { ...
```

```
tryNode.Semantics( )
 1.  terminatesNormally ← false
 2.  tryBody.isReachable ← final.isReachable ← true
 3.  final.Semantics()
 4.  currentCatch ← catches
 5.  localCatches ← throwsInCatches ← null
 6.  while currentCatch ≠ null
 7.        do if not assignable(Throwable, currentCatch.catchIdDecl.type)
 8.              then GenerateErrorMessage("Illegal type for catch identifier")
 9.                   currentCatch.catchIdDecl.type ← errorType
10.           elsif isSubsumedBy(localCatches, currentCatch.catchIdDecl.type)
11.              then GenerateErrorMessage("Catch is Hidden by Earlier Catches")
12.              else localCatches ← append(localCatches,
                                  catchNode(currentCatch.catchIdDecl.type,null)
13.           currentDecl ← Lookup(currentCatch.catchIdDecl.ident)
14.           if currentDecl ≠ null and
                 (currentDecl.kind = Variable or currentDecl.kind = Parameter)
15.              then GenerateErrorMessage("Attempt to redeclare local identifier")
16.           OpenNewScope()
17.           DeclareLocalVariable(currentCatch.catchIdDecl.ident,
                    currentCatch.catchIdDecl.type, CantBeHidden)
18.           catchBody.isReachable ← true
19.           currentCatch.catchBody.Semantics()
20.           terminatesNormally ← terminatesNormally or
                                  currentCatch.catchBody.terminatesNormally
21.           throwsInCatch ←
                              union(throwsInCatch, currentCatch.catchBody.throwsSet)
22.           CloseScope()
23.           currentCatch ← currentCatch.more
24.  prevCatchList ← catchList
25.  catchList ← append(localCatches, catchList)
26.  tryBody.Semantics()
27.  terminatesNormally ← (terminatesNormally or tryBody.terminatesNormally)
                           and final.terminatesNormally
28.  catchList ← prevCatchList
29.  throwsInTry ← tryBody.throwsSet
30.  currentCatch ← catches
31.  while currentCatch ≠ null
32.        do newSet ← filterOutThrows(throwsInTry, currentCatch.catchIdDecl.type)
33.           if newSet = throwsInTry
34.              then GenerateErrorMessage("No Throws Reach this Catch")
35.              else throwsInTry ← newSet
36.           currentCatch ← currentCatch.more
37.  throwsSet ←  union(throwsInTry, throwsInCatch, final.throwsSet)
```

Figure 10.23 Semantic Analysis for Throw Statements

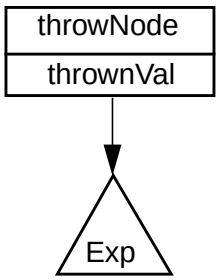

Figure 10.24 Abstract Syntax Tree for a Throw Statement

```
throwNode.Semantics( )
1.  terminatesNormally ← false
2.  throwsSet ← null
3.  thrownType ← thrownVal.ExprSemantics()
4.  if thrownType ≠ errorType
5.      then if not assignable(Throwable, thrownType)
6.               then GenerateErrorMessage("Illegal type for throw")
7.               else if assignable(RuntimeException, thrownType) or
                         assignable(Error, thrownType)
8.                    then return
9.                    throwsSet ← union(thrownVal.throwsSet,
                                        throwItem(thrownType, null))
10.                   throwTargets ← append(catchList, declaredThrowsList)
11.                   while throwTargets ≠ null
12.                         do if assignable(throwTargets.type, thrownType)
13.                                then return
14.                                else thrownType ← thrownType.next
15.                   GenerateErrorMessage("Type thrown not found in
                                            enclosing catch or declared throws list")
```

Figure 10.25 Semantic Analysis for Throw Statements

```
    if (cond)
          throw new ExitComputation();
    if (v < 0.0)
          throw new ArithmeticException();
    else  a = Math.sqrt(v);
... }
catch (e ExitComputation) {return 0;}
```

A new checked exception, `ExitComputation`, is declared. In the try statement, we first check the catch clause. Assuming `e` is not already defined as a variable or parameter, no errors are found. The current catchList is extended with an

entry for type `ExitComputation`. The try body is then checked. Focusing on throw statements, we first process a throw of an `ExitComputation` object. This is a valid subclass of `Throwable` and `ExitComputation` is on the catchList, so no errors are detected. Next the throw of an `ArithmeticException` is checked. It too is a valid exception type. It is an unchecked exception (a subclass of `RuntimeException`), so the throw is valid independent of any try statements that enclose it.

The exception mechanism of C++ is very similar to that of Java, using an almost identical throw/catch mechanism. The techniques developed in this section are directly applicable.

Other languages, like Ada, feature a single exception type that is "raised" rather than thrown. Exceptions are handled in a "when" clause that can be appended to any begin-end block. Again, semantic processing is very similar to the mechanisms developed here.

10.2 Semantic Analysis of Calls

In this section we investigate the semantic analysis of method calls in Java. The techniques we present are also be applicable to constructor and interface calls, as well as calls to subprograms in C, C++ and other languages.

The AST for a callNode is shown in Figure 10.26. The field method is an identifier that specifies the name of the method to be called. The field qualifier is an optional expression that specifies the object or class within which method is to be found. Finally, args is an optional expression list that represents the actual parameters to the call.

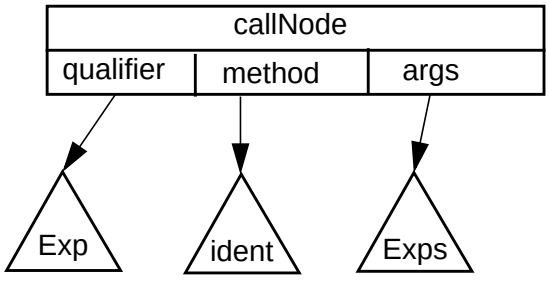

Figure 10.26 Abstract Syntax Tree for a Method Call

The first step in analyzing a call is to determine which method definition to use. This determination is by no means trivial in Java because of **inheritance** and **overloading**.

Recall that classes form an inheritance hierarchy, and all classes are derived, directly or indirectly, from `Object`. An object may have access to methods defined

in its own class, its parent class, its grandparent class, and so on, all the way up to `Object`. In processing a call all potential locales of definition must be checked.

Because of overloading, it is valid to define more than one method with the same name. A call must select the "right" definition, which informally is the nearest accessible method definition in the inheritance hierarchy whose parameters match the actual parameters provided in the call.

Let us begin by gathering all the method definitions that might be applicable to the current call. This lookup process is guided by the kind of method qualifier (if any) that is provided and the access mode of individual methods:

- If no qualifier is provided, we examine the class (call it C) that contains the call being analyzed. All methods defined within C are accessible. In addition, as explained in section xxx.yyy, methods defined in C's superclasses (its parent class, grandparent class, etc.) may be inherited depending on their access qualifiers:
  - ❏ Methods marked public or protected are always included.
  - ❏ A method with default access (not marked public, private or protected) is included if the class within which it is defined is in the same package as C.
  - ❏ Private methods are **never** included (since they can't be inherited)
- If the qualifier is the reserved word `super`, then a call of M in class C must reference a method inherited from a superclass (as defined above). Use of `super` in class `Object` is illegal, because `Object` has no superclass.
- If the qualifier is a type name T (which must be a class name), then T must be in the package currently being compiled, or it must be a class marked public. A call of M must reference a static method (instance methods are disallowed because the object reference symbol `this` is undefined). The static methods that may be referenced are:
  - ❏ All public methods defined in T or a superclass of T.
  - ❏ Methods defined in T or a superclass with default access if the defining class occurs within the current package.
  - ❏ Methods defined in T or a superclass of T marked protected if they are defined in the current package or if the call being analyzed occurs within a subclass of T.
- If the qualifier is an expression that computes an object of type T, then T must be in the package currently being compiled, or it must be a class marked public. A call of M may reference:
  - ❏ All public methods defined in T or a superclass of T.
  - ❏ Methods defined in T or a superclass with default access if the class containing the method definition occurs within the current package.

- Methods defined in T or a superclass of T marked protected if they are defined in the current package or if T is a subclass of C, the class that contains the call being analyzed.

These rules for selecting possible method definitions are codified in Figure 10.27. We assume that methodDefs(ID) returns all the methods named ID in a given class. Similarly, publicMethods(ID) returns all the public methods named ID, etc. The reference currentClass accesses the class currently being compiled; currentPackage references the package being currently compiled.

```
callNode.getMethods( )
 1.  if qualifier = null
 2.      then methodSet ← currentClass.methodDefs(method)
 3.      else methodSet ← null
 4.  if qualifier = null or qualifier = superNode
 5.      then nextClass ← currentClass.parent
 6.      else nextClass ← qualifier.type
 7.  while nextClass ≠ null
 8.      do if qualifier ≠ null and qualifier ≠ superNode and
               nextClass.package ≠ currentPackage and not nextClass.isPublic
 9.           then nextClass ← nextClass.parent
10.                continue
11.         methodSet ← union(methodSet, nextClass.publicMethods(method))
12.         if nextClass.package = currentPackage
13.              then methodSet ← union(methodSet,
                                        nextClass.defaultAccessMethods(method))
14.         if qualifier = null or qualifier = superNode
                 or nextClass.package = currentPackage
                 or (qualifer.kind = type and isAncestor(qualifier.type, currentClass)
                 or (qualifer.kind = value and isAncestor(currentClass, qualifier.type)
15.              then methodSet ← union(methodSet,
                                        nextClass.protectedMethods(method))
16.         nextClass ← nextClass.parent
17.  return methodSet
```

Figure 10.27 Compute Set of Method Definitions that Match the Current Call

Once we have determined the set of definitions that are *possible*, we must filter them by comparing these definitions with the number and type of expressions that form the call's actual parameters. Assume that an expression list is represented by an exprsNode AST (Figure 10.28) and is analyzed using the Semantics method defined in Figure 10.29.

We will assume that each method definition included in the set of accessible methods is represented as a methodDefItem, which contains the fields returnType, argTypes, and classDefIn. returnType is the type returned by the method; classDefIn is the class the method is defined in; argTypes is a linked list of typeItem nodes, one for each declared parameter. Each typeItem contains a type field (the

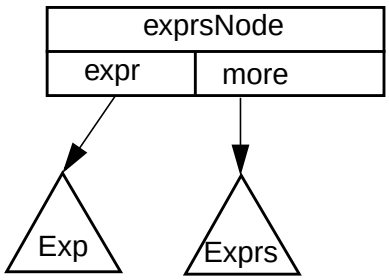

Figure 10.28 Abstract Syntax Tree for an Expression List

```
exprsNode.Semantics( )
1.  expr.ExprSemantics()
2.  more.Semantics()
3.  throwsSet ← union(expr.throwsSet, more.throwsSet)
```

Figure 10.29 Semantic Checks for an Expression List

type of the parameter) and next, (a reference to the next typeItem on the list). We can build a typeItem list for the actual parameters of the call using the getArgTypes method defined in Figure 10.30.

```
getArgTypes( exprList )
1.  if exprList = null
2.      then return null
3.      else return typeItem(exprList.expr.type, getArgTypes(exprList.more))
```

Figure 10.30 Build a Type List for an Expression List.

Once we have a type list for the actual parameters of a call, we can compare it with the declared parameter type list of each method returned by findDefs. But what exactly defines a match between formal and actual parameters? First, both argument lists must have the same length—this is easy to check. Next, each actual parameter must be "bindable" to its corresponding formal parameter.

Bindable means that it is legal to use an actual parameter whenever the corresponding formal parameter is referenced. In Java, bindable is almost the same as assignable. The only difference is that an integer literal, used as an actual parameter, may not be bound to a formal of type `byte`, `short` or `char`. (This requirement was added to make it easier to select among overloaded definitions). We will use the predicate bindable(T1,T2) to determine if an actual of type T2 may be bound to a formal of type T1.

Now checking the feasibility of using a particular method definition in a call is straightforward—we check that the number of parameters is correct and that each parameter is bindable. This is detailed in Figure 10.31, which defines the method applicable(formalParms,actualParms). If applicable returns true, a particular method definition can be used; otherwise, it is immediately rejected as not applicable to the call being processed.

```
applicable( formalParms, actualParms )
1.    if  formalParms = null and actualParms = null
2.          then  return  true
3.    elsif  formalParms = null or actualParms = null
4.          then  return false
5.    elsif  bindable(formalParms.type, actualParms.type)
6.          then  return applicable(formalParms.next, actualParms.next)
7.           else  return false
```

Figure 10.31 Test if Actual Parameters are Bindable to Corresponding Actual Parameters.

When analyzing a method call we will first select the set of method definitions the call might reference. Next we analyze the actual parameters of the call, and build an actual parameters list. This list is compared with the formal parameters list of each method under consideration, filtering out those that are not applicable (because of an incorrect argument count or an argument type mismatch).

At this stage we count the number of method definitions still under consideration. If it is zero, we have an invalid call—no accessible method can be called without error. If the count is one, we have a correct call.

If two or more method definitions are still under consideration, we need to choose the most appropriate definition. Two issues are involved here. First, if a method is redefined in a subclass, we want to use the redefinition. For example, method `M()` is defined in both classes `C` and `D`:

```
class C { void M() { ... } }
class D extends C { void M() { ... } }
```

If we call `M()` in an instance of class `D`, we want to use the definition of `M` in `D`, even though `C`'s definition is visible and type-correct.

It may also happen that one definition of a method `M` takes an object of class `A` as a parameter, whereas another definition of `M` takes a subclass of `A` as a parameter. An example of this is

```
class A { void M(A parm) { ... } }
class B extends A { void M(B parm) { ... } }
```

Now consider a call `M(b)` in class `B`, where `b` is of type `B`. Both definitions of `M` are possible, since an object of class `B` may always be used where a parameter of its parent class (`A`) is expected.

In this case we prefer to use the definition of `M(B parm)` in class `B` because it is a "closer match" to the call `M(b)` this is being analyzed.

We formalize the notion of one method definition being a "closer match" than another by defining one method definition D to be more specific than another definition E if D's class is bindable to E's class and each of D's parameters is bindable to the corresponding parameter of E. This definition captures the notion that we prefer a method definition is a subclass to an otherwise identical definition in a parent class (a subclass may be assigned to a parent class, but not vice-versa). Similarly, we prefer arguments that involve a subclass over an argument that involves a parent class (as was the case in the example of `M(A parm)` and `M(B parms)` used above).

A method moreSpecific(Def1,Def2) that tests whether method definition Def2 is more specific than method definition Def1 is presented in Figure 10.32.

```
moreSpecific( Def1, Def2 )
1.   if bindable(Def1.classDefIn, Def2.classDefIn)
2.       then arg1 ← Def1.argTypes
3.            arg2 ← Def2.argTypes
4.            while arg1 ≠ null
5.                 do if bindable(arg11.type, arg2.type)
6.                       then arg1 ← arg1.next
7.                            arg2 ← arg2.next
8.                       else return false
9.            return true
10.      else return false
```

Figure 10.32 Test if One Method Definition is More Specific than Another Definition.

Whenever we have more than one accessible method definition that matches a particular argument list in a call, we will filter out less specific definitions. If, after filtering, only one definition remains (called the **maximally specific** definition), we know it is the correct definition to use. Otherwise, the choice of definition is **ambiguous** and we must issue an error message.

The process of filtering out less specific method definitions is detailed in Figure 10.33, which defines the method filterDefs(MethodDefSet).

```
filterDefs( MethodDefSet )
1.   changes ← true
2.   while changes
3.        do changes ← false
4.           for def1 ∈ MethodDefSet
5.                do for def2 ∈ MethodDefSet
6.                     do if def1 ≠ def2 and moreSpecific(def1,def2)
7.                          then MethodDefSet ← MethodDefSet – {def1}
8.                               changes ← true
9.   return MethodDefSet
```

Figure 10.33 Remove Less Specific Method Definitions.

After we have reduced the set of possible method definitions down to a single definition, semantic analysis is almost complete. We must check for the following special cases of method calls:

- If an unqualified method call appears in a static context (the body of a static method, a static initializer, or an initialization expression for a static variable), then the method called must be static. (This is because the object reference symbol `this` is undefined in static contexts).
- Method calls qualified by a class name (`className.method`) must be to a static method.

- A call to a method that return void may not appear in an expression context (where a value is expected).

The complete process of checking a method call, as developed above, is defined in Figure 10.34.

```
callNode.Semantics( )
 1.  qualifier.ExprSemantics()
 2.  methodSet ← getMethods()
 3.  args.Semantics()
 4.  actualArgsType ← getArgTypes(args)
 5.  for def ∈ methodSet
 6.        do if not applicable(def.argsType,actualArgsType)
 7.              then methodSet ← methodSet – {def}
 8.  throwsSet ← union(qualifier.throwsSet, args.throwsSet)
 9.  terminatesNormally ← true
10.  if size(methodSet) = 0
11.       then GenerateErrorMessage("No Method matches this Call")
12.            return
13.  elsif size(methodSet) > 1
14.       then methodSet ← filterDefs(methodSet)
15.  if size(methodSet) > 1
16.       then GenerateErrorMessage("More than One Method matches this Call")
17.  elsif inStaticContext() and methodSet.member.accessMode ≠ static
18.       then GenerateErrorMessage("Method Called Must Be Static")
19.  elsif inExpressionContext() and methodSet.member.returnType = Void
20.       then GenerateErrorMessage("Call Must Return a Value")
21.        else throwsSet ← union(throwsSet, methodSet.member.declaredThrowsList)
```

Figure 10.34 Semantic Checks for a Method Call

As an example of how method calls are checked, consider the call of `M(arg)` in following code fragment

```
class A { void M(A parm) {...)
          void M() {...} }
class B extends A { void M(B parm) {...}
                    void test(B arg) {M(arg);}}
```

In method `test`, where `M(arg)` appears, three definitions of `M` are visible. All are accessible. Two of the three (those that take one parameter) are applicable to the call. The definition of `M(B parm)` in `B` is more specific than the definition of `M(A parm)` in `A`, so it is selected as the target of the call.

In all rules used to select among overloaded definitions, it is important to observe that the result type of a method is **never** used to decide if a definition is applicable. Java does not allow two method definitions with the same name that have identical parameters, but different result types to co-exist. Neither does C++. For example, the following two definitions force a multiple definition error:

```
int add(int i, int j) {...}
double add(int i, int j) {...}
```

This form of overloading is disallowed because it significantly complicates the process of deciding which overloaded definition to choose. Not only must the number and types of arguments be considered, but also the context within which result types are used. For example in

```
int i = 1 - add(2,3);
```

a semantic analyzer would have to conclude that the definition of `add` that returns a double is inappropriate because a double, subtracted from 1 would yield a double, which cannot be used to initialize an integer variable.

A few languages, like Ada, do allow overloaded method definitions that differ only in their result type. An analysis algorithm that can analyze this more general form of overloading may be found in [Baker 1982].

Interface and Constructor Calls. In addition to methods, Java allows calls to interfaces and constructors. The techniques we have developed apply directly to these constructs. An interface is an abstraction of a class, specifying a set of method definition without their implementations. For purposes of semantic analysis, implementations are unimportant. When a call to an interface is made, the methods declared in the interface (and perhaps its superinterfaces) are searched to find all declarations that are applicable. Once the correct declaration is identified, we can be sure that a corresponding implementation will be available at run-time.

Constructors are similar to methods in definition and structure. Constructors are called in object creation expressions (use of `new`) and in other constructors; they can never be called in expressions or statements. A constructor can be recognized by the fact that it has no result type (not even void). Once a constructor call is recognized as valid (by examining where it appears), the techniques developed above to select the appropriate declaration for a given call can be used.

Subprogram Calls in Other Languages. The chief difference between calls in Java and in languages like C and C++ is that subprograms need not appear within classes. Rather, subprograms are defined at the global level (within a compilation unit). Languages like Algol, Pascal and Modula 2 and 3 also allow subprograms to be declared locally, just like local variables and constants. Some languages allow overloading; other require a unique declaration for a given name.

Processing calls in these languages follows the same pattern as in Java. Using scoping and visibility rules, possible declarations corresponding to a given call are gathered. If overloading is disallowed, the nearest declaration is used. Other wise, a set of possible declarations is gathered. The number and type of arguments in the call is matched against the possible declarations. If a single suitable declaration isn't selected, a semantic error results.