Chapter 9 Semantic Analysis

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: April 27, 2010

current version: August 13, 2023

Chapter outline: Semantic Analysis

1. Semantic analysis of control structures
2. Semantic analysis of calls
3. Summary

Many rules in the specification of a programming language are checked by the semantic routines. They are not reflected in the binary code. This implies that certain binary code cannot be generated from a high-level-language program.

Reference: John B. Goodenough, Exception Handling: Issues and a Proposed Notation, CACM, vol. 18, no. 12, Dec. 1975, pp. 683-696.

Semantic processing of various features of languges:

Example. Exception Handling

1. What is an exception?
2. How are exceptions used in programs?
3. What are the rules in Java regarding exceptions?
4. How do exceptions interact with other features, such as inheritance, of the programming language?
5. How does the compiler enforce the rules of exceptions?
6. What code is generated for the exceptions?

Overview

node type	SemanticsVisitor	ReachabilityVisitor	ThrowsVisitor
IfTesting			
WhileLooping			
DoWhileLooping			
ForLooping			
LabeledStmt			
Continuing			
Breaking			
Returning			
Switching			
CaseItem			
LabelList			
Trying, Catching, Throwing			
Calling			

§9.1 Semantic analysis for control structures

Control structures are used to specify the operations in a program. Common control structures include `if`, `while`, `switch`, `case`, `break`, `return`, `throw`, `continue`, `goto`, etc.

We may design many semantic analysis methods, one for each kind of AST node. We shall focus on three aspects of semantic analysis: type correctness, reachability and termination, and exceptions.

Each construct must have a certain type. For instance, the predicate of an `if` statement must yield a boolean value. We can implement visitor classes to establish type correctness.[a]

Reachability and termination analysis determines if a construct will ever be executed and will terminate normally. Reachability and termination analysis is only a conservative estimation.

Constructs may throw exceptions rather than terminate normally. Java requires accounting for all checked exceptions.

[a] Type inference is more difficult than type checking.

For exceptions, each AST node that contains an expression or a statement have a `throwsSet` field. This field contains the set of exception types that may be thrown in the subtree rooted at the AST node. It will be propagated as AST is analyzed.

In summary, there are three visitor classes in this chapter:

1. `SemanticsVisitor`: check the types of predicates, parameters, etc.
2. `ReachabilityVisitor`: analyze the control structures for reachability and proper termination.
3. `ThrowsVisitor`: collect information for throws that may "escape" from a given construct.

Three rules for Java exceptions:

1. All checked exceptions that may occur in a procedure must be caught by that procedure or are listed in that procedure's `throws` clause.

2. All checked exceptions that may be propagated to a caller from a procedure must be listed in that procedure's `throws` clause.

3. All checked exceptions that are listed in a procedure's `throws` clause must be propagated to a caller from that procedure in some situations.

§9.1.1 Reachability and termination analysis

Certain languages, such as Java, requires unreachable statements be identified. Here is an example of unreachable statement:

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

The assignment statement is unreachable.

Though it is obvious certain statements are unreachable, it is undecidable in the general case even if we know all the input data.

We add two boolean fields `isReachable` and `terminatesNormally` to AST nodes that represent statements and statement lists.

The reachability and termination analysis is *conservative*. Unless the analysis shows a statement definitely terminates (or is definitely reachable, respectively), its `terminatesNormally` (or `isReachable`) flag is set to false.

Definition. A statement *terminates normally* if execution continues to the next statement.

According this criterion, statements such as `break`, `continue`, `return`, etc., do not terminate normally. An infinite loop, such as

```
for ( ; ; ) { a = a + 0; }
```

does not terminate normally, either.

The rules for `isReachable` and `terminatesNormally` are as follows:

1. If `isReachable` is true for a statement list, it is also true for the first statement in the list. (top-down)
2. If `terminatesNormally` is false for the last statement in a statement list, it is also false for the whole statement list. (bottom-up)
3. The statement list that comprises the body of a method, constructor, or a static initializer is always considered reachable.
4. A local variable declaration or an expression statement (assignment, method call, heap allocation, variable increment or decrement) always have the `terminatesNormally` true (even if the statement is not reachable).
5. A null statement or a null statement list never generates an error message if its `isReachable` is false. Rather, the `isReachable` value is propagated to the successor.
6. A statement is reachable if and only if its predecessor terminates normally. (Note that the predecessor may not be reachable.)

Consider the following example:

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

The method body is assumed to be reachable. So the declaration is reachable. The declaration and the increment terminate normally. So the `return` statement is reachable but does not terminate normally. The `null` statement is not reachable. So the first assignment is not reachable and hence generates an error message. However, the first assignment is considered to terminate normally (in order not to generate additional error messages). Hence, the second assignment is reachable and terminates normally. Hence the body of the function terminates normally.

We require most statements to terminate normally. However, a function that returns a non-void value must execute a `return` or throw an exception. Thus, the function body must not terminate normally otherwise an error message will be issued.

§9.1.2 IF statements

The abstract syntax tree for an `if` statement is shown in Figure 9.2.

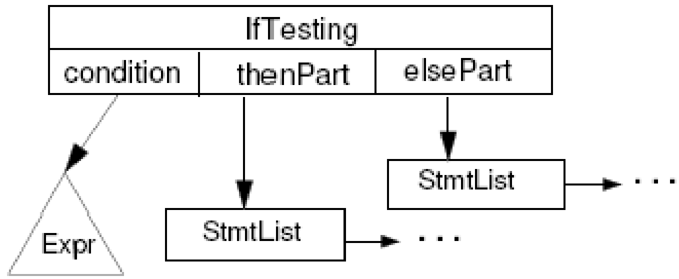


Figure 9.2: Abstract Syntax Tree for an If Statement

For an `if` statement, the compiler will check whether the predicate has the `boolean` or `errorType` type. Otherwise an error message is issued. The `SemanticsVisitor` for `IfTesting` first decides the types of the predicate, and process the `then` and the `else` parts with the `VisitChildren` call. Finally, it calls `CheckBoolean` on the predicate.

It does not care if the predicate is a constant.

For reachability analysis, the predicate is assumed to yield either true or false, that is the predicate always terminates normally (even if it is not reachable). Thus, the `then` and `else` parts are both reachable. On the other hand, an `if` statement terminates normally if either the `then` or the `else` part terminates normally.

It is also possible that an exception is thrown in the three subtree of `IfTesting`. The `GatherThrows(IfTesting)` (Figure 9.4) call gathers all possible exceptions in the three subtrees.

Consider the following example:

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

First the compiler checks that the predicate `b` produces a boolean value. The two branches will be type-checked as well to ensure they are valid statements. Since the two assignments always terminate normally, so does the whole `if` statement.

```
class NodeVisitor
   procedure VisitChildren(n)
      foreah c in n.GetChildren() do call c.Accept(this)
   end
end

class SemanticsVisitor extends NodeVisitor
   procedure CheckBoolean(c)
      if c.type != Boolean and c.type != errorType
      then call error("Need a boolean.")
   end

   procedure Visit(IfTesting ifn)
      call VisitChildren(ifn)
      call CheckBoolean(ifn.condition)
   end

   procedure Visit(WhileLooping wn)    ... later ...  end
```

```
    procedure Visit(DoWhileLooping dwn) ... later ...  end

    procedure Visit(ForLooping fn)      ... later ...  end

    procedure Visit(LabeledStmt ls)  // fig 9.11

    procedure Visit(Continuing cn)   // fig 9.12

    procedure Visit(Breaking bn)     // fig 9.15

    procedure Visit(Returning rn)    // fig 9.18
end
```

Figure 9.1 Semantic Analysis Visitors

```
class ReachabilityVisitor extends NodeVisitor
   procedure Visit(IfTesting ifn)
      ifn.thenPart.isReachable := true
      ifn.elsePart.isReachable := true
      call VisitChildren(ifn)
      thenNormal := ifn.thenPart.terminatesNormally
      elseNormal := ifn.elsePart.terminatesNormally
      ifn.terminatesNormally := thenNormal or elseNormal
   end

   procedure Visit(WhileLooping wn)      // fig 9.6

   procedure Visit(DoWhileLooping dwn)  // fig 9.7

   procedure Visit(ForLooping fn)       // fig 9.8

   procedure Visit(LabeledStmt ls)
      ls.stmt.isReachable := ls.isReachable
```

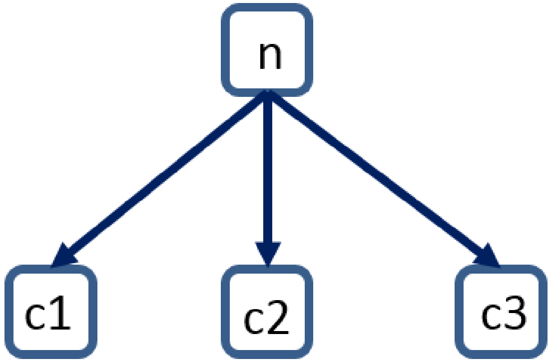
```
        call VisitChildren(ls)
        ls.terminatesNormally := ls.stmt.terminatesNormally
    end

    procedure Visit(Continuing cn)
        cn.terminatesNormally := false
    end

    procedure Visit(Breaking bn)        // fig 9.16

    procedure Visit(Returning rn)
        rn.terminatesNormally := false
    end
end
```

Figure 9.3 Reachability Analysis Visitors



GatherThrows(n):

n.throwsSet := c1.throwsSet ∪ c2.throwsSet ∪ c3.throwsSet

```
class ThrowsVisitor extends NodeVisitor
   procedure GatherThrows(n)
      call VisitChildren(n)
      ans := { }
      foreach c in n.GetChildren() do ans := ans union c.throwsSet
      n.throwsSet := ans
   end

   procedure Visit(IfTesting ifn)
      call GatherThrows(ifn)
   end

   procedure Visit(WhileLooping wn)
      call GatherThrows(wn)
   end

   procedure Visit(DoWhileLooping dwn)
      call GatherThrows(dwn)
   end
```

```
procedure Visit(ForLooping fn)
   call GatherThrows(fn)
end

procedure Visit(LabeledStmt ls)
   call GatherThrows(ls)
end

procedure Visit(Continuing cn)
   cn.throwsSet := { }
end

procedure Visit(Breaking bn)
   bn.throwsSet := { }
end

procedure Visit(Returning rn)
   call GatherThrows(rn)
end
```

```
    procedure Visit(Switching sn)
        call GatherThrows(sn)
    end

    procedure Visit(CaseItem cn)
        call GatherThrows(cn)
    end

    procedure Visit(labelList lln)
        // Constant-valued expressions cannot throw exceptions
        lln.throwsSet := { }
    end
end
```

Figure 9.4 Throws analysis visitors

§9.1.3 While, Do and Repeat loops

The AST for a `while` statement is shown in Figure 9.5.

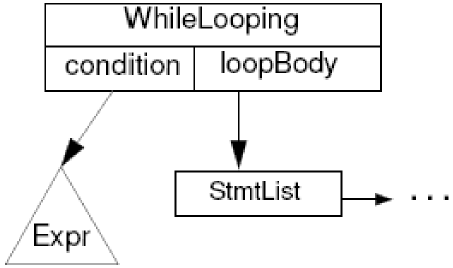


Figure 9.5: Abstract Syntax Tree for a While Statement

```
procedure       ( WhileLooping wn )
    wn.terminatesNormally ← true                                    (21)
    wn.loopBody.isReachable ← true
    constExprVisitor ← new ConstExprVisitor( )
    call wn.condition.        ( constExprVisitor )
    conditionValue ← wn.condition.exprValue
    if conditionValue = true
    then
        wn.terminatesNormally ← false                               (22)
    else
        if conditionValue = false
        then
            wn.loopBody.isReachable ← false                         (23)
    call wn.loopBody.        ( this )                               (24)
end
```

Figure 9.6: Reachability Analysis for a While Statement

A `WhileLooping` node is very similar to an `IfTesting` node, except that there are two subtrees, one for the predicate and one for the loop body. The predicate must have the `boolean` type and the loop body must be type-checked for validity. See the `Visit` method in Figure 9.1.

The `ReachabilityVisitor` for a `WhileLooping` node is shown in Figure 9.6. The `ConstExprVisitor` attempts to decide if the predicate is a constant (i.e., always true or always false). In Figure 9.6, if the predicate is always false, the loop body is marked as unreachable. On the other hand, if the predicate is always true (i.e., an infinite loop), the loop body will not terminate normally. However, if the loop body has a reachable `break` statement, the

```
call wn.loopBody.Accept(this)
```

will reset the `terminatesNormally` flag of the `WhileLooping` node.

If the predicate is not a constant, the `terminatesNormally` flag remains true because we assume the loop will terminate after some iterations.

```
procedure Visit(WhileLooping wn)
   call VisitChildren(wn)
   call CheckBoolean(wn.condition)
end

procedure Visit(DoWhileLooping dwn)
   call VisitChildren(dwn)
   call CheckBoolean(dwn.condition)
end
```

Semantic Analysis of a while-loop (from Figure 9.1)

```
procedure Visit(WhileLooping wn)
   wn.terminatesNormally := true
   wn.loopBody.isReachable := true
   constExprVisitor := new ConstExprVisitor()
   call wn.condition.Accept(constExprVisitor)
   conditionValue := wn.condition.exprValue
   if conditionValue == true        // i.e., always true
   then wn.terminatesNormally := false
   else if conditionValue == false  // i.e., always false
        then wn.loopBody.isReachable := false
   // else conditionValue may be true or false
   call wn.loopBody.Accept(this)
end
```

Figure 9.6 Reachability analysis for a While statement

If an expression always yields a constant value as determined by the `ConstExprVisitor`, the value is set to the `exprValue` field of the expression. A `ConstExprVisitor` could be simple or sophisticated. A simple `ConstExprVisitor` recognizes only expressions made of literals and operators. A sophisticated `ConstExprVisitor` depends on constant propagation to recognizes more constant expressions.

Exceptions may be generated in the predicate and the loop body. The `ThrowsVisitor` (in Figure 9.4) for a `WhileLooping` node collects all the possible exceptions in the loop.

Test cases for your `ConstExprVisitor`:

- `true`
- `3 == 3`
- `2 + 1 == 3`
- `2 + 1 == 5 - 2`
- `a == a`
- `3 - 3 + a == a`
- `3 + a - 3 == a`
- `3 + a - 3 == a`
- `3 + a - 2 -1 == a`
- `2 + a - 3 + 1 == a`

Consider the following example:

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

The compiler first checks if the predicate "`i >= 0`" is a valid boolean expression. Then the loop body is checked for semantic errors. Since the predicate is not a constant, the loop body is assumed to be reachable and the loop is marked as `terminatesNormally`.

Do-While and Repeat loops

The `do-while` and `repeat` loops are similar to a traditional `while` loop. They may share the same AST (as in Figure 9.5). The `SemanticsVisitor` (Figure 9.1) and `ThrowsVisitor` (Figure 9.4) remain the same.

The `ReachabilityVisitor` for `do-while` is shown in Figure 9.7. For a `do-while` node, the loop body is always reachable whether or not the predicate is the constant false.

The `terminatesNormally` flag of the loop is set to false initially. It may be set to true if the loop body contains a reachable `break` statement. This is done during the following call:

```
call dwn.loopBody.Accept(this)
```

The loop also terminates normally if the predicate is not a constant true and if the loop body terminates normally (which is determined during the call:

```
call dwn.loopBody.Accept(this)
```

A `repeat` loop is essentially a `do-while` loop except the the loop ends when the predicate becomes true.

```
procedure Visit(DoWhileLooping dwn)
   dwn.loopBody.isReachable := true
   dwn.terminatesNormally   := false
   call dwn.loopBody.Accept(this)
   constExprVisitor := new ConstExprVisitor()
   call dwn.condition.Accept(constExprVisitor)
   conditionValue   := dwn.condition.exprValue
   if conditionValue != true // i.e., not always true
   then bodyNormal := dwn.loopBody.terminatesNormally
        dwn.terminatesNormally :=
            dwn.terminatesNormally or bodyNormal
end
```

Figure 9.7 Reachability analysis for a Do-While statement

Question: How to change Figure 9.7 for a `repeat` loop?

§9.1.4 For loops

A `for` loop is used for an index to go through a range in a regular way. In C, C++, C#, and Java, a `for` loop is a `while` loop.

The AST for a `for` loop is shown in Figure 9.9. There are four subtrees: `initializer`, `condition`, `increment`, and `loopBody`.

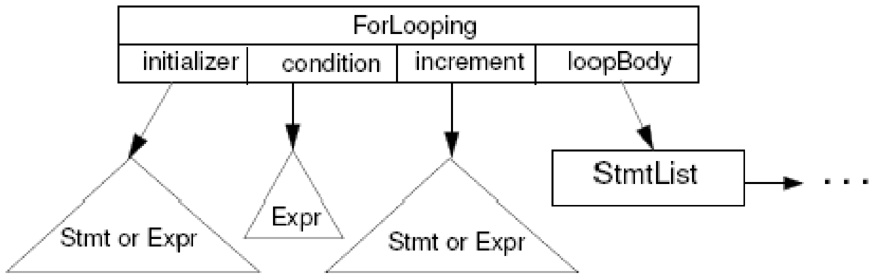


Figure 9.9: Abstract Syntax Tree for a For Loop

There are a few variations. The `condition` could be null, which means an infinite loop. In C++, C#, and Java, a new index variable local to the `for` loop may be declared. So a new symbol-table name scope must be opened and will be closed later.

```
procedure Visit(ForLooping fn)
   call OpenScope()
   call VisitChildren(fn)
   if fn.condition != null
   then call CheckBoolean(fn.condition)
   call CloseScope()
end
```

Semantic Analysis of a for-loop (from Figure 9.1)

```
procedure Visit(ForLooping fn)
   fn.loopBody.isReachable := true
   fn.terminatesNormally   := true
   if fn.condition != null
   then constExprVisitor := new ConstExprVisitor()
        call fn.condition.Accept(constExprVisitor)
        conditionValue := fn.condition.exprValue
        if conditionValue == true // i.e., always true
        then fn.terminatesNormally := false
        else if conditionValue == false // i.e., always false
             then fn.loopBody.isReachable := false
   else fn.terminatesNormally := false // always true
   call fn.loopBody.Accept(this)
end
```

Figure 9.8 Reachability analysis of a for-loop

The `Visit(ForLooping)` of `SemanticsVisitor` in Figure 9.1 first opens a new scope, then visits each children for semantic check, and checks the predicate for the boolean type.

The `Visit(ForLooping)` of `ReachabilityVisitor` in Figure 9.8 is very similar to the `Visit(DoWhileLooping)` in Figure 9.7. A null predicate and a constant-`true` predicate make the loop non-terminating. On the other hand, a constant-`false` predicate makes the loop body unreachable.

The `Visit(ForLooping)` of `ThrowsVisitor` in Figure 9.4 simply gathers all the `throwsSet`s.

Consider the following `for` loop:

```
for (int i = 0; i < 10; i ++) { a = a + i; }
```

A new name scope is created for the loop index `i`. The loop predicate has the boolean type and is not a constant, the loop is considered reachable and is assumed to terminate normally. Finally, the new scope will be closed.

There are variations of the `for` loops:

```
for x := 1 to 20 step 3 do y := y + 5;
```

The loop index could be an existing variable (hence, not creating a new scope). Some programming languages enforce that the index variable cannot be changed inside the loop.

The initial, final, and step values must have the same type as the loop index.

Question: The above code did not consider the following `for`-loop will never execute:

```
for (i = 10; i < 9; i ++) { . . . }
```

How to fix this problem?

§9.1.5 Labels, continue, break, return, and goto statements

Continue

The `continue` statement transfers control to the bottom of the loop in order to start a new iteration. The compiler must make sure that a `continue` may appear only within a loop. A loop label may be specified in a `continue` statement. The label must reference an enclosing loop.

Every statement in Java may carry a label. The AST of a `LabeledStmt` is shown in Figure 9.10, which contains a string-valued `stmtLabel` field and a `stmt` field that references another `LabeledStmt` node.

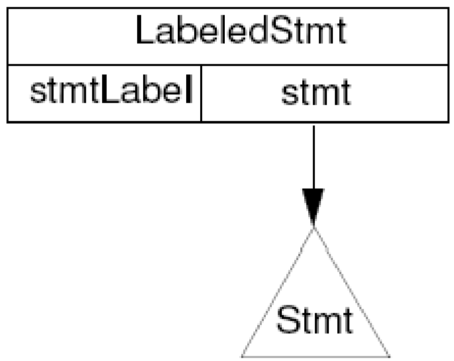


Figure 9.10: Abstract Syntax Tree for a Labeled Statement

In most programming languages, such as Java, C, C++, the same identifier may be used as a label and as a variable. The name space of labels is separate. This is because labels are used in very limited contexts (in `continue`, `break`, `goto`) and cannot be assigned to variables, returned from functions, read from files, etc.

We will maintain a list of labels that are visible to the AST node that is being analyzed. The `GetLabelList` function returns this list of visible labels. `SetLabelList` sets up this list. The list of visible labels is null at the beginning of a method, a constructor, or a static initializer.

Figure 9.13 shows an example of a label list. There are three fields in a `LabelList` node:

1. `label`: which is a string or null.
2. `kind`: which could be *iterative*, *switch*, or *other*.
3. `AST`: which points to the AST node of the labelled statement.

Example. Consider the following code fragment:

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

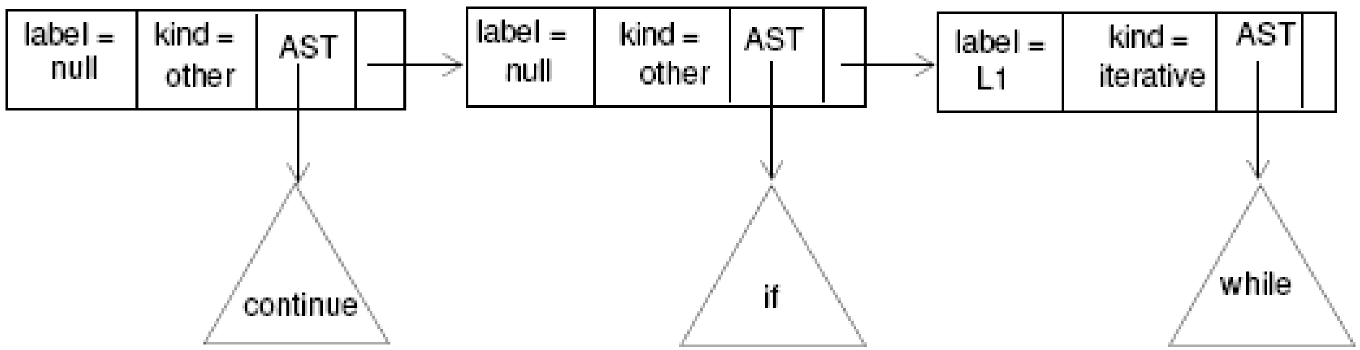


Figure 9.13: Example of a label list in a Continue Statement

```
procedure Visit(LabeledStmt ls)
  newNode := new LabelList(ls.stmtLabel, GetKind(ls.stmt), ls.stmt)
  newList := cons(newNode, GetLabelList())  //add a new label
  call SetLabelList(newList)
  call VisitChildren(ls)
  call SetLabelList( tail(getLabelList()) ) //remove the new label
end
```

Figure 9.11 Semantic analysis of a labeled statement

The `Visit(LabeledStmt)` function in Figure 9.11 adds the label (which could be null) of the current statement to the (front of the) list of visible labels, then visits each child in turn, and finally removes that new label node.

Note that in a list of visible labels, such as Figure 9.13, the label of the innermost enclosing statement is at the front of the list.

The `ReachabilityVisitor(LabeledStmt)` is shown in Figure 9.3, p. 16.

The `ThrowsVisitor(LabeledStmt)` is shown in Figure 9.4, p. 26, which simply gathers all possible exceptions.

```
procedure Visit(LabeledStmt ls)
   ls.stmt.isReachable := ls.isReachable
   call VisitChildren(ls)
   ls.terminatesNormally := ls.stmt.terminatesNormally
end
```

Figure 9.3 Reachability Analysis of a labeledStmt
(from Figure 9.3)

§ `Continue` statement

Two kinds of `continue`:

- `continue`: trasfer to the end of the innermost enlosing loop.
- `continue label5`: trasfer to the end of the designated enlosing loop.

```
procedure Visit(Continuing cn)
   currentList := GetLabelList()
   if cn.stmtLabel == null
   then while currentList != null do  // find innermost iterative
           currentLabel := head(currentList)
           if currentLabel.kind == iterative then return
           currentList := tail(currentList)
        call error("Continue not inside an iterative stmt")
   else while currentList != null do  // search for label
           currentLabel = head(currentList)
           if currentLabel.label == cn.stmtLabel and
              currentLabel.kind  == iterative then return
           currentList := tail(currentList)
        call error("Continue label does not match an iterative stmt")
end
```

Figure 9.12 Semantic analysis of a continue statement

The `Visit(Continuing)` function in Figure 9.12 first finds the target of the `continue` statement.

If the `continue` statement has no label, its target is the innermost iterative statement (e.g., `while`, `repeat`, or `do`). The `Visit(Continuing)` finds the innermost enclosing iterative statement.

Semantic analysis: Is there a target?

If the `continue` statement has a label, the `Visit(Continuing)` finds the iterative statement with that label in the label list from front to end (i.e., from innermost to outermost).

The `ReachabilityVisitor(Continuing)` is shown in Figure 9.3, p. 16. A `continue` statement never terminates normally.

The `ThrowsVisitor(Continuing)` is shown in Figure 9.4, p. 26, which empties the `throwsSet`. A `continue` statement causes no exceptions.

```
procedure Visit(Continuing cn)
   cn.terminatesNormally := false
end
```

Figure 9.3 Reachability Analysis of continue (from Figure 9.3)

Example. Consider the following code fragment:

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

The `continue` statement is inside the `if` statement, which, in turn, is inside the `while` loop. So the list of visible labels contains three items for the `continue` statement, shown in Figure 9.13. The `continue` statement has an appropriate target `L1`.

For C and C++, things are even simpler because a `continue` statement does not carry a label.

Break statements

Two kinds of `break`:

- `break`: trasfer to the successor of the innermost enlosing loop or `switch` statement.
- `break label6`: trasfer to the successor of the designated enlosing loop or `switch` statement.

In Java, an unlabelled `break` is similar to an unlabelled `continue` statement, which breaks out the innermost enclosing `while`, `do`, `for`, or `switch` statement. Thus, a reachable `break` forces the statement it references to terminate normally.

A `break` with a label exits the innermost enclosing statement with the matching label and execution continues to the following statement. The semantic analysis must verify that a suitable target for the `break` exists.

The `FindBerakTarget(Breaking)` method in Figure 9.14 (p. 50) finds the target of a `break`. It calls `GetlabelList` to obtain all the visible labels. For an unlabelled `break`, it locates the innermost *switch* or *iterative* statement. For a labelled `break`, it locates the innermost *switch* or *iterative* statement with a matching label.

```
function FindBreakTarget(Breaking bn) returns Label
   currentList := GetLabelList()
   if bn.stmtLabel == null
   then while currentList != null do  // search innermost switch
                                      // or iterative label
           currentLabel := head(currentList)
           if currentLabel.kind == switch or
              currentLabel.kind == iterative
           then return(currentLabel)  // a label
           currentList := tail(currentList)
        return(null)
   else while currentList != null do  // search
           currentLabel = head(currentList)
           if currentLabel.label == bn.stmtLabel
           then return(currentlabel)  // a label
           currentList := tail(currentList)
        return(null)
end
```

Figure 9.14 Function to find the target of a break

The semantic analysis for a `break` (in Figure 9.15, p. 49) makes sure that `FindBreakTarget` can find a valid target.

Question. There is a minor bug in the algorithm in Figure 9.14. Can you find and fix it?

```
procedure Visit(Breaking bn)
   target := FindBreakTarget(bn)
   if target == null     // cannot find a target
   then if bn.stmtLabel == null
        then call error("Break not inside a switch or iterative stmt")
        else call error("Break label does not match any label.")
end
```

Figure 9.15 Semantic analysis for a break

In the `ReachabilityVisitor(Breaking)` in Figure 9.16, a `break` statement never terminates normally. On the other hand, if a `break` is reachable, the target of the `break` will terminate normally. This method can analyze a `do-forever` loop that terminates by a `break` statement.

```
procedure Visit(Breaking bn)
   bn.terminatesNormally := false
   target := FindBreakTarget(bn)
   if target != null and bn.isReachable
      // target is a loop or a switch statement
   then target.AST.terminatesNormally := true
end
```

Figure 9.16 Reachability analysis for a break

In the following example, the `for` loop, which is an infinite loop, will be marked as not `terminatesNormally` (in the `Visit(ForLooping)` in Figure 9.8, p. 31, line 8). The `Visit(Breaking)` in Figure 9.16 will mark `terminatesNormally` as true again.

```
for { ; ; } {
    . . .
    break;
    . . .
}
```

The `ThrowsVisitor` in Figure 9.4 on page 27 simply notes that the `throwsSet` is empty for a `break` statement.

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;
          . . .
```

The `break` is inside the `if` statement, which is inside the `for` loop, which is inside another `for` loop. The list of visible labels is shown in Figure 9.17. `FindBreakTarget` will find the correct label `L1` in the list.

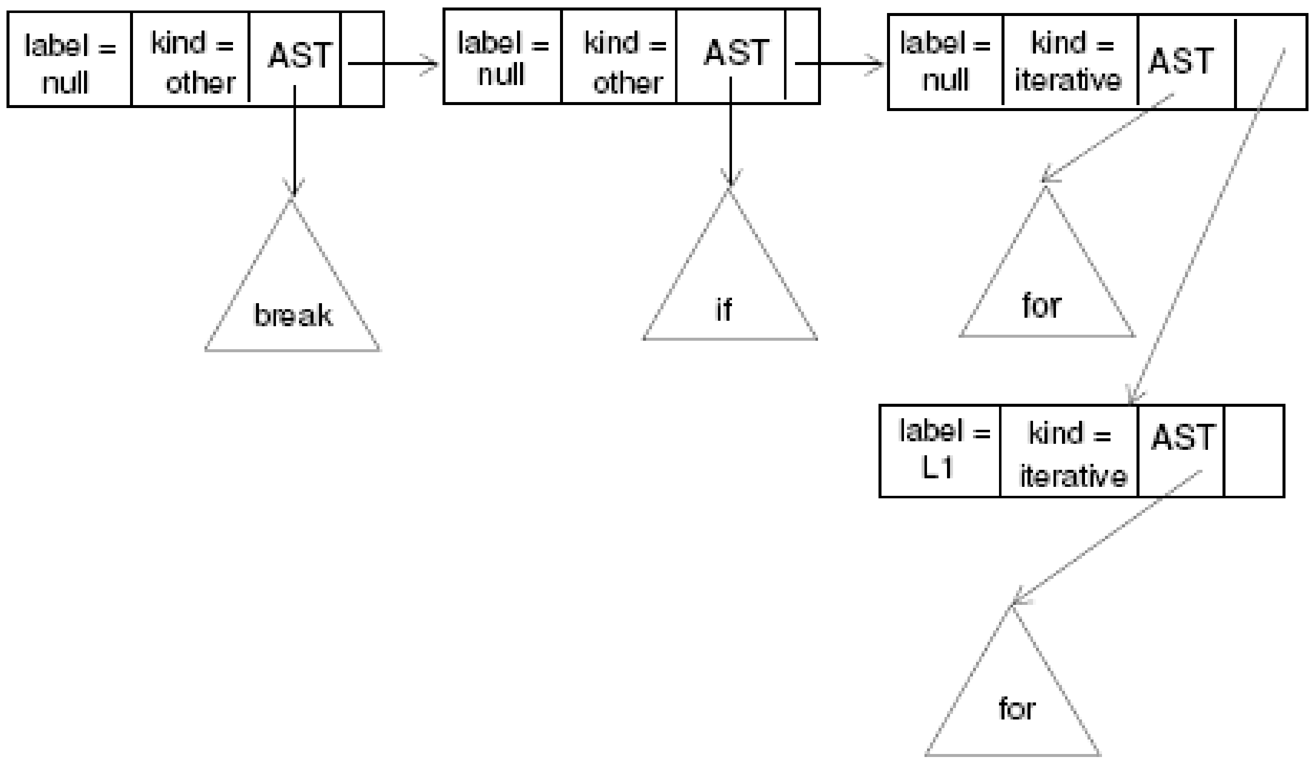


Figure 9.17: Example of a label list in a Break Statement

```
procedure       (Returning rn)
    call      C         (rn)
    currentMethod ←    C        M       (rn)
    if rn.returnVal ≠ null
    then
        if currentMethod = null
        then
            call        ("A value may not be returned from a constructor")
        else
            if not               (currentMethod.returnType, rn.returnValue.type)
            then call         ("Illegal return type")
    else
        if currentMethod ≠ null and currentMethod.returnType ≠ void
        then   call        ("A value must be returned")
end
```

Figure 9.18: Semantic Analysis for a Return

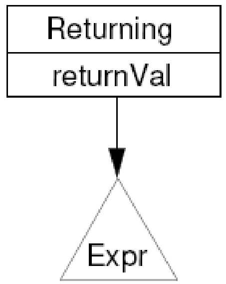


Figure 9.19: Abstract Syntax Tree for a Return Statement

Return statements

The AST for a **`return`** is shown in Figure 9.19. If no value is returned, **`returnVal`** is null. Otherwise it is an AST node representing an expression.

```
procedure Visit(Returning rn)
    call VisitChildren(rn)   // visit the expression child
    currentMethod := GetCurrentMethod(rn)
    if rn.returnVal != null
    then if currentMethod == null
         then call error("Constructor does not return a value")
         else if not Assignable(currentMethod.returnType,
                                rn.returnVal.type)
              then call error("illegal return type")
    else if currentMethod != null and
            curentMethod.returnType != void
         then call error("A value must be returned")
end
```

Figure 9.18 Semantic analysis for a return statement

A function must either return a value (or `void` if declared so) or throw an exception (if the `terminatesNormally` flag of the statement list of the function's body is false). The semantic analysis, shown in Figure 9.18, needs to check the type of the return value. In C, all functions may return without a value.

There are two auxiliary functions:

1. `GetCurrentMethod`
2. `GetCurrentConstructor`

The two methods return a reference (as an `Attributes` structure) to the current method or constructor or null if not applicable.

The `Assignable` function enforces the type compatibility rules of the programming language.

```
procedure Visit(Returning rn)
   call VisitChildren(rn)
   currentMethod := GetCurrentMethod(rn)
   if rn.returnVal != null
   then if currentMethod == null
        then call error("Constructor does not return a value")
        else if not Assignable(currentMethod.returnType,
                               rn.returnValue.type)
             then call error("Illegal return type")
   else if currentMethod != null and
           currentMethod.returnType != void
        then call error("A value must be returned.")
end
```

Figure 9.18 Semantic analysis for a return statement

§ Goto statements

Java allows no goto statements. C, C++, C#, etc., allow *intraprocedural* goto statements. We may write the following statement in C:

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

Identifiers used as labels are kept in a separate table (the `declaredlabels`) since they are in a different name space. Because programming languages allow *forward goto*, checking labels is performed in two steps:

1. First the AST representing the entire function body is traversed and labels are collected in the `declaredlabels` table. During this time, *duplicate labels* are also checked.

2. Later, when a `goto` statement is checked, the label is located in the table.

The difference between `goto` and `continue`/`break` is a `goto` must not be nested inside the statement with the destination label.

Furthermore, `continue`/`break` can reference a label that has already been seen while `goto` may reference to a label that has not occurred yet.

```
     goto L9
     ...
     ...
L9:  ...
```

Some languages allow *non-local goto*. For non-local goto, the compiler maintains a stack of `declaredlabels` tables, one for each nested procedure. A `goto` is valid if its target appears in one of these tables.

```
proc A {
    proc B {
        proc C {
            goto L1;
        }
        . . .
    }
    . . .
    L1: . . .
    . . .
}
```

It is also reasonable to forbid transferring control to arbitrary point in a procedure. For instance, we should *not* jump into the middle of a loop directly. (Why?) Thus, even if the scope of a label is the entire procedure, there are places where the label cannot be referenced in a `goto` statement. We may mark individual labels as *active* or *inactive* for this restriction. For example, a label is active only within its loop body.

There are long-jump and set-jump in C.

§9.1.6 Switch and case statements

Example. This example has three `CaseItem`s.

```
switch (p):
  case 2:
  case 3:
  case 5:
  case 7:  isPrime = true; break;
  case 2*2:
  case 2*3:
  case 16/2:
  case 9:  isPrime = false; break;
  default: isPrime = checkIfPrime(p);
}
```

The AST for a `switch` statement is shown in Figure 9.20.

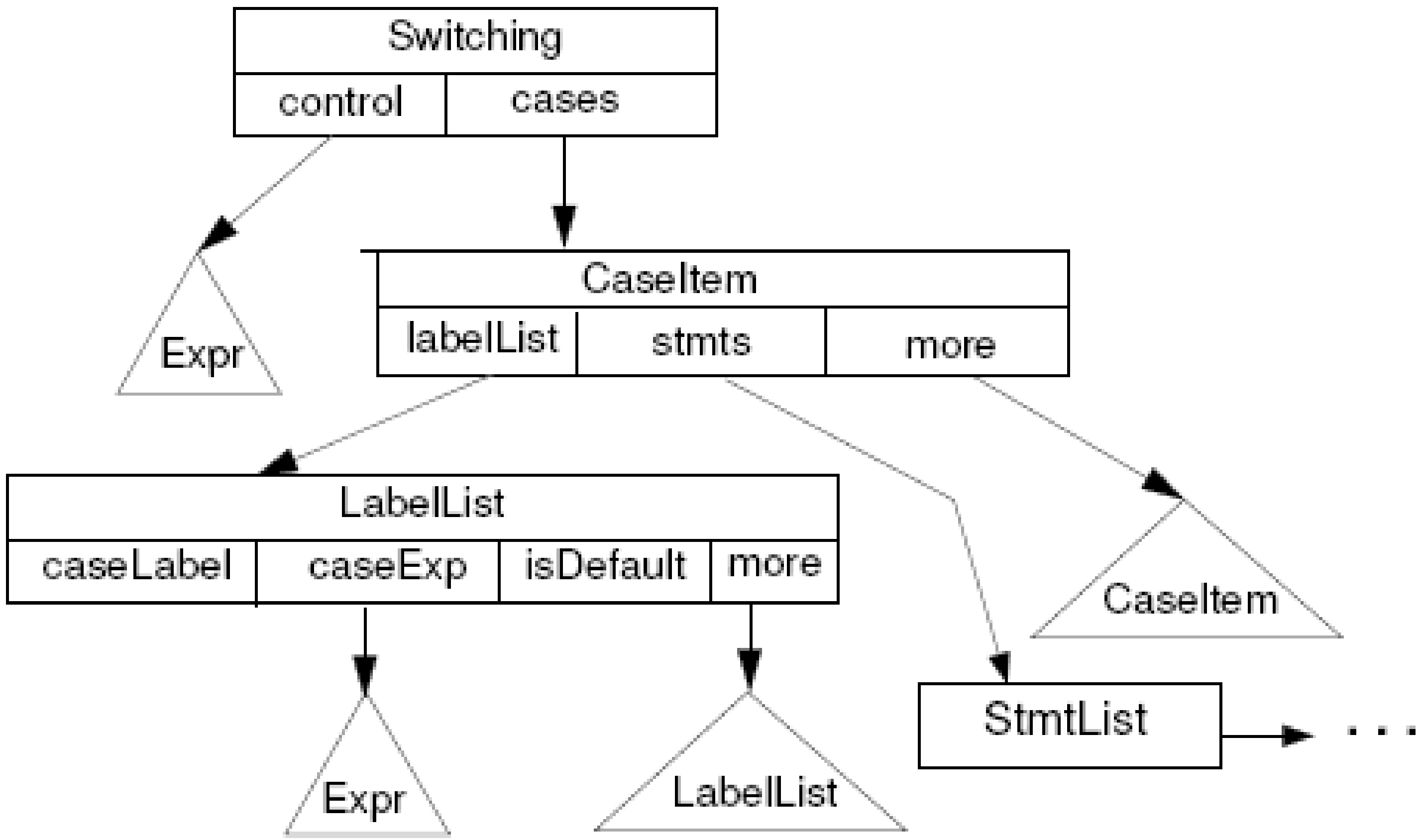


Figure 9.20: Abstract Syntax Tree for a Switch Statement

There is a `control` and a list of `CaseItems`. Each `CaseItem` has a `labelList`, a `stmts`, a `more` field, which points to the remaining `CaseItems`.

A `labelList` contains a `caseLabel`, a `caseExp` (which points to the AST node of an `Expr`), an `isDefault` field (which is a boolean field), and a `more` field (which points to the remaining `labelList` in the same `caseItem`).

The `caseExp` points to an AST node that represents a constant expression. After it is evaluated, its value is stored in the `caseLabel`.

The control and all the statements in the `case` body must be type-checked. The `control` must have an integer (or enumeration) type. Each `case` label must have the same type as the `control`. No two `case` labels may have the same value. There may be at most one default label.

Quiz. Is a `caseExp` a constant expression?

Optimization opportunity. What if the `control` is a constant expression?

```
class NodeVisitor
   procedure VisitChildren(n)
      foreach c in n.GetChildren() do call c.Accept(this)
   end

class SemanticsVisitor extends NodeVisitor // extends Fig 9.1
   procedure Visit(Switching sn)
      call sn.control.Accept(this)  // evaluate the control
      if sn.control.type != errorType and
         not Assignable(int, sn.control.type)
      then call error("Illegal type for a control exp.")
           call SetSwitchType(errorType)
      else call SetSwitchType(sn.control.type)
      call sn.cases.Accept(this)    // evaluate all caseItems
      labelList := Sort(GatherLabels(sn.cases))
      call CheckForDuplicates(labelList)
      if CountDefaults(sn.cases) > 1
      then call error("More than one default case label")
      // default should also appear as the last case.
   end
```

```
    procedure Visit(CaseItem cn)
       call VisitChildren(cn)     // recursive call
    end

    procedure Visit(LabelList lln)
       call VisitChildren(lln)    // recursive call
       lln.caseLabel := null
       if lln.caseExp.type != errorType
       then if not Assignable(GetSwitchType(), lln.caseExp.type)
            then call error("Invalid case label type")
            else constExprVisitor := new ConstExprVisitor()
                 call lln.caseExp.Accept(constExprVisitor)
                 labelValue := lln.caseExp.exprValue
                 if labelValue == null    // not a constant
                 then call error("Label must be a constant")
                 else lln.caseLabel = labelValue
    end
end
```

Figure 9.21 Semantic analysis of a swich statement

The semantics visitor for a `switch` statement is shown in Figure 9.21. Utility functions are shown in Figure 9.22.

The `GatherLabels` collects all labels in a `labelList` in an integer list. Note that `GatherLabels` is an overloaded function.

The `CheckForDuplicates` takes a sorted integer list and checks if there are duplicates labels in the list.

The `CountDefaults` counts the number of defaults in a list of `caseItem`s. Note that `CountDefaults` is also an overloaded function.

The `Visit(LabelList)` in Figure 9.21 uses a `ConstExprVisitor` to evaluate the expression (which is supposed to be a constant). The result is put in the `exprValue` field. If the expression does not yield a constant, a null is used instead.

```
function GatherLabels(CaseItem cn) return intList
   if cn.more == null
   then return( GatherLabels(cn.labelList) )
   rest := GatherLabels(cn.more)
   return( Append(Gatherlabels(cn.labelList), rest) )
end

function GatherLabels(LabelList lln) returns intList
   if lln == null then return(null)
   rest := GatherLabels(lln.more)
   if lln.caseLabel == null then return(rest)  // non-constant
   return( cons(lln.caseLabel, rest) )
end

procedure CheckForDuplicates(intList il)
   if Length(il) > 1
   then if head(il) == head(tail(il))
        then call error("Duplicate case label")
        else call CheckForDuplicates(tail(il))
end
```

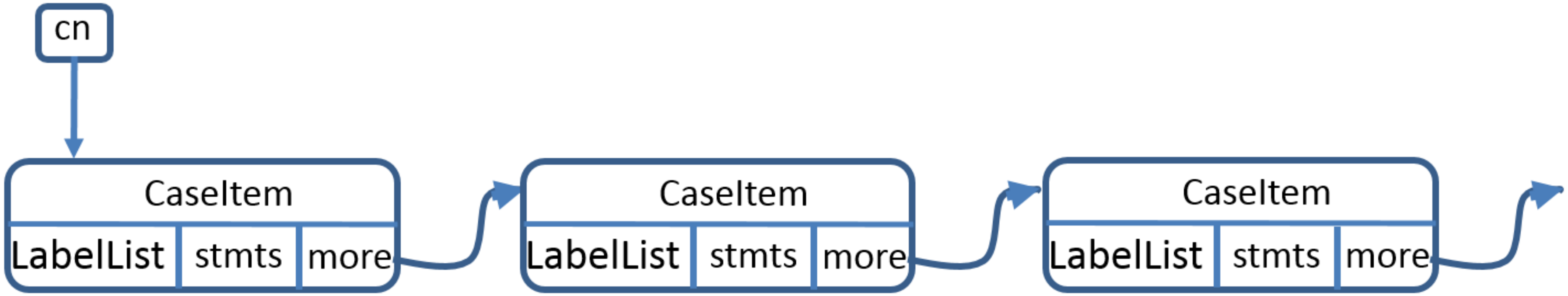
```
function CountDefaults(CaseItem cn) returns int
   if cn == null then return(0)
   return(CountDefaults(cn.labelList) + CountDefaults(cn.more))
end

function CountDefaults(LabelList lln) returns int
   if lln == null then return(0)
   if lln.isDefault then return( 1+ContDefaults(lln.more) )
   return( ContDefaults(lln.more) )
end
```

Figure 9.22 Utility semantic methods for switch statements

Tail recursion:

```
CountDefaults(cn) = CountDefaults(cn.labelList) +
                    CountDefaults(cn.more)
```



The `ReachabilityVisitor(Switching)` is shown in Figure 9.23, p. 75.

An empty `switch` body trivially terminates normally.

If the last `caseItem` terminates normally, so does the entire `switch` statement since the execution falls through to the succeeding statements.

If any of the statements in the `switch` body contains a reachable `break`, the entire `switch` terminates normally.

```
class ReachabilityVisitor extends NodeVisitor // extends fig 9.3
   procedure Visit(Switching sn)
      sn.terminatesNormally := false
      call VisitChildren(sn)
      if sn.cases == null
      then sn.terminatesNormally := true
      else sn.terminatesNormally :=
           sn.terminatesNormally or sn.cases.terminatesNoramlly
   end

   procedure Visit(CaseItem cn)
      cn.stmts.isReachable := true
      call VisitChildren(cn)
      if cn.more == null
      then cn.terminatesNormally := cn.stmts.terminatesNormally
      else cn.terminatesNormally := cn.more.terminatesNormally
   end
end
```

Figure 9.23 Reachability analysis for a switch statement

Example.

```
switch (p):
  case 2:
  case 3:
  case 5:
  case 7:  isPrime = true; break;
  case 4:
  case 6:
  case 8:
  case 9:  isPrime = false; break;
  default: isPrime = checkIfPrime(p);
}
```

In the above example, the expression `p` is checked for a valid integer expression. The label list is built by examining each `caseItem` and `labelList`. We verify that each case label is a valid constant that is assignable to p.

Since the last case (the `default` case) terminates normally, so does the entire `switch` statement.

The label list returned by `GatherLabels` is $\{2, 3, 5, 7, 4, 6, 8, 9\}$. After sorting and comparing adjacent entries in the list, we conclude that there are no duplicate labels. Furthermore, there is exactly one default case.

In C#, there is no "fall-through" from one leg of a `switch` statement to the next. For example, the following example is illegal in C# (but is legal in C, C++, and Java).

```
switch (p):
  case 0:  isZero = true;
  case 1:  print(p);
}
```

The compiler can check for this error by requiring that, in each `caseItem`, `stmts.terminatesNormally` is false.

Other languages may allow enumeration types as well as the integer type for the `switch` expression.

Ada allows a range of values in a case such as `5..7`. The compiler needs to check for duplicates and complete coverage of all possible values.

§9.1.7 Exception handling

Up to this point, the `ThrowsVisitor` mostly gathers the exceptions that may occur in an expression (in a `throwsSet`), an assignment, an `if` statement, a `while` loop, and a `switch` statement. Next we will study what exceptions might occur in a `try` statement and a `throw` statement. They are related to the declared exceptions in the header of a function.

You need to understand Java exception handling: `throw`, `throws`, `try`, `catch`, and `finally` in Java.

Terminology: `throws` clause, `try` block, `catch` clause, `throw`, and `finally` block.

Two issues:

- no redundant handlers (`SubsumesLaterCatches` is for this purpose)
- no missing handlers

Here is an example.

```
public void writelist() throws ArrayIndexOutOfBoundsException {
  PrintStream pStr = null;
  try {
    pStr = new PrintStream(
                   new FileOutputSteam("outfile") );
    pStr.println("The 9th element is " +
                 victor.elementAt(9));
    // do not need    pStr.close()   here
    // since close() is done at the finally-block
  } catch (IOException e) {
    System.err.println("i/o error");
  } catch (ABCException e) {
    System.err.println("user-defined error");
  } finally { if (pStr != null) pStr.close(); }
}
```

```
int foo() throws AException, BException {
  ... statements before the try block ...
  try {
    ...
    ... throw (new BException()) ...
    ...
  } catch (CException e) {
    ... this is the first handler ...
  } catch (DException e) {
    ... this is the second handler ...
  } catch (EException e) {
    ... this is the third handler ...
  } finally { ... this is the finally block ... }
  ... statements after the try block ...
}
```

Most modern programming languages provide the exception mechanism. Exceptions may be thrown explicitly (with a `throw`/`raise` statement) or implicitly (due to an execution error).

Thrown exceptions may be propagated several times and then finally are caught by a handler.

Exceptions are cleaner and more efficient than error flags or gotos.

We will focus on Java exceptions. Exceptions in other programming languages are similar.

Java exceptions are typed. All exceptions are subclasses of the `Throwable` class.

There are two kinds of exceptions: *checked* and *unchecked*. A checked exception must be caught by an enclosing `try` statement or is listed in the `throws` clause of the enclosing method or constructor. Here is an example of a Java code fragment:

```
class ExitComputation extends Exception { . . . }
try { . . .
      if (a > b) throw new ExitComputation();
      if (v < 0.0) throw new ArithemticException();
      else a =Math.sqrt(v)
} catch (ExitComputation e) { print(e); return(0); }
  catch (ABCException e)    { print(e); return(5); }
  finally                   { a := b + c;          }
```

An unchecked exception (i.e., a `RuntimeException` or an `Error`) may optionally be handled in a `try` statement. If not caught by the program, it will be passed to the Java runtime system. Eventually, the program will terminate. The `ArithemticException`, a subclass of `RuntimeException`, is an unchecked exception.

The AST for a `try` is shown in Figure 9.24. The AST for a `catch` clause is shown in Figure 9.25.

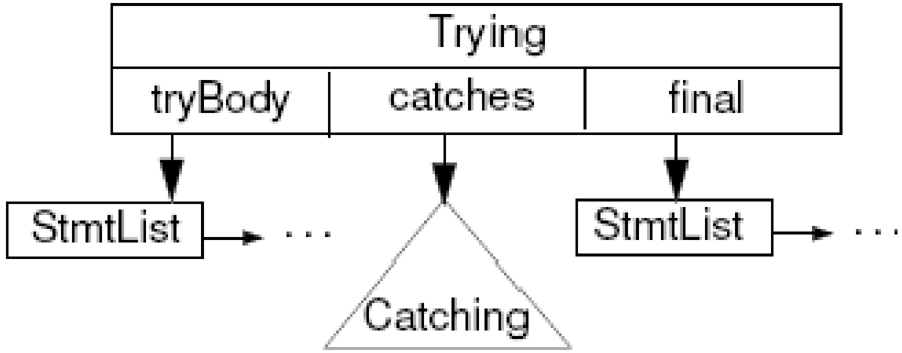


Figure 9.24: Abstract Syntax Tree for a Try Statement

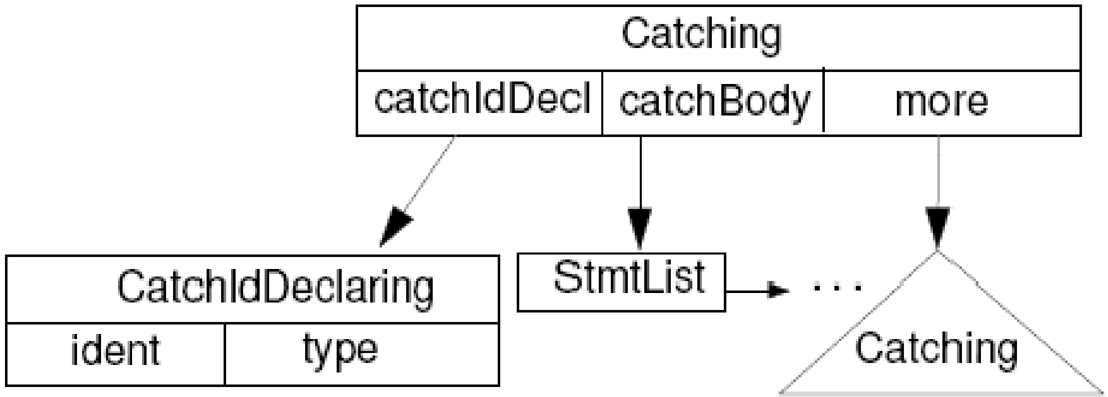


Figure 9.25: Abstract Syntax Tree for a Catch Block

A `Trying` node contains three fields: `tryBody` (which is a list of statements), `catches` (which is a list of `Catching` nodes), and `finally` (which is also a list of statements). A `Visit(Trying)` call (in Figure 9.26) simply invokes the `Visit` method on each of the three children.

A `catch` clause is analyzed during a `Visit(Catching)` call. A `Catching` node contains three fields:

1. `catchIdDecl`: which contains the type of the exception and an identifier as the parameter of the catch clause.
2. `catchBody`: which is a list of statements.
3. `more`: which forms a list of `catch` clauses.

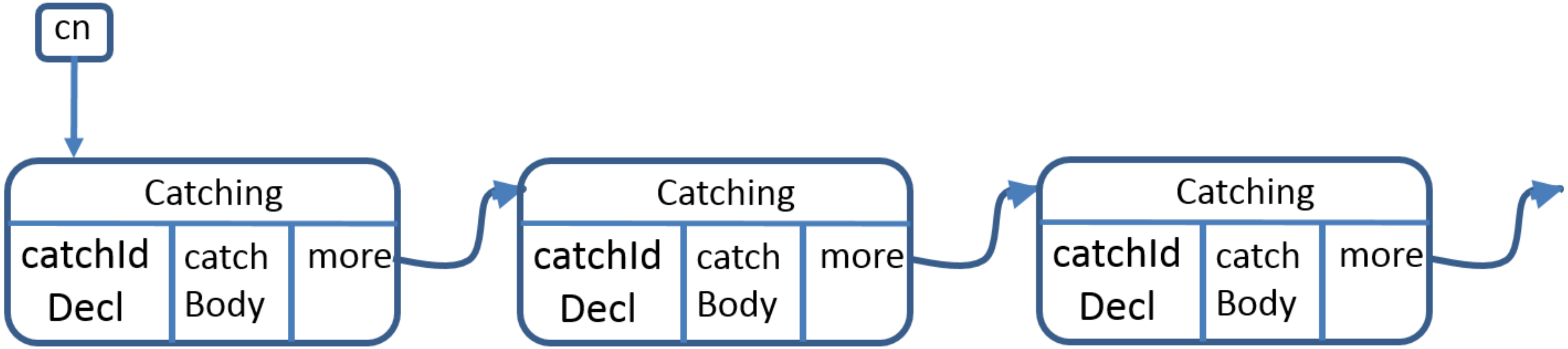


Figure 1: A list of `Catching` nodes suitable for tail recursion in the `Visit(Catching)` method.

Each `catch` forms a separate scope with the addition of the parameter.

A `try` body may be attached to by many `catch` clauses. In order to make every attached `catch` clause useful in some circumstances, it is required that no `catch` clause can completely subsume any later `catch` clause. (An exception A *subsumes* another exception B means that A is an ancestor class of B.) This is done by the `SubsumesLaterCatches` method in Figure 9.27.

The `Visit(Catching)` call iteratively visits each `Catching` node on the list linked by the `more` field.

For each `Catching` node (i.e., a handler), it checks if indeed an exception (i.e., a subclass of `Throwable`) is thrown and if the exception cannot subsume (i.e., hide) a later exception in the list. If all is fine, a scope is opened and the parameter is entered into the symbol table. In the new scope the body of the exception handler is analyzed. When all is done, the new scope is closed.

Each `throw` statement is analyzed in the `Visit(Throwing)` call in Figure 9.26. There is an argument for a `throw` statement. It simply checks if indeed it is an exception object that is thrown.

Three problems:

- No redundant handlers: `subsumesLaterCatches`
- No useless handlers: `ProcessCatch`
- No missing handlers: `Visit(Throwing)`

```
class NodeVisitor
   procedure VisitChildren(n)
      foreach c in n.GetChildren() do call c.Accept(this)
   end

class SemanticsVisitor extends NodeVisitor // extends Fig 9.21
   procedure Visit(Trying tn)
      call VisitChildren(tn)
   end

   procedure Visit(Catching cn)
      if not Assignable(Throwable, cn.catchIdDecl.type)
      then call error("Illegal type for catch identifier")
           cn.catchIdDecl.type := errorType
      else if SubsumesLaterCatches(cn.catchIdDecl.type, cn.more)
           then call error("This catch hides later catches.")
      call OpenScope()
      attr := new Attributes(variableAttributes)
      attr.kind := variableAttributes
```

```
        attr.variableType := cn.catchIdDecl.type.GetTypeDescriptor()
        call currentSymbolTable.EnterSymbol( // parameter of handler
             cn.catchIdDecl.ident.name, attr)
        call cn.catchBody.Accept(this)  // process the handler body
        call CloseScope()
        if cn.more != null then call cn.more.Accept(this)
                           // process next handler, tail recursion
    end

    procedure Visit(Throwing tn)       // throw an exception
        call VisitChildren(tn)
        if tn.thrownVal.type != errorType and
           not Assignable(Throwable, tn.thrownVal.type)
        then call error("Illegal type for throw")
    end
end
```

Figure 9.26 Semantic analysis visitors (part 3)

There are a few utility functions for semantics analysis:

- `function SubsumesLaterCatches(exceptionType, Catching cn) returns Boolean`
- `procedure ProcessCatch(SetOfType throwsSet, Catching cn)`
- `function FilterThrows(SetOfType throwsSet, exceptionType) returns SetOfType`
- `function FilterCatches(SetOfType throwsSet, Catching cn) returns SetOfType`
- `procedure GetCatchList( )`
- `procedure SetCatchList( )`
- `procedure UpdateCatchList(Catching cn)`

```
function SubsumesLaterCatches(handlerType, Catching cn)
                 returns Boolean
   // Is handlerType an ancestor class of any exception in cn?
   // cn is the list of the remaining exception handlers.
   if cn == null then return(false)
   if Assignable(handlerType, cn.catchIdDecl.type)
   then return(true)
   return( SubsumesLaterCatches(handlerType, cn.more) )
end

procedure ProcessCatch(SetOfType throwsSet, Catching cn)
   // throwsSet is a set of the exceptions that might be thrown.
   // cn is a list of exception handlers.
   // Check each handler in cn handles some exceptions in throwsSet.
   filteredThrowsSet :=FilterThrows(throwsSet,cn.catchIdDecl.type)
   // filteredThrowsSet are exceptions not caught by cn
   if filteredThrowsSet == throwsSet
   then call error("No throws reach this catch--a useless handler")
   else if cn.more != null
```

```
        then call ProcessCatch(filteredThrowsSet, cn.more)
        // tail recursion
end

function FilterThrows(SetOfType throwsSet, handlerType)
               returns SetOfType
   // throwsSet is a set of the exceptions that might be thrown.
   // Filter out the exceptions from throwsSet that are handled
   // by the handler handlerType.
   answer := { }
   foreach t in throwsSet do
      if not Assignable(handlerType, t)
      // exceptionType is an ancestor class of t
      then answer := answer union { t }      // t is not caught
   return(answer)
end
```

```
function FilterCatches(SetOfType throwsSet, Catching cn)
               returns SetOfType
   // Filter out the exceptions from throwsSet that are handled
   // by any handler in cn.
   newThrowsSet := filterThrows(throwsSet, cn.catchIdDecl.type)
   if cn.more == null then return( newThrowsSet )
   return( filterCatches(newThrowsSet, cn.more) )
end

procedure UpdateCatchList(Catching cn)
   // add new handlers to the catchList.
   call ExtendCatchList( cn.catchIdDecl.type )
   if cn.more != null then call UpdateCatchList(cn.more)
end
```

Figure 9.27 Utility semantic methods for try and throw statements

The `ThrowsVisitor` class in Figure 9.28 (p. 90) performs throws analysis. The `Visit(Trying tn)` call first invokes

```
tn.catches.Accept(this)
```

which, in turn, invokes the `Visit(Catching)` method. The `Visit(Catching)` method simply gathers together all possible thrown exceptions in the `try` block (using the `GatherThrows` method).

The `Visit(Trying tn)` call then invokes

```
tn.final.Accept(this)
```

to analyze each of the statements in the statement list in `tn.final`.

After analyzing `tn.final`, the `GetCatchList` and `UpdateCatchList` methods collect all the exceptions that might be caught. Note that `try` blocks may be nested, as follows:

```
. . .                // currentCatchList = { }
try {
   . . .             // currentCatchList = { B, C }
   try {
       . . .         // currentCatchList = { A, B, B, C }
   } catch (A e) { . . . }
     catch (B e) { . . . }
   finally { . . . }
   . . .             // currentCatchList = { B, C }
} catch (B e) { . . . }
  catch (C e) { . . . }
finally { . . . }
. . .                // currentCatchList = { }
```

There is a `currentCatchList`, which records a list of all the current handlers for a `try` block.

The `GetCatchList` method returns the list (i.e., `currentCatchList`) of exceptions that are caught by all the enclosing `try-catch` blocks. The `UpdateCatchList` method adds the `catch` clauses of the inner `try` block.

The body of the `try` block is analyzed last by

```
tn.tryBody.Accept(this)
```

After this is done, the catch list is restored with the `SetCatchList` method. The throws analysis results in the list of all exceptions that might be thrown in the `try` body.

The `ProcessCatch` method in Figure 9.28 verifies some exceptions can reach each of the `catch` clauses attached to the `try` block (otherwise the `catch` clause is useless).

The `FilterCatch` method in Figure 9.28 accumulates all the exceptions that can *escape* the current `try-catch` block.

Procedure `Visit(Trying tn)` calculates the set of all possbile exceptions that may occur in the `try` block but are *not* handled by the handlers of the `try` block. These include those in the try-body, the handlers, and the `finally` block.

```
class ThrowsVisitor extends NodeVisitor // extends fig 9.4
   procedure GatherThrows(n)
      call VisitChildren(n)  // find the throwsSet of each child
      ans := { }
      foreach c in n.GetChildren() do ans := ans union c.throwsSet
      n.throwsSet := ans
   end

   procedure Visit(Trying tn)
      call tn.catches.Accept(this)
      call tn.final.Accept(this)
      currentCatchList := GetCatchList()
      call UpdateCatchList(tn.catches)  // add new handlers
      call tn.tryBody.Accept(this)
      call SetCatchList(currentCatchList) // remove the handlers
      call ProcessCatch(tn.tryBody.throwsSet, tn.catches)
      tn.throwsSet := FilterCatches(tryBody.throwsSet, tn.catches)
      tn.throwsSet := tn.throwsSet union tn.catches.throwsSet
                      union tn.final.throwsSet
   end
```

```
procedure Visit(Catching cn)
   call GatherThrows(cn)
end

procedure Visit(Throwing tn)             // throw an exception
   call VisitChildren(tn)
   thrownType := tn.thrownVal.type       // exception type
   tn.throwsSet := tn.thrownVal.throwsSet union { thrownType }
   if Assignable(RuntimeException, thrownType) or
      Assignable(Error, thrownType) then return
   throwTargets := GetCatchList() union GetDeclThrowsList()
   filteredTargets := FilterThrows(throwTargets, thrownType)
   if Size(throwTargets) == Size(filteredTargets)
   then call error("Type thrown not found in enclosing catch
                    or in throws clause")
end
```

```
   procedure Visit(Calling cn) // call a method: call foo(exp, ...)
      call GatherThrows(cn)
      if cn.calledMethod != null
      then cn.throwsSet := cn.throwsSet union
                           cn.calledMethod.declaredThrowsList
   end
end
```

Figure 9.28 Throws analysis Visitors (Part 2)

Example.

```
int xyz( . . . ) throws A, B, C {
    try {
        . . .
        throw new D;
        . . .
    } catch (E e) { }
      catch (F f) { }
      catch (G g) { }
      finally { }
    . . .
}
. . .
call xyz(e1, e2, ...)
```

Reachability analysis is shown in Figure 9.29 (p. 101). In the `Visit(Trying tn)` call, the `try` body and `finally` block are always reachable. A `try` block will terminate normally if (1) the `finally` block terminates normally and (2) either any `catch` clause terminates normally or the `try` body terminates normally.

```
class ReachabilityVisitor extends NodeVisitor // extends fig 9.23
   procedure Visit(Trying tn)
      tn.tryBody.isReachable := true
      tn.final.isReachable   := true
      call VisitChildren(tn)
      catchOrTryOK := tn.catches.terminatesNormally or
                      tn.tryBody.terminatesNormally
      tn.terminatesNormally := catchOrTryOK and
                               tn.final.terminatesNormally
   end

   procedure Visit(Catching cn)
      cn.catchBody.isReachable := true
      call VisitChildren(cn)
      cn.terminatesNormally := cn.catchBody.terminatesNormally
      if cn.more != null
      then cn.terminatesNormally := cn.terminatesNormally or
                                    cn.more.terminatesNormally
   end
```

```
    procedure Visit(Throwing tn)
        tn.terminatesNormally := false // never terminates normally
    end

    procedure Visit(Calling cn)
        cn.terminatesNormally := true
        // always assumed to terminate normally
        // is this wrong?
    end
end
```

Figure 9.29 Reachability analysis visitors (Part 3)

The AST for a `throw` statement is shown in Figure 9.30

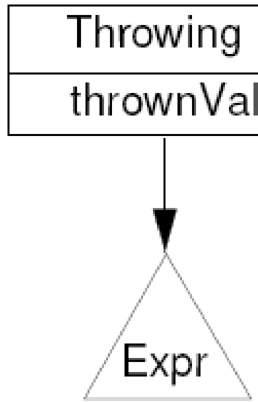


Figure 9.30: Abstract Syntax Tree for a Throw Statement

In semantics analysis, the `Visit(Throwing)` method in Figure 9.26 (p. 85) checks if indeed an exception, but not anything else, is thrown.

In throws analysis, the `Visit(Throwing)` method in Figure 9.28 (p. 95) verifies that when a checked exception is thrown, an enclosing `try` block will catch the exception or an enclosing method/constructor puts the exception in its `throws` list.

On the other hand, if the thrown exception is unchecked (i.e., `RuntimeException` or `Error`), it need not be caught explicitly.

The `GetCatchList()` method collects all exceptions caught by all enclosing `try` blocks. The `GetDeclThrowsList()` method collects all exceptions on the current method/constructor's `throws` clause.

```
public void wlist() throws AException, BException ...
```

Their union is the allowed exceptions. The

```
FilterThrows(throwTargets, thrownType)
```

call verifies if the `thrownType` belongs to `throwTargets`. If not, an error message is issued.

Error case 1.

```
int mm() throws A, B {
  try {
        throw new E;
  } catch (C e) { ... }
    catch (D e) { ... }
}
```

Error case 2.

```
int mm() throws A, B {
  try {
       ...  throw new A;
       ...  throw new B;
       ...  throw new C;
       // never throw new D;
  } catch (C e) { ... }
    catch (D e) { ... }  // this D handler is redundant
}
```

§9.2 Semantic analysis of calls

Calling a method in Java looks like:

```
pStr.println("The 9th element is ");
```

The AST for a call is shown in Figure 9.31.

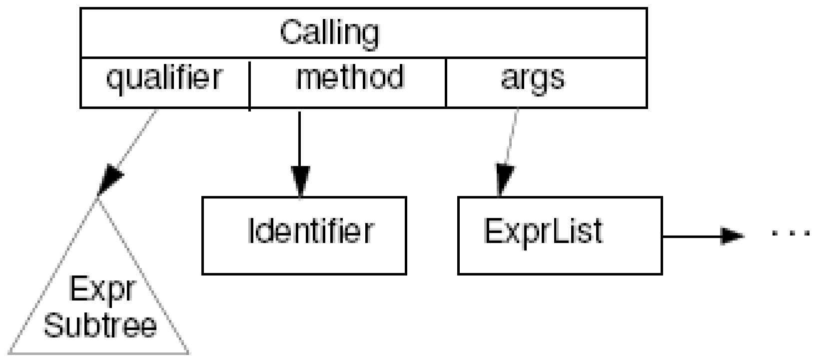


Figure 9.31: Abstract Syntax Tree for a Method Call

The `qualifier` is an optional expression specifying the object or class within which the method is to be found. For example, `super` is a qualifier. The `identifier` is the name of the method. The `args` is a list of expressions, which are intended as arguments of the function call.

When calling a function, the compiler needs to determine which function is called. If all functions have different names, this determination is easy. We may simply look up the function name in the current symbol table.

Modern programming languages allow *overloading*, that is, many functions may have identical names. Furthermore, object-oriented languages provide *inheritance*. Determining the actual function to be invoked is difficult. Consider the following Java example:

Java overloading and inheritance

```
class A           { . . . }
class B extends A { . . . }
class C extends B { . . . }
class D           { int h(A x) { return 1; } }
class E extends D { int h(C x) { return 2; } }
class F extends E { int h(A x) { return 3; } }
class G extends F { int h(B x) { return 4; }
                    int h(C x) { return 5; } }
class H {
      main(. . .) {
      B b = new C();
      E e = new G();
      . . . e.h(b) . . . --- 3
      }       }
```

In Java, all classes are subclasses of the `Object` class, directly or indirectly.[a] Java provides four access modifiers for methods: `public`, `protected`, `private`, and default (i.e., no modifiers). An object may use the methods defined its class and in all the ancestor classes with appropriate access modifiers. Thus, for a method call, the compiler needs to search all these relevant classes in the order specified by the Java programming language.

The semantics analysis of a call first gathers all method definitions that might be the target of the call.

If the qualifier is `super`, the compiler will look for the method in the parent class. If the qualifier is a class name, the compiler will look for a *static method* (or *class method*). (There are two kinds of methods in Java: class methods and instance methods.)

Different programming languages may enforce different rules to look for a method. These rules are enforced by the `GetMethods` function in

[a] Java's case is easier because Java adopts single inheritance. C++ adopts multiple inheritance.

Figure 9.32 (p. 111). The `GetMethods` function returns a set of `Attributes` structures, each representing an accessible method.

The `GetCurrentClass` function returns the class under compilation. The `MethodDef(ID)` function returns all (the `Attributes` structures for) the methods in a class with name `ID`. Similarly, the `VisibleMethods(ID)` function returns the public and protected methods with name `ID`.

In the semantics analysis `Visit(Calling)` (in Figure 9.34, p. 116) uses the `GetMethods` to obtain the set of candidate method definitions. `Visit(Calling)` uses the following function:

```
Applicable(GetArgs(def.signature), actualArgsType)
```

to select those candidate methods of which the number and the types of parameters match the actual arguments in the call statement.

If more than one candidate method definition is selected, the `FilterDefs` function (in Figure 9.33, p. 114) is invoked to choose the *most specific* candidate method definitions. The `MoreSpecific` function in Figure 9.33 determines if one method definition is more specific than the other.

If all is fine, there should be exactly one candidate method definition. Two last checks will be performed in `Visit(Calling)`:

1. If the called method has a qualifier and the qualifier is not `super`, then the candidate method definition must be a static method.
2. If the called method needs to return a value (using the `InExpressionContext` function), such as

   ```
   a + ff(b) + c
   ```

   then the candidate method definition must return a non-void value.

Other utility functions

In the implementation, we assume each method definition is represented by an `Attributes` structure, which contains three fields:

1. `returnType` is the type of the return value
2. `signature` is the type signature of the method
3. `classDefinedIn` is the class in which the method is defined

The `GetArgTypes` function (in Figure 9.32) builds a list of types, each of which corresponds an expression in a list of expressions. The list of expressions are the actual arguments of the method calls.

The list of types built by the `GetArgTypes` function is used to match against the *signature* of a method definition, that is, the list of types of formal parameters and of the return value.

The `Bindable` function used in Figure 9.33 determines if an actual argument can match a formal parameter. `Bindable` is almost the same as `Assignable`. When interfaces are considered, a class object may sometimes be bound even if it may not be directly assigned.

The `Applicable` function in Figure 9.32 makes use of the `Bindable` function to match each actual argument to a formal parameter in a method definition. If all arguments match the corresponding formal parameters, the `Applicable` function returns `true`, which means the method definition can be used in a method call. Otherwise, the method definition is removed from the list of candidate method definitions.

The notion of *more specific*

When there are more than one appropriate method definition, we will use the *most specific* one. There are two issues:

1. If a method is redefined in a subclass, we will use the redefinition. For example,

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

   The `M` method in class `D` is more specific.

2. If a method is redefined in a subclass whose parameter is also a subclass, the redefinition is more specific. For example,

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

   The `M(B)` method in class `B` is more specific.

Question. Which `M` is more specific?

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

Answer. Neither is more specific than the other.

We define a method definition X is *more specific* than another definition Y if X's class is bindable to (or a subclass of) Y's class and each parameter of X is bindable to the corresponding parameter of Y.

The `MoreSpecific(def1, def2)` function in Figure 9.33 determines if definition `def2` is more specific than `def1`.

Example.

```
class C           { . . . }
class D extends C { . . . }
class A           { void M(C x) { . . . }
                    void M(D x) { . . . } }
class B extends A { void M(D x) { . . . }
                    void test(D arg)    { M(arg); } }
```

At the call `M(arg)`, three definitions of `M` are visible. Two are applicable to the call. The last definition is more specific and is preferred. □

Note that in Java, the result type is not used in overload resolution. Java does not allow overloading of two methods that have the same name, the same parameter types, but different result types. Neither do C++ and C#. For example, the following example causes an error:

```
int   add(int i; int j) { . . . }
float add(int i; int j) { . . . }
```

If the result type is also considered, say in Ada, overload resolution would be more complicated.[a]

[a]Four issues in dispatching a method in Java:

1. overloading: signature (types of parameters and return value)
2. inheritance: inherit methods from ancestor classes
3. static typing: no run-time type errors after compiler checkup
4. dynamic dispatch: search a method with a specific signature from the actual (not real) class of the object

```
function GetMethods(Calling cn) returns SetOfAttributes
   currentClass := GetCurrentClass()
   if cn.qualifier == null
   then methodSet := currentClass.MethodDefs(cn.method)  // same ID
   else methodSet := { }
   if cn.qualifier == null or cn.qualifier == superNode
   then nextClass := currentClass.parent
   else nextClass := cn.qualifier.type
   while nextClass != null do
      if cn.qualifier != null and cn.qualifier != superNode
         and not nextClass.isPublic
      then nextClass := nextClass.parent
           continue
      methodSet := methodSet union
                   nextClass.VisibleMethods(cn.method)  // same ID
      nextClass := nextClass.parent
   return(methodSet)
end
```

```
function GetArgTypes(ExprList el) returns ListOfTypes
   // returns the list of types of expressions in el
   typeList := null
   foreach expr in el do
      typeList := Append(typeList, List(expr.type))
   return(typeList)
end

function Applicable(formalParms, actualParms) returns Boolean
   if formalParms == null and actualParms == null
   then return(true)
   if formalParms == null or actualParms == null
   then return(false)  // numbers of arguments mismatch.
   if Bindable(head(formalParms), head(actualParms))
   then return( Applicable(tail(formalParms), tail(actualParms)) )
   return(false)
end
```

Figure 9.32 Utility semantic methods for method calls (I)

```
function MoreSpecific(def1, def2) returns Boolean
   // returns true if def2 is more specific than def1
   if Binadable(def1.classDefinedIn, def2.classDefinedIn)
   then arg1 := def1.argtypes
        arg2 := def2.argTypes
        // arg1 and arg2 must have the same lengths.
        while arg1 != null do
           if Bindable(head(arg1), head(arg2))
           then arg1 := tail(arg1)
                arg2 := tail(arg2)
           else return(false)
        return(true)
   return(false)
end  // end of function
```

```
function FilterDefs(methodDefSet) returns SetOfDefs
   changes := true
   while changes do
      changes := false
      foreach def1 in methodDefSet do
         foreach def2 in methodDefSet do
            if def1 != def2 and MoreSpecific(def1, def2)
            then methdoDefSet := methodDefSet - { def1 }
                 changes := true
   return(methodDefSet)
end
```

Figure 9.33 Utility semantic methods for method calls (II)

```
class NodeVisitor
   procedure VisitChildren(n)
      foreach c in n.GetChildren() do call c.Accept(this)
   end
end  // end of class

class SemanticsVisitor extends NodeVisitor // extends fig 9.26
   procedure Visit(Calling cn)
      call VisitChildren(cn)    // parameters
      cn.calledMethod := null
      methodSet := GetMethods(cn)
      actualArgsType := GetArgTypes(cn.args)
      foreach def in methodSet do
         if not Applicable(GetArgs(def.signature), actualArgsType)
         then methodSet := methodSet - { def }
      if Size(methodSet) == 0
      then call error("No method matches this call")
           return
      if Size(methodSet) > 1
      then methodSet := filterDefs(methodSet)
```

```
        if Size(methodSet) > 1
        then call error("More than one method matches this call")
             return
        m := the singleton member of methodSet
        cn.calledMethod := m
        if cn.qualifier != null and cn.qualifier != superNode and
           m.accessMode != static
        then call error("Method called must be static.")
        else if InExpressionContext(cn) and m.returnType == void
             then call error("call must return a value")
    end
end
```

Figure 9.34 Semantic analysis visitors (Part 4)

Java also includes interfaces and constructors. Their analysis methods are similar to the above methods.

An *interface* is an abstraction of a class specifying a set of method definitions without their implementation. Actual implementation is insignificant during semantics analysis.

Constructors are similar to ordinary methods but they do not return any value.

Subprograms are similar to methods, except that methods are defined within a class. Subprograms are defined at the global level (i.e., within a compilation unit). Semantics analysis of subprograms are similar to that of methods.

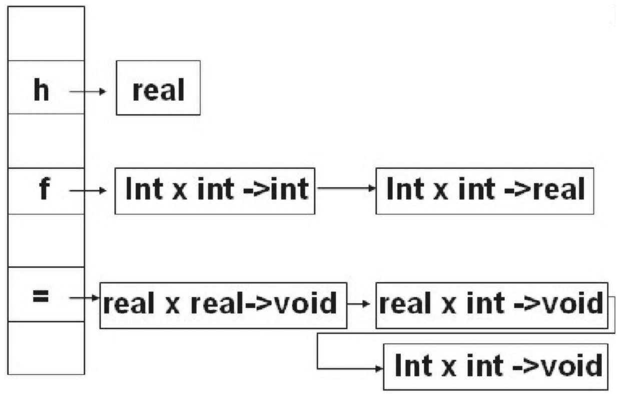
For compiling methods in an object-oriented language, the first parameter is the implicit `this`.

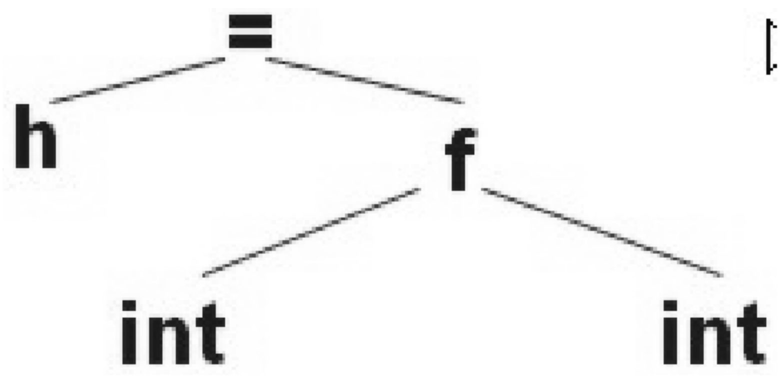
Appendix. Overload resolution

Consider

```
h = f(1,2);
```

we have the following overloaded definitions in the symbol table and the abstract syntax tree:





Top-down approach

```
int count(tree, rtype) {
   if tree is a leaf then
      if tree.rtype = rtype then return 1 else return 0;
   counter = 0;
   for each definition d of tree.root do
      if result type of d    == rtype and
         # arguments in d == # subtrees of tree then {
         comb = 1; i = 1;
         for each subtree s of tree do
            comb = comb * count(s, d.arg[i].rtype);
            i    = i + 1;
         end for
         counter = counter + comb;
      }
   end for
   return counter;
}
```

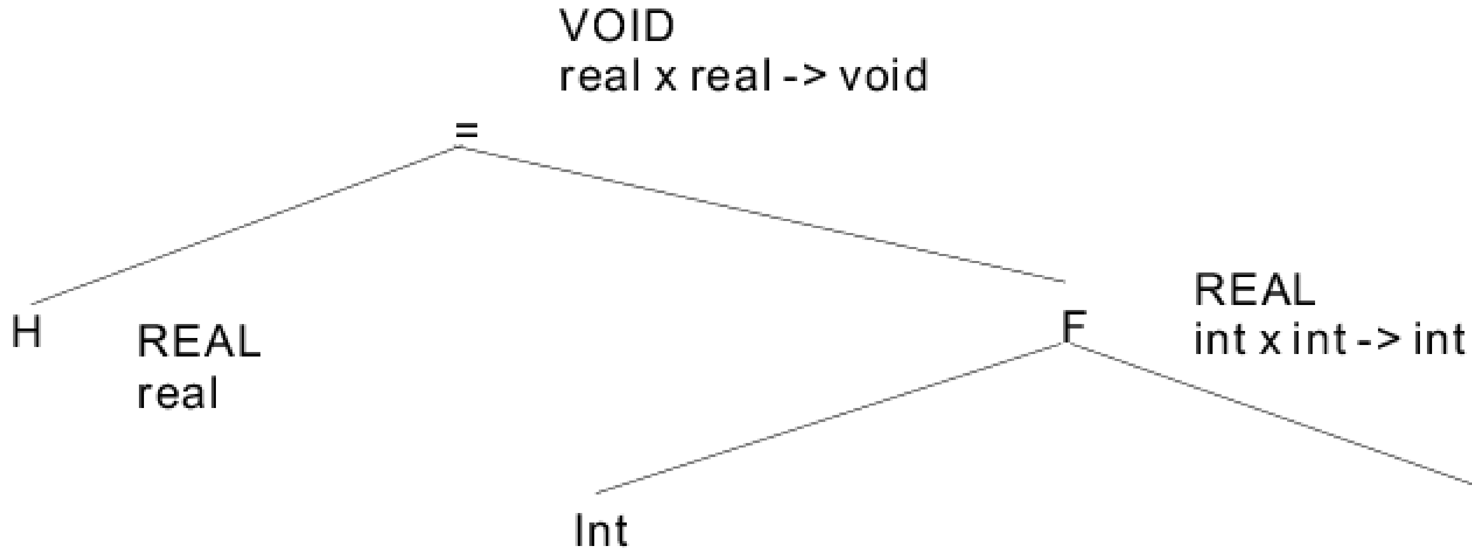


Figure 2: Overload resolution (1)

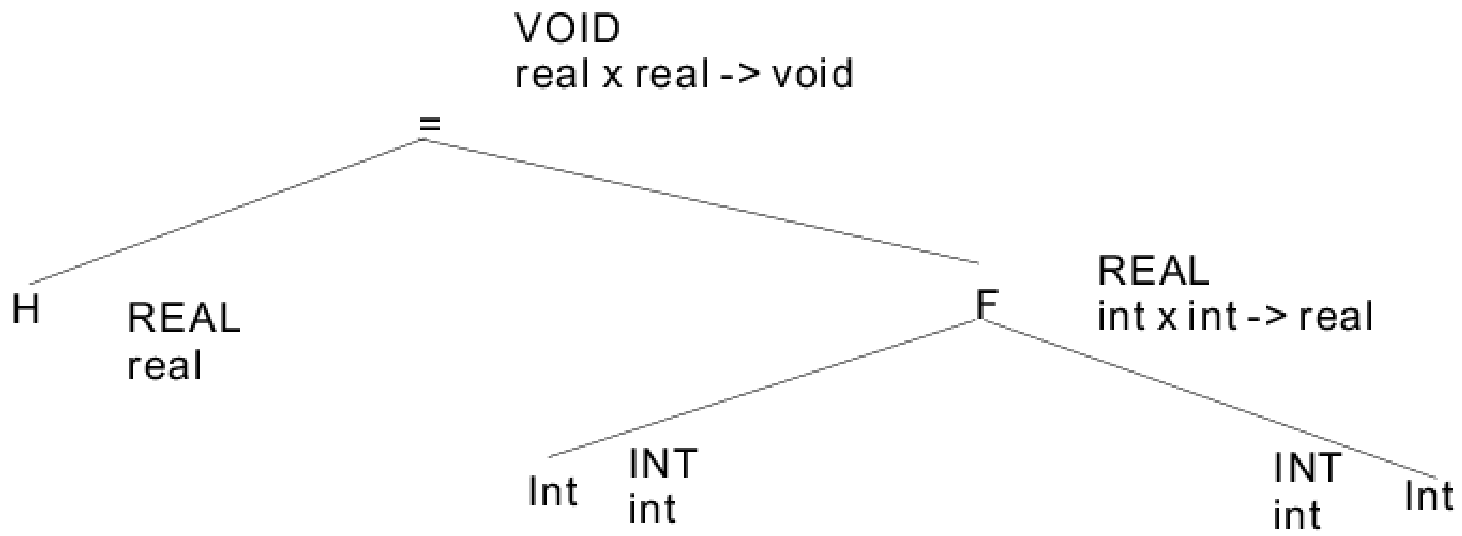


Figure 3: Overload resolution (2)

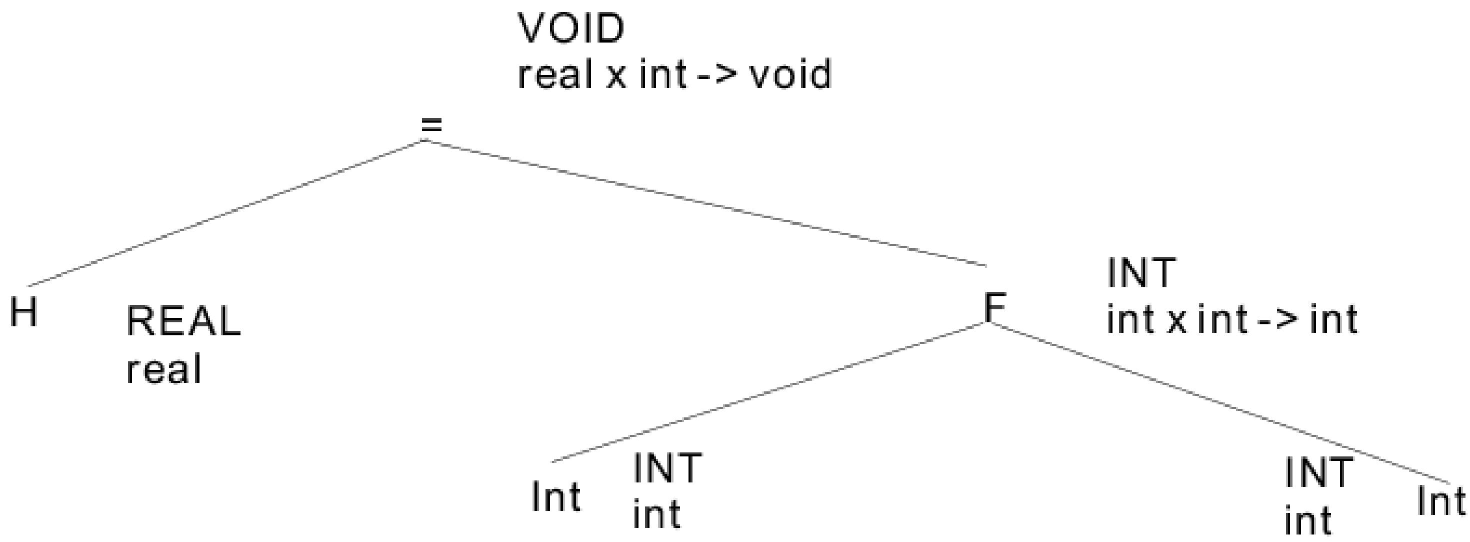


Figure 4: Overload resolution (3)

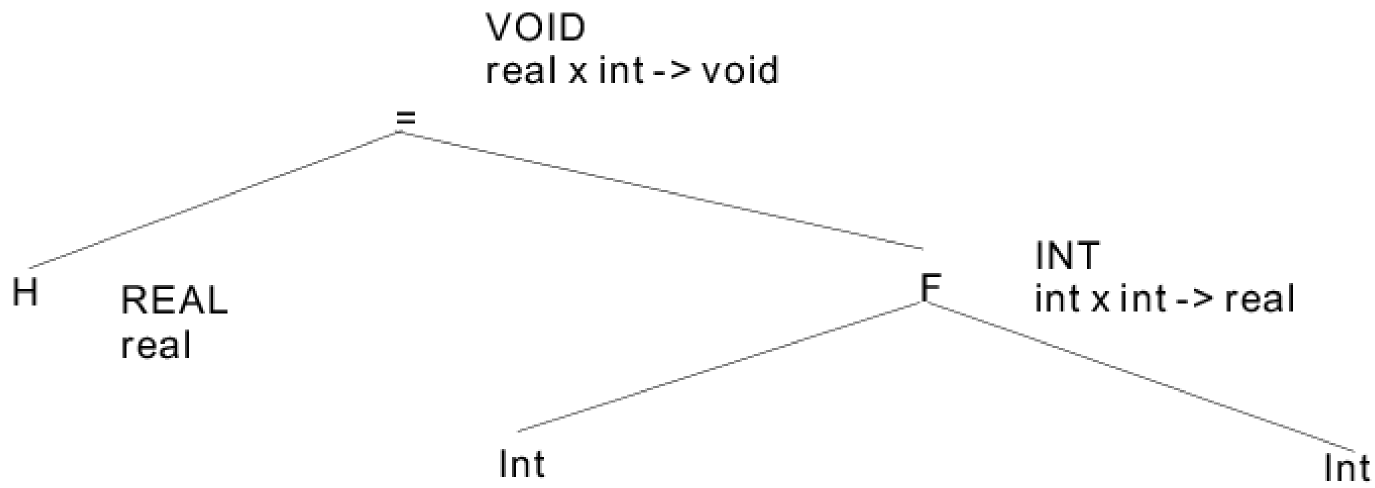


Figure 5: Overload resolution (4)

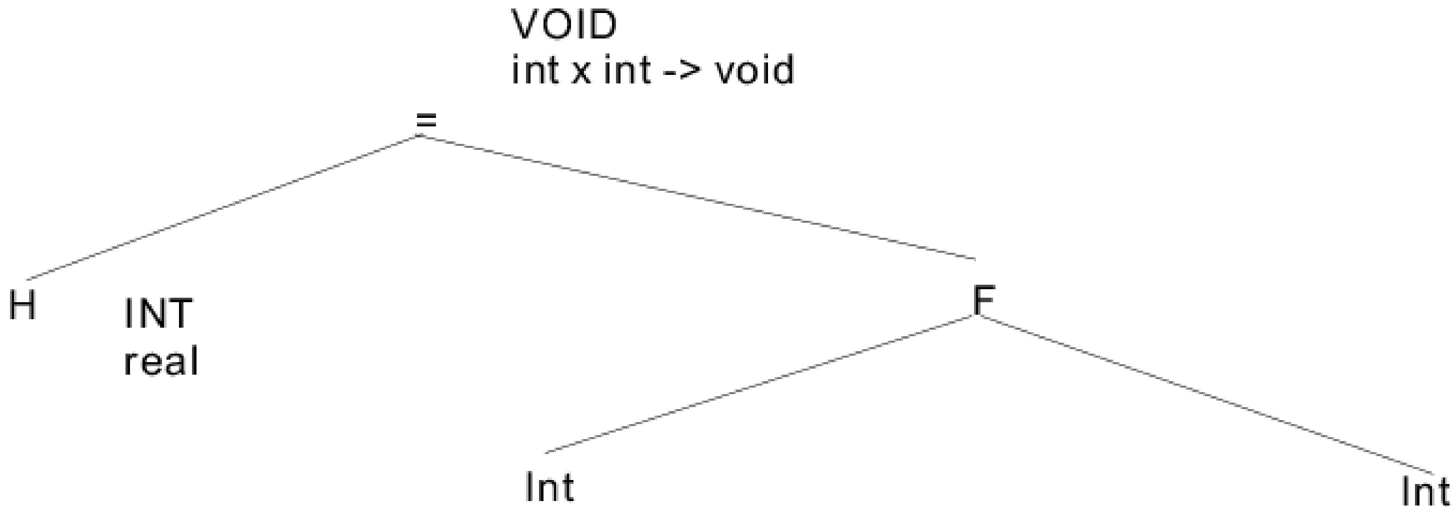


Figure 6: Overload resolution (5)

Example of semantic check (in C++)

What is wrong with the following program? Can the compiler find the error? How can the compiler find the error? What checks must a compiler perform?

```
class Printer {
public:
  Printer(ostream& pstream) : m_stream(pstream) {}
  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }
  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};

Printer{myStream}.println("hello").println(500);

class CoutPrinter : public Printer {
public:
  CoutPrinter() : Printer(cout) {}
  CoutPrinter& SetConsoleColor(Color c){  ... ;  return *this; }
```

```
};

// v----- we have a 'Printer' here, not a 'CoutPrinter'
CoutPrinter().print("Hello ").SetConsoleColor(Color.red).println("Printer!");
// compile error
```

Example. Semantic errors in LLVM bitcode.

LLVM bitcode verification (llvm-as)

1. Both of a binary operator's parameters are of the same type.
2. Verify that the indices of mem access instructions match other operands.
3. Verify that arithmetic and other things are only performed on first-class types. Verify that shifts and logicals only happen on integrals f.e.
4. All of the constants in a switch statement are of the correct type.
5. The code is in valid SSA form.
6. It is illegal to put a label into any other type (like a structure) or to return one.
7. Only phi nodes can be self referential: `%x = add i32 %x, %x` is invalid.
8. PHI nodes must have an entry for each predecessor, with no extras.
9. PHI nodes must be the first thing in a basic block, all grouped together.
10. PHI nodes must have at least one entry.
11. All basic blocks should only end with terminator insts, not contain them.
12. The entry node to a function must not have predecessors.
13. All Instructions must be embedded into a basic block.
14. Functions cannot take a void-typed parameter.
15. Verify that a function's argument list agrees with its declared type.
16. It is illegal to specify a name for a void value.
17. It is illegal to have an internal global value with no initializer.
18. It is illegal to have a `ret` instruction that returns a value that does not agree with the function return value type.
19. Function call argument types match the function prototype.
20. All other things that are tested by asserts spread about the code.

Figure 7: LLVM bitcode verification.