13 *Code Generation: Control Structures and Subroutines*

13.1 Code Generation for Control Structures

Control structures, whether conditionals like if and switch statements, or while and for loops, involve generation of conditional branch instructions. That is, based on a boolean value or relation, statements may be executed or not. In translating these constructs we will utilize jump code just as we did for conditional expressions. Depending on the nature of the construct, we will utilize either JumpIfTrue or JumpIfFalse form.

13.1.1 If Statements

The AST corresponding to an if statement is shown in Figure 13.1. An IfNode has three subtrees, corresponding to the condition controlling the if, the then statements and the else statements. The code we will generate will be of the form

1. If condition is false go to 4.
2. Execute statements in thenPart.
3. Go to 5.
4. Execute statements in elsePart.
5. Statements following the if statement.

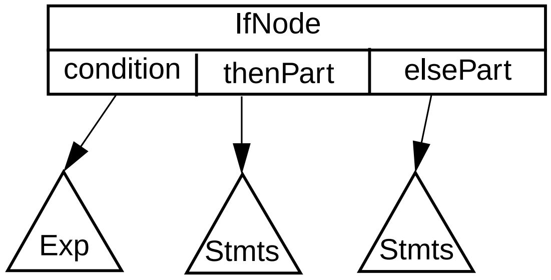

Figure 13.1 Abstract Syntax Tree for an if statement

At run-time the code we generate first evaluates the boolean-valued condition. If it is false, the code branches to a label at the head of the instructions generated for the else part. If the condition is true, no branch is taken; we "fall through" to the instructions generated for the then statements. After this code is executed, we branch to the very end of the if statement (to avoid incorrectly executing the statements of the else part).

The CodeGen function for an IfNode is shown in Figure 13.2.

As an example, consider the following statement

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

Assume `a` and `b` are local variables with indices of 1 and 2. We know `b` is a boolean, limited to either 1 (true) or 0 (false). The code we generate is:

```
IfNode.CodeGen( )
1.  ElseLabel ← CreateUniqueLabel()
2.  condition.Result ← AddressNode(JumpIfFalse, ElseLabel)
3.  condition.CodeGen()
4.  thenPart.CodeGen()
5.  OutLabel ← CreateUniqueLabel()
6.  GenGoTo(OutLabel)
7.  GenLabel(ElseLabel)
8.  elsePart.CodeGen()
9.  GenLabel(OutLabel)
```

Figure 13.2 Code Generation for If Statements Using Jump Code

```
    iload      2  ; Push local #2 (b) onto the stack
    ifeq       L1 ; Goto L1 if b is 0 (false)
    iconst_1      ; Push 1
    istore     1  ; Store stack top into local #1 (a)
    goto       L2 ; Skip around else statements
L1: iconst_2      ; Push 2
    istore     1  ; Store stack top into local #1 (a)
L2:
```

Improving If Then StatementsThe code we generate for IfNodes is correct even if the elsePart is a nullNode. This means we correctly translate both “if then” and “if then else” statements. However, the code for an if then statement is a bit clumsy, as we generate a `goto` to the immediately following statement. Thus for

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

we generate

```
    iload      2  ; Push local #2 (B) onto the stack
    ifeq       L1 ; Goto L1 if B is 0 (false)
```

```
      iconst_1        ; Push 1
      istore      1   ; Store stack top into local #1 (a)
      goto        L2  ; Skip around else statements
  L1:
  L2:
```

We can improve code quality by checking whether the elsePart is a nullNode before generating a goto to the end label, as shown in Figure 13.3.

```
IfNode.CodeGen( )
 1.  ElseLabel ← CreateUniqueLabel()
 2.  condition.Result ← AddressNode(JumpIfFalse, ElseLabel)
 3.  condition.CodeGen()
 4.  thenPart.CodeGen()
 5.  if elsePart = null
 6.      then GenLabel(ElseLabel)
 7.      else OutLabel ← CreateUniqueLabel()
 8.           GenGoTo(OutLabel)
 9.           GenLabel(ElseLabel)
10.           elsePart.CodeGen()
11.           GenLabel(OutLabel)
```

Figure 13.3 Improved Code Generation for If Statements

13.1.2 While, Do and Repeat Loops

The AST corresponding to a while statement is shown in Figure 13.4. A whileNode has two subtrees, corresponding to the condition controlling the loop and the loop body. A straightforward translation is

1. If condition is false go to 4.
2. Execute statements in loopBody.
3. Go to 1.
4. Statements following the while loop.

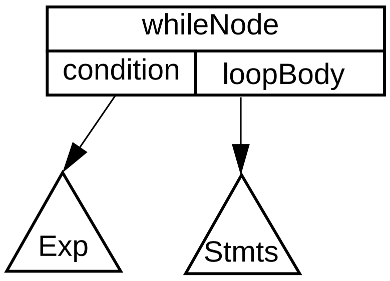

Figure 13.4 Abstract Syntax Tree for a While Statement

This translation is correct, but it is not as efficient as we would like. We expect a loop to iterate many times. Each time through the loop we will execute an unconditional branch to the top of the loop to reevaluate the loop control expression, followed by a conditional branch that will probably fail (it succeeds only after the last iteration). A more efficient alternative is to generate the conditional expression and conditional branch *after* the loop body, with an initial goto around the loop body (so that zero iterations of the loop are possible). That is, we generate code structured as

1. Go to 3.
2. Execute statements in loopBody.
3. If condition is true go to 2.

Now our translation executes only one branch per iteration rather than two.

While loops may contain a continue statement, which forces the loop's termination condition to be evaluated and tested. As detailed in Section 13.1.4, we assume that the function getContinueLabel() will return the label used to mark the target of a continue.

The code generator for while loops is shown in Figure 13.5.

```
WhileNode.CodeGen( )
 1.   ConditionLabel ← getContinueLabel()
 2.   GenGoTo(ConditionLabel)
 3.   TopLabel ← CreateUniqueLabel()
 4.   GenLabel(TopLabel)
 5.   loopBody.CodeGen()
 6.   GenLabel(ConditionLabel)
 7.   condition.Result ← AddressNode(JumpIfTrue, TopLabel)
 8.   condition.CodeGen()
```

Figure 13.5 Code Generation for While Statements

As an example, consider the following while loop, where `I` is an integer local with a variable index of 2 and `L1` is chosen as the ConditionLabel

```
while (I >= 0) { I--;}
```

The JVM code generated is

```
    goto        L1 ; Skip around loop body
L2:
    iload       2  ; Push local #2 (I) onto the stack
    iconst_1       ; Push 1
    isub           ; Compute I-1
    istore      2  ; Store I-1 into local #2 (I)
L1:
    iload       2  ; Push local #2 (I) onto the stack
    ifge        L2 ; Goto L2 if I is >= 0
```

Do and Repeat LoopsJava, 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. In our translation for while loops we placed evaluation and testing of the termination condition after the loop body—just where the do while loop wants it!

We need change very little of the code generator we used for while loops to handle do while loops. In fact, all we need to do is to eliminate the initial goto we used to jump around the loop body. With that change, the remaining code will correctly translate do while loops. For example, the statement

```
do { I--;} while (I >= 0);
```

generates the following (label `L1` is generated in case the loop body contains a continue statement):

```
L2:
    iload       2  ; Push local #2 (I) onto the stack
    iconst_1       ; Push 1
    isub           ; Compute I-1
    istore      2  ; Store I-1 into local #2 (I)
L1:
    iload       2  ; Push local #2 (I) onto the stack
    ifge        L2 ; Goto L2 if I is >= 0
```

Some languages, like 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. Again, only minor changes are needed to handle a repeat loop. As was the case for do loops, the initial branch around the loop body is removed. This is because repeat loops always test for termination at the end of an iteration rather than at the start of an iteration.

The only other change is that we wish to continue execution if the condition is false, so we evaluate the termination condition in the JumpIfFalse form of jump code. This allows the repeat until loop to terminate (by falling through) when the condition becomes true.

13.1.3 For Loops

For loops are translated much like while loops. As shown in Figure 13.6, the AST for a for loop adds subtrees corresponding to loop initialization and increment.

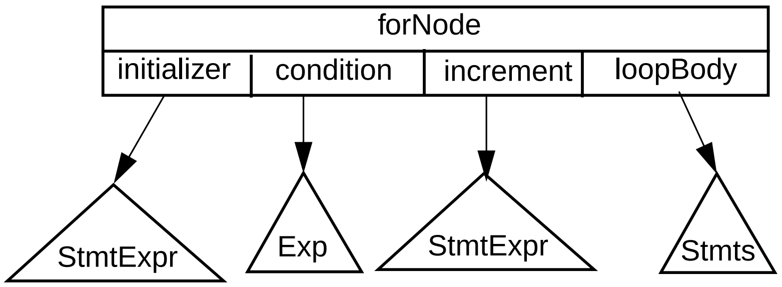

Figure 13.6 Abstract Syntax Tree for a For Statement

For loops are expected to iterate many times. Therefore after executing the loop initializer, we again skip past the loop body and increment code to reach the termination condition, which is placed at the bottom of the loop:

1. Execute initializer code.
2. Go to 5.
3. Execute statements in loopBody.
4. Execute increment code.
5. If condition is true go to 3.

Any or all of the expressions used to define loop initialization, loop increment or loop terminations may be null. Null initialization or increment expressions are no problem. They appear as nullNodes in the AST and generate nothing when CodeGen is called. However, the termination condition is expected to produce a conditional branch to the head of the loop. If it is null, we must generate an unconditional branch back to the top of the loop body. (For loops without a termination condition can only be exited with a break or return statement within the loop body.)

The code generator shown in Figure 13.7 is used to translate a forNode.

As an example, consider the following for loop (where `i` and `j` are locals with variable indices of 1 and 2)

```
ForNode.CodeGen( )
 1.  initializer.CodeGen()
 2.  SkipLabel ← CreateUniqueLabel()
 3.  GenGoTo(SkipLabel)
 4.  TopLabel ← CreateUniqueLabel()
 5.  GenLabel(TopLabel)
 6.  loopBody.CodeGen()
 7.  ContinueLabel ← getContinueLabel()
 8.  GenLabel(ContinueLabel)
 9.  increment.CodeGen()
10.  GenLabel(SkipLabel)
11.  if condition = null
12.      then GenGoTo(TopLabel)
13.      else condition.Result ← AddressNode(JumpIfTrue, TopLabel)
14.           condition.CodeGen()
```

Figure 13.7 Code Generation for For Loops

```
for (i=100;i!=0;i--) {

     j = i;

}
```

The JVM code we generate is

```
   bipush      100; Push 100 onto the stack

   istore      1  ; Store 100 into local #1 (i)

   goto        L1 ; skip around loop body and increment

L2:

   iload       1  ; Push local #1 (i) onto the stack

   istore      2  ; Store i into local #2 (j)

L3:               ; Target label for continue statements

   iload       1  ; Push local #1 (i) onto the stack

   iconst_1       ; Push 1

   isub           ; Compute i-1
   istore      1  ; Store i-1 into local #1 (i)
L1:
   iload       1  ; Push local #1 (i) onto the stack
   ifne        L2 ; Goto L2 if i is != 0
```

The special case of

```
      for (;;) {}
```

(which represents an infinite loop) is properly handled, generating

```
                  goto     L1
   L1: L2: L3:
                  goto     L2
```

Java and C++ allow a local declaration of a loop index as part of initialization, as illustrated by the following for loop

```
   for (int i=100; i!=0; i--) {
      j = i;
   }
```

Local declarations are automatically handled during code generation for the initialization expression. A local variable is declared within the current procedure with a scope limited to the body of the loop. Otherwise translation is identical.

13.1.4 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 StatementsLike 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 is enforced by the parser or semantic analyzer. 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 verifies that an enclosing loop with the proper label exists.

Any statement in Java may be labeled. 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.

```
labeledStmt.CodeGen( )
1.  if stmt ∈ { whileNode, doNode, forNode }
2.      then continueList ← continueItem(stmtLabel, CreateUniqueLabel(),
                                 length(finalList), continueList)
3.  stmt.CodeGen()
```

Figure 13.8 Code Generation for Labeled Statements

The code generator of labeledStmt, defined in Figure 13.8, checks stmt to see if it represents a looping statement (of any sort). If it does, the code generator creates a unique label to be used within the loop as the target of continues. It adds this label, called asmLabel, along with the user-defined label (field stmtLabel), onto a list called continueList. This list also includes an integer finalDepth that represents the number of try statements with finally blocks this statement is nested in (see Section 13.2.6) and a field next that links to the next element of the list. continueList will be examined when a continue statement is translated.

As mentioned earlier, looping statements, when they are translated, call getContinueLabel() to get the label assigned for use by continue statements. This label is simply the assembly-level label (generated by CreateUniqueLabel) at the head of continueList.

We'll also use two new code generation subroutines. GenJumpSubr(Label) will generate a subroutine call to Label. In JVM this is a jsr instruction. GenFinalCalls(N) will generate N subroutine calls (using GenJumpSubr) to the labels contained in the first N elements of finalList. This list is created by try statements with finally blocks to record the blocks that must be executed prior to executing a continue, break or return statement (see Section Section 13.2.6 for more details).

A continue statement that references no label will use the first element of continueList. If the continue is labeled, it will search the continueList for a matching label. The field label in continueNode will contain the label parameter, if any, referenced in the continue. A code generator for continue statements is shown in Figure 13.9.

```
ContinueNode.CodeGen( )
1.  if stmtLabel = null
2.      then GenFinalCalls(length(finalList) - continueList.finalDepth)
3.           GenGoTo(continueList.asmLabel)
4.      else listPos ← continueList
5.           while listPos.stmtLabel ≠ stmtLabel
6.               do listPos ← listPos.next
7.           GenFinalCalls(length(finalList) - listPos.finalDepth)
8.           GenGoTo(listPos.asmLabel)
```

Figure 13.9 Code Generation for Continue Statements

In most cases continue statements are very easy to translate. Try statements with finally blocks are rarely used, so a continue generates a branch to the "loop continuation test" of the innermost enclosing looping statement or the looping statement with the selected label. In C and C++ things are simpler still—continues don't use labels so the innermost looping statement is always selected.

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.

A labeled break exits the enclosing statement with a matching label, and continues execution with that statement's successor. For both labeled and unlabeled breaks, semantic analysis has verified that a suitable target statement for the break does in fact exist.

We'll extend the code generator for the labeledStmt node to prepare for a break when it sees a labeled statement or an unlabeled while, do, for or switch statement. It will create an assembler label to be used as the target of break statements within the statement it labels. It will add this label, along with the Java label (if any) of the statement it represents to breakList, a list representing potential targets of break statements. It will also add a flag indicating whether the statement it represents is a while, do, for or switch statement (and hence is a valid target for an unlabeled break). Also included is an integer finalDepth that represents the number of try statements with finally blocks this statement is nested in (see Section 13.2.6).

The code generator will also generate a label definition for the break target after it has translated its labeled statement. The revised code generator for labeledStmt is shown in Figure 13.10.

```
labeledStmt.CodeGen( )
1.  if stmt $\in$ { whileNode, doNode, forNode }
2.      then continueList $\leftarrow$ continueItem(stmtLabel, CreateUniqueLabel(),
                                   length(finalList), continueList)
3.  unlabeledBreakTarget $\leftarrow$
              stmt $\in$ { whileNode, doNode, forNode, switchNode}
4.  if unlabeledBreakTarget or stmtLabel $\neq$ null
5.      then breakLabel $\leftarrow$ CreateUniqueLabel()
6.           breakList $\leftarrow$ breakItem(stmtLabel, breakLabel,
                 unlabeledBreakTarget, length(finalList), breakList)
7.  stmt.CodeGen()
8.  if unlabeledBreakTarget or stmtLabel $\neq$ null
9.      then GenLabel(breakLabel)
```

Figure 13.10 Revised Code Generator for Labeled Statements

A break statement, if unlabeled, will use the first object on breakList for which unlabeledBreakTarget is true. If the break is labeled, it will search the breakList for a matching label. Field label in breakNode will contain the label parameter, if any, used with the break. The code generator for break statements is shown in Figure 13.11.

```
BreakNode.CodeGen( )
 1.  listPos ← breakList
 2.  if stmtLabel = null
 3.      then while not listPos.unlabeledBreakTarget
 4.               do listPos ← listPos.next
 5.           GenFinalCalls(length(finalList) - listPos.finalDepth)
 6.           GenGoTo(listPos.breakLabel)
 7.      else while listPos.stmtLabel ≠ stmtLabel
 8.               do listPos ← listPos.next
 9.           GenFinalCalls(length(finalList) - listPos.finalDepth)
10.           GenGoTo(listPos.breakLabel)
```

Figure 13.11 Code Generation for Break Statements

In practice break statements are very easy to translate. Try statements with finally blocks are rarely used, so a break normally generates just a branch to the successor of the statement selected by the break. In C and C++ things are simpler still—breaks don't use labels so the innermost switch or looping statement is always selected.

As an example of how break and continue statements are translated, consider the following while loop

```
while (b1) {
    if (b2) continue;
    if (b3) break;
}
```

The following JVM code is generated

```
    goto        L1 ; Skip around loop body
L2:
    iload       2  ; Push local #2 (b2) onto the stack
    ifeq        L3 ; Goto L3 if b2 is false (0)
    goto        L1 ; Goto L1 to continue the loop
L3:
    iload       3  ; Push local #3 (b3) onto the stack
    ifeq        L4 ; Goto L4 if b3 is false (0)
    goto        L5 ; Goto L5 to break out of the loop
L1: L4:
    iload       1  ; Push local #1 (b1) onto the stack
    ifne        L2 ; Goto L2 if b1 is true (1)
L5:
```

Label `L1`, the target of the continue statement, is created by the labeledStmt node that contains the while loop. `L1` marks the location where the loop termination condition is tested; it is defined when the while loop is translated. Label `L5`, the target of the exit statement, is created and defined by the labeledStmt node just beyond the end of the while loop. Label `L2`, the top of the while loop, is created and defined when the loop is translated. Labels `L3` and `L4` are created and defined by the if statements within the loop's body .

Return StatementsA returnNode represents a return statement. The field returnVal is null if no value is returned; otherwise it is an AST node representing an expression to be evaluated and returned. If a return is nested within one or more try statements that have finally blocks, these blocks must be executed prior to doing the return. The code generator for a returnNode is shown in Figure 13.12. The code generation subroutine GenReturn(Value) generates a return instruction based on Value's type (an `ireturn`, `lreturn`, `freturn`, `dreturn`, or `areturn`). If Value is null, an ordinary `return` instruction is generated.

```
return.CodeGen( )
1.  if returnVal ≠ null
2.      then returnVal.CodeGen()
3.  GenFinalCalls(length(finalList))
4.  GenReturn(returnVal)
```

Figure 13.12 Code Generator for Return Statements

Goto StatementsJava contains no goto statement, but many other languages, including C and C++, do. Most 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. Languages usually do not require that identifiers used as statement labels be distinct from identifiers used for other purposes. Thus in C and C++, the statement

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

is legal. Labels are usually 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 (so that "forward gotos" are possible). Type checking guarantees that all labels used in gotos are in fact defined somewhere in the current procedure. To translate a goto statement (e.g., in C or C++), we simply assign an assembler label to each source-level label, the first time that label is encountered. That assembler label is generated when its corresponding source-level label is encountered during code generation. Each source-level `goto L` is translated to an assembly language goto using the assembler label assigned to `L`.

A few languages, like Pascal, allow **non-local gotos**. A non-local goto transfers control to a label outside the current procedure, exiting the current procedure. Non-local gotos generate far more than a single goto instruction due to the overhead of returning from a procedure to its caller.

Since we have not yet studied procedure call and return mechanisms (see Section 13.2.1), we will not detail the exact steps needed to perform a non-local goto. However, a non-local goto can be viewed as a very limited form of exception handling (see Section 13.2.6). That is, a non-local goto in effect throws an exception that is caught and handled within the calling procedure that contains the target label. Hence the mechanisms we develop for exception handling will be suitable for non-local gotos too.

13.1.5 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.

The AST for a switch statement, rooted at a switchNode is shown in Figure 13.13.

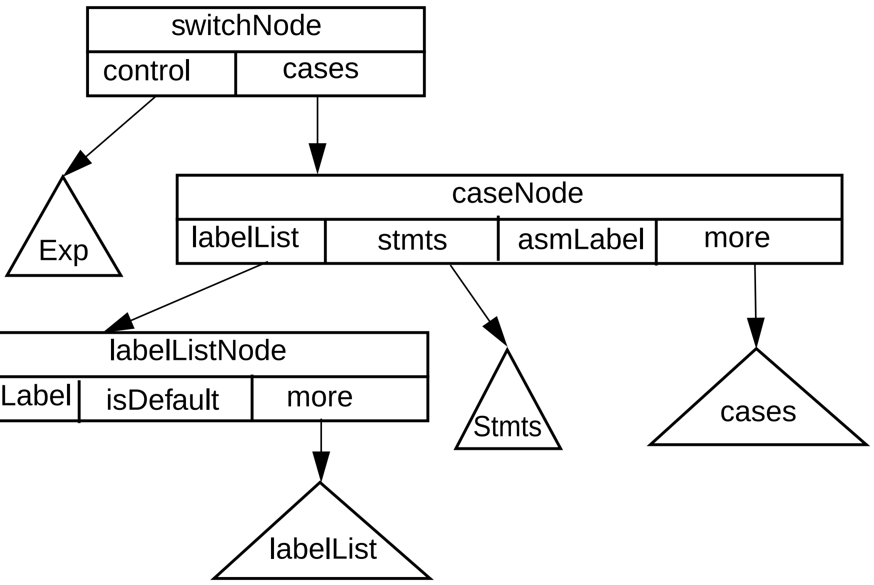

Figure 13.13 Abstract Syntax Tree for a Switch Statement

In the AST control represents an integer-valued expression; cases is a caseNode, representing the cases in the switch. Each caseNode has four 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. asmLabel is an assembler label that will be generated with the statements in stmts. more is either null or another caseListNode, representing the remaining cases in the switch.

A labelListNode contains an integer field caseLabel, 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.

Translating a switch involves a number of steps. The control expression has to be translated. Each of the statements in the switch body has to be translated, and assembler labels have to be added at each point marked by a case constant.

Before labeled cases are translated, we'll use the routines shown in Figure 13.14 to build casePairs, a list of case constants (field caseLabel) and associated assembler labels (field asmLabel). That is, if we see `case 123:` on a statement in the switch, we'll generate an assembler label (say `L234`). Then we'll pair `123` and `L234` together and add the pair to a list of other (caseLabel, asmLabel) pairs. We'll also assign an assembler label to the default case. If no default is specified, we will create one at the very bottom of the switch body.

The utility routines all serve to "preprocess" a switch statement immediately prior to code generation. containsDefault(labelList) examines a list of case labels (a labelListNode) to determine if the special default marker is present. addDefault-IfNecessary(caseNode) adds a default marker (at the very end of a list of case-Nodes) if no user-defined default is present. genCaseLabels(caseNode) creates a new assembly-level label for each case is a caseNode. getDefaultLabel(caseNode) extracts the assembly-level label associated with the default case.

buildCasePairs(labelList, asmLabel) builds a casePairs list for one labelList-Node given an assembly-level label to be used for all case labels in the node. build-CasePairs(cases) takes a caseNode and build a casePairs list for all the cases the node represents.

The reason we need these utility routines is that we wish to generate code that efficiently selects the assembler label associated with any value computed by the

```
containsDefault( labelList )
1.   if  labelList = null
2.         then  return false
3.         else  return labelList.isDefault or containsDefault(labelList.more)

addDefaultIfNecessary( cases )
1.   if  containsDefault(cases.labelList)
2.         then  return
3.   elsif  cases.more = null
4.         then  cases.more ←
                     caseNode(labelListNode(0, true, null), null, null, null)
5.         else   addDefaultIfNecessary(cases.more)

genCaseLabels( cases )
1.   if  cases ≠ null
2.         then  cases.asmLabel ← CreateUniqueLabel()
3.               genCaseLabels(cases.more)

getDefaultLabel( cases )
1.   if  containsDefault(cases.labelList)
2.         then  return  cases.asmLabel
3.         else  return getDefaultLabel(cases.more)

buildCasePairs( labelList, asmLabel )
1.   if  labelList = null
2.         then  return null
3.   elsif  labelList.isDefault
4.         then  return  buildCasePairs(labelList.more, asmLabel)
5.         else  return  casePairs(labelList.caseLabel, asmLabel,
                                     buildCasePairs(labelList.more, asmLabel))

buildCasePairs( cases )
1.   if  cases = null
2.         then  return null
3.         else  return  appendList( buildCasePairs(cases.labelList, asmLabel),
                                       buildCasePairs(cases.more))
```

Figure 13.14 Utility Code Generation Routines for Switch Statements

control expression. For a switchNode, s, buildCasePairs(s.cases) will extract each case constant that appears in the switch and pair it with the assembler label the case constant refers to. We need to decide how to efficiently search this list at run-time. Since the number and numeric values of case constants can vary greatly, no single search approach is uniformly applicable. In fact, JVM bytecodes directly support two search strategies.

Often the range of case constants that appear in a switch is **dense**. That is, a large fraction of the values in the range from the smallest case constant to the greatest case constant appear as actual case constants. For example, given

```
switch(i) {
   case 1: a = 1; break;
   case 3: a = 3; break;
   case 5: a = 5; break;
}
```

The case constants used (1, 3, 5) are a reasonably large fraction of the values in the range 1..5. When the range of case constants is dense, a **jump table** translation is appropriate. As part of the translation we build an array of assembler labels (the jump table) indexed by the control expression. The table extends from the smallest case constant used in the switch to the largest. That is, if case constant `c` labels a statement whose assembly language label is `Li`, then `table[c] = Li`. Positions in the table not explicitly named by a case label get the assembly label of the default case. Control expression values outside the range of the table also select the label of the default case.

As an example, reconsider the above switch. Assume the three cases are assigned the assembly language labels `L1`, `L2`, and `L3`, and the default (a null statement at the very bottom on the switch) is given label L4. Then the jump table, indexed by values in the range 1..5, contains the labels (L1, L4, L2, L4, L3).

The JVM instruction tableswitch conveniently implements a jump table translation of a switch. Its operands are default, low, high, and table. Default is the label of the default case. Low is the lowest case constant represented in table. High is the highest case constant represented in table. Table is a table of high-low+1 labels, each representing the statement to be executed for a particular value in the range low..high. Tableswitch pops an integer at the top of the JVM stack, and uses it as an index into table (if the index is out of range default is used). Control is transferred to the code at the selected label.

For the above example we would generate

```
tableswitch default=L4,low=1, high=5,
        L1, L4, L2, L4, L3
```

Not all switch statements are suitable for a jump table translation. If the value of high-low (and hence the size of the jump table) is too large, an unacceptably large amount of space may be consumed by the tableswitch instruction.

An alternative is to search a list of case constant, assembler label pairs. If the control expression's value matches a particular case constant, the corresponding label is selected and jumped to. If no match is found, a default label is used. If we sort the list of case constant, assembler label pairs based on case constant values, we need not do a linear search; a faster binary search may be employed.

The JVM instruction lookupswitch conveniently implements a search table translation of a switch. Its operands are default, size, and table. default is the label of the default case. `Size` is the number of pairs in `table`. `Table` is a sorted list of case constant, assembler label pairs, one pair for each `case` constant in the switch. `lookupswitch` pops an integer at the top of the JVM stack, and searches for a match with the first value of each pair in `table` (if no match is found `default` is used). Control is transferred to the code at the label paired with the matching `case` constant.

For example, consider

```
switch(i) {
   case -1000000:  a = 1; break;
   case  0:        a = 3; break;
   case  1000000:  a = 5; break;
}
```

where the three cases are assigned the assembly language labels `L1`, `L2`, and `L3`, and the default (a null statement at the very bottom on the switch) is given label `L4`. We would generate

```
lookupswitch default=L4,size=3,
       -1000000:L1,
       0:L2,
       1000000: L3
```

The decision as to whether to use a jump table or search table translation of a switch statement is not always clear-cut. The most obvious factor is the size of the jump table and the fraction of its entries filled with non-default labels. However, other factors, including the programmer's goals (size versus speed) and implementation limitations may play a role. We'll assume that a predicate generate-JumpTable(switchNode) exists that decides whether to use a jump table for a particular switch statement. The details of how generateJumpTable makes its choice are left to individual implementors.

No matter which implementation we choose, a variety of utility routines will be needed to create a jump table or search table from a casePairs list.

sortCasePairs(casePairsList) will sort a casePairs list into ascending order, based on the value of caseLabel. getMinCaseLabel(casePairsList) will extract the minimum case label from a sorted casePairs list. getMaxCaseLabel(case-PairsList) will extract the maximum case label from a sorted casePairs list. gen-Tableswitch(defaultLabel, low, high, table) generates a `tableswitch` instruction given a default label, a low and high jump table index, and a jump table (encoded as an array of assembly labels). Finally, genLookupswitch(defaultLabel,size, case-PairsList) generates a `lookupswitch` instruction given a default label, a list size, a list of casePairs.

We can now complete the translation of a switch statement, using the code generators defined in Figure 13.15.

As an example of code generation for a complete switch statement consider

```
buildJumpTable( casePairsList, defaultLabel )
 1.  min ← getMinCaseLabel(casePairsList)
 2.  max ← getMaxCaseLabel(casePairsList)
 3.  table ← string[max-min+1]
 4.  for    i ← min to max
 5.         do  if casePairList.caseLabel = i
 6.                 then  table[i-min] ← casePairsList.asmLabel
 7.                       casePairsList ← casePairsList.next
                    else  table[i-min] ← defaultLabel
 8.  return  table

caseNode.CodeGen( )
 1.  GenLabel(asmLabel)
 2.  stmts.CodeGen()
 3.  if  more ≠ null
 4.      then  more.CodeGen

 switchNode.CodeGen( )
 1.  control.CodeGen()
 2.  addDefaultIfNecessary (cases)
 3.  genCaseLabels (cases)
 4.  list ← buildCasePairs(cases)
 5.  list ← sortCasePairs(list)
 6.  min ← getMinCaseLable(list)
 7.  max ← getMaxCaseLable(list)
 8.  if  generateJumpTable(cases)
 9.      then  genTableswitch(getDefaultLabel(cases), min, max,
                      buildJumpTable(list, getDefaultLabel(cases)))
10.      else  genLookupwitch(getDefaultLabel(cases), length(list), list)
11.  cases.CodeGen()
```

Figure 13.15 Code Generation Routines for Switch Statements

```
switch(i) {
     case 1:   a = 1;  break;
     case 3:   a = 3;  break;
     case 5:   a = 5;  break;
     default:  a = 0;
}
```

Assuming that `i` and `a` are locals with variable indices of 3 and 4, the JVM code that we generate is

```
    iload       1  ; Push local #3 (i) onto the stack
    tableswitch default=L4, low=1, high=5,
                L1, L4, L2, L4, L3
L1: iconst_1       ; Push 1
    istore      4  ; Store 1 into local #4 (a)
    goto        L5 ; Break
L2: iconst_3       ; Push 3
    istore      4  ; Store 3 into local #4 (a)
    goto        L5 ; Break
L3: iconst_5       ; Push 5
    istore      4  ; Store 5 into local #4 (a)
    goto        L5 ; Break
L4: iconst_0       ; Push 0
    istore      4  ; Store 0 into local #4 (a)
L5:
```

Note that Java, like C and C++, requires the use of a break after each case statement to avoid "falling into" remaining case statements. Languages like Pascal, Ada and Modula 3 automatically force an exit from a case statement after it is executed; no explicit break is needed. This design is a bit less error-prone, though it does preclude the rare circumstance where execution of a whole list of case statements really is wanted. During translation, the code generator for a case statement automatically includes a goto after each case statement to a label at the end of the case.

Ada also allows a case statement to be labeled with a range of values (e.g., `1..10` can be used instead of ten distinct constants). This is a handy generalization, but it does require a change in our translation approach. Instead of pairing case constants with assembly labels, we pair **case ranges** with labels. A single case constant, `c`, is treated as the trivial range `c..c`. Now when casePairs lists are sorted and traversed, we consider each value in the range instead of a single value.

13.2 Code Generation for Subroutine Calls

Subroutines, whether they are procedures or functions, whether they are recursive or non-recursive, whether they are members of classes or independently declared, form the backbone of program organization. It is essential that we understand how to effectively and efficiently translate subroutine bodies and calls to subroutines.

13.2.1 Parameterless Calls

Since calls in their full generality can be complicated, we'll start with a particularly simple kind of subroutine call—one without parameters. We'll also start with calls to static subroutines; that is, subroutines that are individually declared or are members of a class rather than a class instance.

Given a call `subr()`, what needs to be done? To save space and allow recursion, subroutines are normally translated in **closed** form. That is, the subroutine is translated only once, and the same code is used for all calls to it. This means we must branch to the start of the subroutine's code whenever it is called. However, we must also be able to get back to the point of call after the subroutine has executed. Hence, we must capture a **return address**. This can be done by using a special "subroutine call" instruction that branches to the start address of a subroutine *and* saves the return address of the caller (normally this is the next instruction after the subroutine call instruction).

As we learned in Section 11.2, calls normally push a **frame** onto the run-time stack to accommodate locals variables and control information. Frames are typically accessed via a **frame pointer** that always points to the currently active frame. Hence updating the stack pointer (to push a frame for the subroutine being called) and the frame pointer (to access the newly pushed frame) are essential parts of a subroutine call.

After a subroutine completes execution, the subroutine's frame must be popped from the stack and the caller's stack top and frame pointer must be restored. Then control is returned to the caller, normally to the instruction immediately following the call instruction that began subroutine execution.

If the subroutine called is a function, then a **return value** must also be provided. In Java (and many other languages) this is done by leaving the return value on the top of the caller's stack, where operands are normally pushed during execution. On register-oriented architectures, return values are often placed in specially designated registers.

In summary then, the following steps must be done to call a parameterless subroutine.

1. The caller's return address is established and control is passed to the subroutine.
2. A frame for the subroutine is pushed onto the stack and the frame pointer is updated to access the new frame.
3. The caller's stack top, frame pointer, and return address are saved in the newly pushed frame.
4. The subroutine's body is executed.
5. The subroutine's frame is popped from the stack.
6. The caller's stack top, frame pointer and return address are restored.
7. Control is passed to the caller's return address (possibly with a function return value).

The run-time stack prior to step 1 (and after step 7) is illustrated in Figure 13.16(a). The caller's frame is at the top of the stack, accessed by the frame pointer. The run time stack during the subroutine's execution (step 4) is shown in Figure 13.16(b). A frame for the subroutine (the callee) has been pushed onto the

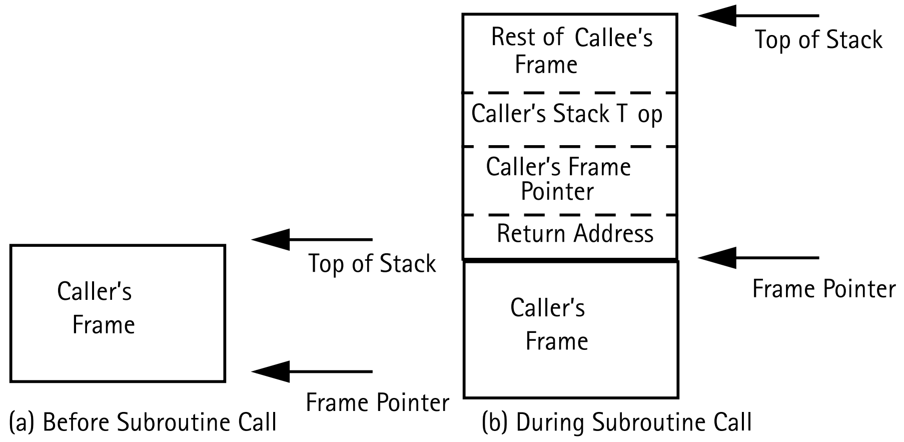

Figure 13.16 Run-time Stack during a Call

stack. It is now accessed through the frame pointer. The caller's stack top, frame pointer and return address are stored within the current frame. The values will be used to reset the stack after the subroutine completes and its frame is popped.

On most computers each of the steps involved in a call takes an instruction or two, making even a very simple subroutine call non-trivial. The JVM is designed to make subroutine calls compact and efficient. The single instruction

```
invokestatic index
```

can be used to perform steps 1-3. `index` is an index into the **JVM constant pool** that represents the static subroutine (class method) being called. Information on the subroutine's frame size and starting address is stored in the corresponding method info entry and class loader entry. Frame creation and saving of necessary stack information are all included as part of the instruction.

A `return` instruction is used to return from a void subroutine. Steps 5-7 are all included in the `return` instruction's operation. For functions, a typed return (`areturn`, `dreturn`, `freturn`, `ireturn`, `lreturn`) is used to perform steps 5-7 along with pushing a return value onto the caller's stack (after the subroutine's frame has be popped).

13.2.2 Parameters

In Java both objects and primitive values (integers, floats, etc.) may be passed as parameters. Both kinds of values are passed through the stack.The actual bit pattern of a primitive value is pushed, in one or two words. Objects are passed by pushing a one-word reference (pointer) to the object.

To call a subroutine with n parameters, we evaluate and push each parameter in turn, then call the subroutine (using an `invokestatic` instruction if the subroutine is a static member of a class).

During execution of the subroutine parameters are accessed as locals, starting with index 0. Names of class members (fields and methods) are referenced through references to the constant pool of the class. Hence a reference to `#5` in JVM code denotes the fifth entry in the constant pool.

As an example, given the class method

```
    static int subr(int a, int b) {return a+b;}
```

and the call `i = subr(1,10);` we would generate

```
Method int subr(int,int)
    iload          0  ; Push local 0, which is a
    iload          1  ; Push local 1, which is b
    iadd              ; Compute a+b onto stack
    ireturn           ; Return a+b to caller
; Now the call and assignment
    iconst_1          ; Push 1 as first parm
    bipush         10 ; Push 10 as second parm
    invokestatic   #4 ; Call Method subr
    istore         1  ; Store subr(1,10) in local #1 (i)
```

This approach to parameter passing works for any number of actual parameters. It also readily supports nested calls such as `f(1,g(2))`. As shown below, we begin by pushing `f`'s first parameter, which is safely protected in the caller's stack. Then `g`'s parameter is passed and `g` is called. When `g` returns, its return value has been pushed on the stack, exactly where `f`'s second parameter is expected. `f` can therefore be immediately called. The code is

```
iconst_1          ; Push f's first parm
iconst_2          ; Push g's first parm
invokestatic #5   ; Call Method g
invokestatic #4   ; Call Method f
```

On register-based architectures nested calls are more difficult to implement. Parameter registers already loaded for an enclosing call may need to be reloaded with parameter values for an inner call (see Exercise 12).

13.2.3 Member and Virtual Functions

Calling Instance MethodsWe have considered how static member functions are called. Now let's look at functions that are members of class instances. Since there can be an unlimited number of instances of a class (i.e., objects), a member of a class instance must have access to the particular object to which it belongs.

Conceptually, each object has its own copy of all the class's methods. Since methods can be very large in size, creating multiple copies of them is wasteful of space. An alternative is to create only one copy of the code for any method. When the method is invoked, it is passed, as an additional invisible parameter, a reference to the object in which the method is to operate.

As an example, consider the following simple class

```
class test {
    int a;
    int subr(int b) { return a+b;}
}
```

To implement the call `tt.subr(2)` (where `tt` is a reference to an object of class `test`), we pass a reference to `tt` as an implicit parameter to `subr`, along with the explicit parameter (2). We also use the instruction `invokevirtual` to call the method.

`invokevirtual` is similar to `invokestatic` except for the fact that it expects an implicit initial parameter that is a reference to the object within which the method will execute. During execution this reference is local 0, with explicit parameters indexed starting at 1. For method `subr` and the call to it we generate

```
Method int subr(int)
```

```
 aload    0        ; Load this pointer from local 0
 getfield #1       ; Load Field this.a
 iload    1        ; Load parameter b (local 1)
 iadd              ; Compute this.a + b
 ireturn           ; Return this.a + b
 ; Code for the call of tt.subr(2)
 aload          5  ; Push reference to object tt
 iconst_2          ; Push 2
 invokevirtual  #7 ; Call Method test.subr
```

Virtual FunctionsIn Java all subroutines are members of classes. Classes are specially designed to support **subclassing**. A new subclass may be derived from a given class (unless it is final). A class derived from a parent class (a superclass) inherits its parent's fields and methods. The subclass may also define new fields and methods and also redefine existing methods. When an existing method, `m`, is redefined it is often important that all references to `m` use the new definition, even if the calls appear in methods inherited from a parent. Consider

```
class C {
    String myClass(){return "class C";}
    String whoAmI(){return "I am " + myClass();}
}
class D extends C {
    String myClass(){return "class D";}
}
class virtualTest {
    void test() {
       C c_obj = new D();
       System.out.println(c_obj.whoAmI());
}   }
```

When `c_obj.whoAmI()` is called, the call to `myClass` is resolved to be the definition of `myClass` in class `D`, even though `c_obj` is declared to be of class `C` and a definition of `myClass` exists in class `C` (where `whoAmI` is defined). Methods that are automatically replaced by new definitions in subclasses are called **virtual functions** (or **virtual methods**).

How does a Java compiler ensure that the correct definition of a virtual function is selected? In Java all instance methods are automatically virtual. The instruction `invokevirtual` uses the class of the object reference passed as argument zero to determine which method definition to use. That is, inside `whoAmI()` the call to `myClass()` generates

```
aload_0          ; Push this, a reference to dobj
invokevirtual #11 ; Call Method myClass() using this ptr
```

The call is resolved inside `dobj`, whose class is `D`. This guarantees that `D`'s definition of `myClass()` is used.

In C++ member functions may be explicitly designated as virtual. Since C++ implementations don't generate JVM code, an alternative implementation is used. For classes that contain functions declared virtual, class instances contain a pointer to the appropriate virtual function to use. Thus in the above example objects of class C would contain a pointer to C's definition of myClass whereas objects of class D contain a pointer to D's definition. When dobj.whoamI() is called, whoAmI is passed a pointer to dobj (the this pointer) which ensures that whoamI will use D's definition of myClass.

Static class members are resolved statically. Given

```
class CC {
    static String myClass(){return "class CC";}
    static String whoAmI(){return "I am " + myClass();}
}
class DD extends CC {
    static String myClass(){return "class DD";}
}
class stat {
    static void test() {
        System.out.println(DD.whoAmI());
}   }
```

"I am class CC" is printed out. The reason for this is that calls to static members use the invokestatic instruction which resolves methods using the class of the object within which the call appears. Hence since the call to myClass is within class C, C's definition is used. This form of static resolution is exactly the same as that found in C++ when non-virtual member functions are called.

13.2.4 Optimizing Calls

Subroutines are extensively used in virtually all programming languages. Calls can be costly, involving passing parameters, object pointers and return addresses, as well as pushing and popping frames. Hence ways of optimizing calls are of great interest. We shall explore a variety of important approaches.

Inlining CallsSubroutines are normally compiled as **closed subroutines**. That is, one translation of the subroutine is generated, and all calls to it use the same code. An alternative is to translate a call as an **open subroutine**. At the point of call the subroutine body is "opened up," or expanded. This process is often called **inlining** as the subroutine body is expanded "inline" at the point of call.

In Java, inlining is usually restricted to static or final methods (since these can't be redefined in subclasses). In C and C++ non-virtual functions may be inlined.

Inlining is certainly not suitable for all calls, but in some circumstances it can be a valuable translation scheme. For example, if a subroutine is called at only one place, inlining the subroutine body makes sense. After all, the subroutine body has to appear *somewhere*, so why not at the only place it is used? Moreover, with inlining the expense of passing parameters and pushing and popping a frame can be avoided.

Another circumstance where inlining is widely used is when the body of a subroutine is very small, and the cost of doing a call is as much (or more) than executing the body itself.

It is often the case that one or more parameters to a call are literals. With inling these literals can be expanded within the subroutine body. This allows simplification of the generated code, sometimes even reducing a call to a single constant value.

On the other hand, inlining large subroutines that are called from many places probably doesn't make sense. Inlining a recursive subroutine can be disastrous, leading to unlimited growth as nested calls are repeatedly expanded. Inlining that significantly increases the overall size of a program may be undesirable, requiring a larger working set in memory and fitting into the instruction cache less effectively. For mobile programs that may be sent across slow communication channels, even a modest growth in program size, caused by inlining, can be undesirable.

Whether a call is translated in the "normal" manner via a subroutine call to a closed body or whether it is inlined, it is essential that the call have the same semantics. That is, since inlining is an optimization it may not affect the results a program computes.

To inline a subroutine call, we enclose the call within a new block. Local variables, corresponding to the subroutine's formal parameters are declared. These local variables are initialized to the actual parameters at the point of call, taking care that the names of formal and actual parameter names don't clash. Then the AST for the subroutine body is translated at the point of call. For functions, a local variable corresponding to the function's return value is also created. A return involving an expression is translated as an assignment of the expression to the return variable followed by a break from the subroutine body.

As an example, consider

```
static int min(int a, int b) {
    if (a<=b) return a;
    else return b;
}
```

and the call `a = min(1,3)`. This call is an excellent candidate for inlining as the function's body is very small and the actual parameters are literals. The first step of the inlining is to (in effect) transform the call into

```
{   int parmA = 1;
    int parmB = 3;
    int result;
body:{
       if (parmA <= parmB) {result = parmA; break body;}
       else                {result = parmB; break body;}
    }
    a=result;
}
```

Now as this generated code is translated, important simplifications are possible. Using **constant propagation** (see Chapter 16), a variable known to contain a constant value can be replaced by that constant value. This yields

```
{   int parmA = 1;
```

```
    int parmB = 3;
    int result;
body:{
        if (1<=3)   {result = 1; break body;}
        else        {result = 3; break body;}
 }
 a=result;
}
```

Now `1<=3` can be simplified (folded) into `true`, and `if (true) ...` can be simplified to its then part. This yields

```
{   int parmA = 1;
    int parmB = 3;
    int result;
body:{
        {result = 1; break body;}
    }
    a=result;
}
```

Now a `break` at the very end of a statement list can be deleted as unnecessary, and `result`'s constant value can be propagated to its uses. W e now obtain

```
{   int parmA = 1;
    int parmB = 3;
    int result;
```

```
  body:{
          {result = 1;}
      }
      a=1;
  }
```

Finally, any value that is never used within its scope is **dead**; its declaration and definitions can be eliminated. This leads to the final form of our inlined call:

```
a=1;
```

Non-recursive and Leaf ProceduresMuch of the expense of calls lies in the need to push and pop frames. Frames in general are needed because a subroutine may be directly or indirectly recursive. If a subroutine is recursive, we need a frame to hold distinct values of parameters and locals for each call.

But from experience we know many subroutines *aren't* recursive, and for non-recursive subroutines we can avoid pushing and popping a frame, thereby making calls more efficient. Instead of using a frame to hold parameters and locals, we can statically allocate space for them.

The Java JVM is designed around frames. It pushes and pops them as an integral part of call instructions. Moreover, locals may be loaded from a frame very quickly and compactly. However, for most other computer architectures pushing and popping a frame can involve a significant overhead. When a subroutine is known to be non-recursive, all that is needed to do a call is to pass parameters (in registers when possible) and to do a "subroutine call" instruction that saves the return address and transfers control to the start of the subroutine.

Sometimes a subroutine not only is non-recursive, but it also calls no subroutines at all. Such subroutines are called **leaf procedures**. Again, leaf procedures need no frames. Moreover, since they do no calls at all, neither parameter registers nor the return address register need be protected. This makes calling leaf procedures particularly inexpensive.

How do we determine whether a subroutine is non-recursive? One simple way is to build a **call graph** by traversing the AST for a program. In a call graph, nodes represent procedures, and arcs represent calls between procedures. That is, an arc from node `A` to node `B` means a call of `B` appears in `A`. After the call graph is built, we can analyze it. If a node `A` has a path to itself, then `A` is potentially recursive; otherwise it is not.

Testing for leaf procedures is even easier. If the body of a method contains no calls (that is, no `invokestatic` or `invokevirtual` instructions), then the method is a leaf procedure

Knowing that a procedure is non-recursive or a leaf procedure is also useful in deciding whether to inline calls of it.

Run-time Optimization of BytecodesSubroutine calls that use the `invokestatic` and `invokevirtual` instructions involve special checks to verify that the methods called exist and are type-correct. The problem is that class definitions may be dynamically loaded, and a call to a method `C.M` that was valid when the call was compiled may no longer be valid due to later changes to class `C`. When a reference to `C.M` is made in an `invokestatic` or `invokevirtual` instruction, class `C` is checked to verify that `M` still exists and still has the expected type. While this check is needed the first time `C.M` is called, it is wasted effort on subsequent calls. The JVM can recognize this, and during execution, valid `invokestatic` and `invokevirtual` instructions may be replaced with the variants `invokestatic_quick` and `invokevirtual_quick`. These instructions have the same semantics as the original instructions, but skip the unnecessary validity checks after their first execution. Other JVM instructions, like those that access fields, also have quick variants to speed execution.

13.2.5 Higher Order Functions and Closures

Java does not allow subroutines to be manipulated like ordinary data (though a Java superset, Pizza [Odersky and Wadler 1997] does). That is, a subroutine cannot be an argument to a call and cannot be returned as the value of a function. However, other programming languages, like ML [Milner et al 1997] and Haskell [Jones et al 1998], are **functional** in nature. They encourage the view that functions are just a form of data that can be freely created and manipulated. Functions that take other functions as parameters or return parameters as results are called **higher-order.**

Higher-order functions can be quite useful. For example, it is sometimes the case that computation of `f(x)` takes a significant amount of time. Once `f(x)` is known, it is a common optimization, called **memoizing**, to table the pair (x,f(x)) so that subsequent calls to f with argument x can use the known value of f(x) rather than recompute it. In ML it is possible to write a function memo that takes a function f and an argument arg. The function memo computes f(arg) *and* also returns a "smarter" version of f that has the value of f(arg) "built into" it. This smarter version of f can be used instead of f in all subsequent computations.

```
fun memo(fct,parm)= let val ans = fct(parm) in
 (ans, fn x=> if x=parm then ans else fct(x))end;
```

When the version of fct returned by memo is called, it must have access to the values of parm, fct and ans, which are used in its definition. After memo returns, its frame must be preserved since that frame contains parm, fct and ans within it.

In general in languages with higher-order functions, when a function is created or manipulated, we must maintain a pair of pointers. One is to the machine instructions that implement the function, and the other is to the frame (or frames) that represent the function's **execution environment**. This pair of pointers is called a **closure**. When functions are higher-order, a frame corresponding to a call may be accessed *after* the call terminates (this is the case in memo). Thus frames can't always be allocated on the run-time stack. Rather, they are allocated in the heap and garbage-collected, just like user-created data. This appears to be inefficient, but it need not be if an efficient garbage collector is used (see Section 11.3).

13.2.6 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.** When an exception is thrown control transfers immediately to the catch clause that handles it. Code at the point where the throw occurred is not resumed.

Exceptions form a clean and general mechanism for identifying and handling unexpected or erroneous situations. They are cleaner 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).

As an example consider the following Java code

```
class NullSort extends Throwable{};
   int  b[] = new int[n];
   try {
      if (b.length > 0)
      // sort b
```

```
          else throw new NullSort(); }
    catch (NullSort ns) {
       System.out.println("Attempt to sort empty array");
    }
```

An integer array `b` is created and it is sorted within the try clause. The length of `b` is tested and a `NullSort` exception is thrown if the array's length is 0 (since trying to sort an empty array may indicate an error situation.) In the example, when a `NullSort` exception is caught, an error message is printed.

Exceptions are designed to be **propagated dynamically**. If a given exception isn't caught in the subroutine (method) where it is thrown, it is propagated back to the caller. In particular, a return is effected (popping the subroutine's frame) and the exception is rethrown at the return point. This process is repeated until a handler is found (possibly a default handler at the outermost program level).

In practice, we'd probably package the sort as a subroutine (method), allowing it to throw and propagate a `NullSort` exception:

```
...
static int [] sort(int [] a) throws NullSort {
    if (a.length > 0)
       // sort a
    else throw new NullSort();
    return a;
 }
...
```

```
int  b[] = new int[n];
try {sort(b);}
catch (NullSort ns) {
    System.out.println("Attempt to sort empty array");
} ...
```

Exception handling is straightforward in Java. The JVM maintains an **exception table** for each method. Each table entry contains four items: fromAdr, toAdr, exceptionClass and handlerAdr. fromAdr is the first bytecode covered by this entry; toAdr is the last bytecode covered. exceptionClass is the class of exception handled. exceptionAdr is the address of the corresponding handler. If an exception is thrown by an instruction in the range fromAdr..toAdr, the class of the thrown exception is checked. If the thrown class **matches** exceptionClass (equals the class or is a subclass of it), control passes to the exceptionAdr. If the class thrown doesn't match exceptionClass or the exception was thrown by an instruction outside fromAdr..toAdr, the next item in the exception table is checked. If the thrown exception isn't handled by any entry in the exception table, the current method is forced to return and the exception is thrown again at the return point of the call.

Since all this checking is automatically handled as part of the execution of the JVM, all we need to do is create exception table entries when try and catch blocks are compiled. In particular, we generate labels to delimit the extent of a try block, and pass these labels to catch blocks, which actually generate the exception table entries.

On architectures other than the JVM, exception processing is more difficult. Code must be generated to determine which exception handler will process an exception of a given class thrown from a particular address. If no handler is defined, a return must be effected (popping the stack, restoring registers, etc.). Then the exception is rethrown at the call's return address.

We'll begin with the translation of a tryNode, whose structure is shown in Figure 13.17. A try statement may have a finally clause that is executed whenever

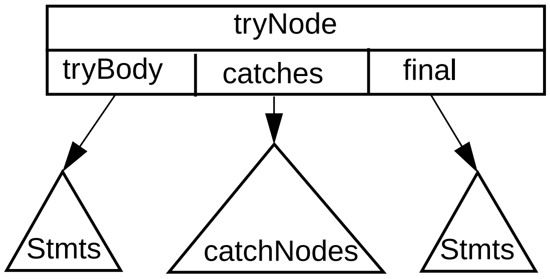

Figure 13.17 Abstract Syntax Tree for a Try Statement

a try statement is exited. A finally clause complicates exception handling, so we'll defer it until the next section. For now, we'll assume final is null. In fact, finally clauses are rarely used, so we will be considering the most common form of exception handling in Java.

First, we will define a few utility and code generation subroutines. GenStore(identNode) will generate code to store the value at the stack top into the identifier represented by identNode. addToExceptionTable(fromLabel, toLabel, exceptionClass, catchLabel) will add the four tuple (fromLabel, toLabel, exceptionClass, catchLabel) to current exception table.

```
tryNode.CodeGen( )
1.  fromLabel ← CreateUniqueLabel()
2.  GenLabel(fromLabel)
3.  tryBody.CodeGen()
4.  exitLabel ← CreateUniqueLabel()
5.  GenGoTo(exitLabel)
6.  toLabel ← CreateUniqueLabel()
7.  GenLabel(toLabel)
8.  catches.CodeGen(fromLabel, toLabel, exitLabel)
9.  GenLabel(exitLabel)
```

Figure 13.18 Code Generation Routine for Try Statements

In translating a try statement, we will first generate assembly-level labels immediately before and immediately after the code generated for tryBody, the body of the try block. catches, a list of catchNodes is translated after the try block. We generate a branch around the code for catch blocks as they are reached only by the JVM's exception handling mechanism. This translation is detailed in Figure 13.18.

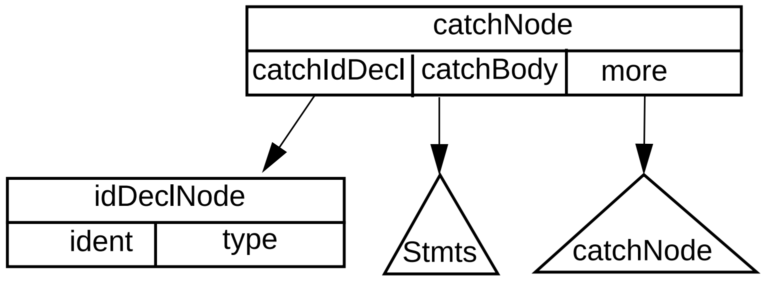

Figure 13.19 Abstract Syntax Tree for a Catch Block

A catchNode is shown in Figure 13.19. To translate a catch block we first define a label for the catch block. When the catch block is entered, the object thrown is on the stack. We store it into the exception parameter ident. We translate catchBody, and then add an entry to the end of the current method's exception table. Finally, more, the remaining catch blocks, are translated. This translation is detailed in Figure 13.20.

```
catchNode.CodeGen( fromLabel, toLabel, exitLabel )
1.  catchLabel ← CreateUniqueLabel()
2.  GenLabel(catchLabel)
3.  catchIdDecl.CodeGen()
4.  GenStore(catchIdDecl.Ident)
5.  catchBody.CodeGen()
6.  addToExceptionTable(fromLabel, toLabel-1,  catchIdDecl.Type,
                         catchLabel)
7.  if  more ≠ null
8.      then  GenGoTo(exitLabel)
9.            more.CodeGen(fromLabel, toLabel, exitLabel)
```

Figure 13.20 Code Generation Routine for Catch Blocks

Since exception table entries are added to the end of the table, our code generator correctly handles nested try blocks and multiple catch blocks.

As an example, consider the following Java program fragment. In the try block, `f(a)` is called; it may throw an exception. If no exception is thrown, `b` is assigned the value of `f(a)`. If exception `e1` is thrown, `b` is assigned 0. If `e2` is thrown, `b` is assigned 1. If any other exception is thrown, the exception to propagated to the containing code. In the exception table, the expression `L-1` denotes one less than the address assigned to `L`.

```
class e1 extends Throwable{};
class e2 extends Throwable{};
try {b = f (a);}
catch (e1 v1) {b = 0;}
catch (e2 v2) {b = 1;}
```

The code we generate is

```
L1: iload          2  ; Push local #2 (a) onto stack
    invokestatic   #7 ; Call Method f
    istore         1  ; Store f(a) into local #1 (b)
    goto           L2 ; Branch around catch blocks
L3:                   ; End of try block
L4: astore         3  ; Store thrown obj in local #3 (v1)
    iconst_0          ; Push 0
    istore         1  ; Store 0 into b
    goto           L2 ; Branch around other catch block
L5: astore         4  ; Store thrown obj in local #4 (v2)
    iconst_1          ; Push 1
    istore         1  ; Store 1 into b
L2:

Exception table:
from     to   catch  exception
 L1     L3-1  L4     Class e1
 L1     L3-1  L5     Class e2
```

Finally blocks and FinalizationWe exit a block of statements after the last statement of the block is executed. This is **normal** termination. However, a block may be exited **abnormally** in a variety of ways, including executing a break or continue, doing a return, or throwing an exception.

Prior to exiting a block, it may be necessary to execute **finalization code**. For example in C++, destructors for locally allocated objects may need to be executed. If reference counting is used to collect garbage, counts must be decremented for objects referenced by local pointers.

In Java, finalization is performed by a finally block that may be appended to a try block. The statements in a finally block must be executed no matter how the try block is exited. This includes normal completion, abnormal completion because of an exception (whether caught or not) and premature completion because of a break, continue or return statement.

Since the finally block can be reached in many ways, it is not simply branched to. Instead it is reached via a simplified subroutine call instruction, the `jsr`. This instruction branches to the subroutine and saves the return address on the stack. No frame is pushed. After the finally statements are executed, a `ret` instruction returns to the call point (again no frame is popped).

A try block that exits normally will “call” the finally block just before the entire statement is exited. A catch block that handles an exception thrown in the try block will execute its body and then call the finally block. An exception not handled in any catch block will have a default handler that calls the finally block and then rethrows the exception for handling by the caller. A break, continue or exit will call the finally block before it transfers control outside the try statement.

```
    tryNode.CodeGen( )
 1.   fromLabel ← CreateUniqueLabel()
 2.   GenLabel(fromLabel)
 3.   if  final ≠ null
 4.       then  finalLabel ← CreateUniqueLabel()
 5.             finalList ← listNode(finalLabel, finalList)
 6.   tryBody.CodeGen()
 7.   if final ≠ null
 8.       then  GenJumpSubr(finalLabel)
 9.             finalList ← finalList.next
10.       else  finalLabel ← null
11.   exitLabel ← CreateUniqueLabel()
12.   GenGoTo(exitLabel)
13.   toLabel ← CreateUniqueLabel()
14.   GenLabel(toLabel)
15.   catches.CodeGen(fromLabel, toLabel, exitLabel, finalLabel)
16.   if final ≠ null
17.       then  defaultHandlerLabel ← CreateUniqueLabel()
18.             GenLabel(defaultHandlerLabel)
19.             exceptionLoc ← GenLocalDecl(Throwable)
20.             GenStoreLocalObject(exceptionLoc)
21.             GenJumpSubr(finalLabel)
22.             GenLoadLocalObject(exceptionLoc)
23.             GenThrow()
24.             addToExceptionTable(fromLabel, defaultHandlerLabel-1,
                                    Throwable, defaultHandlerLabel)
25.             GenLabel(finalLabel)
26.             returnLoc ← GenLocalDecl(Address)
27.             GenStoreLocalObject(returnLoc)
28.             final.CodeGen()
29.             GenRet(returnLoc)
30.   GenLabel(exitLabel)
```

Figure 13.21 Code Generation Routine for Try Statements with Finally blocks

We will consider the complications finally blocks add to the translation of a tryNode and a catchNode. First, we will define a few more utility and code generation subroutines. We'll again use GenLocalDecl(Type), which declares a local variable and returns its frame offset or variable index.

```
catchNode.CodeGen( fromLabel, toLabel, exitLabel, finalLabel )
 1.  catchLabel ← CreateUniqueLabel()
 2.  GenLabel(catchLabel)
 3.  catchIdDecl.CodeGen()
 4.  GenStore(catchIdDecl.Ident)
 5.  catchBody.CodeGen()
 6.  if  finalLabel ≠ null
 7.      then  GenJumpSubr(finalLabel)
 8.  addToExceptionTable(fromLabel, toLabel-1,  catchIdDecl.Type,
                          catchLabel)
 9.  if  more ≠ null or finalLabel ≠ null
10.      then  GenGoTo(exitLabel)
11.  if  more ≠ null
12.      then  more.CodeGen(fromLabel, toLabel, exitLabel, finalLabel)
```

Figure 13.22 Code Generation Routine for Catch Blocks with Finally Blocks

Recall that GenJumpSubr(Label) generates a subroutine jump to Label (a `jsr` in the JVM). GenRet(localIndex) will generate a return using the return address stored at localIndex in the current frame (in the JVM this is a `ret` instruction). GenThrow() will generate a throw of the object currently referenced by the top of the stack (an `athrow` instruction in the JVM).

GenStoreLocalObject(localIndex) will store an object reference into the frame at localIndex (in the JVM this is an `astore` instruction). GenLoadLocalObject(localIndex) will load an object reference from the frame at localIndex (in the JVM this is an `aload` instruction).

An extended code generator for a tryNode that can handle a finally block is shown in Figure 13.21. A non-null final field requires several additional code generation steps. First, in lines 4 and 5, a label for the statements in the finally block is created and added to finalList (so that break, continue and return statements can find the finally statements that need to be executed).

In line 8, a subroutine jump to the finally block is placed immediately after the translated try statements (so that finally statements are executed after statements in the try block). In line 9 the current finalLabel is removed from the finalList (because the tryBody has been completely translated).

After all the catchNodes have been translated, several more steps are needed. In lines 17 to 24 a default exception handler is created to catch all exceptions not caught by user-defined catch blocks. The default handler stores the thrown exception, does a subroutine jump to the finally block, and then rethrows the exception so that it can be handled by an enclosing handler.

In lines 25 to 29, the finally block is translated. Code to store a return address in a local variable is generated, the statements within the finally block are translated, and a return is generated.

An extended code generator for a catchNode that can handle a finally block is shown in Figure 13.22. A finalLabel parameter is added. In line 7 a subroutine call to the finalLabel is added after the statements of each catchBody.

As an example, consider the translation of the following try block with an associated finally block.

```
class e1 extends Throwable{};
try {a = f(a);}
catch (e1 v1) {a = 1;}
finally {b = a;}
```

The code we generate is

```
L1: iload          2  ; Push a onto stack
```

```
    invokestatic   #7 ; Call Method f
    istore         2  ; Store f(a) into local #2 (a)
    jsr            L2 ; Execute finally block
    goto           L3 ; Branch past catch & finally block
L4:                   ; End of try block
L5:
    astore         3  ; Store thrown obj in local #3 (v1)
    iconst_1          ; Push 1
    istore         2  ; Store 1 into a
    jsr            L2 ; Execute finally block
    goto           L3 ; Branch past finally block
 ; Propagate uncaught exceptions
L6:
    astore         4  ; Store thrown obj into local #4
    jsr            L2 ; Execute finally block
    aload          4  ; Reload thrown obj from local #4
    athrow            ; Propagate exception to caller
L2: astore         5  ; Store return adr into local #5
    iload          2  ; Load a
    istore         1  ; Store a into local #1 (b)
    ret            5  ; return using adr in local #5
L3:
Exception table:
```

```
from   to    catch  exception
 L1     L4-1  L5    Class e1
 L1     L6-1  L6    Class Throwable
```

13.2.7 Support for Run-Time Debugging

Most modern programming languages are supported by a sophisticated symbolic debugger. Using this tool, programmers are able to watch a program and its variables during execution. It is essential, of course, that a debugger and compiler effectively cooperate. The debugger must have access to a wide variety of information about a program, including the symbolic names of variables, fields and subroutines, the run-time location and values of variables, and the source statements corresponding to run-time instructions.

The Java JVM provides much of this information automatically, as part of its design. Field, method, and class names are maintained, in symbolic form, in the run-time class file, to support dynamic linking of classes. Attributes of the class file include information about source files, line numbers, and local variables. For other languages, like C and C++, this extra information is added to a standard ".o" file, when debugging support is requested during compilation.

Most of the commands provided by a debugger are fairly easy to implement given the necessary compiler support. Programs may be executed a line at a time by extracting the run-time instructions corresponding to a single source line and executing them in isolation. Variables may be examined and updated by using information provided by a compiler on the variable's address or frame offset.

Other commands, like setting breakpoints, take a bit more care. A JVM `breakpoint` instruction may be inserted at a particular line number or method header. Program code is executed normally (to speed execution) until a `breakpoint` is executed. Normal execution is then suspended and control is returned to the debugger to allow user interaction. `breakpoint` instructions may also be used to implement more sophisticated debugging commands like "watching" a variable. Using compiler-generated information on where variables and fields are updated, `breakpoint` instructions can be added wherever a particular variable or field is changed, allowing its value to be displayed to a user whenever it is updated.

Optimizations can significantly complicate the implementation of a debugger. The problem is that an optimizer often changes how a computation is performed to speed or simplify it. This means what is actually computed at run-time may not corresponding exactly to what a source program seems to specify. For example, consider the statements

```
a = b + c;
d = a + 1;
a = 0;
```

We might generate

```
iload      1  ; Push b onto stack
iload      2  ; Push c onto stack
```

```
iadd             ; Compute a = b + c
iconst_1         ; Push 1
iadd             ; Compute a + 1
istore       3   ; Store d
iconst_0         ; Push 0
istore       0   ; Store a
```

a is computed in the first assignment statement, used in the second and reassigned in the third. Rather than store a's value and then immediately reload it, the generated code uses the value of a computed in the first assignment without storing it. Instead, the second, final value of a is stored.

The problem is that if we are debugging this code and ask to see a's value immediately after the second statement is executed, we'll not see the effect of the first assignment (which was optimized away). Special care is needed to guarantee that the "expected" value is seen at each point, even if that value is eliminated as unnecessary!

For a more thorough discussion of the problems involved in debugging optimized code see [Adl-Tabatabai and Gross 1996].

Exercises

1. Since many programs contain a sequence of if-then statements, some languages (like Ada) have extended if statements to include an **elsif** clause:

   ```
   if expr1
         stmt1
   elsif expr2
         stmt2
   elsif expr3
         stmt3
   ...
   else  stmtn
   ```

   Each of the expressions is evaluated in turn. As soon as a true expression is reached, the corresponding statements are executed and the if-elsif statement is exited. If no expression evaluates to true, the else statement is executed.

   Suggest an AST structure suitable for representing an if-elsif statement. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

2. Assume that we create a new kind of conditional statement, the **signtest**. Its structure is

   ```
   signtest expression
      neg:  statements
      zero: statements
      pos:  statements
   ```

```
end
```

The integer-valued `expression` is evaluated. If it is negative, the statements following `neg` are executed. If it is zero, the statements following `zero` are executed. If it is positive, the statements following `pos` are executed.

Suggest an AST structure suitable for representing a signtest statement. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

3. Assume we add a new kind of looping statement, the exit-when loop. This loop is of the form

```
loop
      statements1
      exit when expression
      statements2
end
```

First `statements1` are executed. Then `expression` is evaluated. If it is true, the loop is exited. Otherwise, `statements2` followed by `statements1` are executed. Then `expression` is reevaluated, and the loop is conditionally exited. This process repeats until `expression` eventually becomes true (or else the loop iterates forever).

Suggest an AST structure suitable for representing an exit-when loop. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

4. In most cases switch statements generate a jump table. However, when the range of case labels is very sparse, a jump table may be too large. A search table, if sorted, can be searched using a logarithmic time binary search rather than a less efficient linear search

If the number of case labels to be checked is small (less than 8 or so), the binary search can be expanded into a series of nested if statements. For example, given

```
switch(i) {
   case -1000000:  a = 1; break;
   case  0:        a = 3; break;
   case  1000000:  a = 5; break;
}
```

we can generate

```
if (i <= 0)
   if (i == -1000000)
           a = 1;
   else    a = 3;
else       a = 5;
```

Explain why this "expanded" binary search can be more efficient than doing a binary search of a list of case label, assembly label pairs.

How would you change the code generator for switch statements to handle this new translation scheme?

5. Some languages, like Ada, allow switch statements to have cases labeled with a range of values. For example, with ranges, we might have the following in Java or C.

```
switch (j) {
   case 1..10,20,30..35 : option = 1; break;
   case 11,13,15,21..29 : option = 2; break;
   case 14,16,36..50    : option = 3; break;
}
```

   How would you change the code generator of Figure 13.15 to allow case label ranges?

6. Switch and case statements normally require that the control expression be of type integer. Assume we wish to generalize a switch statement to allow control expressions and case labels that are floating point values. For example, we might have

```
switch(sqrt(i)) {
   case 1.4142135: val = 2; break;
   case 1.7320508: val = 3; break;
   case 2:         val = 4; break;
   default:        val = -1;
}
```

   Would you recommend a jump table translation, or a search table translation, or is some new approach required? How would you handle the fact that floating point equality comparisons are "fuzzy" due to roundoff errors. (That is, sometimes what should be an identity relation, like `sqrt(f)*sqrt(f) == f` is *false*.)

7. In Section 13.2.1 we listed seven steps needed to execute a parameterless call. Assume you are generating code for a register-oriented machine like the Mips or Sparc or x86 (your choice). Show the instructions that would be needed to implement each of these seven steps.

   Now look at the code generated for a parameterless call by a C or C++ compiler on the machine you selected. Can you identify each of the seven steps needed to perform the call?

8. Consider the following C function and call

```
char select(int i, char *charArray)
   return charArray[i];
c = select(3,"AbCdEfG");
```

   Look at the code generated for this function and call on your favorite C or C++ compiler. Explain how the generated code passes the integer and character array parameters and how the return value is passed back to the caller.

9. Many register-oriented computer architectures partition the register file(s) into two classes, **caller-save** and **callee-save**. Caller-save registers in use at a call site must be explicitly saved prior to the call and restored after the call (since they may be freely used by the subroutine that is to be called). Callee-save registers that will be used within a subroutine must be saved by the sub-

routine (prior to executing the subroutine's body) and restored by the subroutine (prior to returning to the caller). By allocating caller-save and callee-save registers carefully, fewer registers may need to be saved across a call.

Extend the seven steps of Section 13.2.1 to provide for saving and restoring of both caller-save and callee-save registers.

10. Assume we know that a subroutine is a leaf procedure (i.e., that is contains no calls within its body). If we wish to allocate local variables of the subroutine to registers, is it better to use caller-save or callee-save registers? Why?

11. A number of register-oriented architectures require that parameters be placed in particular registers prior to a call. Thus the first parameter may need to be loaded into register `parm1`, the second parameter into register `parm2`, etc. Obviously, we can simply evaluate parameter expressions and move their values into the required register just prior to beginning a call. An alternative approach involves **register targeting**. That is, prior to evaluating a parameter expression, the desired target register is set and code generators for expressions strive to evaluate the expression value directly into the specified register.

    Explain how the Result mechanism of Chapter 12 can be used to effectively implement targeting of parameter values into designated parameter registers.

12. As we noted in Section 13.2.2, nested calls are easy to translate for the JVM. Parameter values are simply pushed onto the stack, and used when all required parameters are evaluated and pushed.

However, matters become more complex when parameter registers are used. Consider the following call

```
a = p(1,2,p(3,4,5));
```

Note that parameter registers loaded for one call may need to be saved (and later reloaded) if an "inner call" is encountered while evaluating a parameter expression.

What extra steps must be added to the translation of a function call if the call appears within an expression that is used as a parameter value?

13. We have learned that some machines (like the JVM) pass parameters on the run-time stack. Others pass parameters in registers. Still another approach (a very old one) involves placing parameter values in the program text, immediately after the subroutine call instruction. For example `f(1,b)` might be translated into

```
call f
1
value of b
instructions following call of f
```

Now the "return address" passed to the subroutine actually points to the first parameter value, and the actual return point is the return address value + n, where n parameters are passed.

What are the advantages and disadvantages of this style of parameter passing as contrasted with the stack and register-oriented approaches?

14. Assume we have an AST representing a call and that we wish to “inline” the call. How can this AST be rewritten, prior to code generation, so that it represents the body of the subroutine, with actual parameter values properly bound to formal parameters?

15. The Java JVM is unusual in that it provides explicit instructions for throwing and catching exceptions. On most other architectures, these mechanisms must be “synthesized” using conventional instructions.

To throw an exception object `O`, we must do two things. First, we must decide whether the throw statement appears within a try statement prepared to catch `O`. If it is, `O` is passed to the innermost catch block declared to handle `O` (or one of `O`’s superclasses).

If no try block that catches `O` encloses the throw, the exception is propagated—a return is effected and `O` is rethrown at the return point.

Suggest run-time data structures and appropriate code to efficiently implement a throw statement. Note that ordinary calls should not be unduly slowed just to prepare for a throw that may never occur. That is, most (or all) of the cost of a throw should be paid only after the throw is executed.

16. When a run-time error occurs, some programs print an error message (with a line number), followed by a call trace which lists the sequence of pending subroutine calls at the time the error occurred. For example, we might have:

```
Zero divide error in line 12 in procedure
    "WhosOnFirst"
```

```
   Called from line 34 in procedure "WhatsOnSecond"
   Called from line 56 in procedure "IDontKnowsOnThird"
   Called from line 78 in procedure "BudAndLou"
```

Explain what would have to be done to include a call trace (with line numbers) in the code generated by a compiler. Since run-time errors are rare, the solution you adopt should impose little, if any, cost on ordinary calls. All the expense should be borne after the run-time error occurs.