Chapter 10

Code Generation

In this chapter we turn to what is normally the final phase of a compiler, the translation of the intermediate representation into machine code for a specific processor. Like the optimization phase, code generation is made complicated by the wide variation in the way in which processors operate, and the facilities they provide to the compiler writer. In this chapter we will overview some of the more commonly used techniques, using the assembler for a generic machine as our target language. Along the way, we will describe in more detail the task of code generation for a few specific machines.

A major way in which processors differ is in the number and variety of *registers* they make available to the programmer. Almost all machines devote one register to pointing to the top of the activation record stack and one register to holding the current activation frame pointer. Another register holds the program counter, which points to the assembly language instruction currently being executed. Typically function results are returned in yet another register. Since this latter register is obviously not saved across procedure invocations, it is frequently used as a temporary register during calculations when it is not otherwise employed as the function result. Other general purpose registers can be used for computations. These may be or may not be divided into integer and floating point registers. For example, the x86 class machines (286, 386, 486, Pentium and so on), use the name `%esp` to describe the top of stack, `%ebp` to represent the frame pointer, and `%eax, %ebx` and `%ecx` to describe general purpose integer registers. The floating point co-processor unit maintains its own set of floating point registers, separate from the integer registers.

The task of the code generator is made considerably easier if we assume we are producing symbolic assembly language, and that this language will then be processed by further tools, such as an assembler, linker and loader, on the way to generating the final executable file. There are techniques to go directly from the intermediate representation to an object-code representation, but they introduce complications that are tangental to the issues we wish

```
Globals
    GlobalVar
    Procedure

InternalStatement
    BranchStatement
    ConditionalBranchStatement
    AssignmentStatement
    ExpressionStatement
    ReturnStatement
    CompoundStatement

Ast
    FramePointer
    IntegerConstant
    RealConstant
    StringConstant
    UnaryExpression
    BinaryExpression
    FunctionCall

Label
```

Figure 10.1: Class Hierarchy for Intermediate Code

to discuss in this chapter.[1]

The structure of this chapter will mirror the structure of the intermediate representation we have developed in earlier chapters (Figure 10.1). At the highest level, a program is represented by a list of globals, which can either be global variables or procedures. The body of a procedure is a compound statement, which can hold a sequence of other statements. Including the parent class InternalStatement, used to represent a null statement, there are seven different statement types found in our intermediate representation. Many of these must maintain expressions. Expressions are represented by abstract syntax trees, which are also built out of seven different forms of node.

It is important to note that although this intermediate representation is sufficiently complete to permit an entire program to be translated into the internal form before any code generation begins, it need not be used in this fashion. Code can be generated for each

[1] The major difficulty in producing object files directly is the problem of *forward references*. How do you produce the code for a branch to an address that has not yet been seen? Typically a hole is left in the file for the assembly language, and later filled when the adderess is known. Using symbolic labels we can avoid this problem altogether.

global as it is encountered, or even for each statement as it is processed within a function. The appropriate technique will depend upon the quality of code to be generated, and the selection of optimization techniques employed.

10.1 Code Generation for Globals

10.1.1 Code Generation for Global Variables

For unitialized global variables it is only necessary to indicate the name of variable and the amount of storage it will occupy. Oftentimes this can be done by a single directive. On an x86 class machine, for example, this is accomplished by the comm directive:

```
    .comm    name,size
```

Initialized data values are more difficult. The assembler normally provides techniques for initializing primitive data values; integers, reals, characters and the like. More complex structures must then be built out of these. The following, for example, intializes a three element integer array with the values 16, 25 and 32:

```
.globl    name
.data
name:
    .long 16
    .long 25
    .long 32
```

Not all global values in the assembler correspond to global variables in the original source language. As we noted in Chapter **??**, any value with a fixed size and a lifetime that extends over the entire execution of a program can be maintained in global storage. Values that satisfy this property include static fields in classes, and the internal virtual method table that may be part of a class representation. A virtual method table can be considered to be an array of pointers to methods. Such a table might be initialized by the following:

```
.globl    name
.data
name:
    .long 16
    .long 25
    .long 32
```

10.1.2 Code Generation for Functions

Code generation for functions is more complex than for variables, but has many of the same characteristics. In some ways code for a function is nothing more than an initialized global value. Like a global, the function defines a name, and the values that will be stored at the location denoted by the name. Unlike a variable, the values are produced by assembler language instructions.

The code for a function can be thought of as wrapping around the code for the statements that compose the body of the function. Thus, we can divide the description into the function prolog and the function epilog. The function epilog can, in turn, be divided into two sections, an instruction part that holds the commands executed to return from a procedure, and a constant pool that describes those constant values that cannot be represented directly in assembler instructions. (On some machines, the constant pool is not actually physically contiguous to the instruction values, but is actually placed in a different segment).

picture

Function Prolog

The function prolog must perform several activities:

- It must finish the completion of the activation record, by reserving the space that will be used by the bookkeeping information and by the local variables. (Space for the first part, the parameters, will already have been created by the calling procedure).
- It must establish the control link.
- It must save the value of any registers that will be used by the procedure and restored to their original value before return.
- It must create a label that will be used by statements internal to the function to represent the end of the function.
- It must intialize the data structures that will be used to represent the constant pool.

The second of these tasks, saving register values, is difficult if a the statements that compose the body of the procedure have not yet been analyzed, for example if code is being generated statement by statement as each is processsed. In this situation it may not be possible to predict what registers will be necessary for the code to execute propery. Possible solutions are to save all register values (generally far too costly an alternative), or to generate code for statements that make minimal use of registers. These possibilities will be discussed in more detail in Section **??**.

Ignoring the saving of registers, on an x86 a typical function prolog looks something like the following:

```
.globl    name
```

```
    .type name,@function
name:
    pushl    %ebp
    movl    %esp,%ebp
    subl    $12,%esp
```

The first push instruction saves the current frame pointer (held in register %ebp) on to the stack. The move instruction then copies the top of stack pointer into the frame pointer, making the new frame pointer. The final instruction creates space for local variables. Here 12 bytes are being allocated.

The constant pool is initialized by the prolog routine, filled as part of the code generation routines for expressions (see Section ??), and emptied by the epilog routine.

Function Epilog

The function epilog produces the statement that will restore the previous value of the activation frame pointer, then return control to the calling program. Generally this sequence is preceded by a label, which is used as a target for branch instructions generated by return statements. Following the code for procedure return, code is generated for each constant in the constant pool. Each constant is given its own label. The following is an example epilog generated for an x86:

```
.L1:
    leave
    ret
    .align 4
.L2:
    .double 3.43
.L3:
    .string "abc"
```

Here the constant pool consists of two values. A double precision constant has been created and labeled with the name L2, and a string value generated labeled with the name L3.

10.2 Code Generation for Statements

There are seven statement types in our intermediate representation. Each statement can be preceded by an optional label. Null statements produce no further code. An expression statement simply invokes the code generation procedure for expressions. Compound statements loop over their collection of statement values and generate code for each in turn. Each of the others produces code as described in the following sections.

10.2.1 Code Generation for Return Statements

As we noted at the beginning of this chapter, the procedure calling conventions for a typical machine set aside a specific register to hold function results. For such values, a return statement loads the value into the register and generates an unconditional branch to the ending label for the procedure.

source language	*generated assembly*
return x;	load 8(fp),r0
	branch L0

As we saw in section **??**, large values that cannot be held in a register are generally handled in a different fashion. In these cases space for the return value is actually allocated by the caller, and a pointer to this space passed to the procedure as a by-reference parameter. The return statement then assigns values to this parameter, before branching to the ending label.

10.2.2 Code Generation for Assignments

Code generation for assignment statements will depend in part on the approach used to generate code for expressions (see Section **??**). Generating code in a stack like fashion, for example, the value of the right hand side will be pushed on to the stack, then popped off and stored at the appropriate location. The following instruction, for example, stores an integer value into the location 12 bytes from the frame pointer:

```
generate code for right hand side
popl    12(%ebp)
```

When code is generated in a register-like fashion, the right hand side will end up being held in a register. This register value is then stored at the appropriate location.

10.3 Code Generation for Branches

An unconditional branch is generally implemented directly by a single assembly language instruction, as in the following x86 instruction:

```
jmp     .L23
```

On some machines jump instructions may be complicated by different instructions used to branch to near locations (for example, within 128 bytes of the current instruction) than for far instructions. However, even on these machines the proper choice of the appropriate

instruction will often be determined by the assembler, and need not concern the writer of the code genrator.

Expressions are evaluated in programming languages for two major purposes. We have seen one of these already in the assignment statement. In an assignment statement, an expression is evaluated for the purpose of generating a result. This result is left (depending upon the style of code generation being used) either on the top of the stack, or in a register. A conditional branch, on the other hand, evaluates expressions for a different purpose. Here the objective is to produce code that will branch to a specific location should the result of the expression satisfy a certain property.

Code generation for conditional branches can be simplified by providing two complementary code generation routines in the abstract syntax tree class. These two routines are named genBranchIfTrue and genBranchIfFalse. The first will generate code that will, when executed, branch to the given label if the expression evaluates as true. The second will generate code, but will branch to the label if the condition is false. How these two routines are used in the generation of code is described in the following sections:

Conditional branch instructions are more complex. In this case the instruction is build out of two elements, an AST that represents an expression to evaluate, and a label to which control should pass if the expression evaluates as true.

10.4 Code Generation for Expressions

Expressions in computer programs are examined for two different reasons. The most common reason to consider an expression is to produce a *value*, that is, the result of the evaluation of the expression. This is the objective in using an expression in an assignment statement, such as the following:

```
x := a * b + c * d;
```

Here the value of the right hand side will be computed, and then assigned to the location described by the left hand side of the assignment. Arguments being passed to functions are similarly processed in order to generate a value. However, there is another reason that expressions are evaluated. This is to effect a control flow transfer. In the execution of the statement:

```
if (a < b) and (c < d) then ...
```

we aren't really interested in the result of the test expression, indeed it may not ever be computed as a value. Instead, we simply want to produce code that will transfer control if the condition is false.

These different objectives for code generation are reflected in different methods in class Ast that are invoked to drive the code generation process. The method genEvaluate() will

be invoked to produce code that, when executed, will evaluate an expression. The method genBranchIfTrue(Label), on the other hand, is used to generate code that, when executed, produces a transfer of control to a given label if the expression is true. As we will see when we examine the control transfer operators, the generation of code will be simplified if we add another method, genBranchIfFalse(Label). When invoked, this method generates code that will branch if the argument expression is false.

All operators can be divided into those the produce a value, and those that are used only for a transfer of control. The transfer of control operators are unary negation, the logical connectives *and* and *or*, and the comparison operators. All other operators are value producing. Each will be discussed in turn in the following sections.

10.4.1 Code Generation for Value Producing Operators

We consider first the most common case, which is the generation of code that will have the effect of evaluating an expression. Code generation will be produced as a side effect of passing the message genEvaluate to an instance of class Ast. Each of the various subclasses of Ast will implement this method, and emit code appropriate to the operation being defined.

There are two major approches to handing the code generation of expressions that yield a value. The difference between these two approaches is how they store the intermediate values that arise during the course of execution, while the remainder of a computation is performed. Consider, for example, the expression:

```
a * b + c * d
```

We might begin by evaluating the expression `a * b`. The plus operation cannot be performed until the second multiplication is executed. Therefore, this multiplication should be performed next. But while we are performing the second multiplication, the results of the first evaluation cannot be forgotten, although they have no part in the evaluation.

The two approaches to code generation can be described as stack-style and as register style. The relative advantages and disadvantages of these two approaches are contrasted in Figure 10.2.

10.4.2 Stack Style Code Generation

In the stack style technique, the activation record is used as an operand stack for holding intermediate values. In the expression above, code would be generated that would have the effect, when executed, of pushing the value `a * b` on to the stack. Next, code is generated that will push the value `c * d` on to the stack. Finally, code is generated that adds the top two items on the stack, leaving the result once more on the stack. A typical instruction sequence in this style for the above expression might look as follows:

```
push 8(fp)     ; push variable a
```

	stack style	**register style**
pro	simple to do registers need not be saved and restored on procedure entry/exit	code is often both shorter and faster
con	code is often long and slow	difficult to do well registers must be saved/restored on procedure entry and exit

Figure 10.2: Comparison and Stack and Register Style Code Generation

```
push 12(fp)    ; push variable b
mult        ; multiply top two values
push 16(fp)    ; push variable c
push 20(fp)    ; push variable d
mult        ; multiply top two values
add        ; add top two values
```

The result is left on the top of the activation record stack.

Register style code generation, on the other hand, uses the machine registers for intermediate results. A typical instruction sequence for this expression might be as follows:

```
move 8(fp),r1    ; move variable a into register 1
move 12(fp),r2    ; move variable b into register 2
mult r2,r1    ; multiply r2 and r1, result in r1
move 16(fp),r2    ; move variable c into register 2
move 20(fp),r3    ; move variable d into register 3
mult r3,r2    ; multiply
add  r2,r1    ; add
```

The result is left in register 1.

The advantages and disadvantages of the two approaches largely mirror each other. The advantage of the stack approach is simplicity. Code generation is performed by a simple traversal of the abstract syntax tree, and for each operator there is only one code sequence to emit. Furthermore, since registers are not used, they need not be saved and restored across the procedure call boundary.

On the other hand, there are several disadvantages to stack style code generation. Many machines do not have instructions that correspond to commands such as "add the top two elements on the stack". Instead, several instructions must be emitted for this purpose. Figure **??** shows the code sequences that might be produced for execution on an Intel x86.

This can make code generated in the stack style much longer than the corresponding register code. Even when no longer, the stack style code can still be much slower than the register style code. This is because the activation record stack is stored in memory, and thus the execution of almost every instruction must make multiple accesses to memory. Registers are typically stored in a fashion that makes their access much faster than accesses to memory. Instructions that need not reference memory, therefore, will often execute much more quickly.

10.4.3 Register style code generation

Many coding conventions and machine instructions naturally leave their value in a register. As we noted in our earlier discussion of the return statement, a function call will normally leave a returned result in the scratch register. Similarly, on many machines an instruction such as multiply or divide will normally leave a result in a register, rather than on the stack. We can use these facts to produce code that is both much shorter and much faster. For example.

example

However, register style code generation has its own difficulties.

Register Save and Restore

We have already noted, in our discussion of the procedure prologue and epilogue, that every procedure wants to maintain the illusion that it can use its own set of registers with impunity. This effect is typically achived by storing the value of registers at the beginning of a procedure, and restoring those values at the end, immediately prior to returning control to the calling procedure.

This means that either all registers must be saved and restored (generally an overly costly decision), or the exact number of registers required must be determined before the procedure prologue can be written.[2]

Because of the need to save and restore registers, a procedure that uses many registers may be faster in execution, but spend more time in the procedure prologue and epilogue. Balancing these two activities against each other my be difficult.

The Analysis of Expressions

The generation of code using the stack approach is simple, because for each operator there is only one code sequence to emit. An add instruction, for example, pushes the value of the left argument on to the stack, then pushes the value of the right argument, then adds the

[2] Actually, a middle ground is possible, and is sometimes observed. This is achived by having the procedure prolog generate, as its first instruction, a branch to the end of the procedure code. The code at the end, which is produced once all statements have been seen, saves the appropriate registers, then returns to the beginning of the code section.

two values together. Code generation for register instructions is more difficult, particularly if one wishes to make effective use of features such as addressing modes. Consider the add instruction, for example. One might want to consider at least the following four separate cases:

1. Both arguments are of a form that can be loaded as an addressing mode. The left argument is placed into a register, and the right is added to it. The expression x+3, for example, might produce the following:

```
move    8(fp), r1
add    3, r1
```

2. The right argument can be loaded as an addressing mode, but the left argument is more complex. Code generation on the left argument will result in a sequence of instructions that leave the result in a register. The add instruction can use this register. The expression x*y+3, for example, might produce the following:

```
...             ; load x*y into register r1
add   3, r1
```

3. The inverse of case 2. Here the left argument can be loaded as an addressing mode, but the right argument is more complex. In this case it might be better to first perform code generation on the right argument, which will produce instructions that leave the result in a register. This register can then be used by the add instruction. The expression 3+x*y, for example, might produce the following:

```
...             ; load x*y into register r1
add 3, r1
```

   Note that this only works because addition is communtative. If we were generating code for subtraction or division, the answer would be the inverse of the result we desire. For these instructions either this case is ignored, or additional instructions are produced to return the result to the form we require.

4. The final case occurs when both values are sufficiently complex that neither can be loaded directly using an addressing mode. Here two register values are required. The left argument is evaluated and stored in one register. Following this, the right argument is evaluated and stored in a second register. The final instruction adds the two register contents, and frees one of the registers for subsequent use. The expression a*b+c*d, for example, might produce the following:

```
... // code to load a * b into register 1
... // code to load c * d into register 2
add     r2, r1 // r2 is now free
```

Non-orthogonal Instruction Sets

The complexity of the code generation for binary operators might be mitigated somewhat if all binary operators could be treated uniformly. But unfortunately, real machines are seldom this simple. The addressing modes that can be used with an add instruction, for example, are often different from those that can be used with a left shift. We have previously noted that communtative operations, such as addition, may permit optimizations that non-communtative operations, such as subtraction, do not. Thus, each binary operator must be handled as a special case.

Register Spills

Another problem occurs with complex expressions. While the activation record stack can be considered to be essentially unbounded, the number of registers is often severly limited. What happens if it is necessary to evaluate an expression that has more intermediate results than the number of free registers available? This is called a register *spill*. To handle a register spill, the value being held in a register must be saved (*spilled*) at a known location. The register is then used as a temporary in the evaluation of a further portion of the expression. Once used, the spilled value must be restored before its value is required.

example

Registers for Variables

We have so far been discussing the use of registers only as holders for intermediate values during the evaluation of expressions. But another use of registers is to maintain a variable value, if the variable is being used repeatedly in a small part of the program, such as a loop index variable. By maintaining the variable in a register, it is not necessary to continually load the value from memory, or store it back to memory once it has been assigned. Consider, the example, the following simple loop:

expression

Simple register style code for this statement might look as follows:

code

If we maintain the value of the variable i in a register r1, rather than in memory, we can improve the code as follows:

code

This code has approximately half the number of memory accesses as the original.

Now we have two competing uses for registers; as intermediate values and as holders for variables. Which use is more important? For any given register, would it be better used to

hold a variable, or to hold an expression? Each program must be evaluated on a case by case bases, and the trade-offs are not always clear.

Registers for Parameters

Generalizing the idea of using registers for variables, there has been an increasing trend in recent years to use registers for at least some parameter values. The first four general purpose registers, for example, might be allocated to the task of parameter passing. Rather than pushing the actual argument values on the stack, these will instead be assigned to the appropriate registers.

Should the called procedure not need the register for any other purpose, the value can remain in the register and be quickly accessed. If, on the other hand, the called procedure requires the use of the register (say, it must invoke another procedure), then the value will be saved in memory as part of the normal register save operation.

Note that passing registers in parameters complicates many parts of the compiler. Consider the question of how parameter values should be accessed (or addressed). If the corresponding register is not required by the current procedure, then the value can simply be used directly from the register. But if the register must be reused for a different parameter, then the original value will be saved as part of the register save operation, and the value must be retrieved from that memory location. But the compiler may not know whether or not a register is required until all the statements in a procedure have been examined.

Imagine the simple case where there is only one register for passing an integer parameter, and consider the following procedure.

```
function paramTest (i : integer);
begin
    while i < 27 begin
        i := i + 1;
        ...
        foo(i+7);
    end
end
```

p to the point where the call on foo is encountered the compiler could assume that the value of i could be continued to be held in a register. However, the value of the register must be overridden in order to pass the appropriate value to foo, which will expect the argument to also be transmitted by means of a register. Therefore the value of i must be saved as part of the function prolog, and within the procedure the value used as if it is a local variable. This decision, in turn, changes the type of code that will be produced for the while test expression, which will be code that appears before the call on the function which necessitated the change.

Summary–register style code generation

In summary, the difficulty with the register style code generation is that there is a single resource (general purpose registers) and a variety of uses (intermediate results, frequently accessed variables, and parameter values). Deciding how register values should be allocated for these different purposes is a nontrivial task, too complex to discuss here. Interested students are referred to references cited at the end of the chapter.

10.4.4 Code Generation for Control Flow Operators

We now turn to the question of generating code for those operators that are evaluated not for their result, but for their influence on the flow of control. You will recall that there are three classes of such operators. These are the unary negation operator, the logical connectives *and* and *or*, and the comparison operators. In addition, the conditional result operator involves aspects of control flow, although it produces a value.

Even when control flow operators appear at the source level to be producing an expression, a simple internal transformation can be used to ensure this is not the case. If the source language permits values of type boolean, a programmer might write an expression such as the following:

```
flag := a < b;
```

It is a simple matter to translate this internally as the following:

```
flag := (a < b) ? true : false;
```

In this fashion the value is produced by the conditional expression operator, a value producing operator, and not by the comparison operator, a control-flow operator. Similarly, when boolean variables are used in a test expression, it would appear that a value-producing expression is being used to perform control flow. However, it is a simple matter to internally convert:

```
while (flag) ...
```

into

```
while (flag == true) ...
```

The control flow is therefore produced by the comparison operator, and not by the original expression itself.

As we noted earlier, the task of code generation is performed by the nodes in an abstract syntax tree, an instance of class AST. Two methods in class AST are used to provide code

generation for control flow. The method genBranchIfTrue(Label) takes as argument a label and will generate code that, when executed, will have the effect of transferring control to the label if the associated condition evaluates to true. Otherwise, execution continues with the next statement in sequence. The method genBranchIfFalse(Label) is similar, but generates code that will branch if the associated condition is false. Only one of these methods is ever invoked for any single node.

In the following sections we will describe the code produced by each of the control flow changing operators.

Code Generation for Not operator

Code generation for the unary logical not operator is particularly trivial in this framework. The branchIfTrue method for a not operator simply invokes branchIfFalse on its child tree, and vice versa.

Suppose, for example, that the branchIfTrue method is invoked for the following expression, passing as argument the label L4.

picture

The not operator will simply invoke the branchIfFalse method on its child node, using the same label it was given:

picture

By this means, no code is ever generated for the not operator!

Code Generation for Logical Operators

The logical connectives and and or have a subtle difference from their mathematical or logical counterparts. In conventional logic an and of two terms is true if both values are true, and false otherwise. The and operator in computer languages, on the other hand, is generally evaluated in a *short-circuit* manner. This means if the truth or falsity of an expression can be determined by the left operand, the right operand is not even considered. Thus an expression can be true or false, even if the right operand is not well defined.

A common example of this is a relational test used to "guard" an array element access. Assume, for example, that x is an array indexed 0 to 9. The following expression has a clear boolean value, even if the index value held in variable I is larger than 9.

```
(I < 10) and (a[i] <> 32)
```

Consider first the *or* operator, and code generated by the branchIfTrue method. Let lab be the label passed as argument to this method. An *or* is true if either argument is true; thus, this operator simply creates code for each child expression in turn:

```
left.genBranchIfTrue(lab);
right.genBranchIfTrue(lab);
```

If the code generated by the left argument evaluates to true, control will be passed to the given label. If the left argument is false, control will fall through to the following statement, which in this case evaluates the right argument. If the right argument is true, then control is passed to the label. If both left and right expressions are false, control falls all the way through this code, and continues with whatever statement follows next.

Branching if the condition is false is more difficult. An or expression is false only if both child expressions are false. If the left child expression is true there is no way the entire expression can be false, and the right child expression should not even be considered. To do this, a new internal label is created, and the code branches around the right part:

```
Label L2 = new Label();
left.genBranchIfTrue(L2);
right.genBranchIfFalse(lab);
L2.placeLabel();
```

The following, for example, shows the complete assembly language that would be generated to evaluate the expression $(a < b) or (c < d)$, assuming a, b, c and d were global variables, and branch to label L4 if the expression is false. The label L5 is the new label being generated internally to the expression.

```
    push a
    push b
    compare
    branchGT L5
    push c
    push d
    compare
    branchGE L4
L5:
```

The *and* operator is symmetric. If code to branch if the condition is false is desired, the falsity of either child is sufficient to determine the outcome. On the other hand, if code to branch if the condition is true is desired, a new internal label is created, and the right clause is ignored if the left term is false.

Code Generation for Comparison Operators

As we saw in the example just shown, comparison operators emit code that will evaluate their left and right arguments, pushing the results on to the stack or loading them into a register, then emit a comparison instruction. This is followed by a conditional branch instruction,

which will branch if the appropriate condition is satisfied. The following illustrates the type of code that could be generated on a x86 for such a comparison:

```
move    12(fp), r1
move    16(fp), r2
comp    r1, r2
jle    L7
```

The code loads the variable stored at offset 12 from the frame pointer into register 1, the variable at offset 16 into register 2, then compares registers 1 and 2. The final statement will perform a jump to label L7 if the the first quantity was less than or equal to the second.

Inverting the condition, that is, code generated by genBranchIfFalse, can be performed by simply changing the opcode on the final branch statement. Evaluating whether the test of one quantity is less than or equal to another is false is the same as saying that the first is greater than the second. In this way, each relational has a natural inverse, as shown in the following table:

relation	lt	le	eq	ne	ge	gt
inverse	ge	gt	ne	eq	lt	le

The code generated by genBranchIfFalse is similar to the code generated by genBranchIfTrue, differing only in the opcode used by the branch instruction.

10.4.5 The Condtional Result Operator

The conditional result operator is a three valued function that selects one of two alternatives, depending upon the outcome of the test. Code generated for this function is very similar to code generated for the if statement. To generate code, a pair of new internal labels are created. Code is then produced for the test condition, using the branchIfFalse method. Code is then generated for each of the child expressions, separeted by a branch around the second expression. Depending upon the expression evaluation technique being used, the result is either left sitting on the top of the stack, or in a register. In the latter case, care must be exercised to ensure that the result is left in the same register regardless of which expression yields the result.

In this fashion, an expression which we represented as

```
testExpression ? trueExpression : falseExpression
```

would execute the following sequence of instructions in the process of producing code:

```
L1 = new Label();
L2 = new Label();
```

```
test.genBranchIfTrue(L1);
trueExpression.genEvaluate();
L2.genBranchInstruction();
L1.genLabel();
falseExpression.genEvaluate();
L2.genLabel();
```

Further Reading

Exercises

1. Stuff