12
Simple Code Generation

In this chapter we will begin to explore the final phase of compilation—code generation. At this point the compiler has already created an AST and type-checked it, so we know we have a valid source program. Now the compiler will generate machine-level instructions and data that correctly implement the semantics of the AST.

Up to now, our compiler has been completely target-machine independent. That is, nothing in the design of our scanner, parser and type-checker depends on exactly which computer we will be using. Code generation, of course, *does* depend on target-machine details. Here we'll need to know the range of instructions and addressing modes available, the format of data types, the range of operating system services that are provided, etc. Using this knowledge, the code generator will

translate each AST into an **executable form** appropriate for a particular target machine.

12.1 Assembly Language and Binary Formats

We'll first need to choose the exact form that we'll use for the code that we generate. One form, commonly used in student projects and experimental compilers, is **assembly language**. We simply generate a text file containing assembler instructions and directives. This file is then assembled, linked and loaded to produce an executable program.

Generating assembly language frees us from a lot of low-level machine-specific details. We need not worry about the exact bit patterns used to represent integers, floats and instructions. We can use symbolic labels, so we need not worry about tracking the exact numeric values of addresses. Assembly language is easy to read, simplifying the debugging of our code generator.

The disadvantage of generating assembly language is that an additional assembly phase is needed to put our program into a form suitable for execution. Assemblers can be rather slow and cumbersome. Hence, rather than generating assembly language, many production quality compilers (like the Java, C and C++ compilers you use) generate code in a **binary format** that is very close to that expected by the target machine.

To support independent compilation of program components, compilers usually produce a **relocatable object file** (a ".o" or ".class" file) that can be linked with other object files and library routines to produce an executable program.

Each instruction or data value is translated into its binary format. For integers this is typically the two's complement representation. For floats, the IEEE floating point standard is widely used. The format of instructions is highly machine-specific. Typically, the leftmost byte represents the operation, with remaining bytes specifying registers, offsets, immediate values or addresses.

Each instruction or data value is placed at a fixed offset within a **section**. When object files are linked together, all members of a section are assigned adjacent memory addresses in the order of their offsets. A section might represent the translated body of a subroutine, or the global variables of a ".c" file.

When references to the address of an instruction or data value are found, they are of the form "section plus offset." Since we don't yet know exactly where in memory a section will be placed, these addresses are marked as **relocatable**. This means that when the linker is run, these address will be *automatically* adjusted to reflect the address assigned to each section.

An object file may reference labels defined in other object files (e.g., `printf`). These are **external references**. They are marked for relocation by the value of the external label, when the linker determines the actual value of the label. Similarly, labels of subroutines and exported global variables may be marked as **external definitions**. Their values may be referenced by other object files.

While this object file mechanism may seem complicated (and in many ways *it is*), it does allow us to build programs from separately compiled pieces, including standard library routines. Not all the object files need be created by the same compiler; they need not even be programed in the same source language. The linker

will relocate addresses and resolve cross-file references. Some compilers, like those for ML and Java, perform an **inter-module dependency analysis** on separately compiled modules to check interface and type consistency.

In Java, execution begins with a single class file. Its fields are initialized and its methods are loaded. In the interests of security, methods may be **verified** to guarantee that they do not violate type security or perform illegal operations. Other classes that are accessed may be **dynamically linked** and loaded as they are referenced, thereby building a program class by class.

In some cases all we want to do is translate and execute a program immediately. The generation and linkage of object files would be unnecessarily complex. An alternative is to generate **absolute machine code**. This comprises machine-level instructions and data with all addresses fully resolved. As instructions and data are generated, they are given fixed and completely resolved addresses, typically within a large block of memory preallocated by the compiler. If an address's value isn't known yet (because it represents an instruction or data value not yet generated), the address is **backpatched**. That is, the location where the address is needed is stored, and the address is "patched in" once it becomes known. For example, if we translate `goto L` before `L`'s address is known, we provisionally generate a jump instruction to location 0 (`jmp 0`) with the understanding that address 0 will be corrected later. When `L` is encountereed, its address becomes known (e.g., 1000) and `jmp 0` is updated to `jmp 1000`.

Library routines are loaded (in fully translated form) as necessary. After all instructions and data are translated into binary form and loaded into memory,

execution begins. Files like `stdin` and `stdout` are opened, and control is passed to the entry point address (usually the first instruction of `main`).

12.2 Translating ASTs

Intermediate representations of programs, like ASTs, are oriented toward the source language being translated. They reflect the source constructs of a language, like loops and arrays and procedures, without specifically stating how these constructs will be translated into executable form. The first step of code generation is therefore **translation**, in which the meaning of a construct is reduced to a number of simple, easily implemented steps. Thus a binary expression that applies an operator to left and right operands (e.g., `a[i]+b[j]`) might be translated by the rule:

> Translate the left operand, then translate the right operand,
>
> then translate the operator

After a construct is translated, it becomes necessary to choose the particular machine language instructions that will be used to implement the construct. This is called **instruction selection**. Instruction selection is highly target-machine dependent. The same construct, translated the same way, may be implemented very differently on different computers. Even on the same machine, we often have a choice of possible implementations. For example, to add one to a value, we might load the literal one into a register and do a register to register add. Alternatively, we might do an add immediate or increment instruction, avoiding an unnecessary load of the constant operand.

12.3 Code Generation for ASTs

Let us first consider the steps needed to design and implement a code generator for an AST node. Consider the **conditional expression**, `if exp1 then exp2 else exp3`. (Conditional expressions are represented in Java, C and C++ in the form `exp1?exp2:exp3`).

We first consider the **semantics** of the expression—what does it *mean*? For conditional expressions, we must evaluate the first expression. If its value is non-zero, the second expression is evaluated and returned as the result of the conditional expression. Otherwise, the third expression is evaluated and returned as the value of the conditional expression.

Since our code generator will be translating an AST, the exact syntax of the conditional expression is no longer visible. The code generator sees an AST like the one shown in Figure 12.1. Rectangles represent single AST nodes; triangles represent AST subtrees. The type of the node is listed first, followed by the fields it contains.

The type checker has already been run, so we know the types are all correct. Type rules of the source language may require conversion of some operands. In such cases explicit conversion operators are inserted as needed (e.g., `a<b?1:1.5` becomes `a<b?((float)1):1.5`).

We first outline the translation we want. A portion of the translation will be done at ConditionalExprNode; the remainder will be done by *recursively visiting*

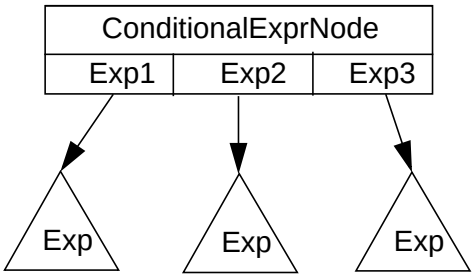

Figure 12.1 Abstract Syntax Tree for a Conditional Expression

its subtrees. Each node's translation will mesh together to form the complete translation.

In choosing a translation it is vital that we fully understand the semantics of the construct. We can study a language reference manual, or consult with a language designer, or examine the translations produced by some reference compiler.

The code we choose will operate as follows:

1. Evaluate Exp1.
2. If Exp1 is zero go to 5.
3. Evaluate ans ← Exp2.
4. Go to 6.
5. Evaluate ans ← Exp3.
6. Use the value computed in ans as the result of the conditional expression.

We have not yet chosen the exact instructions we will generate—only the specific steps to be performed by the conditional expression. Note that alternative translations may be possible, but we must be sure that any translation we choose

is *correct*. Thus evaluating *both* Exp2 and Exp3 would not only be inefficient, it would be *wrong*. (Consider `if a==0 then 1 else 1/a`).

Next we choose the instructions we wish to generate. A given translation may be implemented by various instruction sequences. We want the fastest and most compact instruction sequence possible. Thus we won't use three instructions when two will do, and won't use slower instructions (like a floating-point multiply) when a faster instruction (like a floating-point add) might be used instead.

At this stage it is important that we be very familiar with the architecture of the machine for which we are generating code. As we design and choose the code we will use to implement a particular construct, it is a good idea actually to try the instruction sequences that have been chosen to verify that they will work properly. Once we're sure that the code we've chosen is viable, we can include it in our code generator.

In the examples in this chapter we'll use the Java Virtual Machine (JVM) instruction set [Lindholm and Yellin 1997]. This architecture is clean and easy to use and yet is implemented on virtually all current architectures. We'll also use Jasmin [Meyer and Downing 1997], a symbolic assembler for the JVM.

In Chapter 15 we extend our discussion of code generation to include issues of instruction selection, register allocation and code scheduling for mainstream architectures like the MIPS, Sparc and PowerPC. Thus the JVM instructions we generate in this chapter may be viewed as a convenient platform-independent intermediate form that can be expaned, if necessary, into a particular target machine form.

Before we can translate a conditional expression, we must know how to translate the variables, constants and expressions that may appear within it. Thus we will first study how to translate declarations and uses of simple scalar variables and constants. Then we'll consider expressions, including conditional expressions. Finally we'll look at assignment operations. More complex data structures and statements will be discussed in succeeding chapters.

12.4 Declaration of Scalar Variables and Constants

We will begin our discussion of translation with the declaration of simple scalar variables. Variables may be global or local. Global variables are those declared outside any subprogram or method as well as static variables and fields. Non-static variables declared within a subprogram or method are considered local. We'll consider globals first.

A variable declaration consists of an identifier, its type, and an optional constant-valued initialization, as shown in Figure 12.2.

An IdentifierNode will contain an Address field (set by the type-checker) that references an AddressNode. This field records how the value the identifier represents is to be accessed. Identifiers may access the values they denote in many ways: as global values (statically allocated), as local values (allocated within a stack frame), as register values (held only in a register), as stack values (stored on the run-time stack), as literal values, or as indirect values (a pointer to the actual value). The AccessMode field in AddressNode records how an identifier (or any other data object) is to be accessed at run-time.

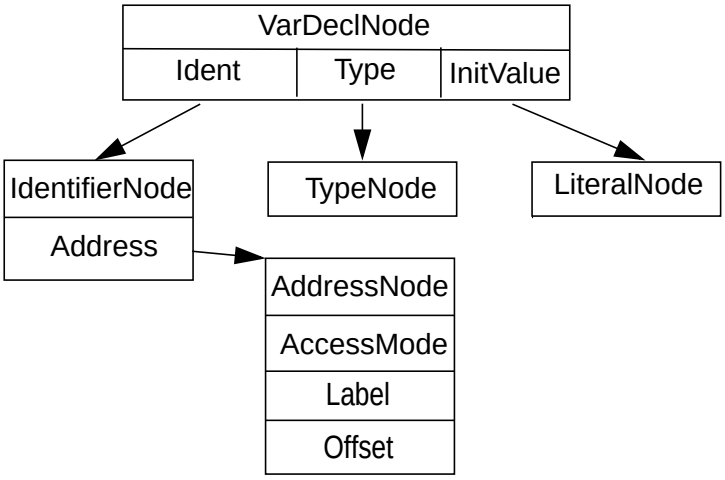

Figure 12.2 Abstract Syntax Tree for Variable Declarations

Within a program, all uses of a given identifier will share the same IdentifierNode, so all information added to an AST node during declaration processing will be visible when uses of the identifier are processed.

To generate code, we'll use a variety of "code generation" subroutines, each prefixed with "Gen." A code generation subroutine generates the machine level instructions needed to implement a particular elementary operation (like loading a value from memory or conditionally branching to a label). With care, almost all machine-specific operations can be compartmentalized within these subroutines.

In our examples, we'll assume that code generation subroutines are generating JVM instructions. However, it is a simple matter to recode these subroutines to generate code for other computers, like the Sparc, or x86 or PowerPC. Our discussion is in no sense limited to just Java and the JVM.

We'll also assume that our code generation subroutines generate assembly language. Each subroutine will append one or more lines into an output file. When code generation is complete, the file will contain a complete assembly language program, ready for processing using Jasmin or some other assembler.

In translating variable declarations, we will use the code generation subroutine GenGlobalDecl(Label, Type, InitialVal). Label is a unique label for the declaration. It may be generated using the subroutine CreateUniqueLabel() that creates a unique label each time it is called (e.g., `Lab001`, `Lab002`, etc.). Alternatively, the label may be derived from the identifier's name. Be careful however; many assemblers treat certain names as reserved (for example, the names of op codes). To avoid an unintentional clash, you may need to suffix identifier names with a special symbol (such as $).

Type is used to determine the amount of memory to reserve. Thus an `int` will reserve a word (4 bytes), a `char` a single byte or halfword, etc.

InitialVal is the literal value the global location will be initialized to. It is obtained from the LiteralValue field of the LiteralNode. If no explicit initial value is provided, InitialVal is null and a default initial value (or none at all) is used.

For example, if we are generating Jasmin JVM instructions, GenGlobalDecl(Label, Type, InitialVal) might generate

```
.field   public static Label TypeCode = Init
```

In the JVM globals are implemented as static fields. *`TypeCode`* is a type code derived from the value of Type. *`Init`* is the initial value specified by InitialVal. *`Label`* is an assembly language label specified by Label.

Recall that variables local to a subprogram or method are considered to be part of a stack frame that is pushed whenever the procedure is called (see section 11.2). This means we can't generate individual labels for local variables—they don't have a fixed address. Rather, when a local declaration is processed during code generation the variable is assigned a fixed offset within the frame. If necessary, offsets are adjusted to guarantee that data is properly aligned. Thus a local integer variable will always have an offset that is word aligned, even if it immediately follows a character variable.

The size of the frame is increased to make room for the newly declared variable. After all declarations are processed, the final size of the frame is saved as part of the routine's type information (this size will be needed when the routine is called). A local variable that is initialized is handled as an uninitialized declaration followed by an assignment statement. That is, explicit run-time initialization instructions are needed each time a procedure is called and its frame is pushed. That is why C and C++ do not automatically initialize local variables.

In the JVM, local variables are easy to implement. All local variables, including parameters, are given a **variable index** in the current frame. This index, in units of words, is incremented by one for all types that require one word or less, and is incremented by two for data that require a doubleword.

We'll use the subroutine GenLocalDecl(Type) to process a local declaration. This routine will return the offset (or variable index) assigned to the local variable (based on its Type).

Each AST node will contain a member function CodeGen() that generates the code corresponding to it. This routine will generate assembly language instructions by calling code generation subroutines. The code generator for a VarDeclNode is shown in Figure 12.3. (Implmentation of assignments is discussed in section Section 12.9).

```
VarDeclNode.CodeGen( )
1.    if Ident.Address.AccessMode = Global
2.          then Ident.Address.Label ← CreateUniqueLabel()
3.               GenGlobalDecl(Ident.Address.Label, Type, InitValue)
4.          else Ident.Address.Offset ← GenLocalDecl(Type)
               if InitValue ≠ null
5.                  then /* Generate code to store InitValue in Ident */
```

Figure 12.3 Code Generation for Variable Declarations

Note that either Label or Offset in AddressNode is set during code generation for variable declarations. These fields are used when uses of the identifier are translated.

As mentioned earlier, one of the problems in using global (32 or 64 bit) addresses is that they often "don't fit" within a single instruction. This means that referencing a global variable by its label may generate several instructions rather than one. To improve efficiency, many compilers access globals using an offset relative to a **global pointer** that points to the global data area (see Section 11.1).

This approach is easy to accommodate. As globals are declared their Offset field is set along with their Label field. The Offset can be used to load or store globals in one instruction. The Label field can be used to access global variables in

other compilation units or if the global data area is so large that not all variable addresses can be represented as an offset relative to a single register.

12.4.1 Handling Constant Declarations and Literals

Constant declarations can be handled almost exactly the same as initialized variables. That is, we allocate space for the constant and generate code to initialize it. Since constants may not be changed (the type-checker enforces this), the location allocated to the constant may be referenced whenever the constant's value is needed.

A few optimizations are commonly applied. For named constants whose value fits within the range of immediate operands, we need not allocate a memory location. Rather, we store the explicit numeric value of the constant and "fill it in" as an immediate operand.

Locally declared constants may be treated like static variables and allocated a global location. This eliminates the need to reinitialize the constant each time the routine that contains it is called.

Small integer literals are accessed when possible as immediate operands. Larger integer literals, as well as floating literals, are handled liked named constants. They are allocated a global data location and the value of the literal is fetched from its memory location. Globally allocated literal values may be

accessed through the global pointer if their addresses are too large to fit in a single instruction.

Some compilers keep a list of literals that have been placed in memory so that all occurrences of the same literal share the same memory location. Other compilers place literals in memory each time they are encountered, simplifying compilation a bit at the expense of possibly wasting a few words of memory.

12.5 Translating Simple Expressions

We now consider the translation of simple expressions, involving operands that are simple scalar variables, constants or literals. Our solution generalizes nicely to expressions involving components of arrays or fields in classes or structures, or the results of function calls.

Many architectures require that an operand be loaded into a register before it can be moved or manipulated. The JVM requires that an operand be placed on the run-time stack before it is manipulated.

We will therefore expect that calling CodeGen in an AST that represents an expression will have the effect of generating code to compute the value of that expression into a register or onto the stack. This may involve simply loading a variable from memory, or it may involve a complex computation implemented using many instructions. In any event, we'll expect to have an expression's value in a register or on the stack after it is translated.

12.5.1 Temporary Management

Values computed in one part of an expression must be communicated to other parts of the expression. For example, in `exp1+exp2`, `exp1` must be computed and then held while `exp2` is computed. Partial or intermediate values computed within an expression or statement are called **temporary values** (because they are needed only temporarily). Temporaries are normally kept on the stack or in registers, though storage temporaries are sometimes used.

To allocate and free temporaries, we'll create two code generation subroutines, GetTemp(Type) and FreeTemp(Address1, Adress2, ...). GetTemp, which will return an AddressNode, will allocate a temporary location that can hold an object of the specified type.

On a register-oriented machine, GetTemp will normally return an integer or floating point register. It will choose the register from a pool of registers reserved to hold operands (see Section 15.3.1).

On a stack-oriented machine like the JVM, GetTemp will do almost nothing. This is because results are almost always computed directly onto the top of the stack. Therefore GetTemp will simply indicate in the address it returns that the stack is to be used.

For more complex objects like arrays or strings, GetTemp may allocate memory locations (in the current frame or heap) and return a pointer to the locations allocated.

FreeTemp, called when temporary locations are no longer needed, frees the locations at the specified addresses. Registers are returned to the pool of available registers so that they may be reused in future calls to GetTemp.

For stack-allocated temporaries a pop may need to be generated. For the JVM, operations almost always pop operands after they are used, so FreeTemp need not pop the stack. Temporaries allocated in memory may be released, and their memory locations may be later reused as necessary.

Now we know how to allocate temporaries, but we still need a way of communicating temporary information between AST nodes. We will add a field called Result to AST nodes. Result contains an AddressNode that describes the temporary that will contain the value the AST node will compute. If the AST computes no value into a temporary, the value of Result is ignored.

Who sets the value of Result? There are two possibilities. A parent can set the value, in effect directing a node to compute its value into the location its parent has chosen. Alternatively, a node can set the value itself, informing its parent of the temporary it has chosen to compute its value into.

In most cases the parent doesn't care what temporary is used. It will set Result to null to indicate that it will accept any temporary. However sometimes it is useful to force a value into a particular temporary, particularly when registers are used (this is called **register targeting**).

Reconsider our conditional expression example. If we let each node choose its own result register, `exp2` and `exp3` *won't* use the same register. And yet in this case, that's exactly what we want, since `exp2` and `exp3` are never both evaluated. If the ConditionalExprNode sets Result, it can direct both subtrees to use the same register. In the JVM, results almost always are placed on the stack, so targeting is automatic (and trivial).

12.5.2 Translating Simple Variables, Constants and Literals

We'll now consider how to generate code for AST nodes corresponding to variables, constants and literals. Variables and constants are both represented by IdentifierNodes. Variables and constants may be local or global and of integer or floating type. Non-scalar variables and constants, components of arrays, members of classes, and parameters will be handled in later chapters.

To translate scalar variables and constants, we will use three code generation subroutines:

GenLoadGlobal(Result, Type, Label),

GenLoadLocal(Result, Type, Offset),

GenLoadLiteral(Result, Type, Value).

GenLoadGlobal will generate a load of a global variable using the label and type specified. If Result is non-null, it will place the variable in Result. Otherwise, GenLoadGlobal will call GetTemp and return the temporary location it has loaded.

Similarly, GenLoadLocal will generate a load of a local variable using the offset and type specified. If Result is non-null, it will place the variable in Result. Otherwise, GenLoadLocal will call GetTemp and return the temporary location it has loaded.

GenLoadLiteral will generate a load of a literal value using the type specified. If Result is non-null, it will place the literal in Result. Otherwise, GenLoadLiteral will call GetTemp and return the temporary location it has loaded.

If a constant represents a literal value (e.g., `const float pi = 3.14159`) and Result is not set by a parent AST node, our code generator will set Result to indicate a literal value. This allows operators that have literal operands to produce better code, perhaps generating an immediate form instruction, or **folding** an expression involving only literals into a literal result. The code generator for IdentifierNodes is shown in Figure 12.4.

```
IdentifierNode.CodeGen( )
1.    if Address.AccessMode = Global
2.          then Result ← GenLoadGlobal(Result, Type, Address.Label)
3.    elsif Address.AccessMode = Local
4.          then Result ← GenLoadLocal(Result, Type, Address.Offset)
5.    elsif Address.AccessMode = Literal
6.          then if Result ≠ null
7.                     then Result ← GenLoadLiteral(Result, Type, Address.Value)
8.                     else Result ← AddressNode(Literal, Address.Value)
```

Figure 12.4 Code Generation for Simple Variables and Constants

AST nodes representing literals are handled in much the same manner as IdentifierNodes. If Result is set by a parent node, GenLoadLiteral is used to load the literal value into the request result location. Otherwise, the Result field is set to represent the fact that it is a literal value (AccessMode = Literal). A load into a register or onto the stack is delayed to allow optimizations in parent nodes.

12.6 Translating Predefined Operators

We now consider the translation of simple expressions—those involving the standard predefined operators and variable, constant or literal operands. In most cases these operators are easy to translate. Most operators have a machine instruction that directly implements them. Arithmetic operations usually require operands be on the stack or in registers, which fits our translation scheme well. We translate operands first and then apply the machine operation that corresponds to the operator being translated. The result is placed in a temporary (Result) just as expected.

A few complication can arise. Some machine instructions allow an immediate operand. We'd like to exploit that possibility when appropriate (rather than loading a constant operand unnecessarily onto the stack or into a register). If both operands are literals, we'd like to be able to simply compute the result at compile-time. (This is a common optimization called **folding**.) For example, `a = 100-1` should be compiled as if it were `a = 99`, with no run-time code generated to do the subtraction.

Care must also be taken to free temporaries properly after they are used as operands (lest we "lose" a register for the rest of the compilation).

We'll use a code generation subroutine GenAdd(Result, Type, Left, Right). GenAdd will generate code to add the operands described by Left and Right. These may be temporaries holding an already-evaluated operand or literals. If literals are involved, GenAdd may be able to generate an immediate instruction or it may be able to fold the operation into a literal result (generating no code at all).

GenAdd will use Type to determine the kind of addition to do (integer or floating, single or double length, etc.). It won't have to worry about "mixed mode" operations (e.g., an integer plus a float) as the type checker will have already inserted type conversion nodes into the AST as necessary. If Result is non-null, GenAdd will place the sum in Result. Otherwise, GenAdd will call GetTemp and return the temporary location into which it has computed the sum.

Since addition is commutative (`a+b` ≡ `b+a`) we can easily extend GenAdd to handle cases in which it is beneficial to swap the left and right operands.

The code generator for PlusNodes is shown in Figure 12.5.

```
PlusNode.CodeGen( )
1.  LeftOperand.CodeGen()
2.  RightOperand.CodeGen()
3.  Result ← GenAdd(Result, Type,
                    LeftOperand.Result, RightOperand.Result)
4.  FreeTemp(LeftOperand.Result, RightOperand.Result)
```

Figure 12.5 Code Generation for + Operator

As an example, consider the expression `A+B+3` where `A` is a global integer variable with a label of `L1` in class `C` and `B` is a local integer variable assigned a local index (an offset within the frame) of 4. The JVM code that is generated is illustrated in Figure 12.6.

JVM Code	Comments	Generated By
`getstatic C/L1 I`	Push static integer field onto stack	GenLoadGlobal
`iload 4`	Push integer local 4 onto stack	GenLoadLocal
`iadd`	Add top two stack locations	GenAdd
`iconst_3`	Push integer literal 3 onto stack	GenAdd
`iadd`	Add top two stack locations	GenAdd

Figure 12.6 JVM Code Generated for `A+B+3`

Other arithmetic and logical operators are handled in a similar manner. Figure 12.7 lists a variety of common integer, relational and logical operators and their corresponding JVM op codes. Since unary + is the identity operator, it is implemented by doing nothing (`+b` ≡ `b`). Further, since `!b` ≡ `b==0`, `!` is implemented using the JVM op code for comparison with zero. There are corresponding instructions for long integer operands and single- and double-length floating operands.

Binary Operator	+	-	*	/	&	|
JVM Op Code	iadd	isub	imul	idiv	iand	ior
Binary Operator	^	<<	>>	%		
JVM Op Code	ixor	ishl	ishr	irem		
Binary Operator	<	>	<=	>=	==	!=
JVM Op Code	if_icmplt	if_icmplt	if_icmplt	if_icmplt	if_icmplt	if_icmplt
Unary Operator	+	-	!	~		
JVM Op Code		ineg	ifeq	ixor		

Figure 12.7 Common Operators and Corresponding JVM OP Codes

Relational operators, like == and >=, are a bit more clumsy to implement. The JVM does not provide instructions that directly compare two operands to produce a boolean result. Rather, two operands are compared, resulting in a conditional branch to a given label. Similarly, many machines, including the Sparc, Motorola MC680x0, Intel x86, IBM RS6000 and PowerPC, use **condition codes**. After a comparison operation, the result is stored in one or more **condition code bits** held

in the processor's status word. This condition code must then be extracted or tested using a conditional branch.

Consider the expression `A < B` where `A` and `B` are local integer variables with frame offsets 2 and 3. On the JVM we might generate.

```
      iload       2  ; Push local #2 (A) onto the stack
      iload       3  ; Push local #3 (B) onto the stack
      if_icmplt   L1 ; Goto L1 if A < B
      iconst_0       ; Push 0 (false) onto the stack
      goto        L2 ; Skip around next instruction
L1:   iconst_1       ; Push 1 (true) onto the stack
L2:
```

This instruction sequence uses six instructions, whereas most other operations require only three (push both operands, then apply an operator). Fortunately in the common case in which a relational expression controls a conditional or looping statement, better code is possible.

12.6.1 Short-Circuit and Conditional Evaluation

We have not yet discussed the `&&` and `||` operators used in C, C++ and Java. That's because they are special. Unlike most binary operators, which evaluate both their operands and then perform their operation, these operators work in "short circuit" mode. This is, if the left operand is sufficient to determine the result of the operation, the right operand *isn't evaluated*. In particular `a&&b` is defined as `if a then b else false`. Similarly `a||b` is defined as `if a then`

`true else b`. The conditional evaluation of the second operand isn't just an optimization—it's essential for correctness. Thus in `(a!=0)&&(b/a>100)` we would perform a division by zero if the right operand were evaluated when `a==0`.

Since we've defined `&&` and `||` in terms of conditional expressions, let's complete our translation of conditional expressions, as shown in Figure 12.8. GenLabel(Label) generates a definition of Label in the generated program. GenBranchIfZero(Operand, Label) generates a conditional branch to Label if Operand is zero (false). GenGoTo(Label) generates a goto to Label.

```
ConditionalExprNode.CodeGen( )
 1.  Result ← Exp2.Result ← Exp3.Result ← GetTemp()
 2.  Exp1.CodeGen()
 3.  FalseLabel ← CreateUniqueLabel()
 4.  GenBranchIfZero(Exp1.Result, FalseLabel)
 5.  FreeTemp(Exp1.Result)
 6.  Exp2.CodeGen()
 7.  OutLabel ← CreateUniqueLabel()
 8.  GenGoTo(OutLabel)
 9.  GenLabel(FalseLabel)
10.  Exp3.CodeGen()
11.  GenLabel(OutLabel)
```

Figure 12.8 Code Generation for Conditional Expressions

Our code generator follows the translation pattern we selected in Section 12.3. First `exp1` is evaluated. We then generate a "branch if equal zero" instruction that tests the value of `exp1`. If `exp1` evaluates to zero, which represents false, we branch to the FalseLabel.

Otherwise, we "fall through." `exp2` is evaluated into a temporary. Then we branch unconditionally to OutLabel, the exit point for this construct. Next False-

Label is defined. At this point exp3 is evaluated, into the same location as exp2. Then OutLabel is defined, to mark the end of the conditional expression.

As an example, consider `if B then A else 1`. We generate the following JVM instructions, assuming `A` has a frame index of 1 and `B` has a frame index of 2.

```
      iload       2  ; Push local #2 (B) onto the stack
      ifeq        L1 ; Goto L1 if B is 0 (false)
      iload       1  ; Push local #1 (A) onto the stack
      goto        L2 ; Skip around next instruction
  L1: iconst_1       ; Push 1 onto the stack
  L2:
```

12.6.2 Jump Code Evaluation

The code we generated for `if B then A else 1` is simple and efficient. For a similar expression, `if (F==G) then A else 1` (where `F` and `G` are local variables of type integer), we'd generate

```
      iload       4  ; Push local #4 (F) onto the stack
      iload       5  ; Push local #5 (G) onto the stack
      if_icmpeq   L1 ; Goto L1 if F == G
      iconst_0       ; Push 0 (false) onto the stack
      goto        L2 ; Skip around next instruction
  L1: iconst_1       ; Push 1 (true) onto the stack
  L2: ifeq        L3 ; Goto L3 if F == G is 0 (false)
      iload       1  ; Push local #1 (A) onto the stack
```

```
        goto        L4 ; Skip around next instruction
L3:     iconst_1       ; Push 1 onto the stack
L4:
```

This code is a bit clumsy. We generate a conditional branch to set the value of `F==G` just so that we can conditionally branch on that value. Any architecture that uses condition codes will have a similar problem.

A moment's reflection shows that we rarely actually *want* the value of a relational or logical expression. Rather, we usually only want to do a conditional branch based on the expression's value, in the context of a conditional or looping statement.

Jump code is an alternative representation of boolean values. Rather than placing a boolean value directly in a register or on the stack, we generate a conditional branch to either a **true label** or a **false label**. These labels are defined at the places where we wish execution to proceed once the boolean expression's value is known.

Returning to our previous example, we can generate `F==G` in jump code form as

```
        iload       4  ; Push local #4 (F) onto the stack
        iload       5  ; Push local #5 (G) onto the stack
        if_icmpne   L1 ; Goto L1 if F != G
```

The label `L1` is the "false label." We branch to it if the expression `F == G` is false; otherwise, we fall through, executing the code that follows if the expression

is true. We can then generate the rest of the expression, defining `L1` at the point where the else expression is to be computed:

```
      iload       1  ; Push local #1 (A) onto the stack
      goto        L2 ; Skip around next instruction
L1:   iconst_1       ; Push 1 onto the stack
L2:
```

This instruction sequence is significantly shorter (and faster) than our original translation.

Jump code comes in two forms, JumpIfTrue and JumpIfFalse. In JumpIfTrue form, the code sequence does a conditional jump (branch) if the expression is true, and "falls through" if the expression is false. Analogously, in JumpIfFalse form, the code sequence does a conditional jump (branch) if the expression is false, and "falls through" if the expression is true. We have two forms because different contexts prefer one or the other.

We will augment our definition of AddressNodes to include the AccessMode values JumpIfTrue and JumpIfFalse. The Label field of the AddressNode will be used to hold the label conditionally jumped to.

It is important to emphasize that even though jump code looks a bit unusual, it is just an alternative representation of boolean values. We can convert a boolean value (on the stack or in a register) to jump code by conditionally branching on its value to a true or false label. Similarly, we convert from jump code to an explicit boolean value, by defining the jump code's true label at a load of 1 and the false label at a load of 0. These conversion routines are defined in Figure 12.9.

```
ConvertToJumpCode( Operand, AccessMode, Label )
 1.   if AccessMode = JumpIfFalse
 2.        then GenBranchIfZero(Operand, Label)
 3.        else GenBranchIfNonZero(Operand, Label)
 4.   return AddressNode(AccessMode, Label)

ConvertFromJumpCode ( Target, AccessMode, Label )
 1.   if AccessMode = JumpIfFalse
 2.        then FirstValue ← 1
 3.             SecondValue ← 0
 4.        else FirstValue ← 0
 5.             SecondValue ← 1
 6.   Target ← GenLoadLiteral(Target, Integer, FirstValue)
 7.   SkipLabel ← CreateUniqueLabel()
 8.   GenGoTo(SkipLabel)
 9.   GenLabel(Label)
10.   Target ← GenLoadLiteral(Target, Integer, SecondValue)
11.   GenLabel(SkipLabel)
12.   return Target
```

Figure 12.9 Routines to Convert To and From Jump Code

We can easily augment the code generators for IdentifierNodes and LiteralNodes to use ConvertToJumpCode if the Result field shows jump code is required.

An advantage of the jump code form is that it meshes nicely with the `&&` and `||` operators, which are already defined in terms of conditional branches. In particular if `exp1` and `exp2` are in jump code form, then we need generate *no further code* to evaluate `exp1&&exp2`.

To evaluate `&&`, we first translate `exp1` into JumpIfFalse form, followed by `exp2`. If `exp1` is false, we jump out of the whole expression. If `exp1` is true, we fall through to `exp2` and evaluate it. In this way, `exp2` is evaluated only when necessary (when `exp1` is true).

The code generator for `&&` is shown in Figure 12.10. Note that both JumpIfFalse and JumpIfTrue forms are handled, which allows a parent node to select the form more suitable for a given context. A non-jump code result (in a register or on the stack) can also be produced, using ConvertFromJumpCode.

```
ConditionalAndNode.CodeGen( )
 1.   if  Result.AccessMode = JumpIfFalse
 2.        then  Exp1.Result ← AddressNode(JumpIfFalse,Result.Label)
 3.              Exp2.Result ← AddressNode(JumpIfFalse,Result.Label)
 4.              Exp1.CodeGen()
 5.              Exp2.CodeGen()
 6.   elsif  Result.AccessMode = JumpIfTrue
 7.        then  Exp1.Result ← AddressNode(JumpIfFalse, CreateUniqueLabel())
 8.              Exp2.Result ← AddressNode(JumpIfTrue,Result.Label)
 9.              Exp1.CodeGen()
10.              Exp2.CodeGen()
11.              GenLabel(Exp1.Result.Label)
12.         else
13.              Exp1.Result ← AddressNode(JumpIfFalse, CreateUniqueLabel())
14.              Exp2.Result ← AddressNode(JumpIfFalse,Exp1.Result.Label)
15.              Exp1.CodeGen()
16.              Exp2.CodeGen()
17.              Result ← ConvertFromJumpCode(Result, JumpIfFalse,
                                  Exp1.Result.Label)
```

Figure 12.10 Code Generation for Conditional And

Similarly, once `exp1` and `exp2` are in jump code form, `exp1||exp2` is easy to evaluate. We first translate `exp1` into JumpIfTrue form, followed by `exp2`. If `exp1` is true, we jump out of the whole expression. If `exp1` is false, we fall through to `exp2` and evaluate it. In this way, `exp2` is evaluated only when necessary (when `exp1` is false).

The code generator for `||` is shown in Figure 12.11. Again, both JumpIfFalse and JumpIfTrue forms are handled, allowing a parent node to select the form more

suitable for a given context. A non-jump code result (in a register or on the stack) can also be produced.

```
ConditionalOrNode.CodeGen( )
 1.   if Result.AccessMode = JumpIfTrue
 2.       then Exp1.Result ← AddressNode(JumpIfTrue,Result.Label)
 3.            Exp2.Result ← AddressNode(JumpIfTrue,Result.Label)
 4.            Exp1.CodeGen()
 5.            Exp2.CodeGen()
 6.   elsif Result.AccessMode = JumpIfFalse
 7.       then Exp1.Result ← AddressNode(JumpIfTrue, CreateUniqueLabel())
 8.            Exp2.Result ← AddressNode(JumpIfFalse,Result.Label)
 9.            Exp1.CodeGen()
10.            Exp2.CodeGen()
11.            GenLabel(Exp1.Result.Label)
12.       else
13.            Exp1.Result ← AddressNode(JumpIfTrue, CreateUniqueLabel())
14.            Exp2.Result ← AddressNode(JumpIfTrue,Exp1.Result.Label)
15.            Exp1.CodeGen()
16.            Exp2.CodeGen()
17.            Result ← ConvertFromJumpCode(Result, JumpIfTrue,
                        Exp1.Result.Label)
```

Figure 12.11 Code Generation for Conditional Or

The `!` operator is also very simple once a value is in jump code form. To evaluate `!exp` in JumpIfTrue form, you simply translate `exp` in JumpIfFalse form. That is, jumping to location `L` when `exp` is false is equivalent to jumping to `L` when `!exp` is true. Analogously, to evaluate `!exp` in JumpIfFalse form, you simply translate `exp` in JumpIfTrue form.

As an example, let's consider `(A>0)||(B<0 && C==10)`, where `A`, `B` and `C` are all local integers, with indices of 1, 2 and 3 respectively. We'll produce a JumpIfFalse translation, jumping to label `F` if the expression is false and falling through if the expression is true.

The code generators for relational operators can be easily modified to produce both kinds of jump code—we can either jump if the relation holds (JumpIfTrue) or jump if it doesn't hold (JumpIfFalse). We produce the following JVM code sequence which is compact and efficient.

```
    iload       1     ; Push local #1 (A) onto the stack
    ifgt        L1    ; Goto L1 if A > 0 is true
    iload       2     ; Push local #2 (B) onto the stack
    ifge        F     ; Goto F if B < 0 is false
    iload       3     ; Push local #3 (C) onto the stack
    bipush      10    ; Push a byte immediate value (10)
    if_icmpne   F     ; Goto F if C != 10
L1:
```

First `A` is tested. If it is greater than zero, the expression must be true, so we go to the end of the expression (since we wish to branch if the expression is *false*). Otherwise, we continue evaluating the expression.

We next test `B`. If it is greater than or equal to zero, `B<0` is false, and so is the whole expression. We therefore branch to label `F` as required. Otherwise, we finally test `C`. If `C` is not equal to 10, the expression is false, so we branch to label `F`. If `C` is equal to 10, the expression is true, and we fall through to the code that follows.

As shown in Figure 12.12, conditional expression evaluation is simplified when jump code is used. This is because we no longer need to issue a conditional

branch after the control expression is evaluated. Rather, a conditional branch is an integral part of the jump code itself.

As an example, consider `if (A&&B) then 1 else 2`. Assuming `A` and `B` are locals, with indices of 1 and 2, we generate

```
      iload    1      ; Push local #1 (A) onto the stack
      ifeq     L1     ; Goto L1 if A is false
      iload    2      ; Push local #2 (B) onto the stack
      ifeq     L1     ; Goto L1 if B is false
      iconst_1        ; Push 1 onto the stack
      goto     L2     ; Skip around next instruction
L1:   iconst_2        ; Push 2 onto the stack
L2:
```

```
ConditionalExprNode.CodeGen( )
 1.  Result ← Exp2.Result ← Exp3.Result ← GetTemp()
 2.  FalseLabel ← CreateUniqueLabel()
 3.  Exp1.Result ← AddressNode(JumpIfFalse, FalseLabel)
 4.  Exp1.CodeGen()
 5.  Exp2.CodeGen()
 6.  OutLabel ← CreateUniqueLabel()
 7.  GenGoTo(OutLabel)
 8.  GenLabel(FalseLabel)
 9.  Exp3.CodeGen()
10.  GenLabel(OutLabel)
```

Figure 12.12 Code Generation for Conditional Expressions Using Jump Code

12.7 Pointer and Reference Manipulations

In C and C++ the unary `&` and `*` operators create and dereference pointers. For example,

```
P = &I; // P now points to I's memory location

J = *P; // Assign value P points to (I) to J
```

Java does not allow explicit pointer manipulation, so `&` and `*` are not available.

In most architectures, `&` and `*` are easy to implement. A notable exception is the JVM, which forbids most kinds of pointer manipulation. This is done to enhance program security; arbitrary pointer manipulation can lead to unexpected and catastrophic errors.

Conventional computer architectures include some form of "load address" instruction. That is, rather than loading a register with the byte or word at a particular address, the address itself is loaded. Load address instructions are ideal for implementing the `&` (address of) operator. If `V` is a global variable with label `L`, `&V` evaluates to the address `L` represents. The expression `&V` can be implemented as

```
LoadAddress Register, L.
```

If `W` is a local variable at offset `F` in the current frame, then `&W` represents `F` plus the current value of the frame pointer. Thus `&W` can be implemented as

```
LoadAddress Register, F+FramePointer.
```

The dereference operator, unary `*`, takes a pointer and fetches the value at the address the pointer contains. Thus `*P` is evaluated in two steps. First `P` is evaluated, loading an address (`P`'s value) into a register or temporary. Then that

address is used to fetch a data value. The following instruction pair is commonly generated to evaluate `*P`

```
Load Reg1, P      ; Put P's value into Reg1
Load Reg2, (Reg1) ; Use Reg1's contents as an address
                  ; to load Register 2
```

Pointer arithmetic is also allowed in C and C++. If `P` is a pointer, the expressions `P + i` and `P - i` are allowed, where `i` is an integer value. In these cases, `P` is treated as an unsigned integer. The value of `i` is multiplied by the `sizeof(T)` where `P`'s type is `*T`. Then `i*sizeof(T)` is added to or subtracted from `P`. For example, if `P` is of type `*int` (a pointer to `int`) and `int`s require 4 bytes, then the expression `P+1` generates

```
Load Reg1, P      ; Put P's value into Reg1
AddU Reg2, Reg1,4 ; Unsigned add of 4 (sizeof(int)) to P
```

Since pointer arithmetic is limited to these two special cases, it is easiest implemented by extending the code generation subroutines for addition and subtraction to allow pointer types (as well as integer and floating types).

C++ and Java provide references which are a restricted form of pointer. In C++ a reference type (`T&`), once initialized, cannot be changed. In Java, a reference type may be assigned to, but only with valid references (pointers) to heap-allocated objects.

In terms of code generation, references are treated almost identically to explicit pointers. The only difference is that no explicit dereference operators appear in the AST. That is, if `P` is of type `int*` and `R` is of type `int&`, the two

expressions `(*P)+1` and `R+1` do exactly the same thing. In fact, it is often a good translation strategy to include an explicit "dereference" AST node in an AST that contains references. These nodes, added during type checking, can then cue the code generator that the value of a reference must be used as a pointer to fetch a data value from memory.

Some programming languages, like Pascal and Modula, provide for "var" or "reference" parameters. These parameters are implemented just as reference values are; they represent an address (of an actual parameter) which is used to indirectly fetch a data value when a corresponding formal parameter is used.

12.8 Type Conversion

During translation of an expression, it may be necessary to convert the type of a value. This may be forced by a type cast (e.g., `(float) 10`) or by the type rules of the language (`1 + 1.0`). In any event, we'll assume that during translation and type checking, explicit type conversion AST nodes have been inserted where necessary to signal type conversions. The ConvertNode AST node (see Figure 12.13) signals that the Operand node (an expression) is to be converted to the type specified by the Type field after evaluation. Type-checking has verified that the type conversion is legal.

Some type conversion require no extra code generation. For example, on the JVM (and most other machines) bytes, half words and full words are all manipulated as 32 bit integers on the stack or within registers. Thus the two expressions `'A'+1` and `(int) 'A'+1` generate the same code.

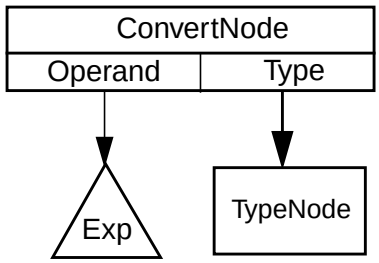

Figure 12.13 Abstract Syntax Tree for Type Conversion

Conversions between single and double length operands and integer and floating representations *do* require additional code. On the JVM special instructions are provided; on other machines library subroutines may need to be called. Four distinct internal representations must be handled—32 bit integer, 64 bit integer, 32 bit floating point and 64 bit floating point. Conversions between each of these representations must be supported.

On the JVM 12 conversion instructions of the form `x2y` are provided, where `x` and `y` can be `i` (for integer), `l` (for long integer), `f` (for float) and `d` (for double). Thus `i2f` takes an integer at the top of the stack and converts it to a floating point value.

Three additional conversion instructions, `i2s`, `i2b` and `i2c` handle explicit casts of integer values to short, byte and character forms.

12.9 Assignment Operators

In Java, C and C++ the assignment operator, `=`, may be used anywhere within an expression; it may also be cascaded. Thus both `a+(b=c)` and `a=b=c` are legal.

Java also provides an assignment statement. In contrast, C and C++ allow an expression to be used wherever a statement is expected. At the statement level, an assignment discards its right-hand side value after the left-hand side variable is updated.

The AST for an assignment to a simple scalar variable is shown in Figure 12.14. Code generation is reasonably straightforward. We do need to be careful about how we handle the variable that is the target of the assignment.

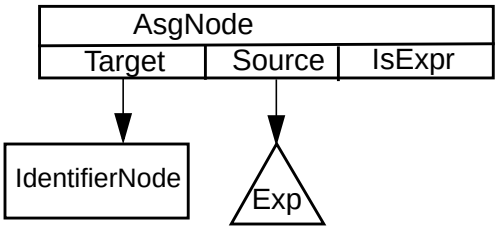

Figure 12.14 Abstract Syntax Tree for Assignments

Normally, we recursively apply the code generator to all the children of an AST node. Were we to do this the IdentifierNode, we'd compute its value and put in on the stack or in a register. This *isn't* what we want. For simple variables, we'll just extract the identifier's label or frame offset directly from the IdentifierNode. In more complex cases (e.g., when the left-hand side is a subscripted variable or pointer expression), we need a special variant of our code generator that evaluates the *address* denoted by an expression rather than the *value* denoted by an expression. To see why this distinction is necessary, observe that `a[i+j]` on the left-hand side of an assignment represents the *address* of an array element; on the right-hand side it represents the *value* of an array element. Whenever we trans-

late an expression or construct, we'll need to be clear about whether we want to generate code to compute an address or a value.

We'll also need to be sure that the expression value to be assigned in on the stack or in a register. Normally, translating an expression does just that. Recall however, that as an optimization, we sometimes delay loading a literal value or translate a boolean-valued expression in jump code form. We'll need to be sure we have a value to store before we generate code to update the left-hand side's value.

Since we are handling only simple variables, translating an assignment is simple. We just evaluate (as usual) the right-hand side of the expression and store its value in the variable on the left-hand side. The value of the expression is the value of the right-hand side. The IsExpr field (a boolean) marks whether the assignment is used as an expression or a statement. If the assignment is used as an statement, the right-hand side value, after assignment, may be discarded. If the assignment is used as an expression, the right-hand side value will be preserved.

We will use two code generation subroutines:

GenStoreGlobal(Source, Type, Label, PreserveSource),

GenStoreLocal(Source, Type, Offset, PreserveSource).

GenStoreGlobal will generate a store of Source into a global variable using the label and type specified. GenStoreLocal will generate a store of Source into a local variable using the offset and type specified.

In both routines, Source is an AddressNode. It may represent a temporary or literal. In the case of a literal, special case instructions may be generated (like a

"store zero" instruction). Otherwise, the literal is loaded into a temporary and stored.

If PreserveSource is true, the contents of Source is preserved. That is, if Source is in a register, that register is not released. If Source is on the stack, its value is duplicated so that the value may be reused after the assignment (store instructions automatically pop the stack in the JVM). If PreserveSource is false, Source is not preserved; if it represents a register, that register is freed after the assignment.

Translation of an assignment operator is detailed in Figure 12.15.

```
AsgNode.CodeGen( )
1.   Source.CodeGen()
2.   if Source.Result.AccessMode ∈ {JumpIfTrue, JumpIfFalse}
3.        then Result ← ConvertFromJumpCode(Result,
                    Source.Result.AccessMode, Source.Result.Label)
4.        else Result ← Source.Result
5.   if Target.Address.AccessMode = Global
6.        then GenStoreGlobal(Result, Type, Target.Address.Label, IsExpr)
7.        else GenStoreLocal(Result, Type, Target.Address.Offset, IsExpr)
```

Figure 12.15 Code Generation for Assignments

As an example, consider `A = B = C + 1` where `A`, `B` and `C` are local integers with frame indices of 1, 2 and 3 respectively. First, `C + 1` is evaluated. Since `B = C + 1` is used as an expression, the value of `C + 1` is duplicated (and thereby preserved) so that it can also be assigned to `A`. The JVM code we generate is

```
iload       3   ; Push local #3 (C) onto the stack
iconst_1        ; Push 1 onto the stack
iadd            ; Compute C +1
dup             ; Duplicate C + 1 for second store
```

```
istore      2   ; Store top of stack into local #2 (B)
istore      1   ; Store top of stack into local #1 (A)
```

In C and C++ we have a problem whenever an expression is used where a statement is expected. The difficulty is that a value has been computed (into a register or onto the stack), but that value will never be used. If we just ignore the value a register will never be freed (and hence will be "lost" for the rest of the program) or a stack value will never be popped.

To handle this problem, we'll assume that whenever an expression is used as a statement, a VoidExpr AST node will be placed above the expression. This will signal that the expression's value won't be used, and the register or stack location holding it can be released. The VoidExpr node can also be used for the left operand of the "," (sequencing) operator, where the left operand is evaluated for side-effects and then discarded.

Note that in some cases an expression used as a statement need not be evaluated at all. In particular, if the expression has no side-effects, there is no point to evaluating an expression whose value will be ignored. If we know that the expression does no assignments, performs no input/output and makes no calls (which themselves may have side-effects), we can delete the AST as useless. This analysis might be done as part of a pre-code generation simplification phase or as a data flow analysis that detects whether or not a value is dead (see Chapter 16).

12.9.1 Compound Assignment Operators

Java, C and C++ contain a useful variant of the assignment operator—the compound assignment operator. This operator uses its left operand twice—first to obtain an operand value, then as the target of an assignment. Informally `a op= b` is equivalent to `a = a op b` except that `a` must be evaluated only once. For simple variables the left-hand side may be visited twice, first to compute a value and then to obtain the address used in a store instruction. If evaluation of the left-hand side's address is non-trivial, we first evaluate the variable's address. We then use that address twice, as the source of an operand and as the target of the result.

The code generator for += is shown in Figure 12.16. Other compound assignment operators are translated in a similar manner, using the operator/op code mapping shown in Figure 12.7.

```
PlusAsgNode.CodeGen( )
1.   LeftOperand.CodeGen()
2.   RightOperand.CodeGen()
3.   Result ← GenAdd(Result, Type,
                        LeftOperand.Result, RightOperand.Result)
4.   FreeTemp(LeftOperand.Result, RightOperand.Result)
5.   if LeftOperand.Address.AccessMode = Global
6.        then GenStoreGlobal(Result, Type,
                        LeftOperand.Address.Label, IsExpr)
7.        else GenStoreLocal(Result, Type,
                        LeftOperand.Address.Offset, IsExpr)
```

Figure 12.16 Code Generation for += Operator

As an example, consider `A += B += C + 1` where `A`, `B` and `C` are local integers with frame indices of 1, 2 and 3 respectively. The JVM code we generate is

```
iload    1  ; Push local #1 (A) onto the stack

iload    2  ; Push local #2 (B) onto the stack

iload    3  ; Push local #3 (C) onto the stack
```

```
iconst_1    ; Push 1 onto the stack
iadd        ; Compute C+1
iadd        ; Compute B+C+1
dup         ; Duplicate B+C+1
istore   2  ; Store top of stack into local #2 (B)
iadd        ; Compute A+B+C+1
istore   1  ; Store top of stack into local #1 (A)
```

12.9.2 Increment and Decrement Operators

The increment (++) and decrement (--) operators are widely used in Java, C and C++. The prefix form `++a` is identical to `a += 1` and `--a` is identical to `a -= 1`. Thus these two operators may be translated using the techniques of the previous section.

The postfix forms, `a++` and `a--` are a bit different in that after incrementing or decrementing the variable, the *original* value of the variable is returned as the value of the expression. This difference is easy to accommodate during translation.

If `a++` is used as a statement (as it often is), a translation of `a += 1` may be used (this holds for `a--` too).

Otherwise, when `a` is evaluated, its value is duplicated on the stack, or the register holding its value is retained. The value of `a` is then incremented or decre-

mented and stored back into a. Finally, the original value of a, still on the stack or in a register, is used as the value of the expression.

For example, consider a+(b++), where a and b are integer locals with frame indices of 1 and 2. The following JVM code would be generated

```
iload       1  ; Push local #1 (A) onto the stack
iload       2  ; Push local #2 (B) onto the stack
dup            ; Duplicate B
iconst_1       ; Push 1 onto the stack
iadd           ; Compute B+1
istore      2  ; Store B+1 into local #2 (B)
iadd           ; Compute A+original value of B
```

As a final observation, note that the definition of the postfix ++ operator makes C++'s name something of a misnomer. After improving C's definition we certainly don't want to retain the *original* language!

Exercises

1. Most C and Java compilers provide an option to display the assembly instructions that are generated. Compile the following procedure on your favorite C or Java compiler and get a listing of the code generated for it. Examine each instruction and explain what step it plays in implementing the prodecure's body.

```
void proc() {
        int a=1,b=2,c=3,d=4;
        a = b + c * d;
        c=(a<b)?1:2;
        a=b++ + ++c;
 }
```

2. Show the code that would be generated for each of the following expressions. Assume `I` and `J` are integer locals, `K` and `L` are integer globals, `A` and `B` are floating point locals and `C` and `D` are floating point globals.

```
I+J+1
K-L-1
A+B+1
C-D-1
I+L+10
A-D-10
I+A+5
I+(1+2)
```

```
I+1+2
```

3. The code generated for `I+1+2` in Exercise 2 is probably not the same as that generated for `I+(1+2)`. Why is this? What must be done so that both generate the same (shorter) instruction sequence?
4. Some languages, including Java, require that all variables be initialized. Explain how to extend the code generator for VarDeclNodes (Figure 12.3) so that both local and global variables are given a default initialization if no explicit initial value is declared.
5. Complete the code generator for VarDeclNodes (Figure 12.3) by inserting calls to the proper code generation subroutines to store the value of InitValue into a local variable.
6. On register-oriented machines, GetTemp is expected to return a different register each time it is called. This means that GetTemp may run out of registers if too many registers are requested before FreeTemp is called.

   What can be done (other than terminating compilation) if GetTemp is called when no more free registers are available?
7. It is clear that some expressions are more complex than others in terms of the number of registers or stack locations they will require.

   Explain how an AST representing an expression can be traversed to determine the number of registers or stack locations its translation will require. Illustrate your technique on the following expression

```
((A+B)+(C+D))+((E+F)+(G+H))
```

8. Explain how to extend the code generator for the `++` and `--` operators to handle pointers as well as numeric values.

9. Assume we add a new operator, the **swap** operator, `<->`, to Java or C. Given two variables, `v1` and `v2`, the expression `v1<->v2` assigns `v2`'s value to `v1` and assigns `v1`'s original value to `v2`, and then returns `v1`'s new value.

   Define a code generator that correctly implements the swap operator.

10. Show the jump code that would be generated for each of the following expressions. Assume `A`, `B` and `C` are all integer locals, and that `&&` and `||` are left-associative.

```
(A<1)&&(B>1)&&(C!=0)

(A<1)||(B>1)||(C!=0)

(A<1)&&(B>1)||(C!=0)

(A<1)||(B>1)&&(C!=0)
```

11. What is generated if the boolean literals `true` and `false` are translated into JumpIfTrue or JumpIfFalse form?

    What code is generated for the expressions `(true?1:2)` and `(false?1:2)`? Can the generated code be improved?

12. If we translate the expression `(double)(long) i` (where `i` is an `int`) for the JVM we will generate an `i2l` instruction (`int` to `long`) followed by an `l2d` (`long` to `double`) instruction. Can these two instructions be combined into an `i2d` (`int` to `double`) instruction?

Now consider `(double)(float) i`. We now will generate an `i2f` instruction (`int` to `float`) followed by an `f2d` (`float` to `double`) instruction. Can these two instructions be combined into an `i2d` instruction?

In general what restrictions must be satisfied to allow two consecutive type conversion instructions to be combined into a single type conversion instruction?

13. In C and C++ the expression `exp1,exp2` means evaluate `exp1`, then `exp2` and return the value of `exp2`. If `exp1` has no side effects (assignments, I/O or system calls) it need not be evaluated at all. How can we test `exp1`'s AST, prior to code generation, to see if we can suppress its evaluation?

14. On some machines selected registers are preassigned special values (like 0 or 1) for the life of a program. How can a code generator for such a machine exploit these preloaded registers to improve code quality?

15. It is sometimes that case that a particular expression value is needed more than once. An example of this is the statement `a[i+1]=b[i+1]+c[i+1]` where `i+1` is used three times.

    An expression that may be reused is called a **redundant** or **common** expression. An AST may be transformed so that an expression that is used more than once has multiple parents (one parent for each context that requires the same value). An AST node that has more than one parent isn't really a tree anymore. Rather, its a **directed-acyclic graph** (a **dag**).

How must code generators for expressions be changed when they are translating an AST node that has more than one parent?