COMPILER CONSTRUCTION **Fall 2019**

Lecture 24: (Memory allocation, basic block, code optimization)

Instructor: K.R. Chowdhary *: Professor of CS*

24.1 Introduction

The final phase in compiler is the code generation. It takes as input the intermediate representation produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program (see Fig. 24.1).

Figure 24.1: Position of code generator.

There are too many requirements imposed on code generator. The target program (machine code) must preserve the semantics of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine, like, CPU, memory, registers, etc. Moreover, the code generating program itself must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily an optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers which need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the intermediate representation into intermediate representation from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the back end, may require multiple passes over the intermediate representation before generating the target program. The techniques presented in the following can be used irrespective of whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: *machine instruction selection*, *register allocation and assignment*, and *instruction ordering*. Instruction selection involves choosing appropriate target-machine instructions to implement the intermediate representation statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

In the following we will discuss about algorithms about code generation that translates IR (Intermediate Representation) into sequence of target language instructions for simple register machine. Some code generators partition the IR instructions into *basic blocks*, that consist of sequence of instructions that are always executed together.

24.2 Target language

The knowledge about the target machine and its instruction set is prerequisite for designing a code generator. The minimum requirement for this is assembly code of the machine. In fact, a target machines models are three-address machine with store and load instructions, with jump – conditional and unconditional. The registers are designated as $R0, R1, ..., R_{n-1}$.

The most common load instructions are $LD\ r,\ x$, where x is memory location and r is register. Alternatively, it can be $LD\ r1,\ r2$.

The store instructions are $ST\ x,\ r$.

Computation instructions are $OP\ dst,\ src_1,\ src_2$, which are not necessarily distinct. OP is operator ADD, SUB.

The instructions $BR\ L$ causes branches to machine instruction at location L, and $Bcond\ r, L$, branches to location L subject to the true result in some register r.

Further it is assumed that the target machine has various addressing modes: register, memory, register indirect, memory indirect, absolute, relative, etc.

24.3 Issues in design of code generator

These issues are dependent on IR, target language, and run-time system, like, instruction selection, register allocation, and assignment, the ordering of the instructions.

Code generator input The input to code generator is IR, and the symbol table, generated due to preceding phases. The choices of IR are: three-address representation, like quadruples, triples; virtual machine representation like bytecode; linear representation like postfix; syntax-trees, and DAGs. Further it is assumed that the input coming to code generator has no errors of syntax and semantics, as they have been removed. Also, the type conversion operators have been inserted where ever needed.

Target Program The most common target architecture are RISC (reduced instruction set computing), CISC (compressed instruction set computing), and stack based. The RISC machines have many registers, three-address instructions, simple addressing modes, and simple instructions.

However, the CISC architecture have relatively few registers, two-address instructions, number of addressing modes, many register classes, and variable length instructions.

Operations on the stack-based machines are done on the stack. These machines are not common now, except in JVM (java virtual machines). The java virtual machine translate the byte code instructions into native hardware instructions at run time.

Instruction Selection The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of this mapping is determined by factors such as,

- level of IR
- nature of instruction set architecture
- desired quality of generated code.

If IR is high level, code generator translate each IR statement into a sequence of machine instructions using code templates. However, such method generate poor code, which requires further optimization. If IR reflects some low level features of the machine, then code generator can use this to generate more efficient code. Some machines do the floating point operations at machine instruction level.

For three-address instruction ($x = y + z$) we can design a skeleton to construct a two-address code, as follows:

```
LD   R0, y    // R0 = y       (load y into register R0)
ADD  R0, z    // R0 = R0 +z   (add z to R0)
ST   x, R0    // x = R0       (store R0 into x)
```

The increment instruction ($a = a + 1$) can be implemented by two address instructions as follows:

```
LD   R0, a
ADD  R0, #1
ST   a, R0
```

Register Allocation The key problem in code generation is deciding of what values to hold in what registers. The registers are the fastest units of a machine, but there are not enough numbers to hold all the values, hence the extra values are stored in memory. The problem of using the registers is subdivided into two parts:

1. *Register allocation.* Selection of variables is done at each point in the program, that will reside in registers.
2. *Register assignment.* Specific registers are picked up for assignment variables to them. This part is complex, as operating system demand certain register usage convention, which is to be observed. In fact the problem is NP-complete.

Evaluation order The order of computation can effect the efficiency of the target code, as some computation requires fever registers than the others. However, picking best order is NP-complete.

24.4 Addresses in Target code

In the following discussions we will show how the names in the IR code can be converted into addresses in the target code using static and dynamic allocation. Each executing program runs in its own logical address space that was partitioned into four code and data areas:

1. The statically determined area code that holds the executable target code. Its size can be determined at compile time.

2. A data area that is statically determined and holds global constants and other data generated by the compiler. Its size can also be determined at compile time.

3. A dynamically managed heap area for holding data that are allocated and freed during the program execution. The size of heap cannot be determined at compiled time.

4. A dynamically managed area stack for holding activation records as they are created and destroyed during the procedure calls and returns. The size of stack also cannot be determined at compile time.

24.4.1 Static Allocation

We will focus on the following thee-address statements: *call callee, return, halt*, and *action*. The *action* is place holder for other three-address statements.

The size and layout of *activation records* are determined by the code generator via the information about names stored in the symbol table. We will first explain how to store the return address in the activation record on a procedure call. For convenience, we assume that first location in the activation record holds the return address.

Let us first consider the code needed to implement the simplest case, static allocation. We give the name of the procedure being called, as *calee* or *called* Here, a *call callee* statement in the intermediate code can be implemented by a sequence of two instructions of a target-machine:

`ST` *callee.staticArea*, # here + 20
`BR` *callee.codeArea*
<- this is the return position

The ST instruction saves return address at the beginning of the activation record for callee, and the `BR` instruction transfers control to the target code for the called procedure *callee*. The attribute before *callee.staticArea* is a constant that gives the address of the beginning of the activation record for *callee*. The attribute *callee.codeArea* is constant referring to the address of the first instruction of the called procedure *callee* in the code area of the run-time memory.

The operand #*here* + 20, in the `ST` instruction is the literal return address; it is the address of the instruction following the `BR` instruction. We assume that "# here" is the address of the current instruction and that the three constants (*calee.staticArea, 20, calee.codeArea*), plus the two instructions (ST, BR) in the calling sequence have length of 5 words or 20 bytes. Note that, this happens when word is of 4-byte length.

The code for the procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is `HALT`, which returns control to the operating system. A `return` statement in *callee* can be implemented by simple jump instruction, like,

`BR` **callee.staticArea*

which transfers control to the address saved at the beginning of the activation record for *callee*. Here, **callee.staticArea* is pointer to return address.

Example 24.1 *Implementing subroutine call.*

Suppose that we have following three-address code:

```
// code for C
action1
call p
action2
halt

//code for P
action3
return
```

Let us assume that code C starts at location 100 and subroutine P starts at location 200. The instructions starting at 100 implement the statements: `action1; cal p; action2; halt`.

The execution starts with the instruction `ACTION1` at address 100. Therefore the `ST` instruction at address 120 saves the return address at 140 in the machine status field, which is the first word in the activation record of P. We assume that each ACTION instruction takes 20 bytes. Also, we assume that activation records for these procedures are statically allocated starting at locations 300 and 364, respectively. The following is target code for static allocation.

```
                //code for c
100:  ACTION1     // code for action1
120:  ST 364, #140 //save return address 140 at 364
132:  BR 200      // call p
140:  ACTION2
160:  HALT        // return to oper. syst.
....              // code for p
200:  ACTION3     //
220:  BR *364     // return to 140
...               // 364-363 holds activation record for c

364:              // return address
368:              // local data for p
```

24.4.2 Stack allocation

When we use relative addresses for storage in the activation records, the static allocation becomes stack allocation. In this type of allocation, the position of an activation record for a procedure is not known until run time. This position is usually stored in a register. The indexed address mode of our target machine is convenient for this purpose.

The relative addresses in an activation record can be taken as offsets from any known position in the activation record. We shall use positive offsets by maintaining in a register `SP` a pointer to the beginning of the activation record on top of stack. When a procedure call occurs, the calling procedure increments `SP` and transfers control to the called procedure. After control returns to the caller, we decrement `SP`, thereby deallocating the activation record of the called procedure.

Following is the code for the first procedure that initializes the stack by setting `SP` to the start of the stack area in memory:

```
LD SP, #stackStart        // initialize the stack
code for the first procedure
HALT                      // terminate execution
```

A procedure call sequence increments `SP`, saves the return address, and transfers control to the called procedure:

```
ADD SP, SP, #caller.recordSize   // increment stack pointer
ST  *SP, #here + 16              // save return address
BR  callee.codeArea              // return to caller
```

The operand *#caller.recordSize* represents the size of an activation record, so the `ADD` instruction makes `SP` point to the next activation record. The operand *#here+16* in the `ST` instruction is the address of the instruction following `BR`; it is saved in the address pointed to by `SP`.

The return sequence consists of two parts. The called procedure transfers control to the return address using,

```
BR  *0(SP)     // return to caller
```

The reason for using `*0(SP)` in `BR` instruction is that we need two levels of indirection: `0(SP)` is the address of the first word in the activation record and `*0(SP)` is return address saved here.

The second part of the return sequence is in the caller, which decrements `SP`, thus restoring `SP` to its previous value. That is, after the subtraction `SP` points to the beginning of activation record of the caller:

```
SUB  SP, SP, #caller.recordSize     // decrement stack pointer
```

24.4.3 Run time addresses for names

The storage-allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed. We have studied that the name in a three-address statement is really a pointer to symbol-table entry for that name. This approach has significant advantages; it makes the compiler more portable, since the front end need not to be changed even when compiler is moved to a different machine when a different run-time organization is needed. On the other hand, generating the specific sequence of access steps while generating intermediate code can be of significant advantage in an optimizing compiler, since it lets the optimizer take advantage of details in would not see the simple three-address statement.

In either case, the name must be replaced ultimately by a code to access storage locations. Consider the three-address copy statement `x = 0`. Suppose the symbol entry x has relative address 12. Consider that `x` is statically allocated in the beginning at address *static*. Thus actual run time address of x is $x + 12$. The assignment `x` is translated to,

```
static[12] = 0
```

If the static area starts at address 100, the target code for this statement is

```
LD 112, #0
```

24.5 Construction of Basic Blocks and Flow Graph

The graph representation of intermediate code is helpful for discussing code generation. To begin with, the graph is not constructed explicitly by a code-generation algorithm. Code generation is benefited from context. For example, we can do a better job of register allocation if we know how values are defined and used. We can select the instructions in a better way, by looking at sequences of three-address statements. The basic block and flow graph representation are done as follows:

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the following properties
   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no direct jumps to enter into the middle of a block.
   (b) Control will leave the block without halting in the block or branching, from inside of a block exiting from the last instruction in the block.
2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks follow to what other blocks.

24.6 Basic Blocks Algorithm

Our first job is to partition a sequence of three-address instructions into *basic blocks*. We begin a new basic block with the first instruction and keep adding instructions until we meet

either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm-1: Partitioning three-address instructions set into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks, such that each instruction is assigned to exactly one basic block.

METHOD: First, we identify those instructions in the intermediate code that are *leader instructions*, that is, the first instructions in some basic block. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

Example 24.2 *Intermediate code to set a* 10×10 *matrix to an identity matrix.*

Consider the pseudocode as below, which turns a 10×10 matrix into an identity matrix.

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j] = 0.0;
for i from 1 to 10 do
   a[i,i] = 1.0;
```

Corresponding to above pseudo-code, the intermediate code in Table 24.1 turns a 10×10 matrix a into an identity matrix. In generating the intermediate code, we have assumed that the real-valued array elements take 4 bytes each, and that the matrix a is stored in row-major form (first row is stored, element by element, then second row, and so on). Note that j is column counter and i is row counter.

First, instruction 1 is a *leader* by rule (1) of Algorithm 1. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

Table 24.1: Intermediate code to set the 10×10 matrix a to an identity matrix

1)	$i = 1$
2)	$j = 1$
3)	$t1 = 10 * i$
4)	$t2 = t1 + j$
5)	$t3 = 4 * t2$
6)	$t4 = t3 - 44$
7)	$a[t4] = 0.0$
8)	$j = j + 1$
9)	$if\ j <= 10\ goto\ (3)$
10)	$i = i + 1$
11)	$if\ i <= 10\ goto\ (2)$
12)	$i = 1$
13)	$t5 = i - 1$
14)	$t6 = 44 * t5$
15)	$a[t6] = 1.0$
16)	$i = i + 1$
17)	$if\ i <= 10\ goto\ (13)$

We conclude that the leaders are instructions $1, 2, 3, 10, 12$, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12 and instruction 13's block is 13 through 17. □

24.6.1 Liveness and Next-Use

For generating good code, it is essential to know when the value of a variable will be used. If the value of a variable, currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The use of a name in a three-address statement is defined as follows: Let a three-address statement i assigns a value to variable x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j uses the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Our algorithm to determine liveness and next-use information makes a back-ward pass over each basic block. We store the information in the symbol table. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 1. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Algorithm 2. Determining the liveness and next-use information for each statement in a basic block.

Input. A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

Output. At each statement $i : x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

Method. We start at the last statement in B and scan backwards to the beginning of B. At each statement $i : x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Here we have used "+" as a symbol representing any operator. If the three-address statement i is of the form $x = +y$ or $x = y$, the steps are the same as above, ignoring z. Note that the order of steps (2) and (3) may not be interchanged because x may be y or x. □

24.6.2 Flow Graph

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a predecessor of C, and C is a successor of B. Often we add two nodes, called the entry and exit in the flow graph, which are not executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, i.e., to the basic block that comes from the first instruction. There is an edge to the exit from any basic block that contains an instruction. If the final instruction of the program is a conditional jump, then the block containing the final instruction of the program is one predecessor of the exit.

Example 24.3 *Construct a flow-graph from intermediate code in Table 24.1.*

The set of basic blocks constructed in Table 24.1 yields the flow graph of Fig. 24.2.

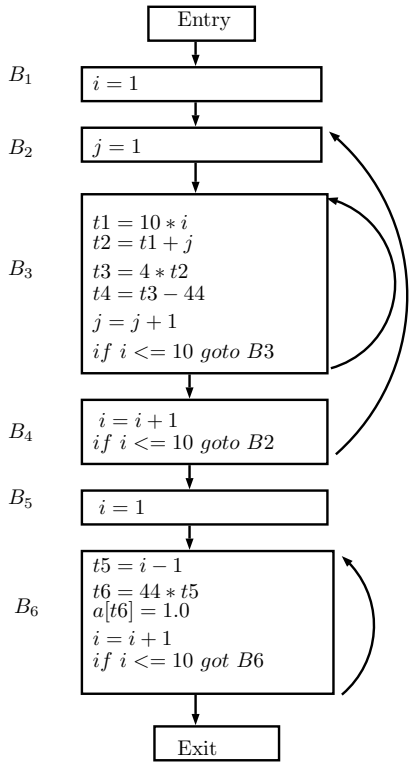

Figure 24.2: Flow graph from Table 24.1.

The entry points to the basic block B_1, as B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4. Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. □

24.6.3 Representation of Flow Graphs

First, note from Fig. 24.2 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks. Recall that every conditional or unconditional jump is directed to the leader instruction of some basic block. The reason for this change is that after constructing the flow graph, it is common to make substantial changes to the instructions in the various basic blocks. If jumps were to instructions, we would have to fix the targets of the jumps every time one of the target instructions was changed.

Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) need their own representation.

We might represent the content of a node by pointer to the leader, together with a count of the number of instructions or a second pointer to the last instruction. However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

24.6.4 Loops

The loops in programs are due to programming-language constructs like *while-statements*, *do-while-statements*, and *for-statements* naturally give rise to loops in programs. Since, virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a loop if

1. There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.

2. Every node in L has a nonempty path, completely within L, to the entry of L.

Example 24.4 *Analyse the loops in flow-graph in Figure 24.2.*

The flow graph of Fig. 24.2 has three loops:

1. B_3 by itself.

2. B_6 by itself.

3. $\{B_2, B_3, B_4\}$

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the second requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a nonempty path to B_2 staying within L. For instance, B_2 has the path $B_2 + B_3 + B_4 + B_2$. □

24.7 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself. More thorough global optimization, which looks at how information flows among the basic blocks of a program.

24.7.1 The DAG Representation of Basic Blocks

Many important techniques for *local optimization* begin by transforming a *basic block* into a DAG (directed acyclic graph). The idea of DAG extends naturally to the collection of expressions that are created within one basic block. We construct a DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
3. Node N is labelled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated *output nodes*. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

1. We can eliminate local common subexpressions, that is, instructions that compute a value that has already been computed.
2. We can eliminate dead code, that is, instructions that compute a value that is never used.
3. We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
4. We can apply algebraic laws to reorder operands of three-address instructions, and sometimes it simplify the computation.

24.7.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, of a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions earlier.

Example 24.5 *Construct the DAG for the block and find the common subexpressions.*

$$\begin{aligned} a &= b + c \\ b &= a - d \\ c &= b + c \\ d &= a - d \end{aligned} \tag{24.1}$$

The corresponding DAG is shown in Fig. 24.3. When we construct the node for the third statement $c = b + c$, we know that the use of b in $b + c$ refers to the node of Fig. 24.3 labeled $-$, because that is the most recent definition of b. Thus, we do not confuse the values computed at statements one and three.

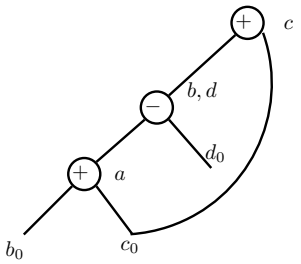

Figure 24.3: DAG for basic block in equation (24.1).

However, the node corresponding to the fourth statement $d = a - d$ has the operator $-$ and the nodes with attached variables a and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled $-$. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 24.3, the basic block can be replaced by a block with only three statements. In fact, if b is not live on exit from the block, then we do not need to compute that variable, and can use d to receive the value represented by the node labeled '$-$'. in Fig. 24.3. The block then becomes the following sequence of expressions:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned} \tag{24.2}$$

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.

Example 24.6 *Basic Block optimization for*

$$\begin{aligned} a &= b + c \\ b &= b - d \\ c &= c + d \\ e &= b + c \end{aligned} \tag{24.3}$$

When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence in equation 24.3 is the same, namely $b_0 + c_0$. That is, even though b and c both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 24.4, but does not exhibit any common subexpressions. However, *algebraic identities* applied to

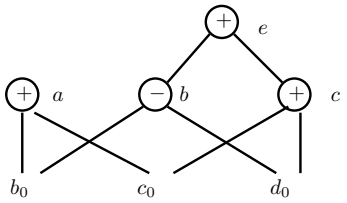

Figure 24.4: DAG for basic block in equation (24.3).

the DAG, are discussed later in next parts, may expose the equivalence. These entities are basically, $x + 0 = 0 + x = x$, $x/1 = x$, etc.

□

24.7.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example 24.7 *Dead Code Elimination.*

If, in Fig. 24.4, a and b are live but c and e are not, we can immediately remove the root labeled e. Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached. □

24.7.4 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three- address statements:

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$

If we think of $a[i]$ as an operation involving a and i, similar to $a + i$, then it might appear as if the two uses of $a[i]$ were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction $z = a[i]$ by the simpler $z = x$. However, since j could equal i, the middle statement may in fact change the value of $a[i]$; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like $x = a[i]$, is represented by creating a node with operator $= []$ and two children representing the initial value of the array, a_0 in this case, and the index i. Variable x becomes a label of this new node.

2. An assignment to an array, like $a[j] = y$, is represented by a new node with operator and three children representing a_0, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on a_0. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Example 24.8 *Construct a DAG for a sequence of array assignments, for basic block:*

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$

The DAG is is shown in Fig. 24.5.

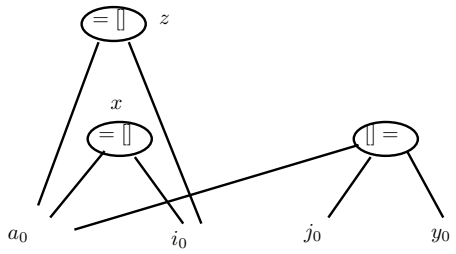

Figure 24.5: DAG for sequence of array assignments.

The node N for x is created first, but when the node labeled [] = is created, N is killed. Thus, when the node for x is created, it cannot be identified with N, and a new node with the same operands a_0 and i_0 must be created. □

24.8 Review Questions

1. What are the challenges in code generation phase of compiler, in respect of following:
   (a) Code generator it self
   (b) Language to be compiled
   (c) The hardware for which it is supposed to generate the code
2. In the code generation phase, what are the major subareas for applying the heuristics?
3. What are the primary tasks of the code generator?
4. Why it is important to segment the intermediate code into blocks?
5. Define block, and flow graph.
6. What are the advantages of representing blocks of code using DAGs?
7. What kind of optimizations are possible in intermediate code using DAGs?
8. How you will determine whether a value is live or not on exit of a block?
9. What are the different forms of target programs? Explain.

10. In what condition, the generated code is further optimized?
11. What feature of IR will contribute to more efficient object code?
12. What are the input to code generator?
13. What are the objectives of code generator?
14. What are the challenges of code generation?
15. Can the size of stack can be determined at compile time?
16. Which addresses can be determined at compile time?
17. Which addresses cannot be determined at compile time?
18. Explain the difference between static and dynamic allocation?
19. What is activation record?

24.9 Exercises

1. Below given is simple matrix addition program.

```
for(i=0; i < 10; i++}
   c[i][j] = a[i][j] + b[i][j];
```

2. Translate the program into three-address statements of the type. Assume the matrix entries are numbers that require 4 bytes, and that matrices are stored in column-major order.
3. Write the algorithm in pseudo code to determine liveness of variables in a block.
4. Construct the flow graph for your code from above.
5. Identify the loops in your flow graph from above.
6. Write your own version of algorithm for dead-code elimination?
7. Write the algorithm in pseudo code to construct the basic blocks out of a given program in intermediate representation.
8. Modify the code in Table 24.1 so that it works for 8-byte data values, consider the array of 5×5 of real numbers.
9. Modify the Table 24.1 so that it represents intermediate code for addition of two matrices of 10×10, with each element of 4-byte size.
10. Construct flow graph for above exercise.
11. Explain the value number method to determine common sub-expressions.
12. Generate the code for the following three-address statements assuming all variables are stored in memory locations.
    (a) $x = 5$

(b) $x = a$

(c) $x = a + 1$

(d) $x = a + b$

(e) $x = b * c$; $y = a + x$

13. Generate the code for the following three-address statements assuming a and b are array whose elements are 4-byte values.

14. The following statement sequence:

    $x = a[i]$

    $y = b[j]$

    $a[i] = y$

    $b[j] = x$

15. The following statement sequence:

    $x = a[i]$

    $y = b[i]$

    $z = x * y$

16. The following sequence:

    $x = a[i]$

    $y = b[x]$

    $a[i] = y$

17. Generate the code for the following three-address sequence assuming that p and q are in memory locations:

    $y = *q$

    $q = q + 4$

    $*p = y$

    $p = p + 4$

18. Generate code for the following sequence assuming that x, y, z are memory locations:

    $if\ x < y$ goto $L1$

    $z = 0$

    $goto\ L2$

    $L1 : z = 1$

19. Determine the cost of the following instruction sequence :

(a) $LD\ R0, y$

    $LD\ R1, z$

$ADD\ R0, R0, R1$

$ST\ x, R0$

(b) $LD\ R0, i$

$MUL\ R0, R0, 8$

$LD\ R1, a(R0)$

$ST\ b, R1$

20. Generate the code for the following three-address statements assuming stack allocation where register `SP` points to the top of the stack.

```
call p
call q
return
call r
return
return
```

21. Generate the code for the following three-address statements assuming stack allocation where register `SP` points to the top of stack.

    (a) $x = 5$
    (b) $x = a$
    (c) $x = a + 1$
    (d) $x = a + b$
    (e) $x = b * c$; $y = a + x$

22. Explain how the procedure call functions?

23. Given the three-address code, explain the step by step procedure to construct the basic blocks for that.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.