

Til INF 310, våren 2011

Fra Aho, Sethi, Ullman: "Compilers, ..."

9.2 THE TARGET MACHINE

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe the nuances of any target machine in sufficient detail to be able to generate good code for a complete language on that machine. In this chapter, we shall use as the target computer a register machine that is representative of several minicomputers. However, the code-generation techniques presented in this chapter have also been used on many other classes of machines.

Our target computer is a byte-addressable machine with four bytes to a word and n general-purpose registers, R_0, R_1, \dots, R_{n-1} . It has two-address instructions of the form

op source, destination

in which *op* is an op-code, and *source* and *destination* are data fields. It has the following op-codes (among others):

- MOV (move *source* to *destination*)
- ADD (add *source* to *destination*)
- SUB (subtract *source* from *destination*)

Other instructions will be introduced as needed.

The source and destination fields are not long enough to hold memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands and/or addresses. The source and destination of an instruction are specified by combining registers and memory locations with address modes. In the following description, *contents(a)* denotes the contents of the register or memory address represented by *a*.

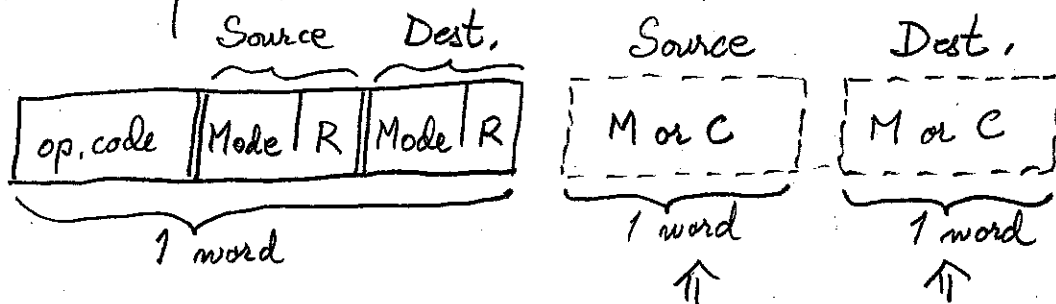
The address modes together with their assembly-language forms and associated costs are as follows:

MODE	FORM	ADDRESS	ADDED COST
<i>absolute</i>	M	M	1
<i>register</i>	R	R	0
<i>indexed</i>	$c(R)$	$c + \text{contents}(R)$	1
<i>indirect register</i>	*R	$\text{contents}(R)$	0
<i>indirect indexed</i>	* $c(R)$	$\text{contents}(c + \text{contents}(R))$	1
<i>literal</i>	#M	The <u>value</u> M	1

← Note the order. It is logical, but if a in R_0
 b in R_1
 then (b) (a)
 SUB R_1, R_0
 will result in:
 $R_0 = a - b$

(only for source)

Instruction format:



If M or C occur in the actual addressing mode.

A memory location M or a register R represents itself when used as a source or destination. For example, the instruction

```
MOV R0, M
```

stores the contents of register $R0$ into memory location M .

An address offset c from the value in register R is written as $c(R)$. Thus,

```
MOV 4(R0), M
```

stores the value

$contents(4 + contents(R0))$

into memory location M .

Indirect versions of the last two modes are indicated by prefix $*$. Thus,

```
MOV *4(R0), M
```

stores the value

$contents(contents(4 + contents(R0)))$

into memory location M .

A final address mode allows the source to be a constant:

MODE	FORM	CONSTANT	ADDED COST
literal	$\#c$	c	1

Thus, the instruction

```
MOV #1, R0
```

loads the constant 1 into register $R0$.

Instruction Costs

We take the cost of an instruction to be one plus the costs associated with the source and destination address modes (indicated as "added cost" in the table for address modes above). This cost corresponds to the length (in words) of the instruction. Address modes involving registers have cost zero, while those with a memory location or literal in them have a cost one, because such operands have to be stored with the instruction.

If space is important, then we should clearly minimize instruction length. However, doing so has an important additional benefit. For most machines and for most instructions, the time taken to fetch an instruction from memory exceeds the time spent executing the instruction. Therefore, by minimizing the instruction length we also tend to minimize the time taken to perform the instruction as well.² Some examples follow.

² The cost criterion is meant to be instructive rather than realistic. Allowing a full word for an instruction simplifies the rule for determining the cost. A more accurate estimate of the time taken by an instruction would consider whether an instruction requires the value of an operand, as well

added

1. The instruction `MOV R0, R1` copies the contents of register R0 into register R1. This instruction has cost one, since it occupies only one word of memory.
2. The (store) instruction `MOV R5, M` copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
3. The instruction `ADD #1, R3` adds the constant 1 to the contents of register 3, and has cost two, since the constant 1 must appear in the next word following the instruction.
4. The instruction `SUB 4(R0), *12(R1)` stores the value $\text{contents}(12 + \text{contents}(R1)) - \text{contents}(4 + R0)$ into the destination `*12(R1)`. The cost of this instruction is three, since the constants 4 and 12 are stored in the next two words following the instruction.

contents (4 + contents (R0))

Some of the difficulties in generating code for this machine can be seen by considering what code to generate for a three-address statement of the form $a := b + c$ where b and c are simple variables in distinct memory locations denoted by these names. This statement can be implemented by many different instruction sequences. Here are a few examples:

1. `MOV b, R0`
`ADD c, R0` cost = 6
`MOV R0, a`
2. `MOV b, a`
`ADD c, a` cost = 6

Assuming R0, R1, and R2 contain the addresses of a, b, and c, respectively, we can use:

3. `MOV *R1, *R0`
`ADD *R2, *R0` cost = 2

Assuming R1 and R2 contain the values of b and c, respectively, and that the value of b is not needed after the assignment, we can use:

4. `ADD R2, R1`
`MOV R1, a` cost = 3

We see that in order to generate good code for this machine, we must utilize its addressing capabilities efficiently. There is a premium on keeping the l- or r-value of a name in a register, if possible, if it is going to be used in the near future.

as its address (found with the instruction), to be fetched from memory.

9.4 BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. In Chapter 10, we use the flow graph of a program extensively as a vehicle to collect information about the intermediate program. Some register assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

```
t1 := a * a  
t2 := a * b
```

$$\begin{aligned}
 t_3 &:= 2 * t_2 \\
 t_4 &:= t_1 + t_3 \\
 t_5 &:= b * b \\
 t_6 &:= t_4 + t_5
 \end{aligned}
 \tag{9.1}$$

A three-address statement $x := y + z$ is said to *define* x and to *use* (or *reference*) y and z . A name in a basic block is said to be *live* at a given point if its value is used after that point in the program, (perhaps in another basic block) *may be*

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 9.1. Partition into basic blocks.

Input. A sequence of three-address statements.

Output. A list of basic blocks with each three-address statement in exactly one block.

Method.

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are the following.

- i) The first statement is a leader.
- ii) Any statement that is the target of a conditional or unconditional goto is a leader.
- iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program. \square

Example 9.3. Consider the fragment of source code shown in Fig. 9.7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in Fig. 9.8.

```

begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + prod a[i] * b[i];
    i := i + 1
  end
  while i <= 20
end
    
```

Fig. 9.7. Program to compute dot product.

← Definition of

- define x

- use x

- x is live

Note: Liveness is independent of basic blocks.

What about procedure calls?

We skip the treatment of those.

Let us apply Algorithm 9.1 to the three-address code in Fig. 9.8 to determine its basic blocks. Statement (1) is a leader by rule (i) and statement (3) is a leader by rule (ii), since the last statement can jump to it. By rule (iii) the statement following (12) (recall that Fig. 9.13 is just a fragment of a program) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block. □

```

(1) prod := 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a [ t1 ]      /* compute a[i] */
(5) t3 := 4 * i
(6) t4 := b [ t3 ]      /* compute b[i] */
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
    
```

Fig. 9.8. Three-address code computing dot product.

Flow Graphs

We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a *flow graph*. The nodes of the flow graph are the basic blocks. One node is distinguished as *initial*; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if

1. there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 , or
2. B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

We say that B_1 is a *predecessor* of B_2 , and B_2 is a *successor* of B_1 .

Example 9.4. The flow graph of the program of Fig. 9.7 is shown in Fig. 9.9. B_1 is the initial node. Note that in the last statement, the jump to statement (3) has been replaced by an equivalent jump to the beginning of block B_2 . □

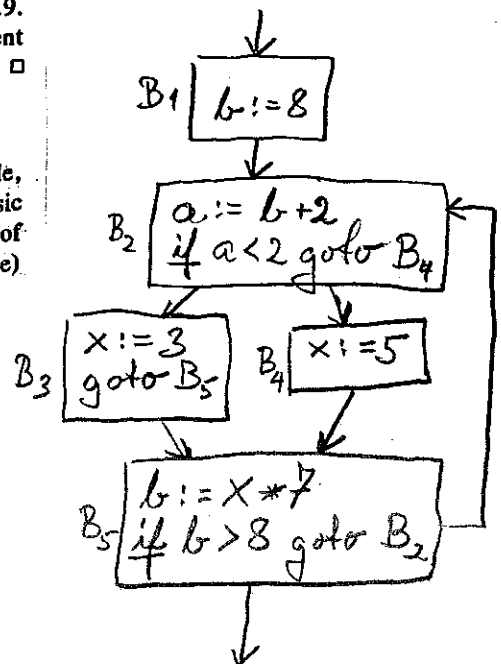
Representation of Basic Blocks

Basic blocks can be represented by a variety of data structures. For example, after partitioning the three-address statements by Algorithm 9.1, each basic block can be represented by a record consisting of a count of the number of quadruples in the block, followed by a pointer to the leader (first quadruple).

```

(1) b := 8
(2) a := b + 2
(3) if a < 2 goto (6)
(4) x := 3
(5) goto (7)
(6) x := 5
(7) b := x * 7
(8) if b > 8 goto (2)
    
```

Flow-graph ⇒



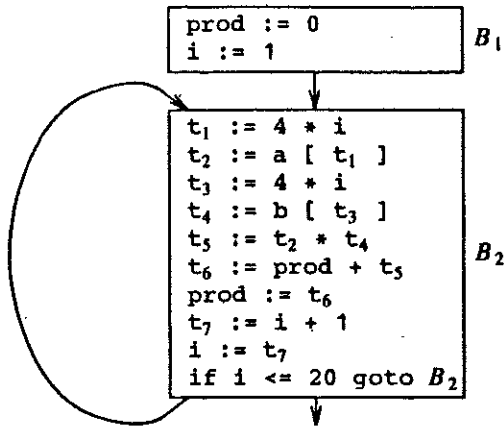


Fig. 9.9. Flow graph for program.

of the block, and by the lists of predecessors and successors of the block. An alternative is to make a linked list of the quadruples in each block. Explicit references to quadruple numbers in jump statements at the end of basic blocks can cause problems if quadruples are moved during code optimization. For example, if the block B_2 running from statements (3) through (12) in the intermediate code of Fig. 9.9 were moved elsewhere in the quadruple array or were shrunk, the (3) in `if i <= 20 goto (3)` would have to be changed. Thus, we prefer to make jumps point to blocks rather than quadruples, as we have done in Fig. 9.9.

It is important to note that an edge of the flow graph from block B to block B' does not specify the conditions under which control flows from B to B' . That is, the edge does not tell whether the conditional jump at the end of B (if there is a conditional jump there) goes to the leader of B' when the condition is satisfied or when the condition is not satisfied. That information can be recovered when needed from the jump statement in B .

Loops

In a flow graph, what is a loop, and how does one find all loops? Most of the time, it is easy to answer these questions. For example, in Fig. 9.9 there is one loop, consisting of block B_2 . The general answers to these questions, however, are a bit subtle, and we shall examine them in detail in the next chapter. For the present, it is sufficient to say that a loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are *strongly connected*; that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and

← We will not make use of this definition, but one should know it.

Note that in the handwritten example on previous page, the set $\{B_2, B_3, B_5\}$ is not a loop, as it can be entered at two places (see top next page) from nodes not in the loop:

- at B_2 (from B_1)
- at B_5 (from B_4)

However $\{B_2, B_3, B_4, B_5\}$ is a loop!

- The collection of nodes has a unique *entry*, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an *inner* loop.

← Better: innermost loop?

9.5 NEXT-USE INFORMATION

In this section, we collect next-use information about names in basic blocks. If the name in a register is no longer needed, then the register can be assigned to some other name. This idea of keeping a name in storage only if it will be used subsequently can be applied in a number of contexts. We used it in Section 5.8 to assign space for attribute values. The simple code generator in the next section applies it to register assignment. As a final application, we consider the assignment of storage for temporary names.

Computing Next Uses

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement *i* assigns a value to *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 *i*.

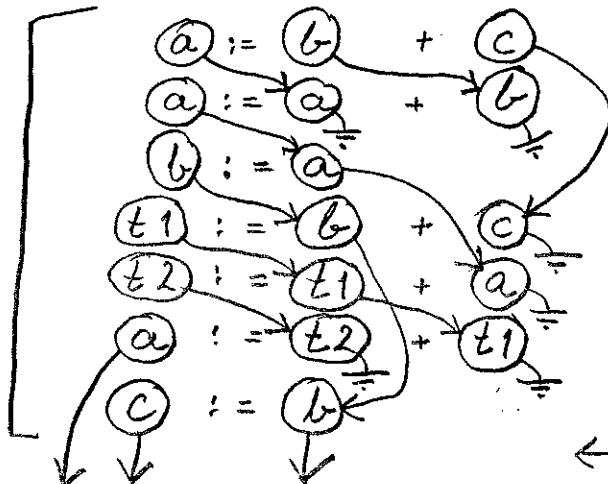
We wish to determine for each three-address statement $x := y \text{ op } z$ what the next uses of *x*, *y*, and *z* are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement but we may, if we wish, attempt to determine whether or not there is such a use by the live-variable analysis technique of Chapter 10.

Our algorithm to determine next uses makes a backward pass over each basic block. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 9.1. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Having found the end of a basic block, we scan backwards to the beginning, recording (in the symbol table) for each name *x* whether *x* has a next use in the block and if not, whether it is live on exit from that block. If the data-flow analysis discussed in Chapter 10 has been done, we know which names are live on exit from each block. If no live-variable analysis has been done, we can assume all nontemporary variables are live on exit, to be conservative. If the algorithms generating intermediate code or optimizing the code permit certain temporaries to be used across blocks, these too must be considered live. It would be a good idea to mark any such temporaries, so we do not have to consider all temporaries live.

Suppose we reach three-address statement *i*: $x := y \text{ op } z$ in our backward scan. We then do the following.

- Attach to statement *i* the information currently found in the symbol table



inside basic blocks

← We do no such analysis, so we must assume this.


← Result of Next Uses analysis

1) ⊗ ← Next use

2) ⊗ ← No next use, and x is not live.

3) ⊗ ← No next use, but x is live.

← Assume a, b, c is live here.

regarding the next use and liveness of x , ~~and 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 . Note that the order of steps (2) and (3) may not be interchanged because x may be y or z .

If three-address statement i is of the form $x := y$ or $x := op y$, the steps are the same as above, ignoring z .

Storage for Temporary Names

Although it may be useful in an optimizing compiler to create a distinct name each time a temporary is needed (see Chapter 10 for justification), space has to be allocated to hold the values of these temporaries. The size of the field for temporaries in the general activation record of Section 7.2 grows with the number of temporaries.

We can, in general, pack two temporaries into the same location if they are not live simultaneously. Since almost all temporaries are defined and used within basic blocks, next-use information can be applied to pack temporaries. For temporaries that are used across blocks, Chapter 10 discusses the data-flow analysis needed to compute liveness.

We can allocate storage locations for temporaries by examining each in turn and assigning a temporary to the first location in the field for temporaries that does not contain a live temporary. If a temporary cannot be assigned to any previously created location, add a new location to the data area for the current procedure. In many cases, temporaries can be packed into registers rather than memory locations, as in the next section.

For example, the six temporaries in the basic block (9.1) can be packed into two locations. These locations correspond to t_1 and t_2 in:

```

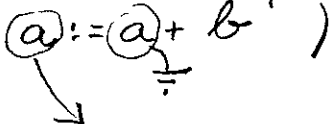
t1 := a * a
t2 := a * b
t2 := 2 * t2
t1 := t1 + t2
t2 := b * b
t1 := t1 + t2
    
```

9.6 A SIMPLE CODE GENERATOR

The code-generation strategy in this section generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact if possible. For simplicity, we assume that

⁴ If x is not live, then this statement can be deleted; such transformations are considered in Section 9.8.

2.5 Attach to y and z the information currently found in their symbol table for y and z . (Otherwise, it will not work for: $a := (a + b)$)



for each operator in a statement there is a corresponding target-language operator. We also assume that computed results can be left in registers as long as possible, storing them only (a) if their register is needed for another computation or (b) just before a procedure call, jump, or labeled statement.⁵

Condition (b) implies that everything must be stored just before the end of a basic block.⁶ The reason we must do so is that, after leaving a basic block, we may be able to go to several different blocks, or we may go to one particular block that can be reached from several others. In either case, we cannot, without extra effort, assume that a datum used by a block appears in the same register no matter how control reached that block. Thus, to avoid a possible error, our simple code-generation algorithm stores everything when moving across basic-block boundaries as well as when procedure calls are made. Later we consider ways to hold some data in registers across block boundaries.

← The general philosophy of this code generator algorithm

We can produce reasonable code for a three-address statement $a := b + c$ if we generate the single instruction `ADD Rj, Ri` with cost one, leaving the result a in register Ri . This sequence is possible only if register Ri contains b , Rj contains c , and b is not live after the statement; that is, b is not used after the statement.

If Ri contains b but c is in a memory location (called c for convenience), we can generate the sequence

```
ADD  c, Ri          cost = 2
```

or

```
MOV  c, Rj          cost = 3
ADD  Rj, Ri
```

provided b is not subsequently live. The second sequence becomes attractive if this value of c is subsequently used, as we can then take its value from register Rj . There are many more cases to consider, depending on where b and c are currently located and depending on whether the current value of b is subsequently used. We must also consider the cases where one or both of b and c is a constant. The number of cases that need to be considered further increases if we assume that the operator $+$ is commutative. Thus, we see that code generation involves examining a large number of cases, and which case should prevail depends on the context in which a three-address statement is seen.

← We do not utilize commutativity

⁵ However, to produce a *symbolic dump*, which makes available the values of memory locations and registers in terms of the source program's names for these values, it may be more convenient to have programmer-defined variables (but not necessarily compiler-generated temporaries) stored immediately upon calculation, should a program error suddenly cause a precipitous interrupt and exit.

⁶ Note we are not assuming that the quadruples were actually partitioned into basic blocks by the compiler; the notion of a basic block is useful conceptually in any event.

Register and Address Descriptors

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case.) As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.
2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method for a name.

← Usually, if the value of a user-defined variable is only in its "home position", then it has no address descriptor.

A Code-Generation Algorithm

The code generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement $x := y \text{ op } z$ we perform the following actions:

1. Invoke a function *getreg* to determine the location L where the computation $y \text{ op } z$ should be performed. L will usually be a register, but it could also be a memory location. We describe the details of *getreg* shortly.
2. Consult the address descriptor for y to determine y' , (one of) the current location(s) of y . Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L , generate the instruction $\text{MOV } y', L$ to place a copy of y in L , else
3. Generate the instruction $\text{OP } z', L$, where z' is a current location of z . Again, prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L . If L is a register, update its descriptor to indicate that it contains the value of x .
4. If the current values of y and/or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of $x := y \text{ op } z$, those registers no longer will contain y and/or z , respectively.

update descriptors so that y is no longer in L .

Move this to a last point 5, otherwise $a := a + b$ will not be treated properly.

If the current three-address statement has a unary operator, the steps are analogous to those above, and we omit the details. An important special case is a three-address statement $x := y$. If y is in a register, simply change the register and address descriptors to record that the value of x is now found only in the register holding the value of y . If y has no next use and is not

The following must obviously be an invariant:

values of variables that are live (with or without a next use) must exist at least in one location

live ~~on exit from the block~~, the register no longer holds the value of y .

If y is only in memory, we could in principle record that the value of x is in the location of y , but this option would complicate our algorithm, since we could not then change the value of y without preserving the value of x . Thus, if y is in memory we use *getreg* to find a register in which to load y and make that register the location of x .

Alternatively, we can generate a `MOV y, x` instruction, which would be preferable if the value of x has no next use in the block. It is worth noting that most, if not all, copy instructions will be eliminated if we use the block-improving and copy-propagation algorithm of Chapter 10.

Once we have processed all three-address statements in the basic block, we store, by `MOV` instructions, those names that are live on exit and not in their memory locations. To do this we use the register descriptor to determine what names are left in registers, the address descriptor to determine that the same name is not already in its memory location, and the live variable information to determine whether the name is to be stored. If no live-variable information has been computed by data-flow analysis among blocks, we must assume all user-defined names are live at the end of the block.

The Function *getreg*

The function *getreg* returns the location L to hold the value of x for the assignment `$x := y \text{ op } z$` . A great deal of effort can be expended in implementing this function to produce a perspicacious choice for L . In this section, we discuss a simple, easy-to-implement scheme based on the next-use information collected in the last section.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as `$x := y$` could cause a register to hold the value of two or more variables simultaneously), and y is not live and has no next use after execution of `$x := y \text{ op } z$` , then return the register of y for L . ~~Update the address descriptor of y to indicate that y is no longer in L .~~
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), if x has a next use in the block, or op is an operator, such as indexing, that requires a register, find an occupied register R . Store the value of R into a memory location (by `MOV R, M`) if it is not already in the proper memory location M , update the address descriptor for M , and return R . If R holds the value of several variables, a `MOV` instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.
4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L .

taking action to preserve.

← Wait until the end of step 2 of Code Gen.

For the statement
 $x := y \text{ op } x$
 we cannot use point (4) above, as this would produce the instructions:
`MOV y, x`
`<op> x, x`
 (which corresponds to:
 $x := y \text{ op } y$)

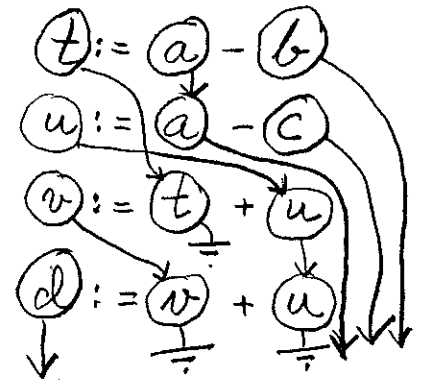
A more sophisticated *getreg* function would also consider the subsequent uses of *x* and the commutativity of the operator *op* in determining the register to hold the value of *x*. We leave such extensions of *getreg* as exercises.

Example 9.5. The assignment $d := (a - b) + (a - c) + (a - c)$ might be translated into the following three-address code sequence

```
t := a - b
u := a - c
v := t + u
d := v + u
```

Also a, b and c live at the end,

with *d* live at the end. The code generation algorithm given above would produce the code sequence shown in Fig. 9.10 for this three-address statement sequence. Shown alongside are the values of the register and address descriptors as code generation progresses. Not shown in the address descriptor is the fact that *a*, *b*, and *c* are always in memory. We also assume that *t*, *u* and *v*, being temporaries, are not in memory unless we explicitly store their values with a *MOV* instruction.



STATEMENTS	CODE GENERATED	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
		registers empty	
$t := a - b$	MOV a, R0 SUB b, R0	R0 contains t	t in R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
$v := t + u$	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Fig. 9.10. Code sequence.

The first call of *getreg* returns R0 as the location in which to compute *t*. Since *a* is not in R0, we generate instructions *MOV a, R0* and *SUB b, R0*. We now update the register descriptor to indicate that R0 contains *t*.

Code generation proceeds in this manner until the last three-address statement $d := v + u$ has been processed. Note that R1 becomes empty because *u* has no next use. We then generate *MOV R0, d* to store the live variable *d* at the end of the block.

The cost of the code generated in Fig. 9.10 is 12. We could reduce this to 11 by generating *MOV R0, R1* immediately after the first instruction and removing the instruction *MOV a, R1*, but to do so requires a more sophisticated code-generation algorithm. The reason for the savings is that it is cheaper to load R1 from R0 than from memory. □

(We do not consider arrays, pointers, procedure calls, or other similar constructions.)