SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC304J– COMPILER DESIGN

UNIT V

UNIT 5

- **BASIC BLOCKS,FLOW GRAPHS**
  - **NEXT-USE INFORMATION**
- **PRINCIPAL SOURCES OF OPTIMIZATION**
  - **FUNCTION PRESERVING TRANSFORMATION**
  - **LOOP OPTIMIZATION**
- **PEEPHOLE OPTIMIZATION**
- **INTRODUCTION TO GLOBAL DATA FLOW ANALYSIS**
  - **COMPUTATION OF GEN AND KILL**
  - **COMPUTATION OF IN AND OUT**
- **OPTIMIZTION OF BASIC BLOCKS**
  - **BUILDING EXPRESSION OF DAG**
- **RUNTIME ENVIRONMENTS**
  - **SOURCE LANGUAGE ISSUES**
  - **STORAGE ORGANIZATION**
  - **ACTIVATION RECORDS**
  - **STORAGE ALLOCATION STRATEGIES**
  - **PARAMETER PASSING**

Basic Blocks and Flow Graphs

Basic Blocks and Flow Graphs

- Graph representation of intermediate code is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm.
- We can do a better job of register allocation if we know how values are defined and used
- We can do a better job of instruction selection by looking at sequences of three-address statements
- The representation is constructed as follows:

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

Basic Block

- 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.

Partitioning three-address instructions into basic blocks

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. 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.

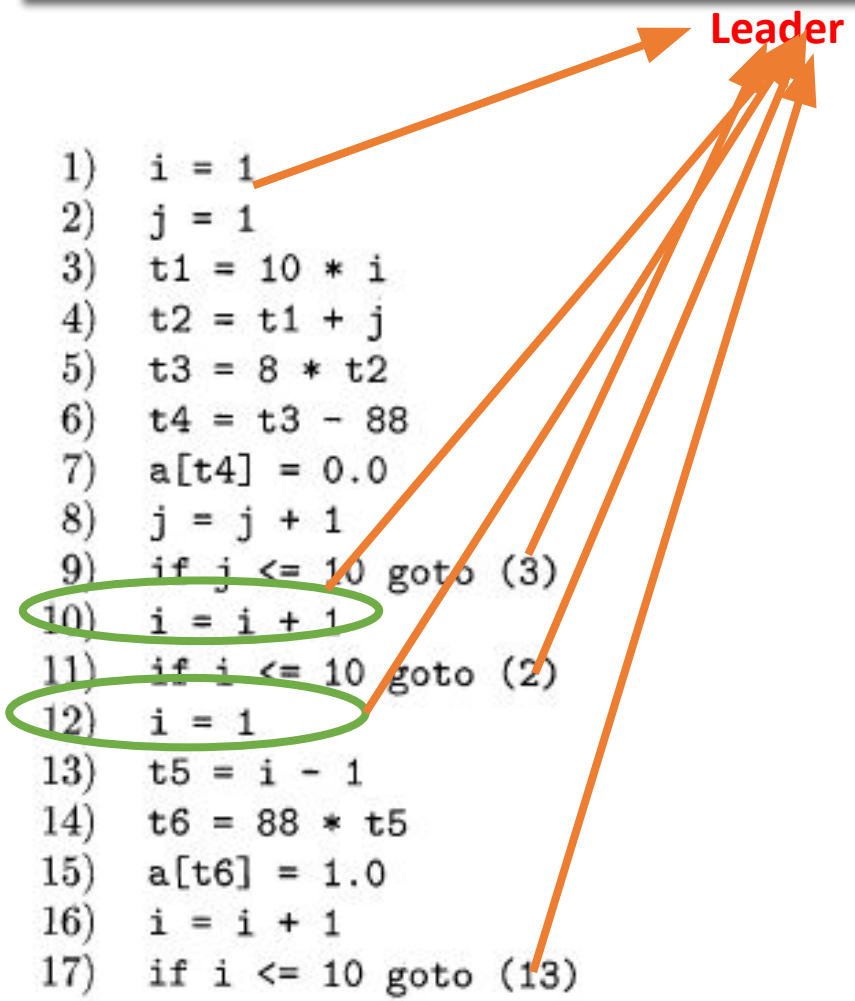
3. Any instruction that immediately follows a conditional or unconditional jump is a leader

Rule 1. The first three-address instruction in the intermediate code is a leader.

Rule 2. Any instruction that is the target of a conditional or unconditional jump is a leader.

Rule 3. Any instruction that immediately follows a conditional or unconditional jump is a leader

Example



- First, instruction 1 is a leader by rule (I) of Algorithm 8.5.
- 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.
- 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

Next-Use Information

- Knowing when the value of a variable will be used next is essential for generating good code
- 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 statement *i.*
- We further say that *x* is *live* at statement *i.*

Next-Use Information

Algorithm: 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

Flow Graphs

- If we have two block B and C
- 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:

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

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
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 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

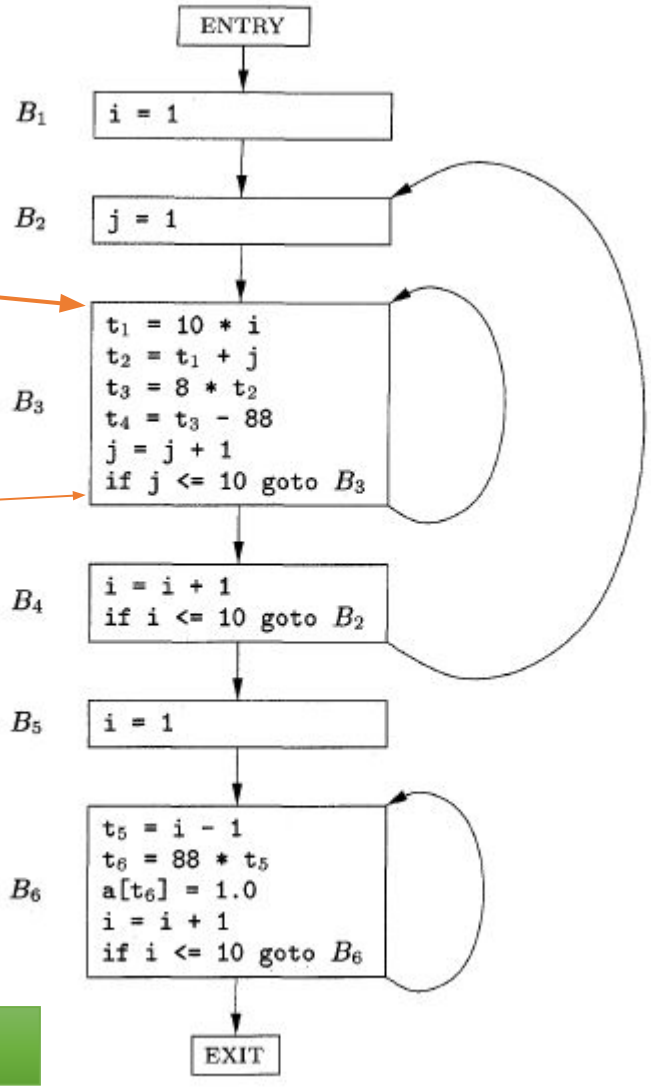


Figure 8.9: Flow graph from Fig. 8.7

The leaders are instructions 1, **2,** 3, 10, 12, and 13

The Principal Sources of Optimization

The Principal Sources of Optimization

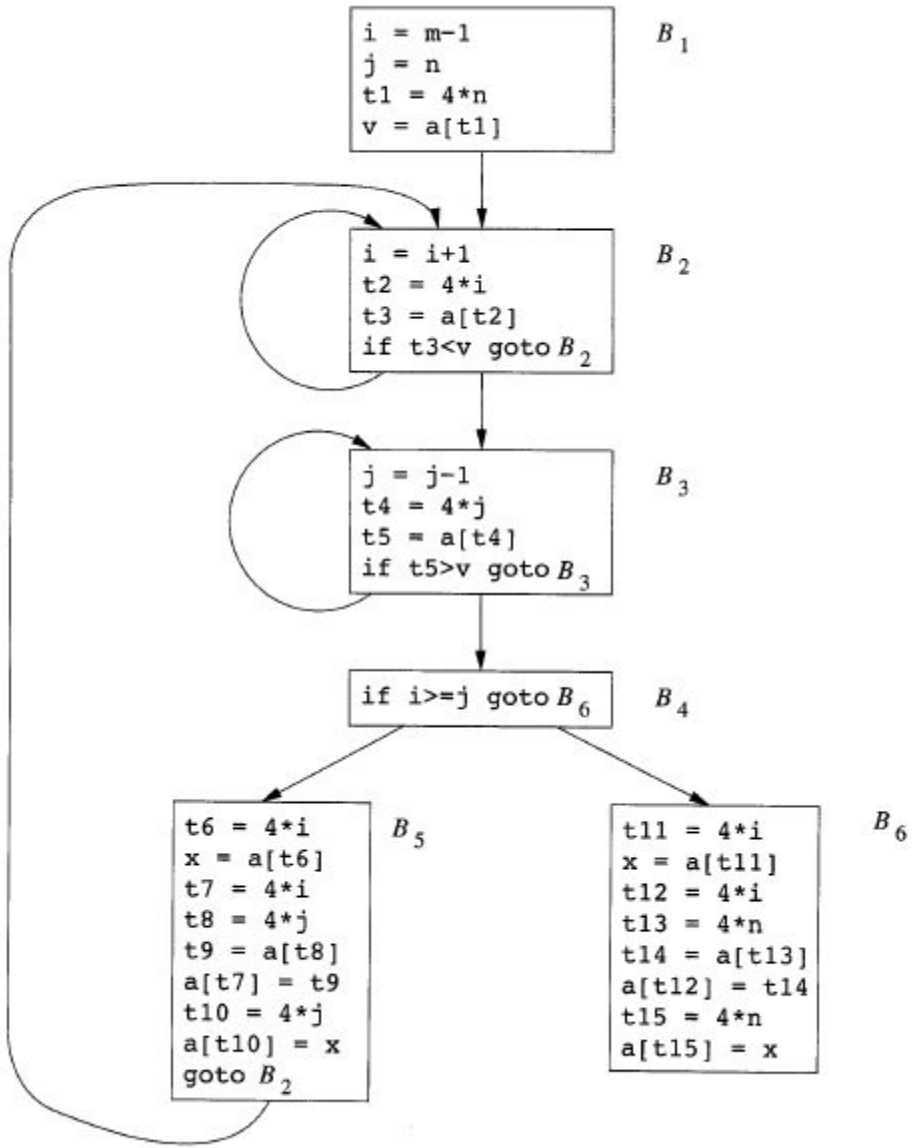
- A compiler optimization must preserve the semantics of the original program.
- Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.
- A compiler knows only how to apply relatively low-level semantic transformations, using general facts

Causes of Redundancy

- There are many redundant operations in a typical program.
- Sometimes the redundancy is available at the source level. For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary.
- But more often, the redundancy is a side effect of having written the program in a high-level language

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] 
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
(1)   i = m-1
(2)   j = n
(3)   t1 = 4*n
(4)   v = a[t1]
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
(13)  if i>=j goto (23)
(14)  t6 = 4*i
(15)  x = a[t6]
(16)  t7 = 4*i
(17)  t8 = 4*j
(18)  t9 = a[t8]
(19)  a[t7] = t9
(20)  t10 = 4*j
(21)  a[t10] = x
(22)  goto (5)
(23)  t11 = 4*i
(24)  x  = a[t11]
(25)  t12 = 4*i
(26)  t13 = 4*n
(27)  t14 = a[t13]
(28)  a[t12] = t14
(29)  t15 = 4*n
(30)  a[t15] = x
```

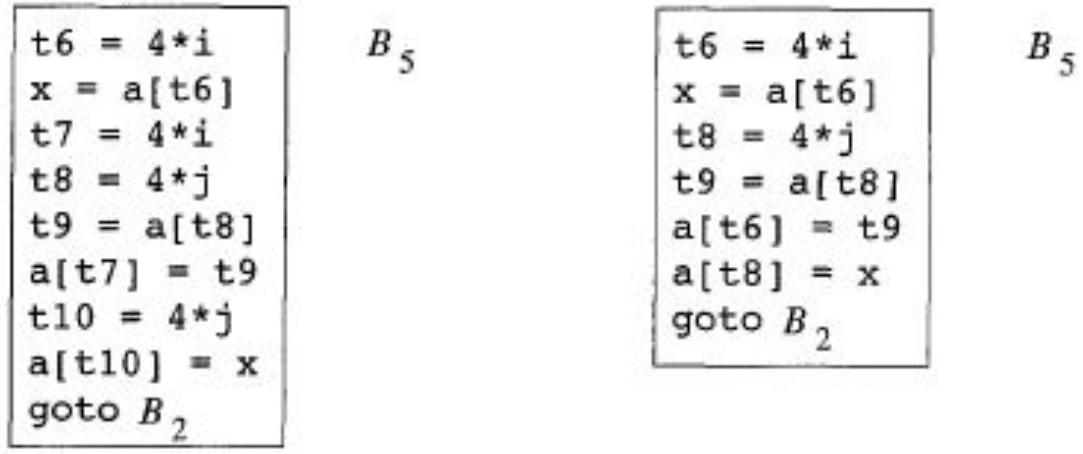

i = m-1
j = n
t1 = 4*n
v = a[t1]
B_1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B_2
B_2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B_3
B_3
if i>=j goto B_6
B_4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B_2
B_5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B_6

Semantics-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- Common-subexpression elimination,
- copy propagation,
- dead-code elimination, and constant folding
- are common examples of such function-preserving (or *semantics-preserving)* transformations

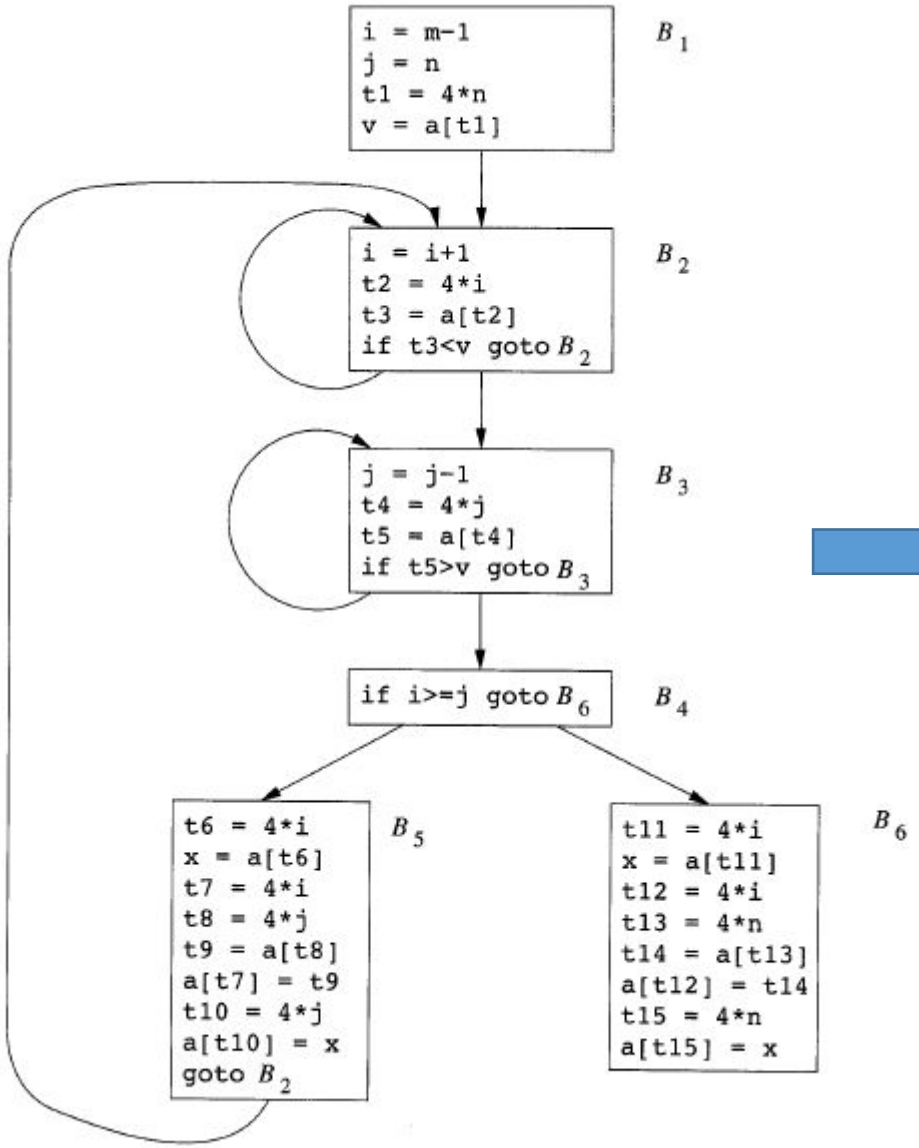
Common-sub expression elimination

- Local common sub expression elimination



(a) Before. (b) After.

Figure 9.4: Local common-subexpression elimination



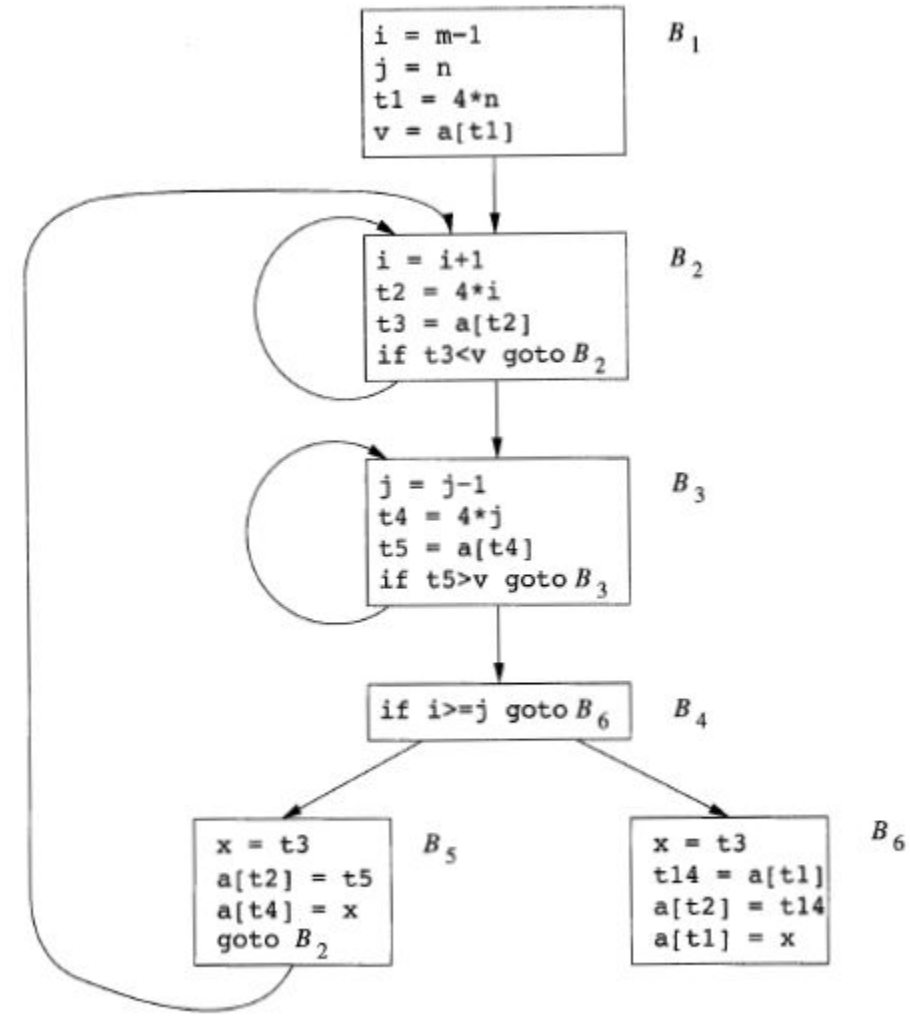


Figure 9.5: B_5 and B_6 after common-subexpression elimination

Copy Propagation

- Block B5 can be further improved by eliminating x, using two new transformations.
- One concerns assignments of the form u = v called copy statements, or copies for short
- The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v.

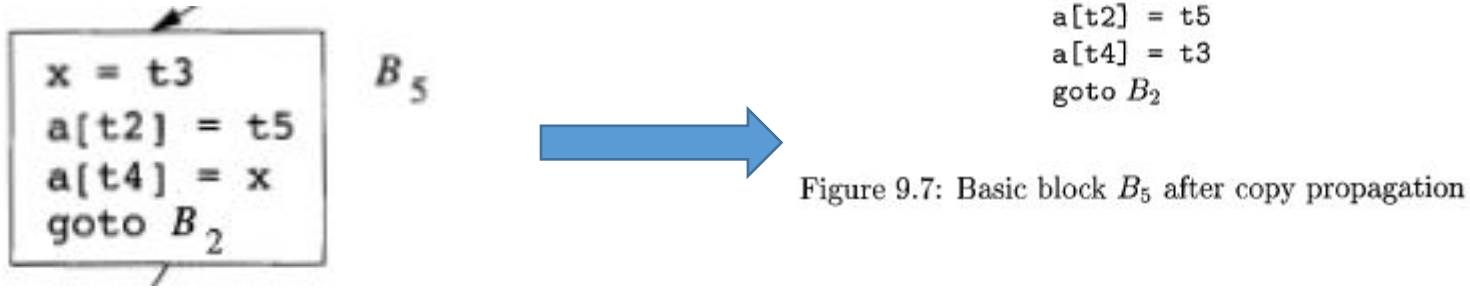


Figure 9.7: Basic block B_5 after copy propagation

Dead-Code Elimination

- A variable is *live* at a point in a program if its value can be used subsequently
- otherwise, it is *dead* at that point.
- A related idea is *dead* (or *useless) code* - statements that compute values that never get used.
- One advantage of copy propagation is that it often turns the copy statement into dead code

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

```
a[t2] = t5
a[t4] = t3
goto B2
```

Figure 9.7: Basic block B_5 after copy propagation

CodeMotion

- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- An important modification that decreases the amount of code in a loop is *code motion*

```
while (i <= limit-2) /* statement does not change limit */
```

Code motion will result in the equivalent code

```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

Induction Variables and Reduction in Strength

- Another important optimization is to find induction variables in loops and optimize their computation.
- A variable x is said to be an "induction variable“ if there is a positive or negative constant **c** such that each time x is assigned, its value increases by c.
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction.*

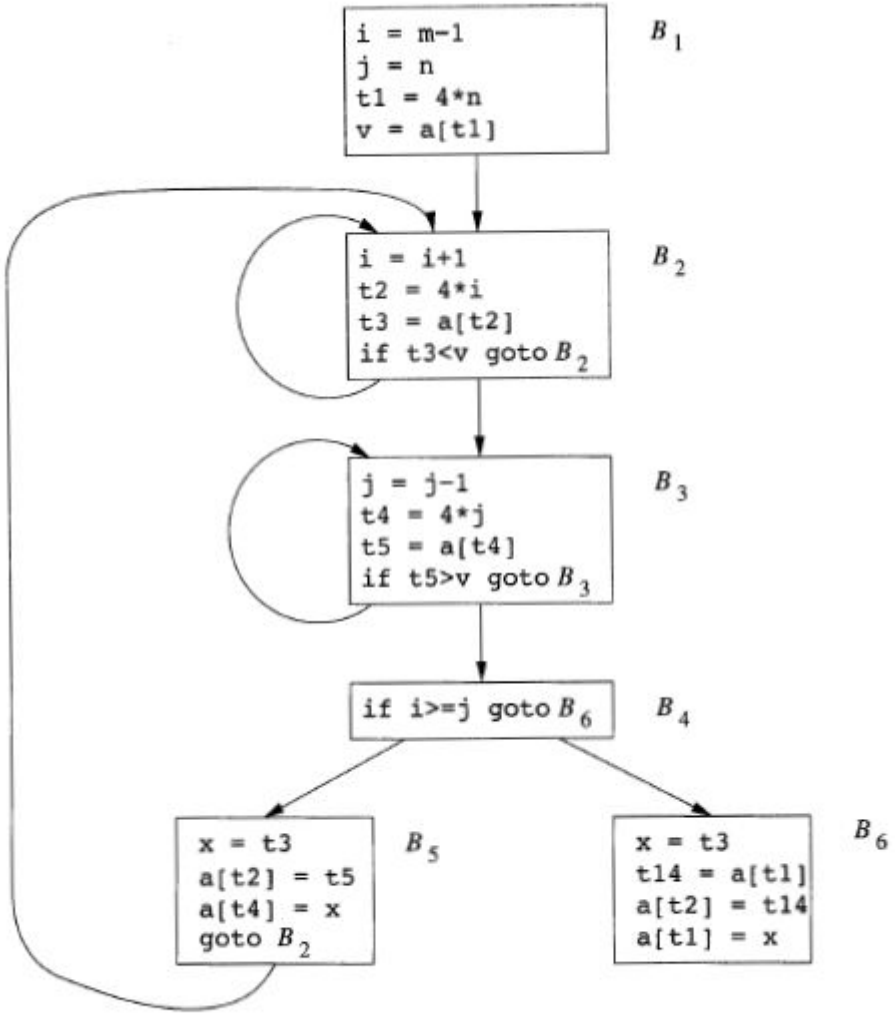


Figure 9.5: B_5 and B_6 after common-subexpression elimination

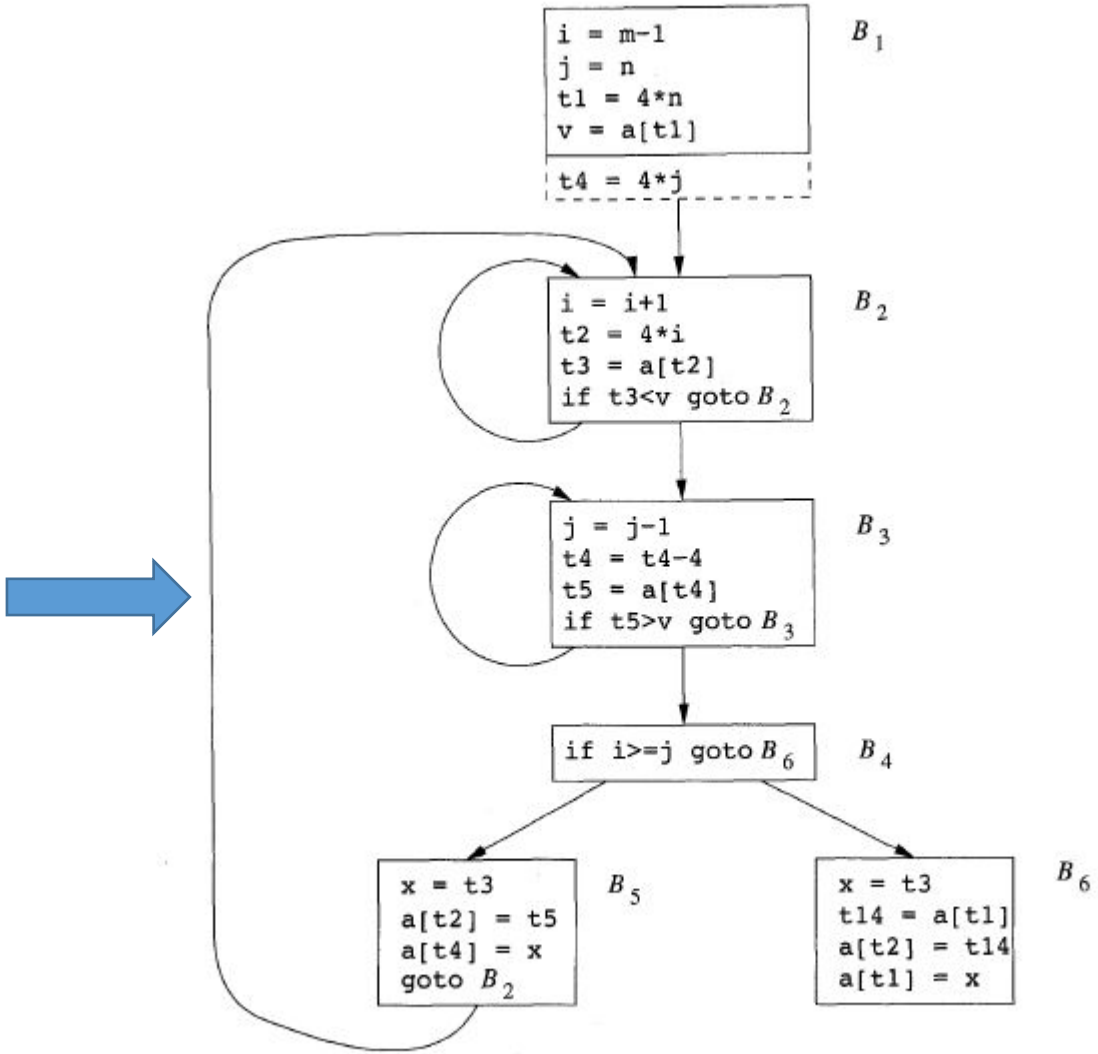


Figure 9.8: Strength reduction applied to $4 * j$ in block B_3

Loop Optimization

Loop Optimization

- **Loop optimization** is the process of increasing execution speed and reducing the overheads associated with Loops
- In loops, especially in the inner loops, programs tend to spend the bulk of their time.
- The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:

- Code motion, which moves code outside a loop
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- An important modification that decreases the amount of code in a loop is cc

```
while (i <= limit-2) /* statement does not change limit */
```

Code motion will result in the equivalent code

```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

Induction Variables and Reduction in Strength

- Another important optimization is to find **induction variables** in loops and optimize their computation.
- A variable x is said to be an "induction variable“ if there is a positive or negative constant **c** such that each time x is assigned, its value increases by c.
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as ***strength reduction***.

Induction Variables and Reduction in Strength

- For instance, ***i*** and *t2* are induction variables in the loop containing ***B2 (shown below)***.

```
i= i+1
t2 = 4*i
t3 = a[t2]
If t3 < v goto B2
```

(Block 2)

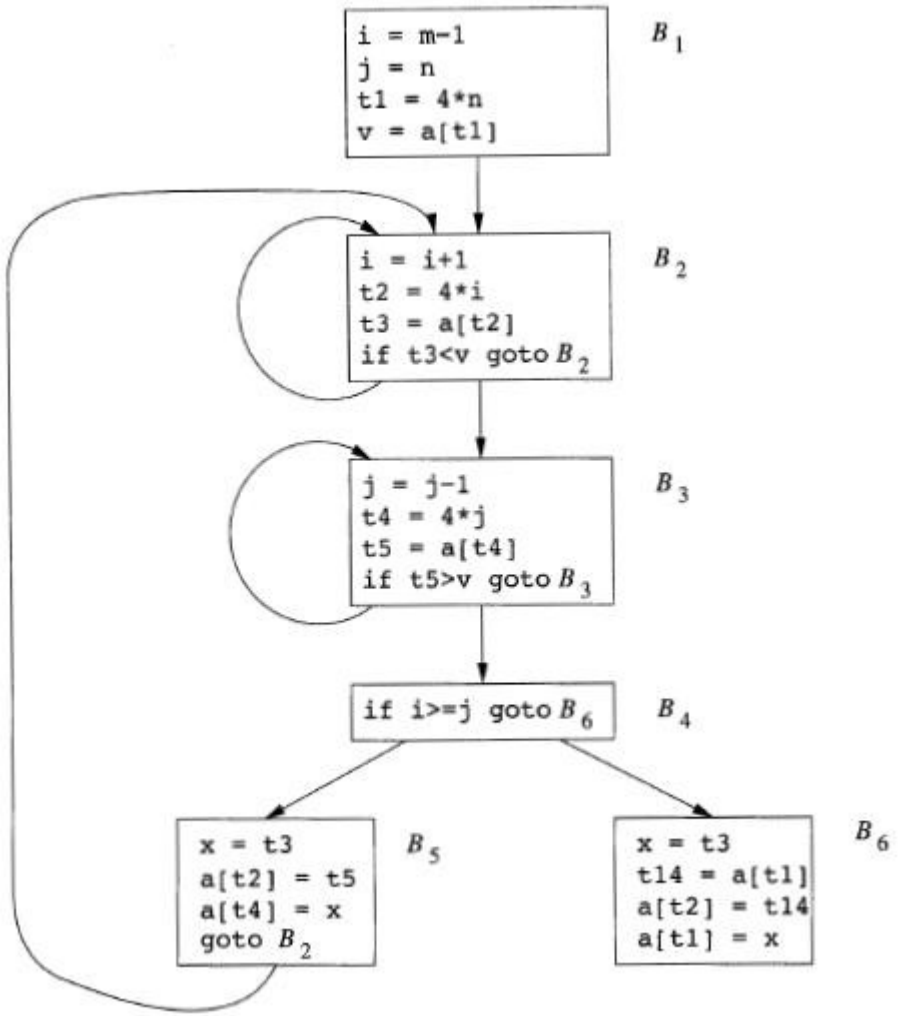
- induction variables not only allow us sometimes to perform a strength reduction;
- often it is possible to eliminate all but one of a group of induction variables whose values remain in **lock step** as we go around the loop.

Induction Variables and Reduction in Strength

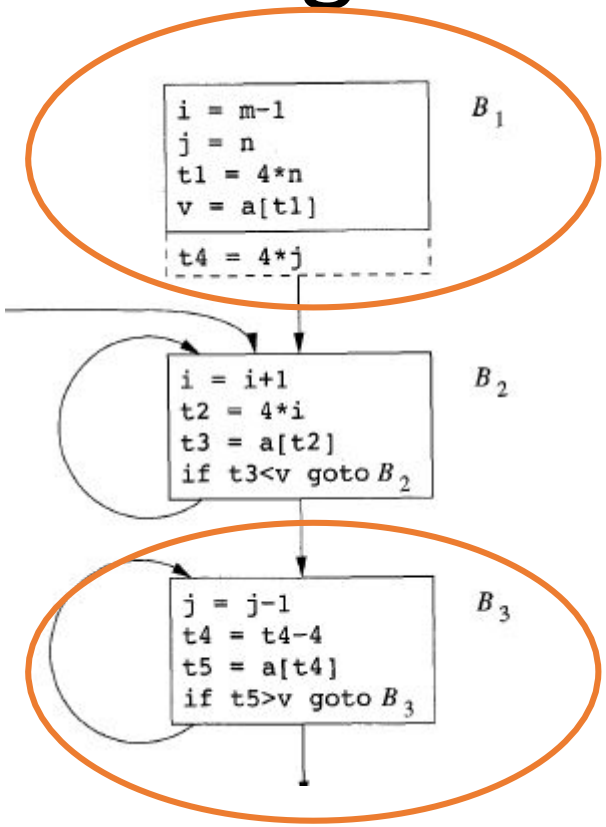
```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

(Block 3)

- Note that the values of j and t4 remain in lock step;
- every time the value of *j* decreases by 1, the value of t4 decreases by 4, because 4 * *j* is assigned to t4.
- These variables, j and t4, thus form a good example of a pair of induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one.
- Here in B3 , we cannot get rid of either j or t4 completely; t4 is used in *B3* and j is used in ***B4.***
- However, we can illustrate reduction in strength and a part of
the process of induction-variable elimination.



Induction Variables and Reduction in Strength

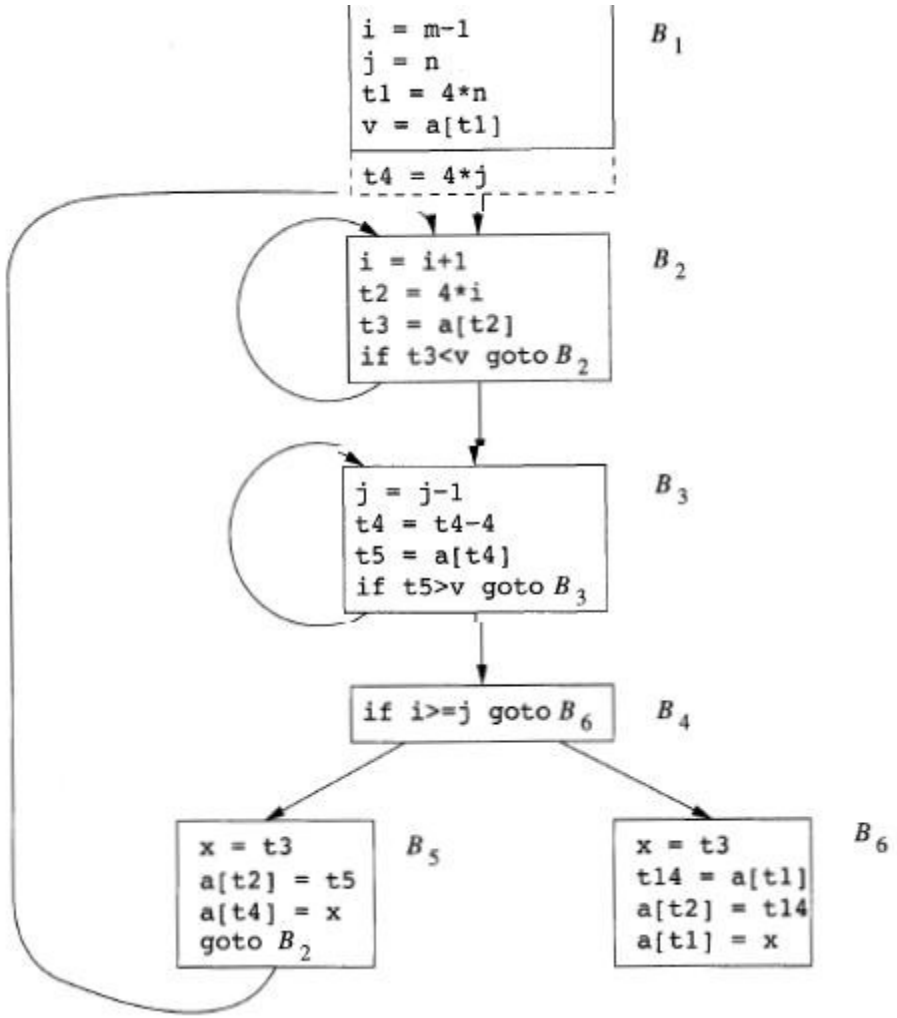


- As the relationship *t4 = 4 * j* surely holds after assignment to *t4* in and *t4* is not changed elsewhere in the inner loop around B3
- it follows that just after the statement j = j -1 the relationship *t4 = 4 * j – 4* must hold.
- **We may therefore replace the assignment t4 = 4* j by t4 = t4-4.**
- The only problem is that *t4* does not have a value when we enter block B3 for the first time.
- Since we must maintain the relationship *t4 = 4 * j* on entry to the block B3, we place an initialization of *t4* at the end of the block where j itself is initialized
- So in block B1 we can added one more instruction, which is executed once in block B1,
- **The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines**

Induction Variables and Reduction in Strength

```
i= i+1
t2 = 4*i
t3 = a[t2]
If t3 < v goto B2
```
(Block 2)

- After reduction in strength is applied to the inner loops around B2 and B3
- The only use of i and *j* is to determine the outcome of the test in block B4.
- We know that the values of i and *t2* satisfy the relationship
  *t2 = 4 * i,*
- while those of j and *t4* satisfy the relationship *t4 = 4* j.*
- Thus, the test *t2 ≥ t4* can substitute for i ≥ j.
- Once this replacement is made, i in block B2 and j in block B3 become dead variables,
- Then the assignments to them in these blocks become dead code that can be eliminated.



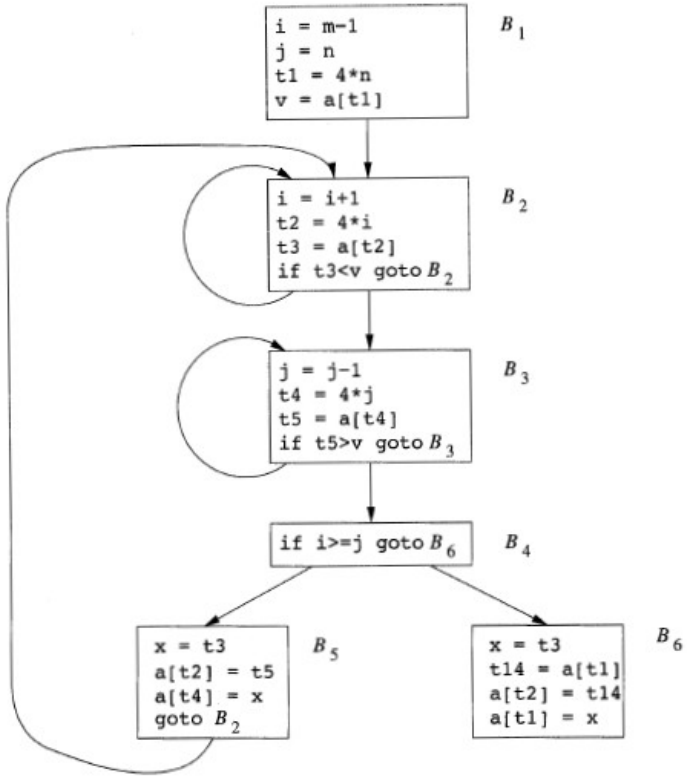
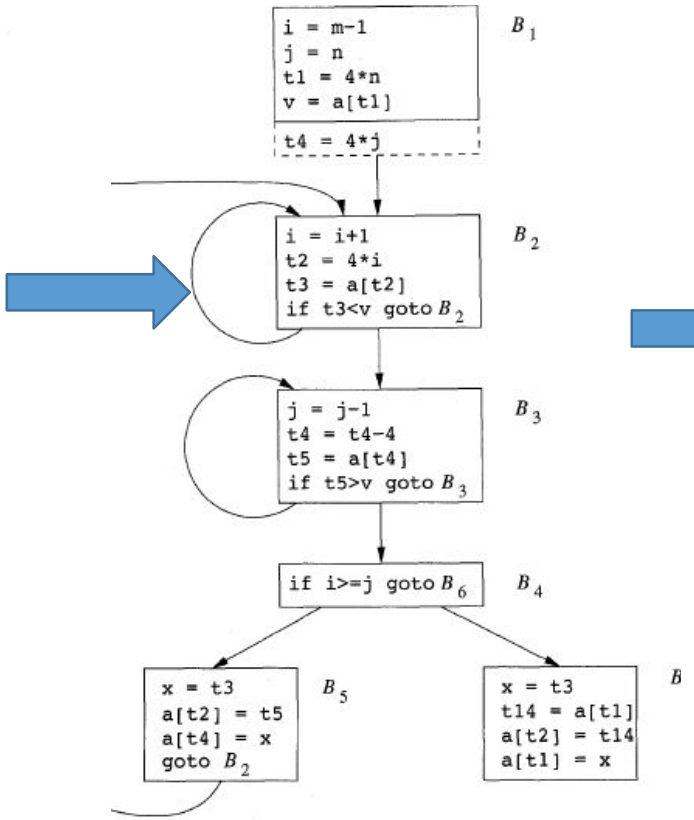


Figure 9.5: B_5 and B_6 after common-subexpression elimination



e 9.8: Strength reduction applied to $4 * j$ in block B_3

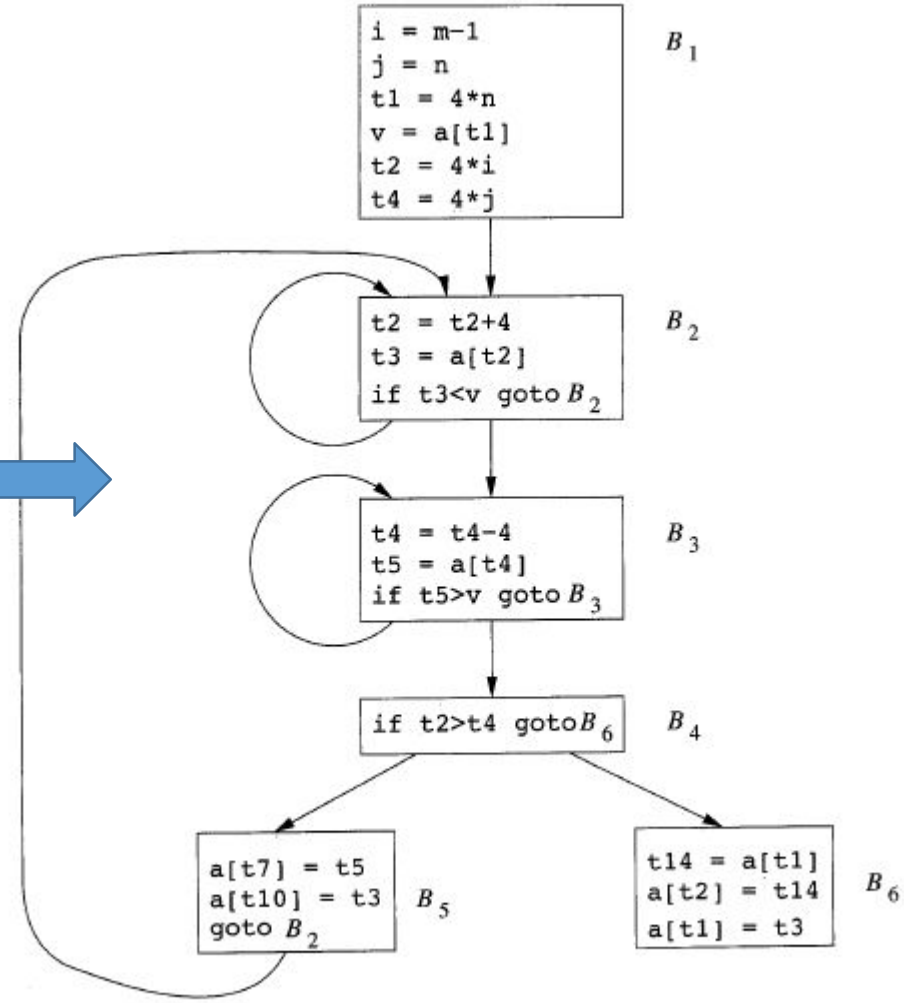


Figure 9.9: Flow graph after induction-variable elimination

Peephole Optimization

Peephole Optimization

- A simple but effective technique for locally improving the target code is peephole optimization
- This is done by examining a ***sliding window*** of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

Peephole Optimization

- we shall see the following examples of program transformations that are characteristic of peephole optimizations:

- **Redundant-instruction elimination**
- **Flow-of-control optimizations**
- **Algebraic simplifications**
- **Use of machine idioms**

Redundant instruction elimination

- At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

```
int add_ten(int x)
    {
    int y, z;
    y = 10;
    z = x + y;
    return z;
    }
```

```
int add_ten(int x)
    {
    int y;
    y = 10;
    y = x + y;
    return y;
    }
```

```
int add_ten(int x)
    {
    int y = 10;
    return x + y;
    }
```

```
int add_ten(int x)
    {
    return x + 10;
    }
```

- Eliminating Redundant Loads and Stores

```
LD a, R0
ST R0, a
```

Unreachable code

- Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

```
void add_ten(int x)
{
   return x + 10;
   printf("value of x is %d", x);
}
```

- In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

- There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

- In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

Algebraic expression simplification

- There are occasions where algebraic expressions can be made simple.
- For example, the expression **a = a + 0** can be replaced by **a** itself
- The expression a = a + 1 can simply be replaced by INC a.

Strength reduction

- There are operations that consume more time and space.
- Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

- For example,

 x * 2 can be replaced by **x << 1**, which involves only one left shift.

 Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Use of Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example,

 some machines have auto-increment and auto-decrement addressing modes.

 These add or subtract one from an operand before or after using its value.

 The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.

Introduction to Data-Flow Analysis

Introduction to Data-Flow Analysis

- All the optimizations depend on data-flow analysis.
- "Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths.
- For example, one way to implement global common sub-expression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program.
- As another example, if the result of an assignment is not used along any subsequent execution path, then we can eliminate the assignment as dead code.

The Data-Flow Abstraction - path

- The execution of an intermediate-code statement transforms an input state to a new output state.

✔ **The input state = the program point before the statement**

✔ **The output state = the program point after the statement**

- For analysing the behaviour of a program, we should consider all the possible sequences of program points -Paths
- This can be done through flow graph

Possible execution paths

- The flow graph tells us about the possible execution paths

Within one basic block, the program point after a statement is the same as the program point before the next statement.

If there is an edge from block B1 to block B2, then the program point after the last statement of B1 may be followed immediately by the program point before the first statement of B2.

Possible execution paths

We may define an execution path (or just path) from point p_1 to point p_n, to be a sequence of points $p_1, p_2, \ldots, P_n$ such that for each $i = 1,2, \ldots, n-1$, either

1. ***p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or***
2. ***p_i is the end of some block and p_{i+1} is the beginning of a successor block.***

"In general, there is an infinite number of possible execution paths through a program, and there is no finite upper bound on the length of an execution path"

The Data-Flow Analysis Schema

- In each application of data-flow analysis, we associate with every program point a data-flow value that represents an abstraction of the set of all possible program states that can be observed for that point.
- We denote the data-flow values before and after each statements by IN[S] and OUT[S], respectively.
- The data-flow problem is to find a solution to a set of constraints on the IN[s]'s and OUT[S]'s, for all statements.
- There are two sets of constraints:
  - Those based on the semantics of the statements ("transfer functions")
  - Those based on the flow of control.

Transfer Functions (semantics of the statements)

- The data-flow values before and after a statement are constrained by the semantics of the statement.
- For example,
  - If variable *a* has value *v* before executing statement b = a, then both *a* and b will have the value *v* after the statement.
  - This relationship between the data-flow values before and after the assignment statement is known as a transfer function.

Transfer Functions

- Transfer functions come in two flavors: information may propagate forward along execution paths, or it may flow backwards up the execution paths
- In a **forward-flow problem**, the transfer function of a statement s, which we shall usually denote f_s takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,

$$OUT[S] = F_s(IN[S])$$

- Conversely, in a **backward-flow problem**, the transfer function *f,* for statement s converts a data-flow value

$$IN[S] = F_s(OUT[S])$$

Control-Flow Constraints

- Within a basic block, control flow is simple.
- If a block B consists of statements s_1 , s_2, . . . , S_n in that order, then the control-flow value out of s_i is the same as the control-flow value into s_{i+1}. That is,

$$IN[S_{i+1}] = OUT[S_i], \text{ for all } i = 1,2,3...n-1$$

- *Control-flow edges between basic blocks create more complex constraints between the last statement of one basic block and the first statement of the following block.*

Data-Flow Schemas on Basic Blocks

- We denote the data-flow values immediately before and immediately after each basic block B by IN[B] and OUT[B],respectively.
- The constraints involving IN[B]and OUT[B] can be derived from those involving IN[S] and OUT[S] for the various statements s in B as follows.
- Suppose block B consists of statements $s_1, \ldots, s_n$ in that order.
- If s_1 is the first statement of basic block B, then

$$IN[B] = IN[s_1],$$

Data-Flow Schemas on Basic Blocks

- Similarly, if s_n is the last statement of basic block B, then

$$OUT[B] = OUT[S_n].$$

- The transfer function of a basic block B, which we denote f_B, can be derived by composing the transfer functions of the statements in the block.
- That is, let f_{Si} be the transfer function of statement s_i.
- Then $f_B = f_{sn} \; o \; f_{s(n-1)} \ldots o \; f_{s2} \; o \; f_{s1}$.
- The relationship between the beginning and end of the block is

$$OUT[B] = f_B(IN[B])$$

Data-Flow Schemas on Basic Blocks

- The constraints due to control flow between basic blocks can easily be rewritten by substituting IN[B] and OUT[B] for IN[s_1] and OUT[S_n], respectively.
- For instance, if data-flow values are information about the sets of constants that *may* be assigned to a variable, then we have a forward- flow problem in which

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Data-Flow Schemas on Basic Blocks

- The data-flow is backwards as we shall soon see in live-variable analysis, the equations are similar, but with the roles of the IN'S and OUT'S reversed.

$$\text{IN}[B] = f_B(\text{OUT}[B])$$

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S].$$

Reaching Definitions

- "Reaching definitions" is one of the most common, and useful data- flow schemas.
- For Example:
- By knowing where in a program each variable x may have been defined when control reaches each point pi we can determine many things about x.
  - For just two examples, a compiler then knows whether x is a constant at point p, and
  - a debugger can tell whether it is possible for x to be an undefined variable, should x be used at p.

Reaching Definitions

- Let in a block definition *d reaches* a point *p* if there is a path from the point immediately following d to p_i such that d is not "killed" along that path.
- We kill a definition of a variable x if there is any other definition of x anywhere along the path.
- A definition of a variable x is a statement that assigns, or may assign, a value to x .
- Procedure parameters, array accesses, and indirect references all may have aliases, and it is not easy to tell if a statement is referring to a particular variable x

Reaching Definitions : Example

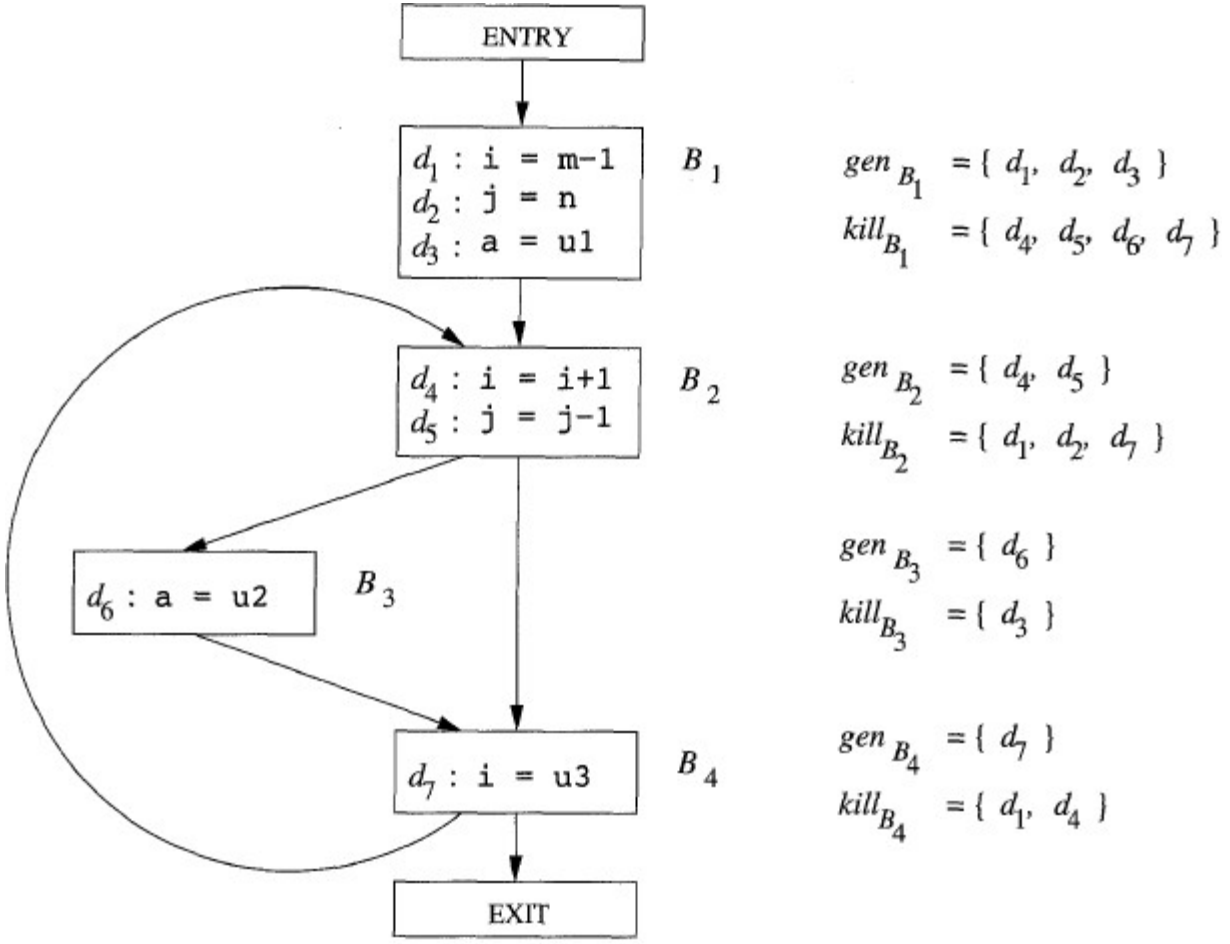


Figure 9.13: Flow graph for illustrating reaching definitions

- The figure is a flow graph with seven definitions.
- Let us focus on the definitions reaching block B2.
- All the definitions in block B1 reach the beginning of block B2.
- The definition d5: j = **j-1** in block B2 also reaches the beginning of block B2, because no other definitions of *j* can be found in the loop leading back to B2.
- This definition, however, kills the definition d2: j = n, preventing it from reaching B3 or B4.
- The statement **d4: i = i+l** in B2 does not reach the beginning of B2 though, because the variable ***i*** is always redefined by d7: i = **u3.**
- Finally, the definition d6 : a = **u2** also reaches the beginning of block B2

Transfer Equations for Reaching Definitions

- Consider a statement

d: u=v+w

- This statement "generates" a definition d of variable u and "kills" all the other definitions in the program that define variable u, while leaving the remaining incoming definitions unaffected.

Transfer Equations for Reaching Definitions

- The transfer function of definition d thus can be expressed as

$$f_d(x) = gen_d \cup (x - kill_d)$$

- If there are two transfer function in a basic block it can be represented a

$$f_1(x) = gen_1 \cup (x - kill_1) \text{ and } f_2(x) = gen_2 \cup (x - kill_2).$$

$$\begin{aligned} f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2) \\ &= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2)) \end{aligned}$$

Transfer Equations for Reaching Definitions

- Thus a block consisting of any number of statements**. Suppose block B has *n* statement**. The equation can be represented as below:

$$f_B(x) = gen_B \cup (x - kill_B),$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

and

$$\begin{aligned} gen_B &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ &\quad \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$

Transfer Equations for Reaching Definitions

–

Basic Block

- A basic block generates a set of definitions and kills a set of definitions.
- The *gen* set contains all the definitions inside the block that are "visible" immediately after the block - we refer to them as *downwards exposed.*
- A definition is downwards exposed in a basic block only if it is not "killed" by a subsequent definition to the same variable inside the same basic block.
- A basic block's *kill* set is simply the union of all the definitions killed by the individual statements.
- The notable fact is *gen* takes precedence, because in *gen-kill* form, the *kill* set is applied before the *gen* set

Control-Flow Equations

- Consider the set of constraints derived from the control flow between basic blocks.
- since a definition cannot reach a point unless there is a path along which it reaches, IN[B] needs to be no larger than the union of the reaching definitions of all the predecessor blocks

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$

Iterative Algorithm for Reaching Definitions

- Let consider every control-flow graph has two empty basic blocks ENTRY and EXIT node
- Initially consider $\text{OUT}[\text{ENTRY}] = \emptyset$
- for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Iterative Algorithm for Reaching Definitions

- **INPUT: A** flow graph for which $kill_B$ and gen_B have been computed for each block B.
- **OUTPUT:** IN[B] and OUT[B], the set of definitions reaching the entry and exit of each block B of the flow graph

```
1)  OUT[ENTRY] = ∅;
2)  for (each basic block B other than ENTRY) OUT[B] = ∅;
3)  while (changes to any OUT occur)
4)      for (each basic block B other than ENTRY) {
5)          IN[B] = ⋃_{P a predecessor of B} OUT[P];
6)          OUT[B] = gen_B ∪ (IN[B] − kill_B);
        }
```

Exampl e

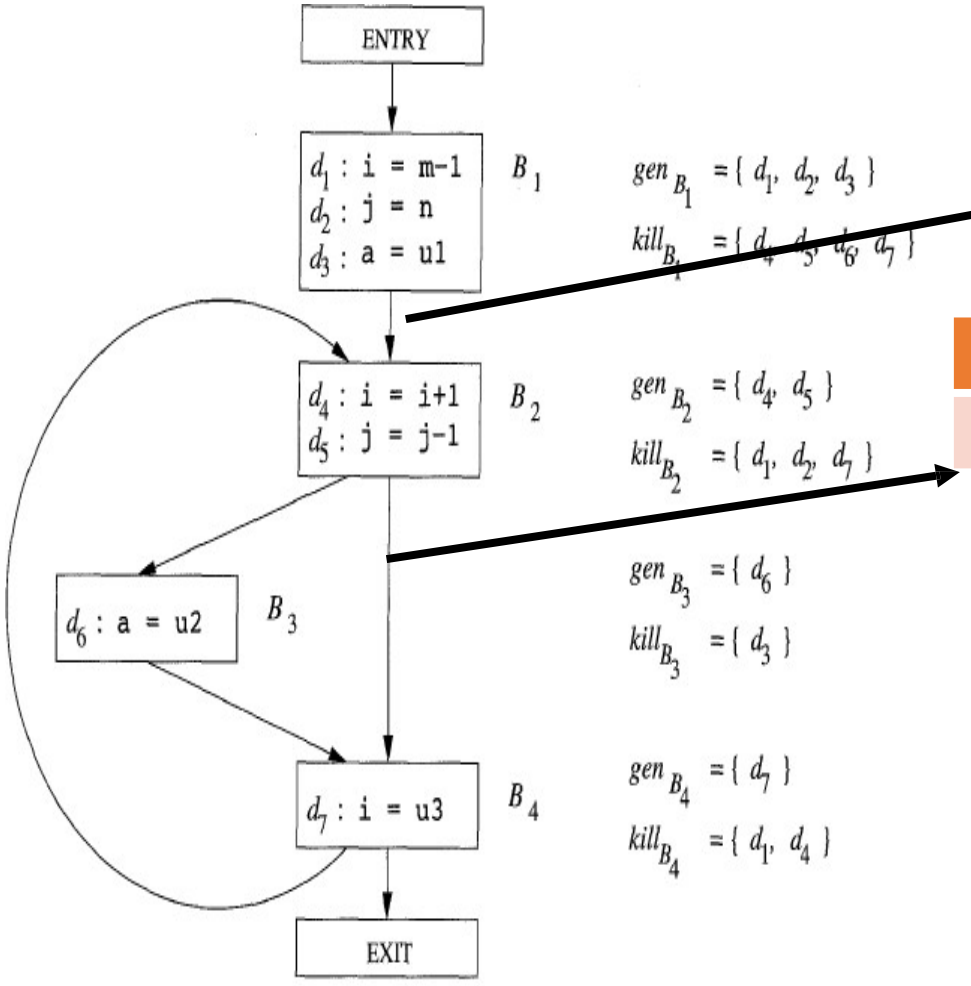


Figure 9.13: Flow graph for illustrating reaching definitions

d1	d2	d3	d4	d5	d6	d7
1	1	1	0	0	0	0

d1	d2	d3	d4	d5	d6	d7
0	0	1	1	1	0	0

Block B	OUT[B]0	IN[B]1	OUT[B]1
B1	000 0000	000 0000	111 0000
B2	000 0000	111 0000	001 1100
B3	000 0000	001 1100	000 1110
B4	000 0000	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111

Example

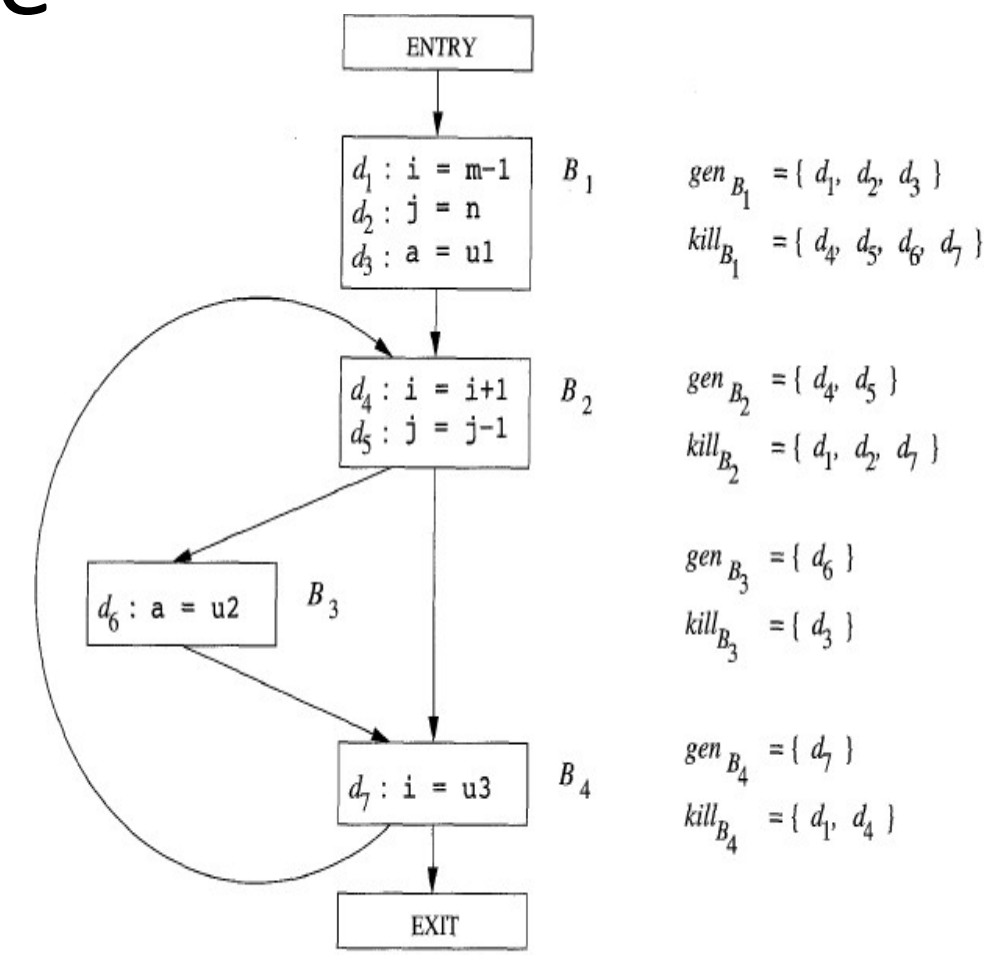


Figure 9.13: Flow graph for illustrating reaching definitions

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Figure 9.15: Computation of IN and OUT

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Then we consider $B = B_2$ and compute

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= gen[B_2] \cup (\text{IN}[B_2]^1 - kill[B_2]) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

Exampl e

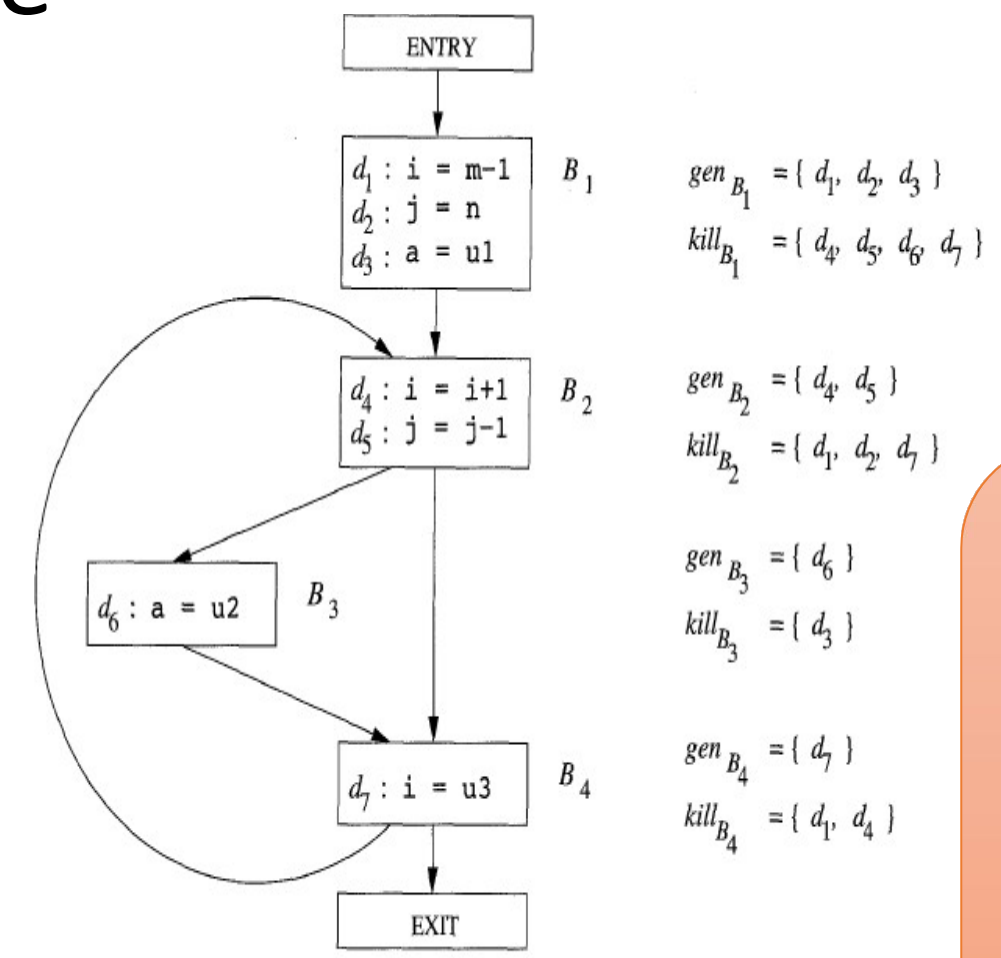


Figure 9.13: Flow graph for illustrating reaching definitions

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Figure 9.15: Computation of IN and OUT

- At the end of the first pass, 0UT[B2]1 = 001 1100, reflecting the fact that *d4* and *d5* are generated in B2, while *d3* reaches the beginning of B2 and is not killed in B2.
- Notice that after the second round, 0UT[B2] has changed to reflect the fact that *d6* also reaches the beginning of B2 and is not killed by B2.
- We did not learn that fact on the first pass, because the path from *d6* to the end of B2, which is *B3* → B4 → B2, is not traversed in that order by a single pass.
- That is, by the time we learn that *d6* reaches the end of B4, we have already computed IN[B2] and OUT[B2] on the first pass.

Optimization of Basic Blocks

The DAG Representation of Basic Blocks - The need

- One of the important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).

Condition for Basic blocks	DAG Representation
For each of the initial values of the variables appearing in the basic block	Considered as **Node**
With each statement s within the block	Considered **Node N** is associated with s (statement)
Statements that are the last definitions, prior to s, and of the operands used by s	Considered **Children's of N**
Variables which are *live on exit* from the block	Considered as ***output nodes***

Calculation of these "live variables" is a matter for global flow analysis

The DAG Representation of Basic Blocks

Using DAG following code improving transformations can be done

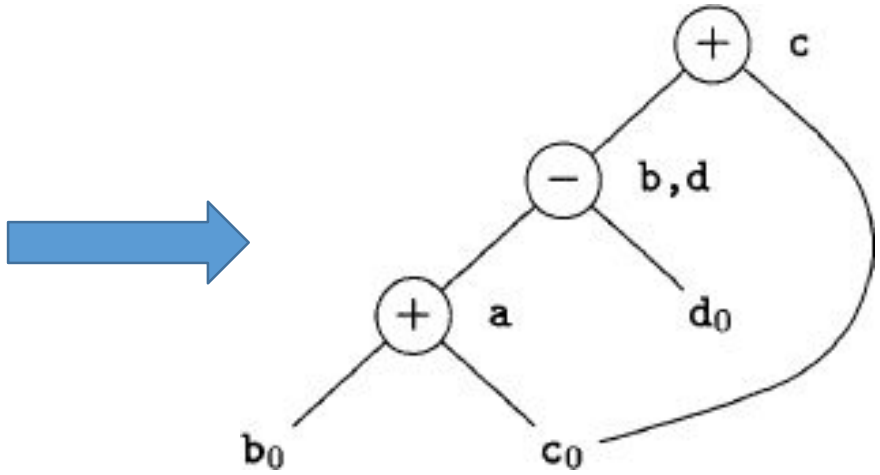
1. We can **eliminate *local common sub-expressions***, that is, instructions that compute a value that has already been computed.
2. We can **eliminate *dead code***, that is, instructions that compute a value that is never used.
3. We can **reorder statements** that do not depend on one another; such 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 hereby simplify the computation

Finding Local Common Sub expressions

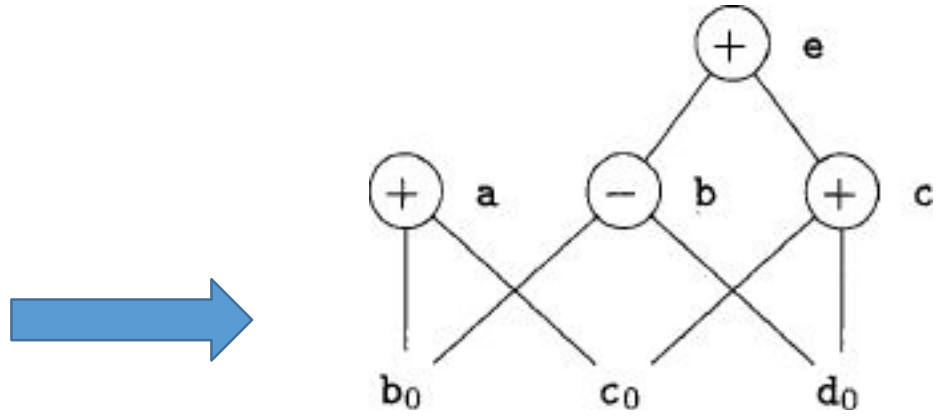
- Common sub-expressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator.
- If so, N computes the same value as M and may be used in its place.
- This technique was introduced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

Example

```
a = b + c
b = a - d
c = b + c
d = a - d
```

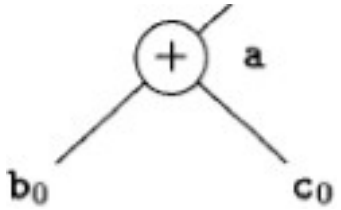



```
a = b + c;
b = b - d
c = c + d
e = b + c
```

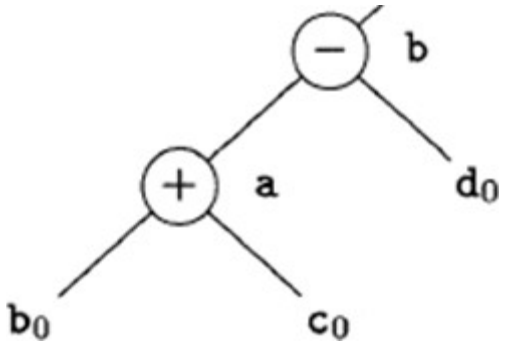


1. a = b + c
2. b = a - d
3. c = b + c
4. d = a - d

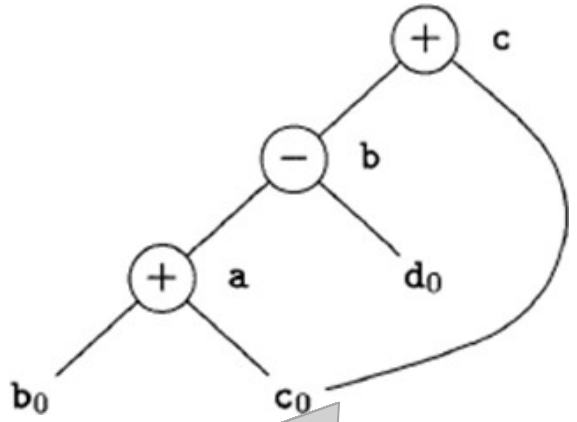
1. a = b + c



2. b = a - d

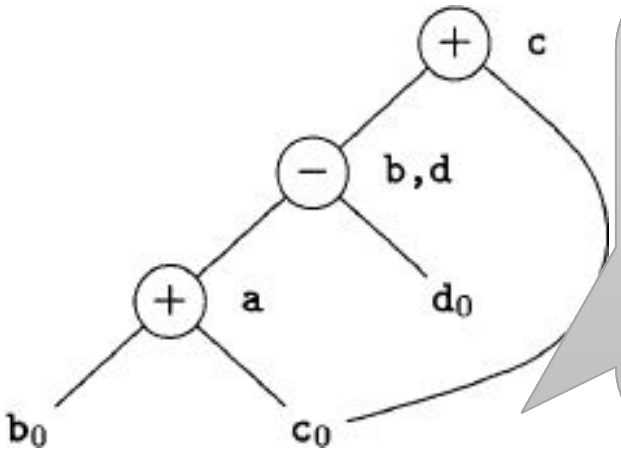


3. c = b + c



- Here b is already used in statement (2) and labelled -
- Also it is the most recent definition of b.
- Thus, we do not confuse the values computed at statements one and three.

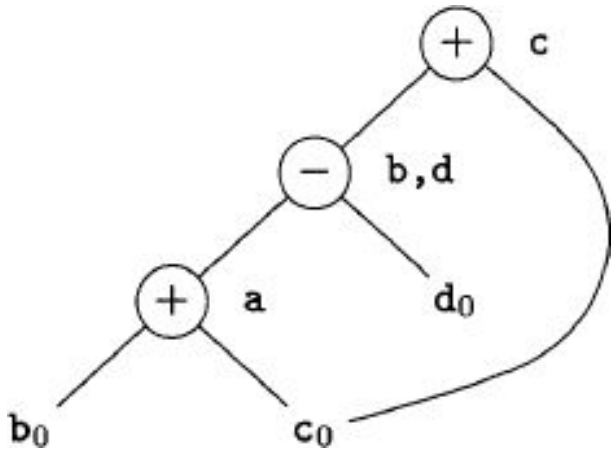
4. d = a - d



- In this statement operator - and the nodes with attached variables a and do as children.
- Since the operator and the children are the same as those for the node corresponding to statement (2)
- No need to create this node, but add d to the list of definitions for the node labeled -.

Finding Local Common Sub expressions

- It might appear that, since there ***are only three nonleaf nodes*** in the DAG , 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 -.
- The block then becomes



1. a = b + c
2. b = a - d
3. c = b + c
4. d = a - d

1. a = b + c
2. d = a - d
3. c = d + c

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.

The Use of Algebraic Identities

1. Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

2. Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

The Use of Algebraic Identities

3. A third class of related optimizations is **constant folding**. Here we evaluate constant expressions at compile time and replace the constant expressions by their value

- Thus the expression *2* * 3.14 would be replaced by 6.28.
- Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

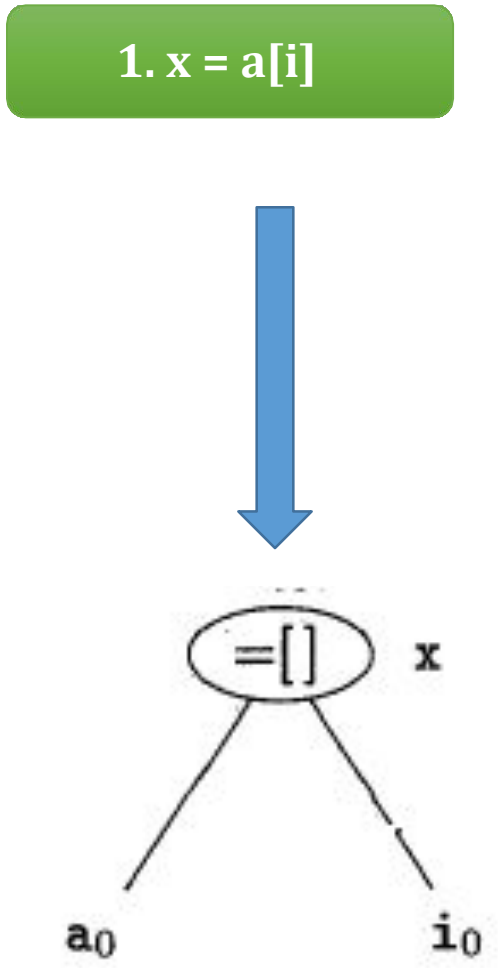
Representation of Array References

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

Now consider a block of code given below:

1. x = a[i]
2. a[j] = y
3. z= a[i]

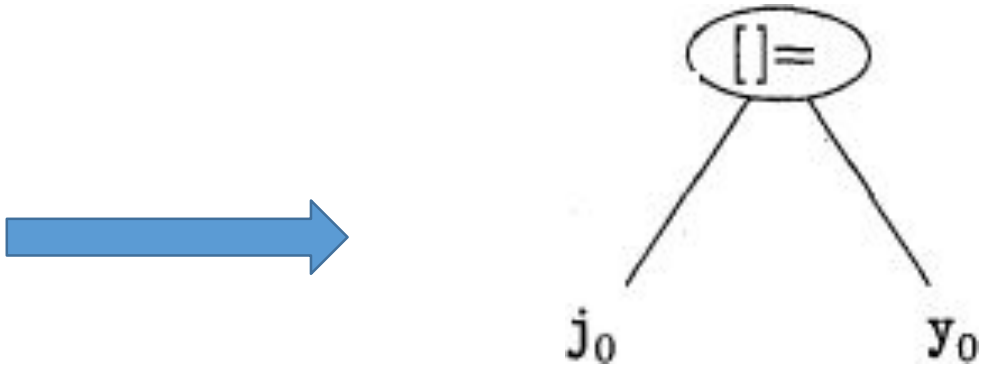
- An assignment from an array, like **x = a[i]** , is represented by creating a node with operator **=[]** and two children representing the initial value of the array, a0 in this case, and the index i. Variable x becomes a label of this new node.



Representation of Array References

- An assignment to an array, like **a[j] = y**, is represented by a new node with operator **[]=** and three children representing a0, j and y. There is no variable labelling this node.

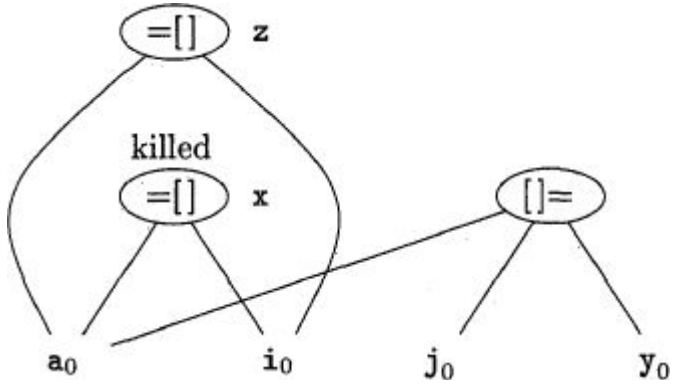
2. a[j] = y



Representation of Array References

- The creation of node according to line 3. will kills all currently constructed nodes whose value depends on **ao**
- **A** node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.
- So the final DAG looks as shown below:

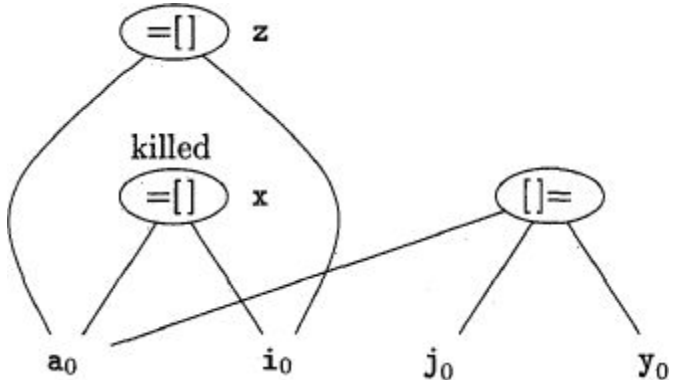
1. x = a[i]
2. a[j] = y
3. z= a[i]



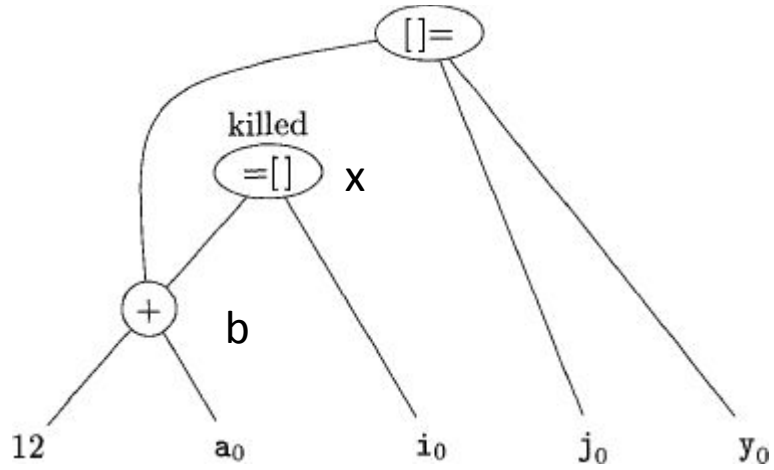
Representation of Array References

- Examples

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




```
b = 12 + a
x = b[i]
b[j] = y
```



Pointer Assignments

- Consider an example:

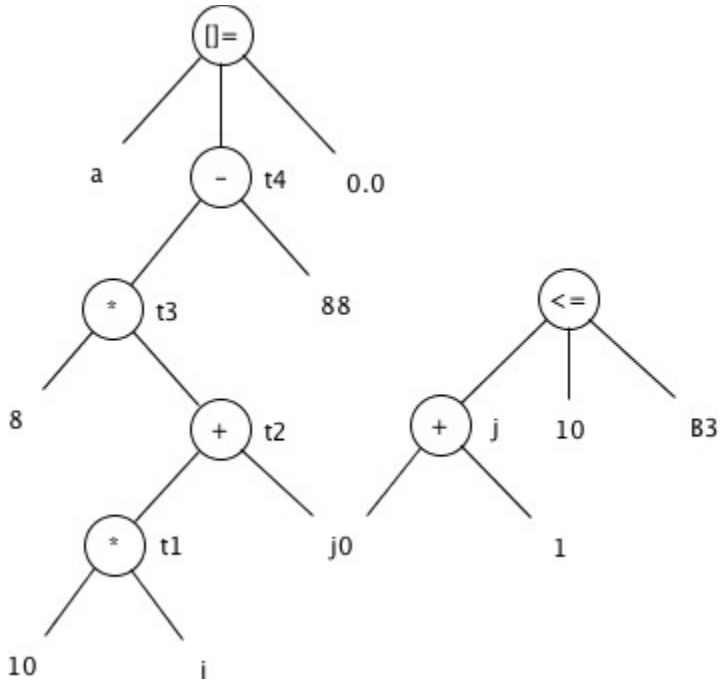
```
x = *p
*q = y
```

- The ***operator =**** will take all nodes that are currently associated with identifiers as arguments, which is relevant for **dead-code elimination**.

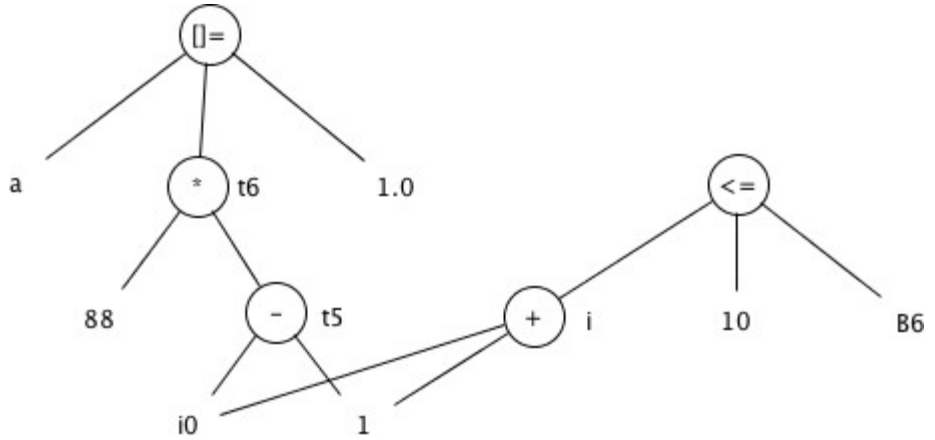
- ***So, There are global pointer analyses one could perform that might limit the set of variables a pointer could reference at a given place in the code.***
- ***Even local analysis could restrict the scope of a pointer***
- ***Procedure calls behave much like assignments through pointers***

Example
s

```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 goto B3
```




```
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```



Examples

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

Fig. 9.15. Three-address code for block B_2.

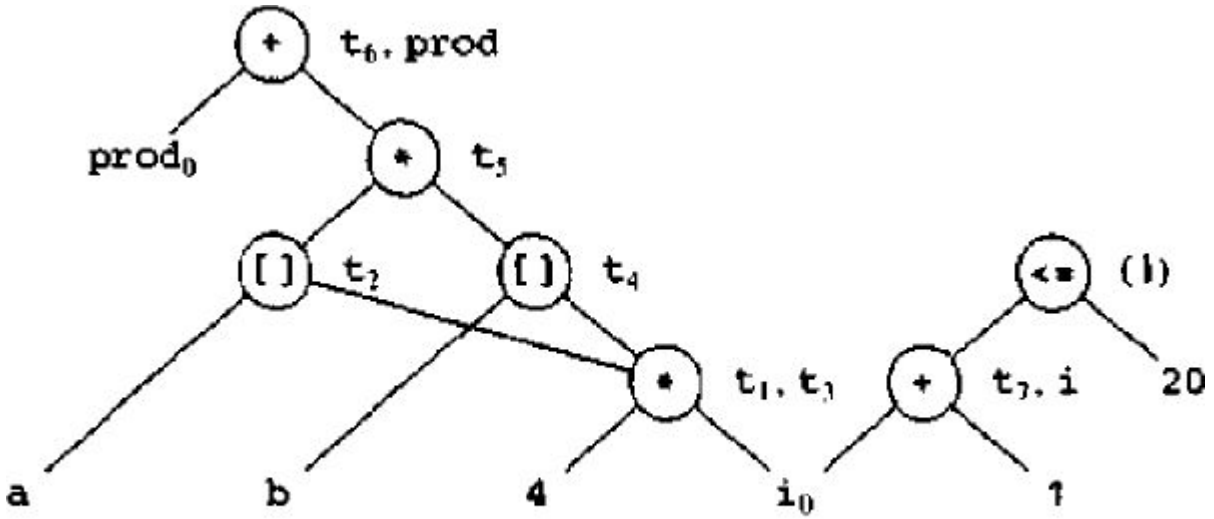
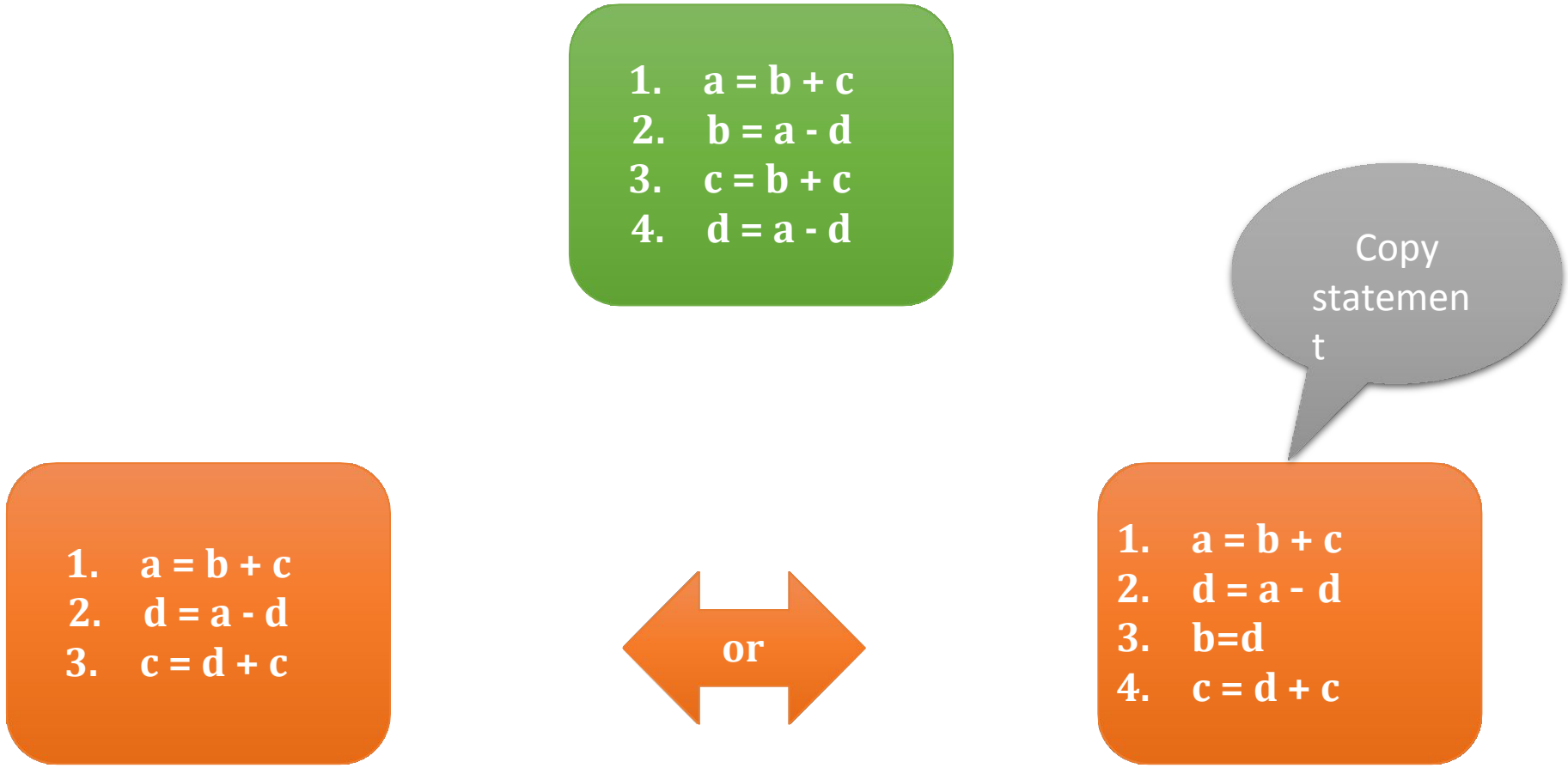


Fig. 9.16. Dag for block of Fig. 9.15.

Reassembling Basic Blocks From DAG's

- After performing whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three- address code.
- For each node that has one or more attached variables, we construct a three- address statement that computes the value of one of those variables.
- We prefer to compute the result into a variable that is ***live on exit from the block***.
- However, ***if we do not have global live-variable*** information to work from, we need to ***assume that every variable of the program is live*** on exit from the block.
- ***If the node has more than one live variable*** attached, then we have to ***introduce copy statements to give the correct value*** to each of those variables.
- Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

Reassembling Basic Blocks From DA



Run time Environment Source Language Issues

Run-Time Environments

- A compiler must accurately implement the abstractions embodied in the source language definition.
- These abstractions typically include names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.
- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

Run-Time Environments - Issues

This environment deals with a variety of issues such as

- The layout and allocation of storage locations for the objects named in the source program
- The mechanisms used by the target program to access variables
- The linkages between procedures
- The mechanisms for passing parameters
- The interfaces to the operating system, input/output devices, and other programs.

Example-Explanation

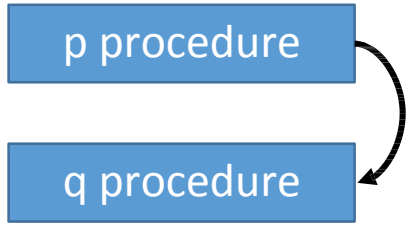
```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

- The main function has three tasks.
  - It calls readArray, sets the sentinels, and
  - Then calls quicksort on the entire data array.
- The possible sequence of call is shown
- In this execution, the call to partition(1,9) returns 4,
- so a[1] through *a[3]* hold elements less than its chosen separator value *v,*
- while the larger elements are in a[5] through a[9].

Procedure activations



- Procedure activations are nested in time.
- If an activation of procedure p calls procedure ***q,*** then that activation of q must end before the activation of p can end.
- There are three common cases:

The activation of q terminates normally	**Then in essentially any language, control resumes just after the point of p at which the call to q was made**
The activation of q, or some procedure q called, either directly or indirectly, aborts	**p ends simultaneously with q.**
The activation of q terminates because of an exception that q cannot handle	**Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues**

Activation Trees

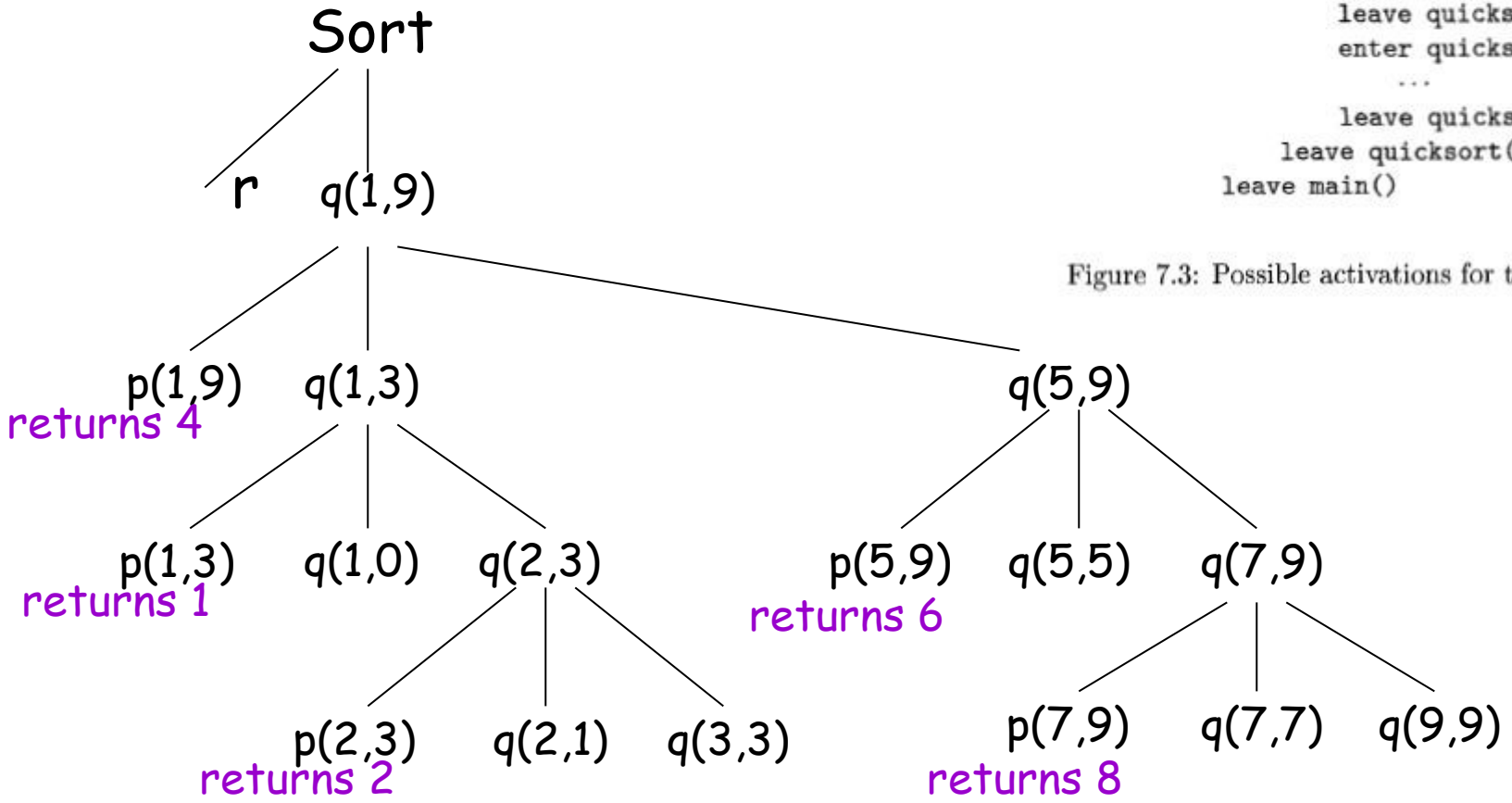
The activations of procedures during the running of an entire program can be represented as tree structure called as activation tree.

- In the tree each node corresponds to one activation
- The root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

Activation Tree

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2



Control stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node n is at the top of the stack the stack contains the nodes along the path from n to the root

Run time Environment-

Storage Organization

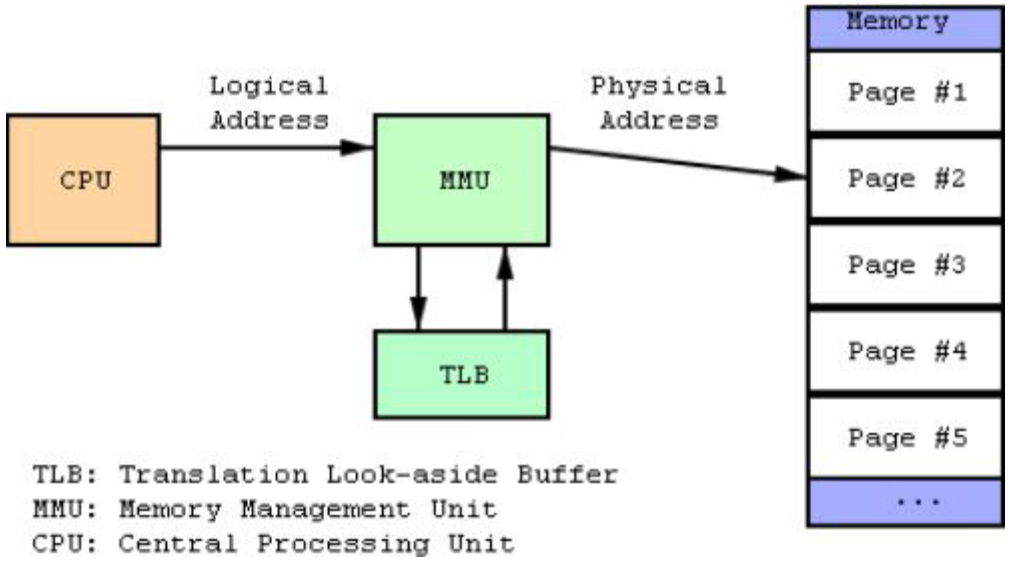
Storage Organization

From the perspective of the compiler writer:

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the ***compiler, operating system, and target machine***.
- The *operating system maps the logical addresses into physical addresses*, which are usually spread throughout memory.

operating system maps the logical addresses into physical addresses

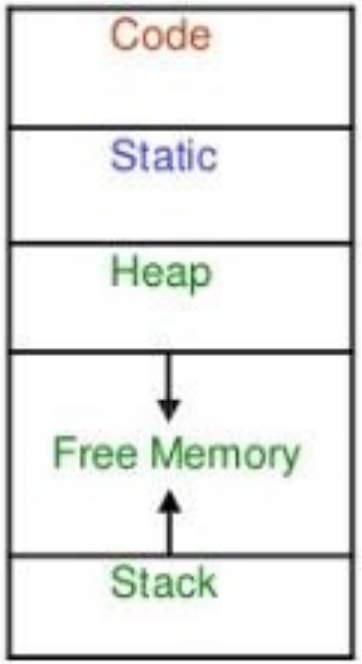
Logical vs Physical address



BASIS FOR COMPARISON	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Address Space	Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
Visibility	The user can view the logical address of a program.	The user can never view physical address of program
Access	The user uses the logical address to access the physical address.	The user can not directly access physical address.

Storage Organization

- The run-time representation of an object program in the logical address space



Memory locations for code are determined at compile time. Usually placed in the low end of memory

Size of some program data are known at compile time – can be placed another statically determined area

Dynamic space areas – size changes during program execution.

- Heap
  - Grows towards higher address
  - Stores data allocated under program control
- Stack
  - Grows towards lower address
  - Stores activation records

Typical subdivision of run-time memory

The stack is used to store data structures called **activation records** that get generated during procedure calls

Runtime Storage management

- The executing program runs in its own logical address space that was partitioned into four code and data areas:

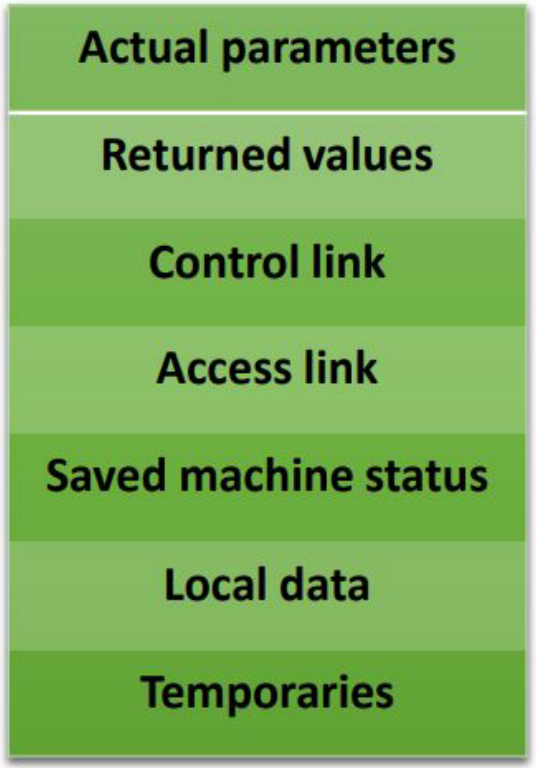
1. A **statically** determined **area Code** that holds the executable target code. The size of the target code can be determined at compile time.
2. A **statically** determined **data area** Static for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
3. A **dynamically** managed **area Heap** for holding data objects that are allocated and freed during program execution. The size of the Heap cannot be determined at compile time.
4. A **dynamically** managed **area Stack** for holding activation records as they are created and destroyed during procedure calls and returns. Like the Heap, the size of the Stack cannot be determined at compile time.

Activation Records

Procedure calls and returns are usually managed by a run-time stack called the control stack. Each live activation has an activation record

- The root of the activation tree will be at the bottom of the stack
- The entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack

A general activation record



Actual parameters	The actual parameters used by the calling procedure
Returned values	Space for the return value of the called function
Control link	A control link, pointing to the activation record of the caller
Access link	An "access link" may be needed to locate data needed by the called procedure
Saved machine status	information about the state of the machine just before the call to the procedure
Local data	Local data belonging to the procedure whose activation record this is
Temporaries	Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers

Run time Environment- Storage Allocation Strategies

Storage Allocation Strategies

The following three different storage allocation strategy were used

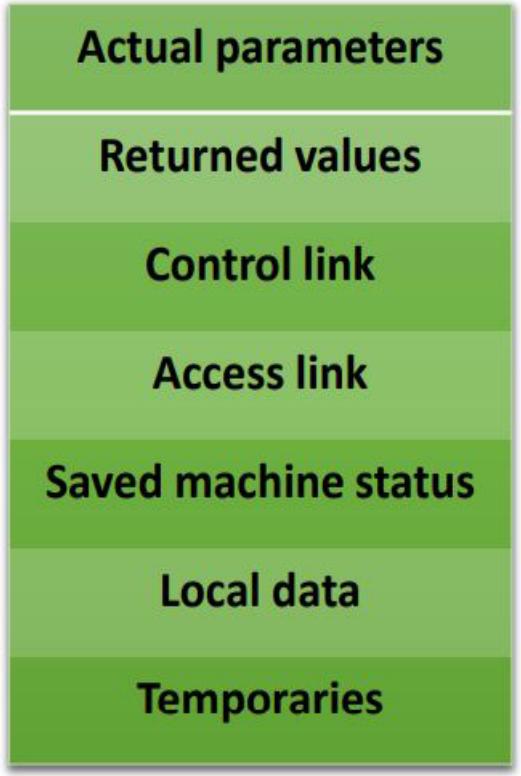
- Static allocation lays out storage for all data objects at compile time
- Stack allocation manages the run time storage as a stack
- Heap allocation allocated and deallocates storages as needed at run time from a data area known as a heap

Stack Allocation of Space

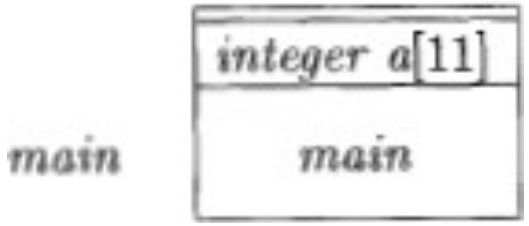
Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.

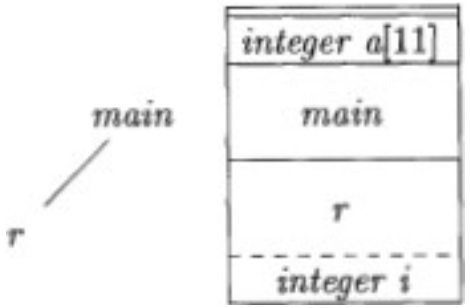
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.
- This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls

A general activation record

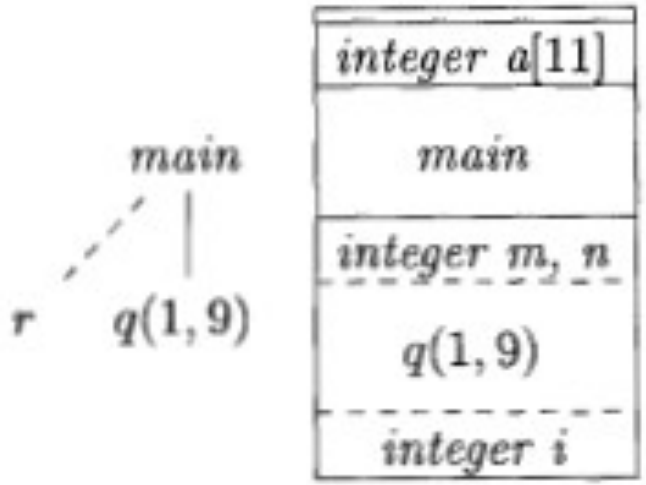


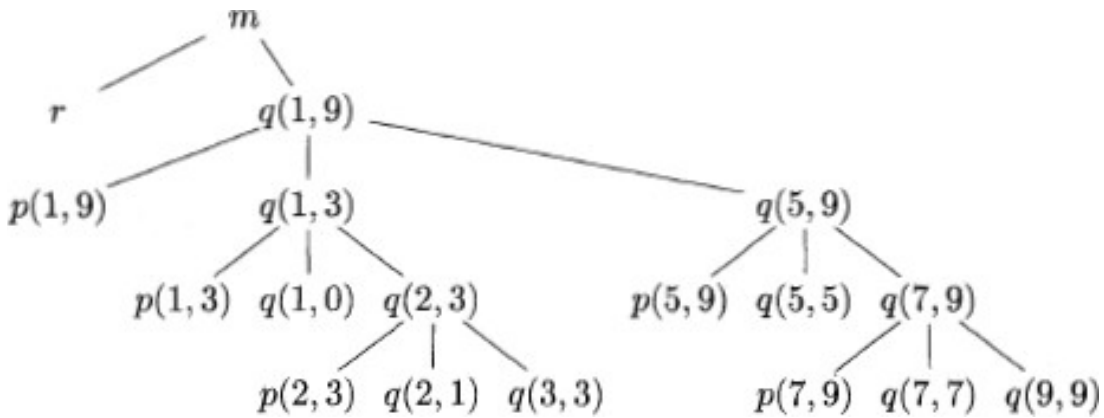
Actual parameters	The actual parameters used by the calling procedure
Returned values	Space for the return value of the called function
Control link	A control link, pointing to the activation record of the caller
Access link	An "access link" may be needed to locate data needed by the called procedure
Saved machine status	information about the state of the machine just before the call to the procedure
Local data	Local data belonging to the procedure whose activation record this is
Temporaries	Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers



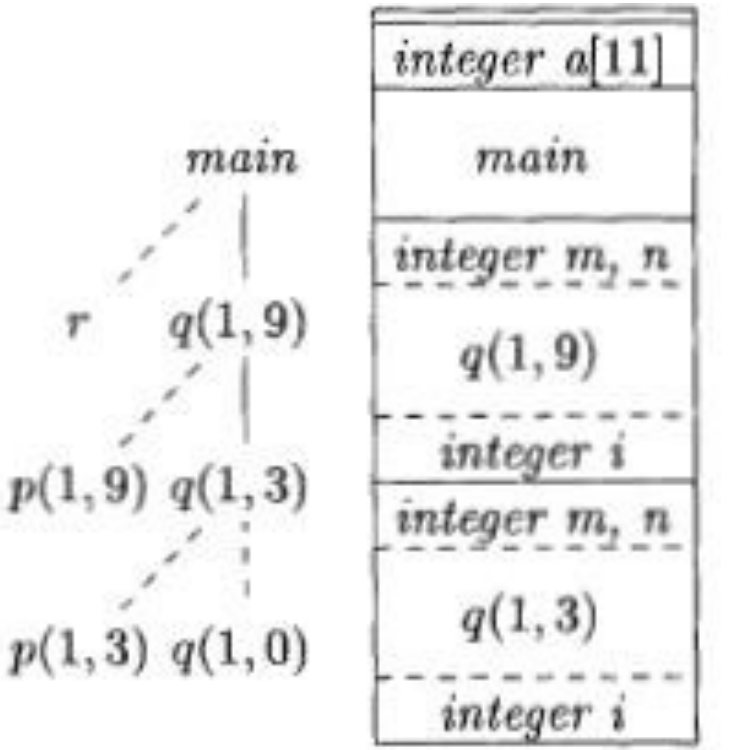


- Procedure *r* is activated When control reaches the first call in the body of main and its activation record is pushed onto the stack.
- The activation record for *r* contains space for local variable i.

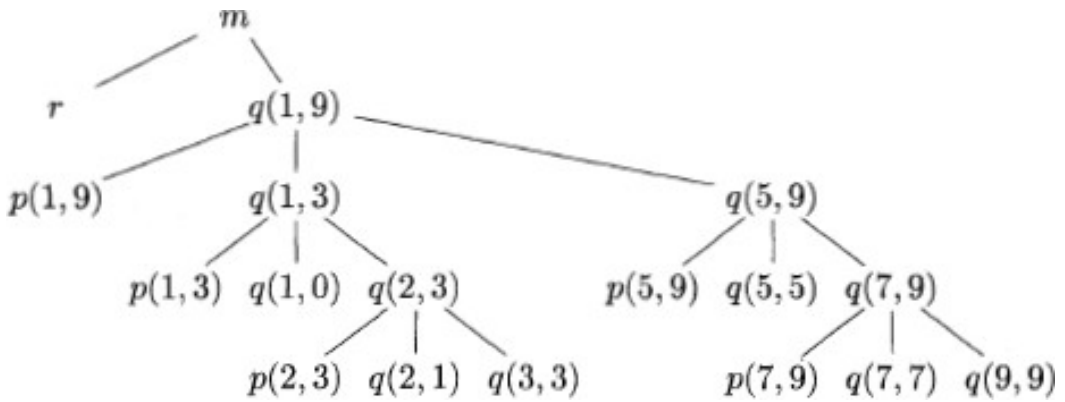




- When control returns from this activation, its record is popped, leaving just the record for main on the stack.
- Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack.



(d) Control returns to $q(1,3)$



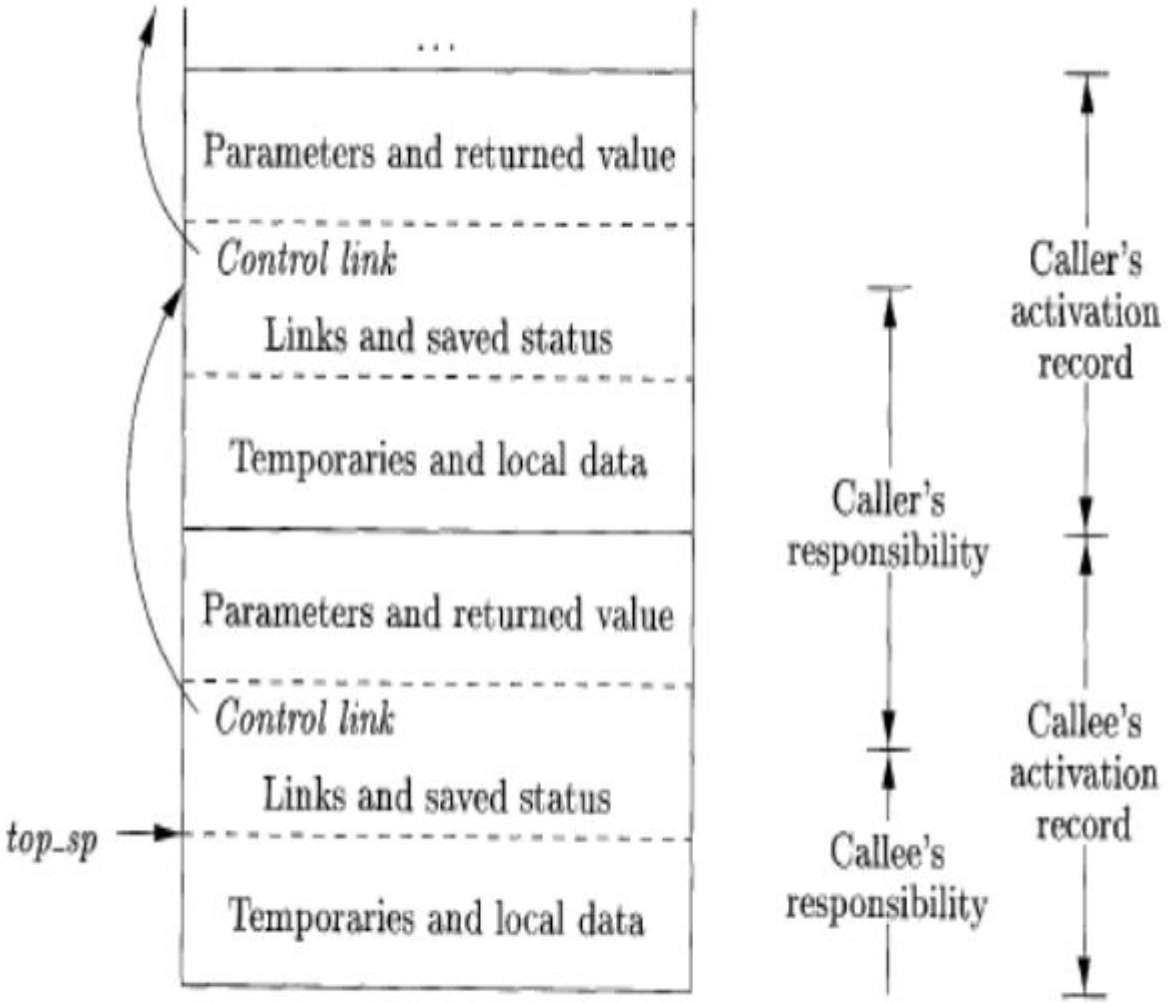
- The activation record for q contains space for the parameters m and n and the local variable ***i***.
- Notice that space once used by the call of *r* is reused on the stack.
- No trace of data local to *r* will be available to q(1,9).
- When q(1,9) returns, the stack again has only the activation record for main.
- A recursive call to q(1,3) was made.
- Activations p(1,3) and q(1,0) have begun and ended during the lifetime of q(l,3), leaving the activation record for q(l,3) on top

Calling Sequences

- **Procedure calls** are implemented bycalling sequences
- It consists of code that allocates an activation record on the stack and enters information into its fields
- The code in a calling sequence is often divided between
  - The calling procedure (the "caller")
  - The procedure it calls (the "callee")

Principles behind Calling Sequences

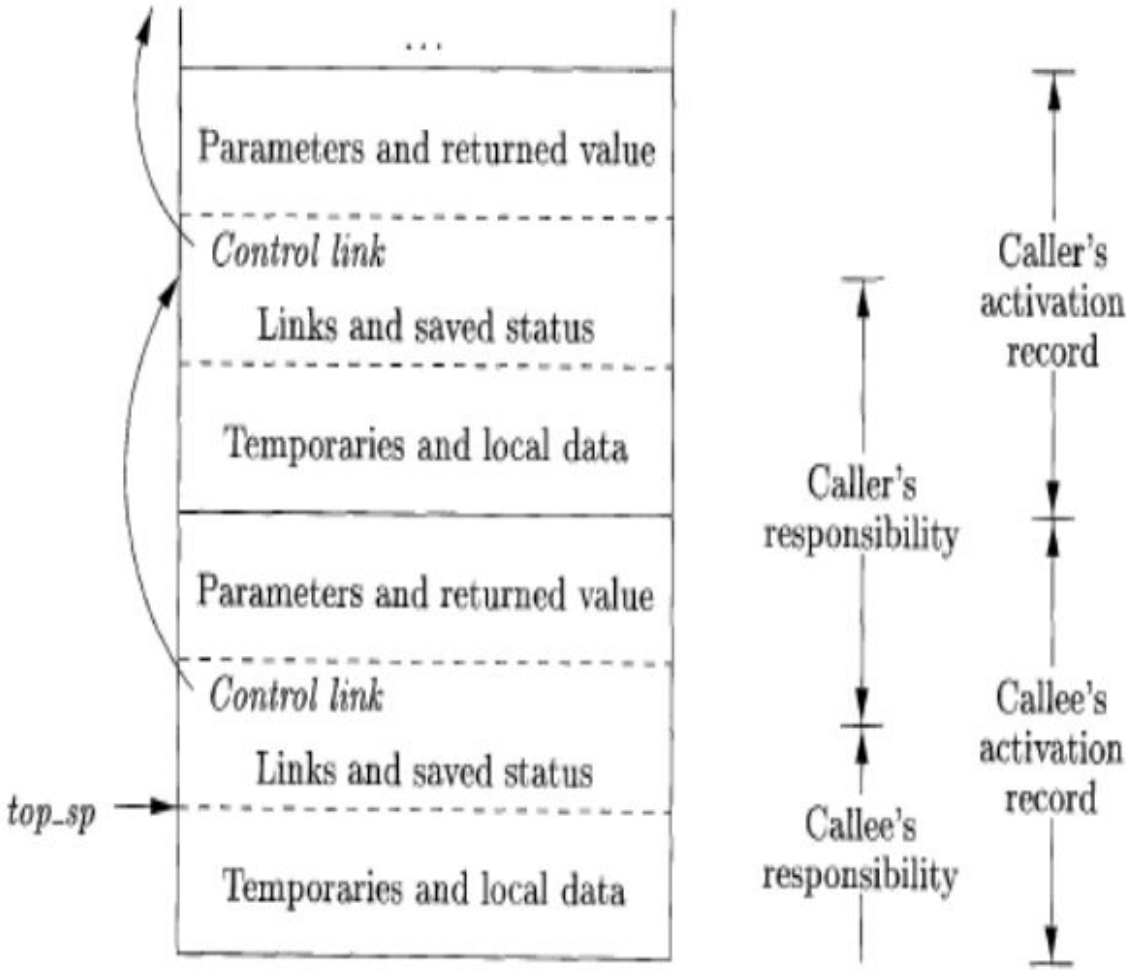
1. Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record
2. Fixed-length items are generally placed in the middle. such items typically include the control link, the access link, and the machine status fields (refer the activation record)
3. Items whose size may not be known early enough are placed at the end of the activation record.
4. We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of the fixed-length fields in the activation record



Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

- ✓ The caller evaluates the actual parameters.
- ✓ The caller stores a return address and the old value of *top-sp* into the callee's activation record.
- ✓ The caller then increments *top-sp* to the position shown in Fig.
- ✓ That is, *top-sp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.
- ✓ The callee saves the register values and other status information.
- ✓ The callee initializes its local data and begins execution

Caller and callee sequence



Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Return sequence

- The callee places the return value next to the parameters.
- Using information in the machine-status field, the callee restores *top-sp* and other registers, and then branches to the return address that the caller placed in the status field.
- Although *top-sp* has been decremented, the caller knows where the return value is, relative to the current value of *top-sp;* the caller therefore may use that value.

Variable-Length Data on the Stack

- The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time
- But which are local to a procedure and thus may be allocated on the stack.
- ***In modern languages, objects whose size cannot be determined at compile time are allocated space in the heap***

- Let consider procedure p has three local arrays, whose sizes be determined at compile time.
- The storage for these arrays is not part of the activation record for p.
- Only a pointer to the beginning of each array appears in the activation record itself.
- Thus, when p is executing, these pointers are at known offsets from the top-of-stack pointer, so the target code can access array elements through these pointers.

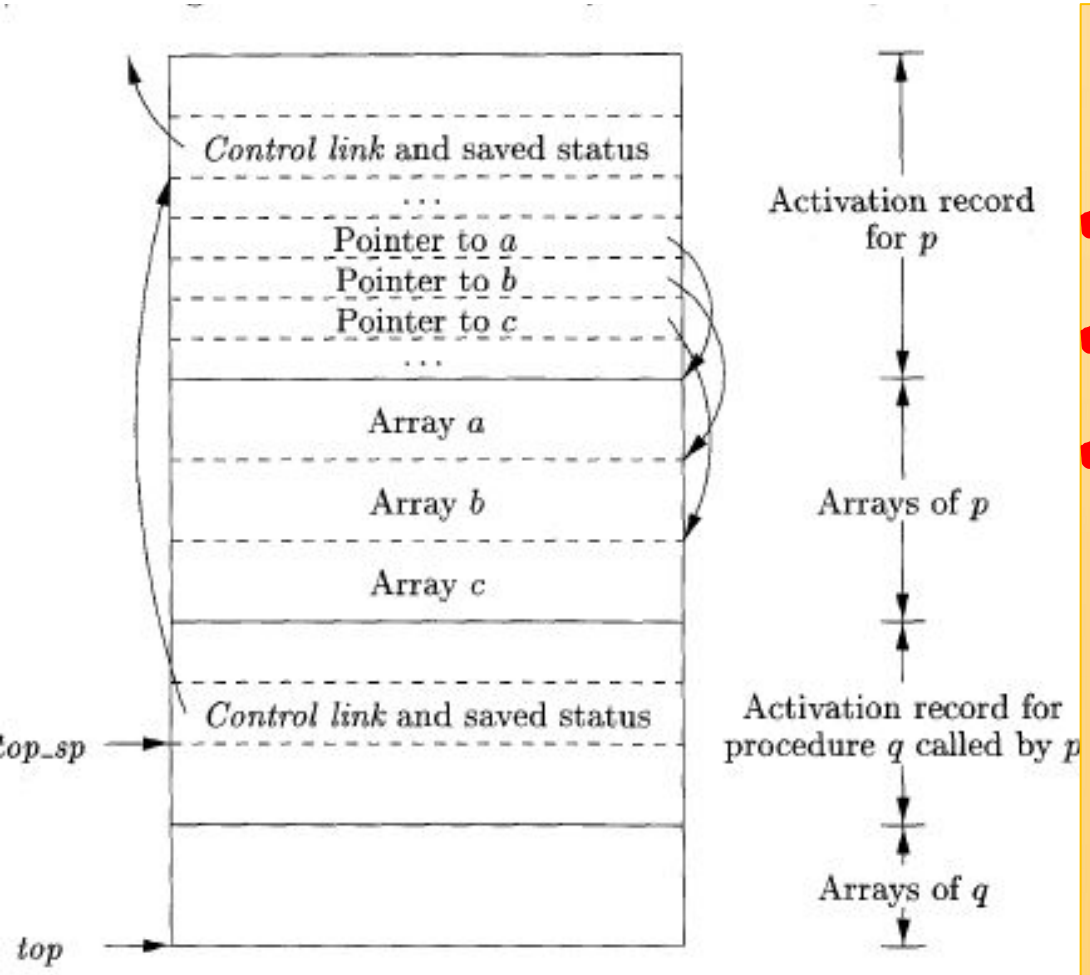


Figure 7.8: Access to dynamically allocated arrays

- The activation record for a procedure q, called by p.
- The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that.
- ✔ **Access to the data on the stack is through two pointers, *top* and *top-sp*.**
- ✔ **Here, *top* marks the actual top of stack; it points to the position at which the next activation record will begin.**
- ✔ **The second, *top-sp* is used to find local, fixed-length fields of the top activation record.**
  - **The code to reposition *top* and *top-sp* can be generated at compile time in terms of sizes that will become known at run time.**
  - **When q returns, *top-sp* can be restored from the saved control link in the activation record for q.**
  - **The new value of *top* is *top-sp* minus the length of the machine-status, control and access link, return-value, and parameter fields in q's activation record.**
  - **This length is known at compile time.**

Heap Allocation

- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes
- Memory Manager: The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions

- **Allocation. When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. If space is exhausted, the memory manager passes that information back to the application program**
- ***Deallocation*. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.**

Required properties of memory management

- ***Space Efficiency***. A memory manager should minimize the total heap space needed by a program
- ***Program Efficiency***. *A* memory manager should make good use of the memory subsystem to allow programs to run faster. By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster
- ***Low Overhead***. Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible

Access to Nonlocal Data on the Stack

Access to Nonlocal Data on the Stack

- Now let us see how a procedure access their data that not belong to its own
- ***That is the mechanism for finding data used within a procedure p but that does not belong to p.***
- Access becomes more complicated in languages where procedures can be declared inside other procedures.

Data Access Without Nested Procedures

The ***global variable*** v has a scope consisting of all the functions that follow the declaration of v, except where there is a local definition of the identifier v

- For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple

1. Global variables are allocated static storage.
   - The locations of these variables remain fixed and are known at compile time.
   - So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.
2. Any other name must be local to the activation at the top of the stack.
   - We may access these variables through the top-sp pointer of the stack.

Data Access With Nested Procedures

- Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule;
- Let us give **nesting depth** 1 to procedures that are not nested within any other procedure

Access Links

- A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the access link to each activation record.
- If procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.
- Note that the nesting depth of q must be exactly one less than the nesting depth of p.

Access Links

- Suppose that the procedure p at the top of the stack is at nesting depth n_p and p needs to access x, which is an element defined within some procedure q that surrounds p and has nesting depth n_q.
- *Then* $n_p \geq n_q$ only if p and q are the same procedure.
- To find x, we start at the activation record for p at the top of the stack and follow the access link $n_p - n_q$ times, from activation record to activation record.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

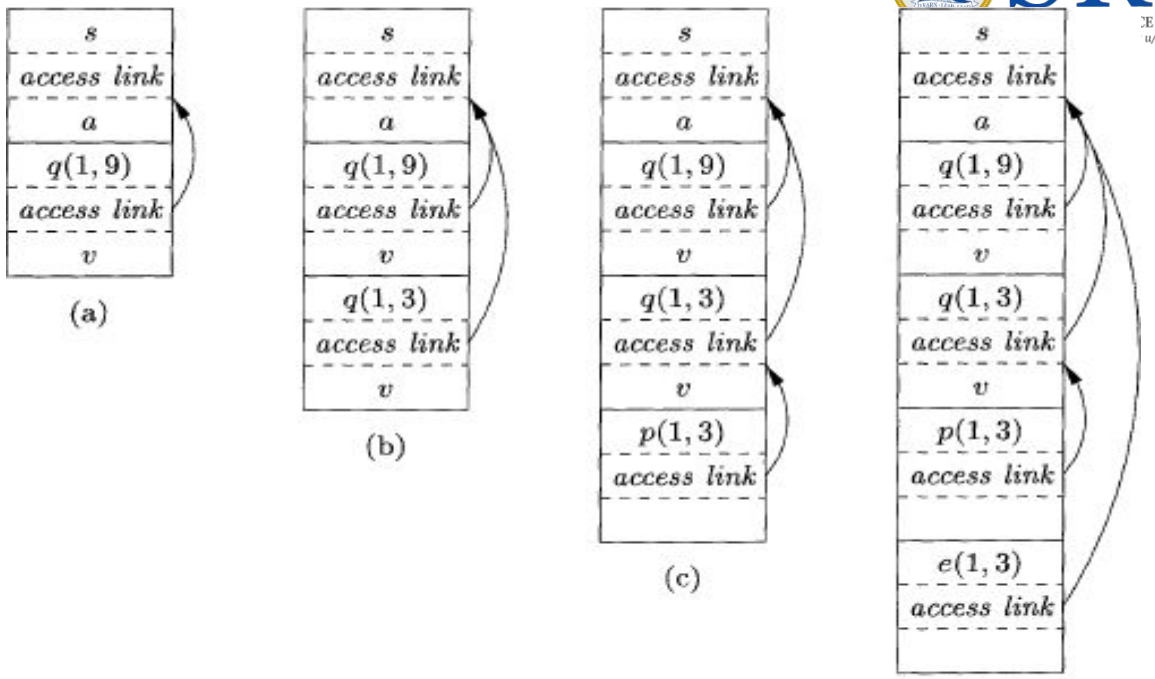


Figure 7.11: Access links for finding nonlocal data

- After sort has called readArray to load input into the array *a* and then called quicksort(1,9) to sort the array.
- The access link from quicksort(1,9) points to the activation record for sort, not because sort called quicksort but because sort is the most closely nested function surrounding quicksort in the program
- In successive steps of recursive call to quicksort(1,3), followed by a call to partition, which calls exchange.
- The quicksort(1,3)'s access link points to sort, for the same reason that quicksort(1,9)'s does.
- The access link for exchange bypasses the activation records for quicksort and partition, since exchange is nested immediately within sort.
- That arrangement is fine, since exchange needs to access only the array *a,* and the two elements it must swap are indicated by its own parameters ***i*** and *j.*

Parameter Passing

- Call by value
  - actual parameters are evaluated and their r-values are passed to the called procedure
  - caller evaluates the actual parameters and places r value in the storage for formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then l-value is passed
  - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

Parameter Passing

- Copy Restore
  - A hybrid between call by value and call by reference
  - Also called as copy-in copy-out/ call by value result
  - actual parameters are evaluated, rvalues are passed by call by value, lvalues are determined before the call
  - when control returns, the current rvalues of the formals are copied into lvalues of the locals
- Call by name
  - The procedure is treated as if it were a macro, its body is substituted for the call in the caller with the actual parameters
  - The local names of the called procedure are kept distinct from the names of the calling procedure
  - The actual parametes are surrounded by parentheses if necessary to preserve their integrity

End of unit 5