CVR COLLEGE OF ENGINEERING

In Pursuit of Excellence

(An Autonomous Institution, NAAC 'A' Grade)

Subject Name : AUTOMATA AND COMPILER DESIGN

Subject Code : 67302

Topic : UNIT V - Code Optimization and Code Generation

Class : III YEAR - I Sem

Faculty Name : Dr. R RAJA

Designation : Associate Professor

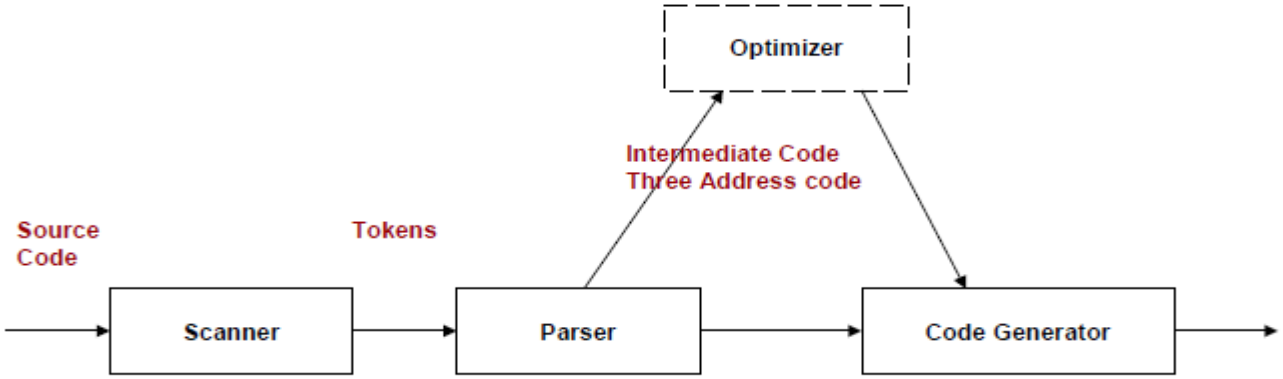
Department : CSIT

UNIT V - Code Optimization and Code Generation

Principle sources of optimization, optimization of basic blocks, flow graphs, data flow analysis. Machine dependent code generation; object code forms, generic code generation algorithm, register allocation and assignment, using DAG representation of Blocks, Peephole optimization.

Introduction

- Code improving transformations – in compiler
- Optimization is a semantics-preserving transformation



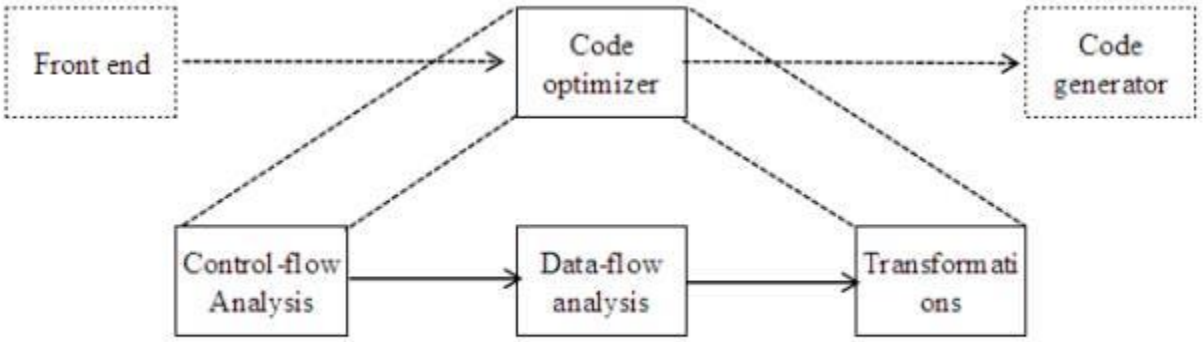


Fig. 5.1 Organization of the code optimizer

- **Necessity of Code Optimization**
  - Loops in programming :- 90:10 rule
  - execution time reduced

- Optimization depends on various factors
  - Memory
  - Algorithm
  - Execution Time
  - Programming Language

Types of Optimizations

- High-level optimizations
  - Function inlining
- Machine-Independent optimizations
  - Without taking any considerations of any properties of the target machine
- Machine-dependent optimizations
  - e.g., peephole optimizations, instruction scheduling
- Local optimizations or Transformation
  - Within basic block
- Global optimizations or Data flow Analysis
  - Across basic blocks

PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

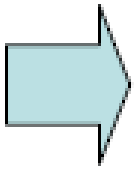
- **Common sub expression elimination**
- **Copy propagation,**
- **Dead-code elimination**
- **Constant folding**

The other transformations come up primarily when global optimizations are performed.

1. Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```



```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

2. Copy Propagation:

- Assignments of the form f : = g called copy statements, or copies for short.
- The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x. For e.g.

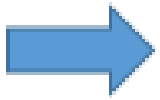
```
x = Pi;

A=x*r*r;
```

- **The optimization using copy propagation can be done as follows: A=Pi*r*r;**
- Here the variable x is eliminated.

3. Dead Code Elimination:

- The dead code may be a variable or the result of some expression computed by the programmer that may not have any further uses.
- By eliminating these useless things from a code, the code will get optimized. For e.g.

```
if(a == b)
{
  b = c;
  .....
  return b;
  c = 0;
}
```



```
if(a == b)
{
  b = c;
  .....
  return b;
}
```

Example:

```
i=0;
if(i==1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

4. Constant folding:

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- The code that can be simplified by user itself, is simplified. For e.g.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

5. Loop Optimizations:

- 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.
- Some loop optimization techniques are:

i) Frequency Reduction (Code Motion):

- In frequency reduction, the amount of code in loop is decreased.
- A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop. For e.g.

Before optimization:	After optimization:

while(i<100)
{
a = Sin(x)/Cos(x) + i;
i++;
}
	t = Sin(x)/Cos(x);
	while(i<100)
	{
	a = t + i;
	i++;
	}

ii) Induction-variable elimination:

- Some loops contain two or more induction variables that can be combined into one induction variable.
- Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both run-time performance and code space.
- Some architectures have auto-increment and auto-decrement instructions that can sometimes be used instead of induction variable elimination.

Before optimization:

```
int a[SIZE];
int b[SIZE];
void f (void)
{
  int i1, i2, i3;

  for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++)
    a[i2++] = b[i3++];
  return;
}
```

After optimization:

```
int a[SIZE];
int b[SIZE];
void f (void)
{
  int i1;

  for (i1 = 0; i1 < SIZE; i1++)
    a[i1] = b[i1];
  return;
}
```

iii) Reduction in Strength:

- The strength of certain operators is higher than other operators.
- For example, strength of * is higher than +. Usually, compiler takes more time for higher strength operators and execution speed is less.
- Replacement of higher strength operator by lower strength operator is called a strength reduction technique
- Optimization can be done by applying strength reduction technique where higher strength can be replaced by lower strength operators. For e.g.

Before optimization:	**After optimization:**

for (i=1;i<=10;i++)
{
sum = i * 7;
printf("%d", sum);
}
	temp = 7;
	for(i=1;i<=10;i++)
	{
	temp = temp + 7;
	sum = temp;
	printf("%d", sum)
	}

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
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

- A name in a basic block is said to *live* at a given point if its value is used after that point in the program, perhaps in another basic block

Algorithm 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:

   - » The first statement is a leader.
   - » Any statement that is the target of a conditional or unconditional goto is a leader.
   - » 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.

Example 3: Consider the fragment of source code, it computes the dot product of two vectors a and b of length 20.

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

- Let us apply Algorithm 1 to the three-address code 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) 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 ]
(5)    t3 := 4*i
(6)    t4 :=b [ t3 ]
(7)    t5 := t2*t4
(8)    t6 := prod +t5
(9)    prod := t6
(10)   t7 := i+1
(11)   i := t7
(12)   if  i<=20  goto (3)
```

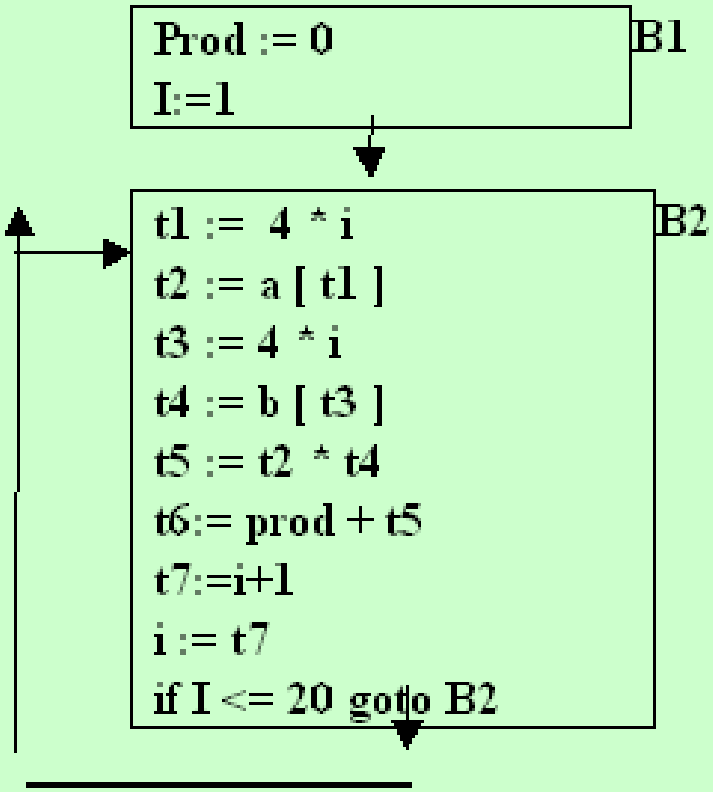


Fig .9 flow graph for program

fig 8. Three-address code computing dot product

Flow graphs:

- A flow graph is a directed graph with flow control information added to the basic blocks.
- The basic blocks serves as nodes of the flow graph. The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- There is a directed edge from block B1 to block B2 if B2 appears immediately after B1 in the code.

- In Fig.9 B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2.
- The target of jump from last statement of B2 is the first statement B2, so there is an edge from B2(last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.

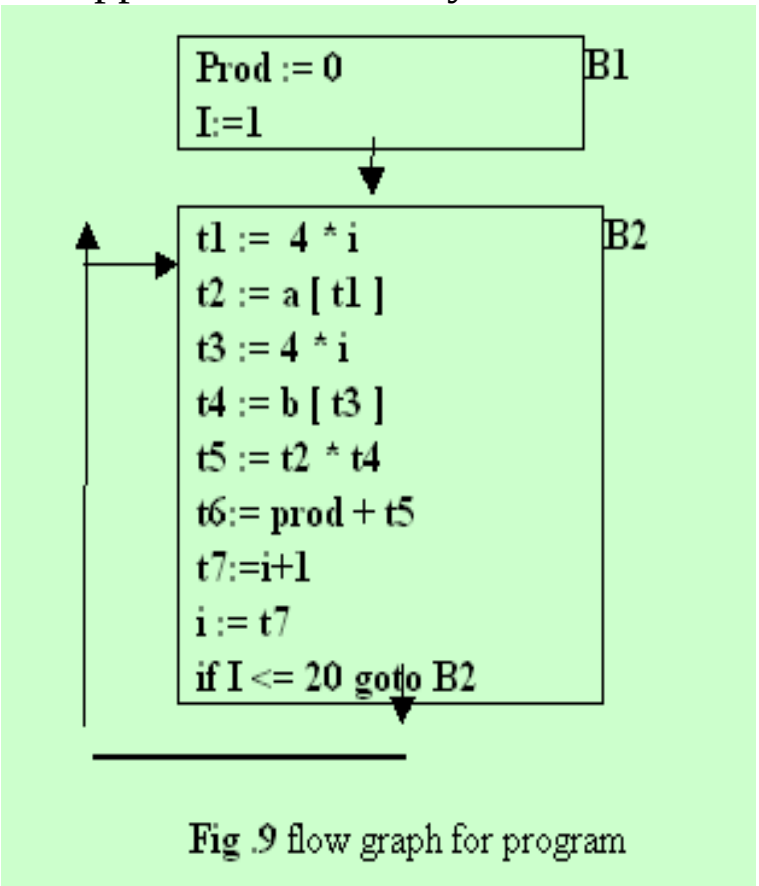


Fig .9 flow graph for program

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without modifying expressions computed by the block. Two important classes of transformation are :

1. Structure Preserving Transformations

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements

2. Algebraic transformation

1. Structure preserving transformations:

a)Common sub-expression elimination: Consider the sequence of statements:

1) a = b + c

2) b = a - d

3) c = b + c

4) d = a - d

Since the second and fourth expressions compute the same expression, the code can be transformed as:

1) a = b + c

2) b = a - d

3) c = b + c

4) d = b

b) Dead-code elimination:

- Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block.
- Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables:

- A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block.
- Such a block is called a normal-form block.

d) Interchange of statements:

- Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

- We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

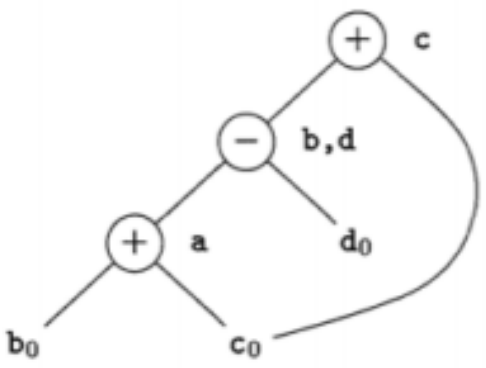
iii) x/1 = x , x/2=x*0.5

Direct Acyclic Graph (DAG):

- DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too.
- DAG provides easy transformation on basic blocks.
- DAG can be understood here:
  - Leaf nodes represent identifiers, names or constants.
  - Interior nodes represent operators.
  - Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assig

Example:

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



Directed Acyclic Graph

Algorithm for Construction of DAG

There are three possible cases to construct DAG on three address code:

Case 1: x = y op z

Case 2: x = op y

Case 3: x = y

DAG can be constructed as follows:

STEP 1:

If y is undefined then create a node with label y. Similarly create a node with label z.

STEP 2:

For case 1, create a node with label op whose left child is node y, and node z will be the right child. Also, check for any common sub expressions.

For case 2, determine if a node is labelled op. such node will have a child node y.

for case 3 node n will be node y.

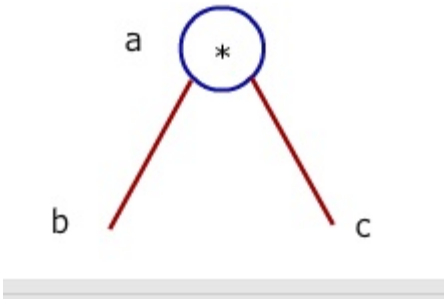
STEP 3:

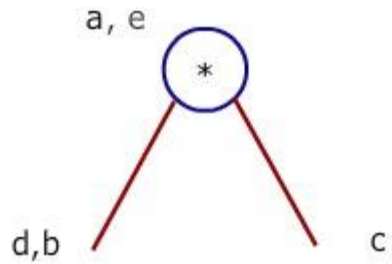
Delete x from list of identifiers for node x. append x to the list of attached identifiers for node n found in step 2.

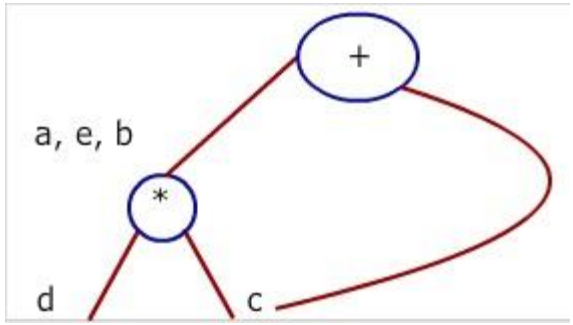
Example:

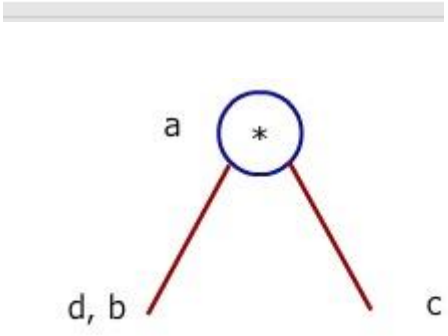
Consider the following three address code statements.

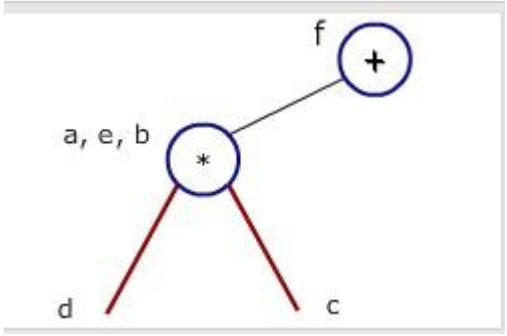
a = b * c
d = b
e = d * c
b = e
f = b + c
g = f + d

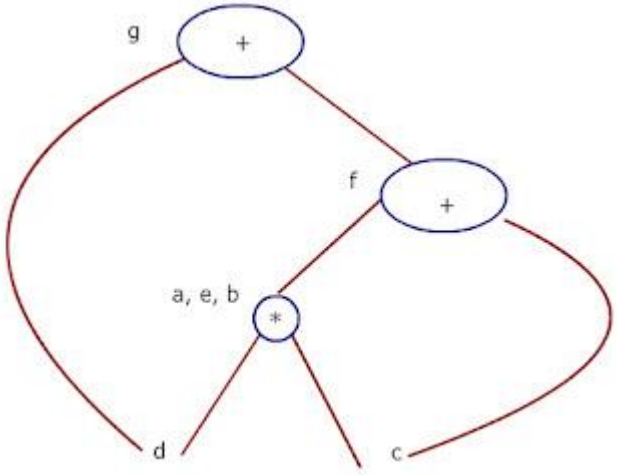












DAG Applications:

- Determines the common sub expressions
- Determines the names used inside the block, and the names that are computed outside the block
- Determines which statements of the block could have their computed value outside the block
- Code may be represented by a DAG describing the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently
- Several programming languages describe systems of values that are related to each other by a directed acyclic graph. When one value changes, its successors are recalculate; each value is evaluated as a function of this predecessors in the DAG

Next – Use Information

- Next-use information is needed for dead-code elimination and register allocation.
- Next-use is computed by a backward scan of a basic block.
- The next-use information will indicate the statement number at which a particular variable that is defined in the current position will be reused.

The following are the steps involved in generating next-use information:

Input: Basic block B of three-address statements

Output: At each statement ***i: x= y op z***, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness 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 next-uses of y and Z to i.

Example live and nextuse() computation

i: a := b + c

j: t := a + b

Statement number	Instruction	Liveness and nextuse initialized	Modified liveness and nextuse after statement 'j'	Modified liveness and nextuse after statement 'i'	Comments
i	a := b + c	live(a) = true, live(b) = true, live(t) = true, nextuse(a)=none, nextuse(b)=none, nextuse(t)=none	live(a) = true, nextuse(a) = j, live(b) = true, nextuse(b) = j, live(t) = false, nextuse(t)= none	live(a) = false, nextuse(a)=none, live(b)=true, nextuse(b) = i, live(c) = true, nextuse(c) = i, live(t) = false, nextuse(t)=none	'a' is said to false and no nextuse. 'b' and 'c' are said to live at statement 'i' and their next use is at 'i' . 't''sliveness and nextuse remains as it is computed in statement 'j'
j	t := a + b	live(a) = true, live(b) = true, live(t) = true, nextuse(a)=none, nextuse(b)=none, nextuse(t =none	live(a) = true, nextuse(a) = j, live(b) = true, nextuse(b) = j, live(t) = false, nextuse(t)= none	live(a) = true, nextuse(a) = j, live(b) = true, nextuse(b) = j, live(t) = false, nextuse(t)=none	'a' and 'b' are used here and hence their nextuse is at 'j' and they are said to be live. 't' is computed and hence nextuse is none and live information is false

Introduction to 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 subexpression 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.

1 The Data-Flow Abstraction

2 The Data-Flow Analysis Schema

3 Data-Flow Schemas on Basic Blocks

4 Reaching Definitions

5 Live-Variable Analysis

6 Available Expressions

1. The Data-Flow Abstraction

- The execution of a program can be viewed as a series of transformations of the program state, which consists of the values of all the variables in the program, including those associated with stack frames below the top of the run-time stack.
- Each execution of an intermediate-code statement transforms an input state to a new output state.
- The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.
- When we analyze the behavior of a program, we must consider all the possible sequences of program points (“paths") through a flow graph that the program execution can take.
- We then extract, from the possible program states at each point, the information we need for the particular data-flow analysis problem we want to solve.
- In more complex analyses, we must consider paths that jump among the flow graphs for various procedures, as calls and returns are executed.

1. The Data-Flow Abstraction

we shall concentrate on the paths through a single flow graph for a single procedure.

Let us see what the flow graph tells us about the possible execution paths.

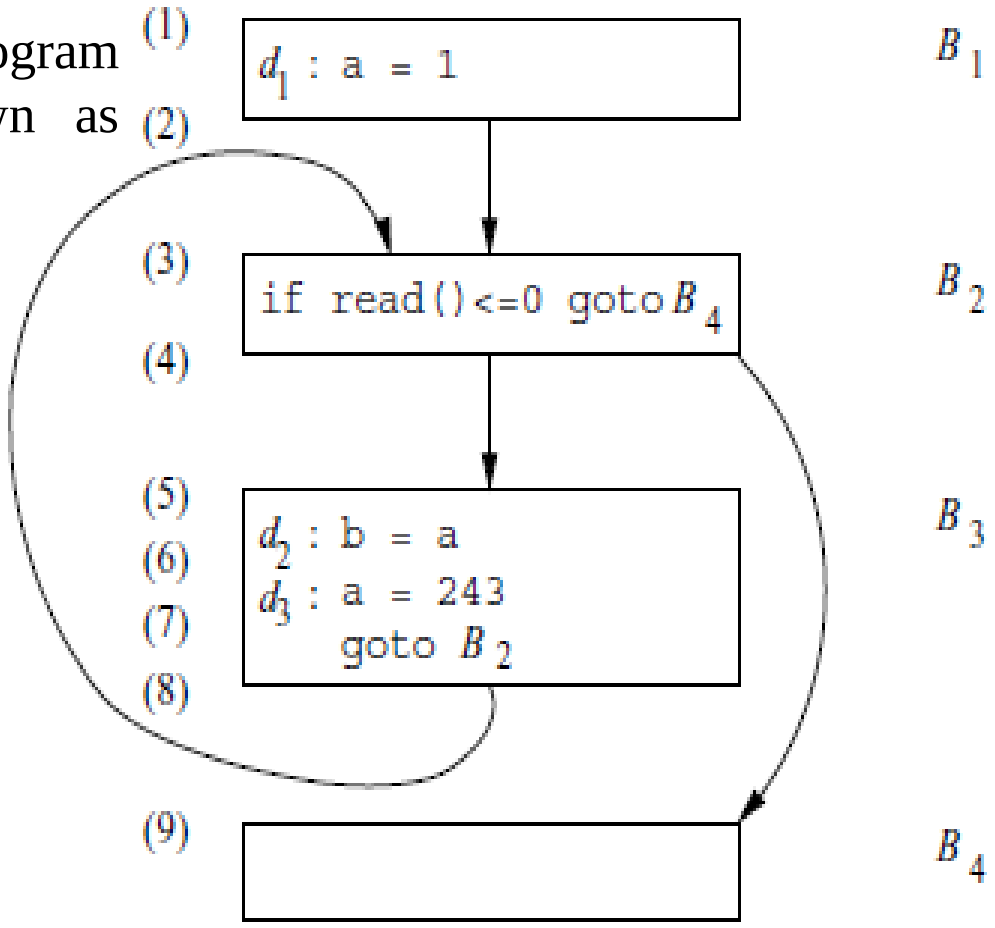
- Within one basic block, the program point after a statement is the same as the program point before the next statement.
- If there is an edge from block B1 to block B2, then the program point after the last statement of B1 may be followed immediately by the program point before the first statement of B2.

Thus, 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; : : : ; n-1,either

1. pi is the point immediately preceding a statement and pi+1 is the point immediately following that same statement, or
2. pi is the end of some block and pi+1 is the beginning of a successor block.

Example program illustrating the data-flow abstraction

The definitions that may reach a program point along some path are known as reaching definitions.



2. 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.
- The set of possible data-flow values is the domain for this application.
- We denote the data-flow values before and after each statement s 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 s.
- There are two sets of constraints:
  - ✓ those based on the semantics of the statements (“transfer functions") and
  - ✓ those based on the flow of control.

Transfer Functions

- 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 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, $\text{OUT}[s] = f_s(\text{IN}[s])$.
- Conversely, in a backward-flow problem, the transfer function fs for statement s converts a data-flow value after the statement to a new data-flow value before the statement. That is,

$$\text{IN}[s] = f_s(\text{OUT}[s]).$$

Control-Flow Constraints

- The second set of constraints on data-flow values is derived from the flow of control. Within a basic block, control flow is simple.
- If a block B consists of statements s1; s2; : : : ; sn in that order, then the control-flow value out of si is the same as the control-flow value into si+1. That is,

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \ldots, n-1.$$

- However, 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.
- For example, if we are interested in collecting all the definitions that may reach a program point, then the set of definitions reaching the leader statement of a basic block is the union of the definitions after the last statements of each of the predecessor blocks.

3. Data-Flow Schemas on Basic Blocks

- While a data-flow schema technically involves data-flow values at each point in the program.
- Control flows from the beginning to the end of the block, without interruption or branching. Thus, we can restate the schema in terms of data-flow values entering and leaving the 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 s1; : : : ; sn, in that order.
- If s1 is the first statement of basic block B, then IN[B] = IN[s1], Similarly, if sn is the last statement of basic block B, then OUT[B] = OUT[sn].
- The transfer function of a basic block B, which we denote fB, can be derived by composing the transfer functions of the statements in the block.

Data-Flow Schemas on Basic Blocks

Then $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$. The relationship between the beginning and end of the block is

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

The constraints due to control flow between basic blocks can easily be rewritten by substituting $\text{IN}[B]$ and $\text{OUT}[B]$ for $\text{IN}[s_1]$ and $\text{OUT}[s_n]$, 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

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

When 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. That is,

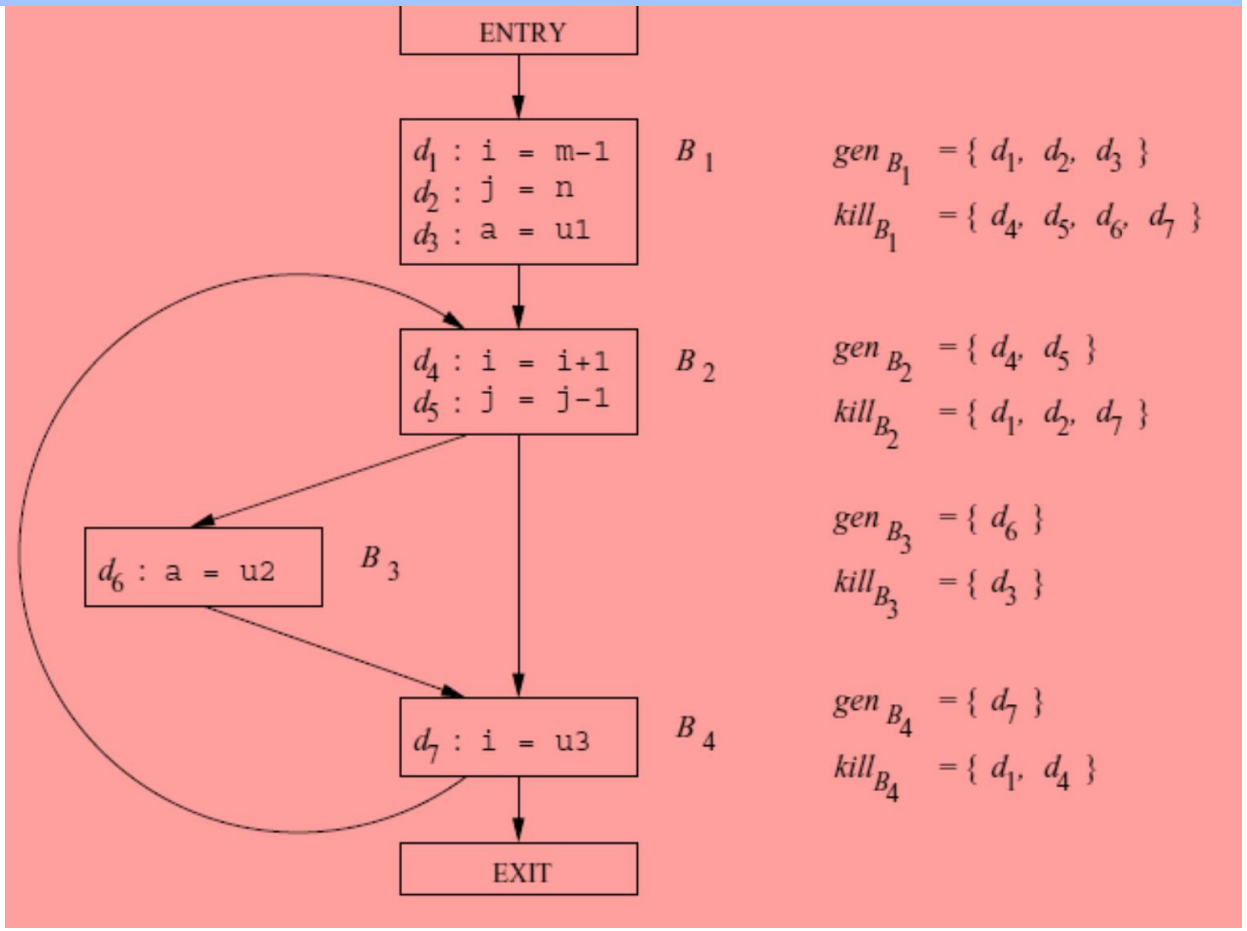
$$\text{IN}[B] = f_B(\text{OUT}[B])$$

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

4. Reaching Definitions

- Reaching definitions is one of the most common and useful data-flow schemas.
- Definition d reaches a point p if there is a path from the point immediately following d to p, 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.
- if we do not know whether a statement s is assigning a value to x, we must assume that it may assign to it; that is, variable x after statement s may have either its original value before s or the new value created by s.

Example 9.9 : Shown in Fig. 9.13 is a flow graph with seven definitions. Let us focus on the definitions reaching block B_2. All the definitions in block B_1 reach the beginning of block B_2. The definition d_5: `j = j-1` in block B_2 also reaches the beginning of block B_2, because no other definitions of j can be found in the loop leading back to B_2. This definition, however, kills the definition d_2: `j = n`, preventing it from reaching B_3 or B_4. The statement d_4: `i = i+1` in B_2 does not reach the beginning of B_2 though, because the variable i is always redefined by d_7: `i = u3`. Finally, the definition d_6: `a = u2` also reaches the beginning of block B_3. □



Transfer Equations for Reaching Definitions

Consider a definition

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

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

where $gen_d = \{d\}$, the set of definitions generated by the statement, and $kill_d$ is the set of all other definitions of u in the program.

Suppose there are two functions

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

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

Transfer Equations for Reaching Definitions

This rule extends to a block consisting of any number of statements. Suppose block B has n statements, with transfer functions $f_i(x) = gen_i \cup (x - kill_i)$ for $i = 1, 2, \ldots, n$. Then the transfer function for block B may be written as:

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

where

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

and

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

5. Live-Variable Analysis

Define

1. def_B as the set of variables *defined* (i.e., definitely assigned values) in B prior to any use of that variable in B, and
2. use_B as the set of variables whose values may be used in B prior to any definition of the variable.

Thus, the equations relating *def* and *use* to the unknowns IN and OUT are defined as follows:

$$\text{IN}[\text{EXIT}] = \emptyset$$

and for all basic blocks B other than EXIT,

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$
$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

The first equation specifies the boundary condition, which is that no variables are live on exit from the program. The second equation says that a variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block. The third equation says that a variable is live coming out of a block if and only if it is live coming into one of its successors.

5. Live-Variable Analysis

Algorithm 9.14: Live-variable analysis.

INPUT: A flow graph with *def* and *use* computed for each block.

OUTPUT: IN[B] and OUT[B], the set of variables live on entry and exit of each block B of the flow graph.

METHOD: Execute the program in Fig. 9.16. □

```
IN[EXIT] = ∅;
for (each basic block B other than EXIT) IN[B] = ∅;
while (changes to any IN occur)
        for (each basic block B other than EXIT) {
                OUT[B] = ⋃_{S a successor of B} IN[S];
                IN[B] = use_B ∪ (OUT[B] − def_B);
        }
```

Figure 9.16: Iterative algorithm to compute live variables

6. Available Expressions

Algorithm 9.17: Available expressions.

INPUT: A flow graph with e_kill_B and e_gen_B computed for each block B. The initial block is B_1.

OUTPUT: IN$[B]$ and OUT$[B]$, the set of expressions available at the entry and exit of each block B of the flow graph.

METHOD: Execute the algorithm of Fig. 9.20. The explanation of the steps is similar to that for Fig. 9.14. □

```
OUT[ENTRY] = ∅;
for (each basic block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
        for (each basic block B other than ENTRY) {
                IN[B] = ⋂_{P a predecessor of B} OUT[P];
                OUT[B] = e_gen_B ∪ (IN[B] − e_kill_B);
        }
```

Figure 9.20: Iterative algorithm to compute available expressions

use-definition (or ud-) chaining is one particular problem

- Given that variable A is used at point p, at which points could the value of A used at p have been defined?
- By a use of variable A means any occurrence of A as an operand.
- By a definition of A means either an assignment to A or the reading of a value for a.
- By a point in a program means the position before or after any intermediate language statement.
  - ✓ Control reaches a point just before a statement when that statement is about to be executed.
  - ✓ Control reaches a point after a statement when that statement has just been executed.

Data Flow equations:

- A definition of a variable **A** reaches a point **p ,**if there is a path in the flow graph from that definition to **p**, such that no other definitions of **A** appear on the path.
- To determine the definitions that can reach a given point in a program requires assigning distinct number to each definition. To compute two sets for each basic block **B:**
  - **GEN[B]-**set of generated definitions within block **B.**
  - **KILL[B]-** set of definitions outside of **B** that define identifiers that also have definition within **B**.

To compute the sets IN[B] and OUT[B]:

IN[B]-set of all definition reaching the point just before the first statement of block B.

OUT[B]-set of definitions reaching the point just after the last statement of B

Data Flow equations:

1. $OUT[B] = IN[B] - KILL[B] \bigcup GEN[B]$
2. $IN[B] = \bigcup_{P\ a\ pred-ecessor\ of\ B} OUT[P]$

Data Flow equations:

1. $OUT[B] = IN[B] - KILL[B] \bigcup GEN[B]$
2. $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$

- The rule (1) says that a definition d reaches the end of the block B if and only if either
  - d is in IN[B], i.e d reaches the beginning of B and is not killed by B ,or
  - d is generated within B i.e., it appears in B and its identifier is not subsequently redefined within B.
- The rule (2) says that a definition reaches the beginning of the block B if and only if it reaches the end of one of its predecessors.

Algorithm for reaching definition:

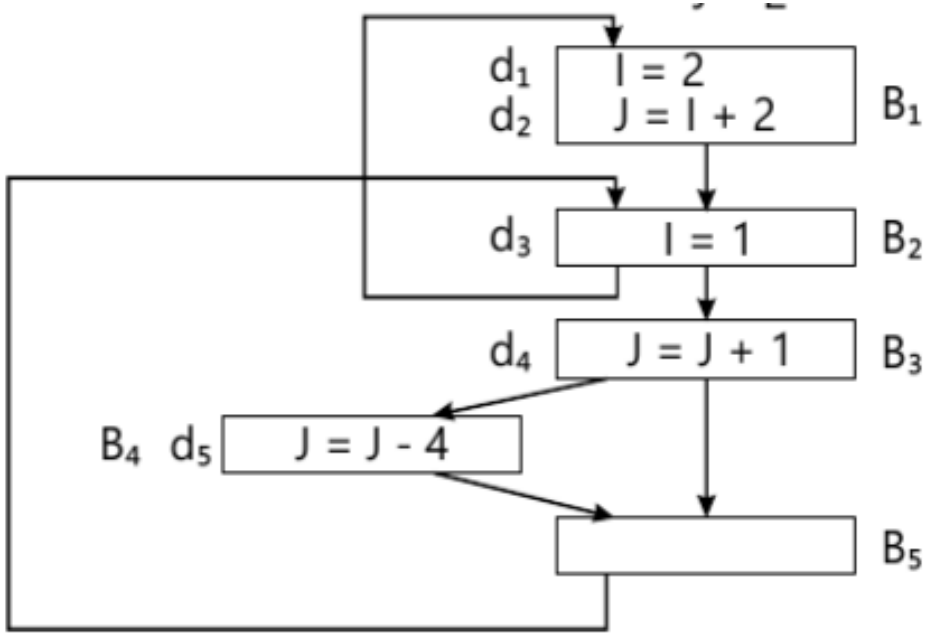
Input: A flow graph for which GEN[B] and KILL[B] have been computed for each block B.

Output: IN[B] and OUT[B] for each block B.

Method: An iterative approach, initializing with IN[B]=Ø for all B and converging to the desired values of IN and OUT.

```
begin
    for each block B do
    begin
            IN[B]=Ø
            OUT[B]=GEN[B]
    end
CHANGE=true
while CHANGE do
    begin
            CHANGE=false
            for each block B do
                begin
                NEWIN = ⋃ OUT [P]   (P a predecessor of B)

                if NEWIN≠IN[B] then CHANGE=true
                IN[B]=NEWIN
                OUT[B] = IN[B] – KILL[B] ⋃ GEN[B]

                end
```

Example: Consider the Flow graph:



Solution:

i) Generate GEN(B) and KILL(B) for all blocks:

Block B	GEN[B]	Bit vector	KILL[B]	Bit vector
B1	$\{d_1,d_2\}$	11000	$\{d_3,d_4,d_5\}$	00111
B2	$\{d_3\}$	00100	$\{d_1\}$	10000
B3	$\{d_4\}$	00010	$\{d_2,d_5\}$	01001
B4	$\{d_5\}$	00001	$\{d_2,d_4\}$	01010
B5	$\emptyset$	00000	$\emptyset$	00000

Computation For block B1 :

NEWIN=OUT[B2] , since B2 is the only predecessor of B1

NEWIN=GEN[B2]=00100 = IN[B1]

OUT[B1] = IN[B1] – KILL[B1] + GEN[B1]

= 00100 – 00111 + 11000

= 11000

For block B2 :

NEWIN=OUT[B1] + OUT[B5] , since B1 and B5 are the predecessor of B2

IN[B2] = 11000 + 00000 = 11000

OUT[B2] = IN[B2] – KILL[B2] + GEN[B2]

= 11000 – 10000 + 00100

= 01100

As iteration 4 and 5 produces same values, the process ends. Hence the value of definition anywhere at any point of time is deduced.

Computing ud-chains:

- From reaching definitions information we can compute ud-chains.
- If a use of variable A is preceded in its block by a definition of A, then only the last definition of A in the block prior to this use reaches the use. – i.e. ud-chain for this use contains only one definition.
  - ✓ If a use of A is preceded in its block B by no definition of A, then the ud-chain for this use consists of all definitions of A in IN[B].
  - ✓ Since ud-chain take up much space, it is important for an optimizing compiler to format them compactly.

Initial pass

Block B	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass-1

Block B	IN[B]	OUT[B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

Pass-2

Block B	IN[B]	OUT[B]
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass-3

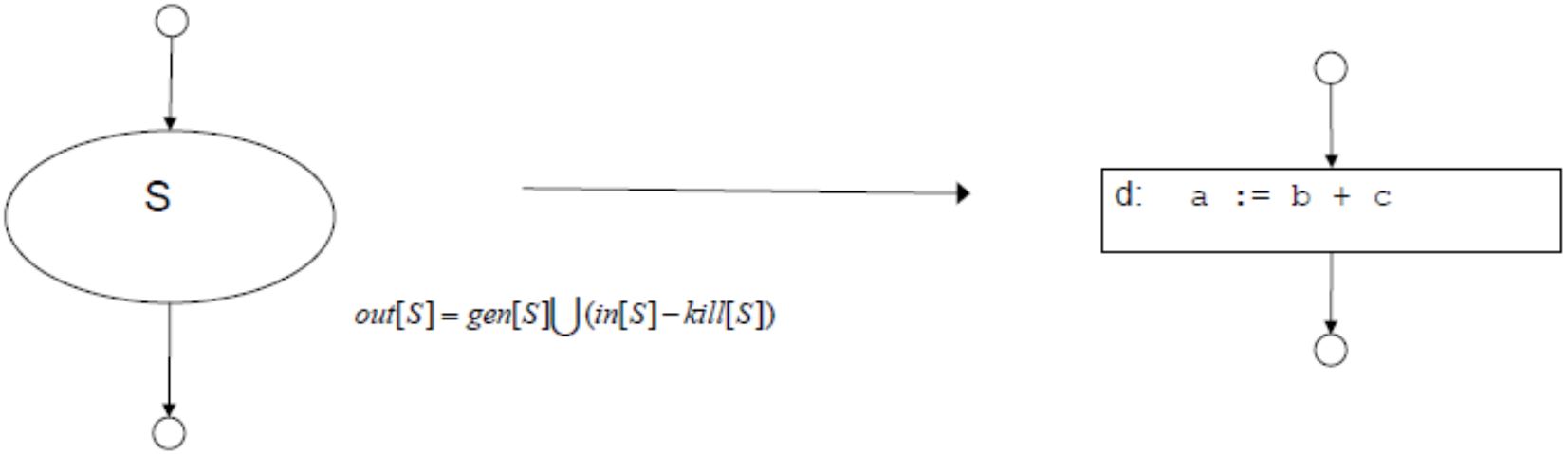
Block B	IN[B]	OUT[B]
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Applications:

- Knowing the use-def and def-use chains are helpful in compiler optimizations including constant propagation and common sub-expression elimination.
- Helpful in dead-code elimination.
- Instruction reordering.
- (Implementation of) scoping/shadowing.

Data Flow Equations Single statement

$$gen[S] = \{d\}$$
$$kill[S] = D_a - \{d\}$$



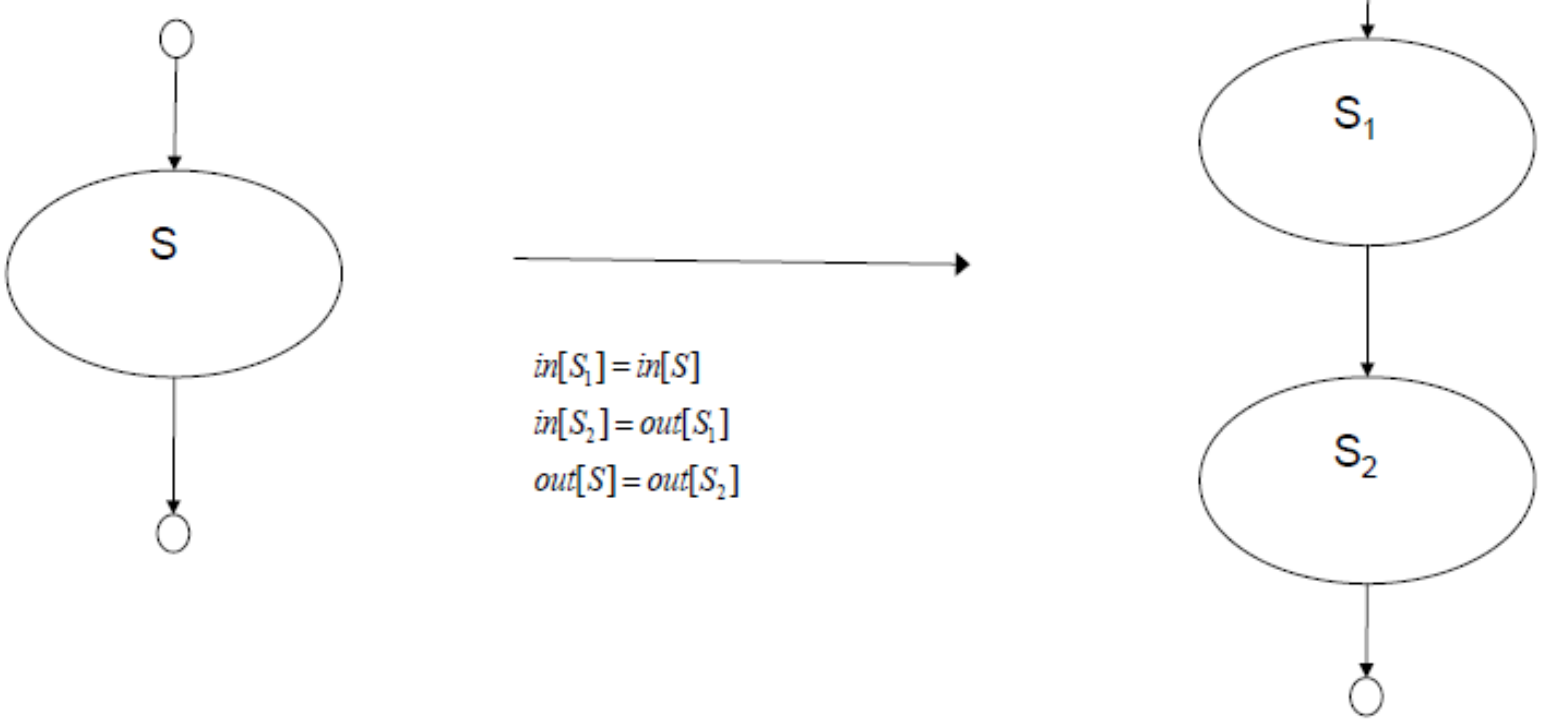
$$out[S] = gen[S] \bigcup (in[S] - kill[S])$$

D_a: The set of definitions in the program for variable a

Data Flow Equations Composition

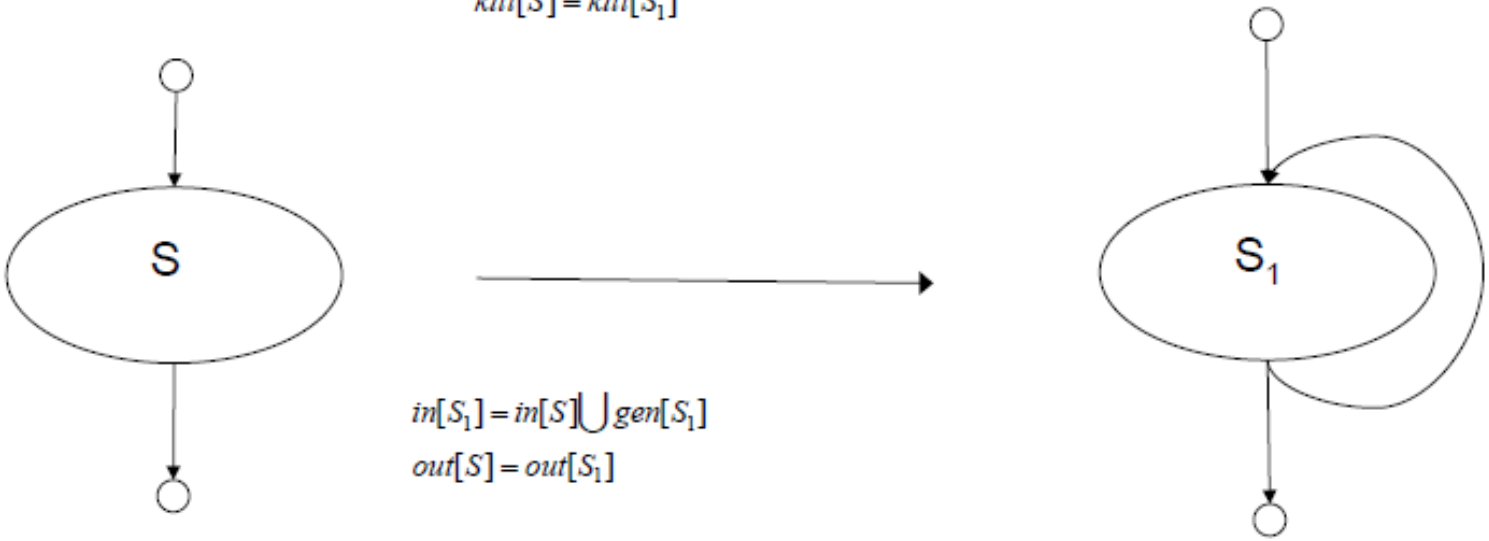
$$gen[S] = gen[S_2] \bigcup (gen[S_1] - kill[S_2])$$

$$kill[S] = kill[S_2] \bigcup (kill[S_1] - gen[S_2])$$



Data Flow Equations if-then-else

$$gen[S] = gen[S_1]$$
$$kill[S] = kill[S_1]$$



$$in[S_1] = in[S] \bigcup gen[S_1]$$
$$out[S] = out[S_1]$$

Issues in the Design of a Code Generator

- General tasks in almost all code generators: instruction selection, register allocation and assignment.
  - ❖ The details are also dependent on the specifics of the intermediate representation, the target language, and the run-time system.
- The most important criterion for a code generator is that it produce correct code.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

1.Input to the Code Generator

- The input to the code generator is
  - the intermediate representation of the source program produced by the front end along with
  - information in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the IR.

- Choices for the IR
  - Three-address representations: quadruples, triples, indirect triples
  - Virtual machine representations such as byte codes and stack-machine code
  - Linear representations such as postfix notation
  - Graphical representation such as syntax trees and DAG's

- Assumptions
  - Relatively lower level IR
  - All syntactic and semantic errors are detected.

3. Instruction Selection

- The nature of the instruction set of the target machine determines the difficulty of the instruction selection.
- Uniformity and completeness of the instruction set are important
- Instruction speeds is also important

Say, **x = y + z**

- ***Mov y, R0*** ***//load y into register R0***
- ***Add z, R0*** ***//add Z to R0***
- ***Mov R0, x*** ***// store R0 into X***

- Statement by statement code generation often produces poor code

- The quality of the generated code is determined by its speed and size.
- Cost difference between the different implementation may be significant.
  - Say a = a + 1

    Mov a, R0

    Add #1, R0

    Mov R0, a
  - If the target machine has increment instruction (INC), we can write

    inc a

4.Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
  - During register allocation, we select the set of variables that will reside in register at a point in the program.
  - During a subsequent register assignment phase, we pick the specific register that a variable will reside in.
- Certain machine requires register-pairs for some operands and results.

5 Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- However, picking a best order in the general case is a difficult NP-complete problem

Instruction cost

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
absolute	M	M	Add R0, R1	1
register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1
indirect register	*R	contents(R)	ADD * 100	0
indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

Instruction cost

Cost of an instruction = 1 + cost associated with the source and destination address modes

❖This cost corresponds to the length of the instruction.

❖Here, cost 1 means that it occupies only one word of memory.

❖Address modes involving registers have cost zero, while those with a memory location or literal in them have cost one.

Example:

1. Move register to memory R0 → M

MOV R0, M

cost = 1+1+1 (since address of memory location M is in word following the instruction)

2. Indirect indexed mode:

MOV * 4(R0), M

cost = 1+1+1 (since one word **for** memory location M, one word **for** result of *4(R0) and one **for** instruction)

3. Literal Mode: MOV #1, R0

cost = 1+1+1 = 3 (one word **for** constant 1 and one **for** instruction)

Code Generation:

1. Introduction;

- Consider each statement
- Remember if operand is in a register
- Uses Descriptors to keep track of register Contents and Address for names

2. Register descriptor

- Keep track of what is currently in each register.
- Whenever a new register is needed it will be consulted
- Initially all the registers are empty
- In progress register will hold the value of '0/More' Names

3. Address descriptor

- Keep track of location where current value of the name can be found at runtime
- The location might be a register, stack, memory address or a set of these
- Information about this is stored in the symbol table
- Information can be used to determine the accessing method for a name.

Generation of target code for a sequence of 3-Address Statement: X := Y op Z

1. Basic Assumptions

- ∀operators in SL ∃ a target Language Operator

- Computed results can be left in register until either
  - ✓ If the register is needed for another computation

    OR
  - ✓ Just before a procedure call / jump / labeled statement.

2. Code Gen Algorithm

Input: A sequence of three address statements that constitutes a basic block.

Ex. X := Y op Z

Actions

Step1: Invoke getReg() function

- ❖ To determine the location L where the result of the computation y op z should be stored.
- ❖ L will be a register or be a memory location.

Step 2: Access the Address descriptor for Y

- To determine the current location of Y
- If the value of Y is currently both in memory and a register. then choose Register. for Y'.
- If (value of Y is not in L) then
  - Generate the instruction MOV Y,L to place a copy in L

Step 3: Generate the Instruction op Z', L

- Z' is the current location of Z
- To determine the current location of Z
- If the value of Z is currently both in memory and a register. then choose Register. for Z'.
- Update the address descriptor of X to indicate that X is in current location L
- If L is in R update its descriptor that it contains the value of X
- Remove X from all other descriptors

Step 4: If Y & Z have No Next Use && are not live on exit from block

- Then alter the register descriptor

If the 3 – Address Statement. Has a Unary Operator
X := Y

- Same as previous
- If Y is in Register
  - Change the register and address Descriptor.
  - To record the value of X is in the register holding the value of Y
  - If Y has no next use && Not live on exit
    - The register no longer holds the value of Y
- Else Y is only in memory
  - Use getreg () to find a register to load Y and make that register. as the location of X
  - Alternate Solution: Generate a MOV Y, X instruction. If X has no next use.

- After processing
  - Store the names that “Live on Exit” && “Not in their Memory Locations.” by using MOV instruction.

- Steps Involved
  - Use Register Descriptor to determine what names are left in the registers
  - Use Address Descriptor to determine that the same name is not already in its memory location
  - Use the Live variable information. To determine if the name is to be stored
  - If no Live information. Then assume that the entire user defined names live at the end of the block.

The Functions getreg

- 1. If (Y is in register that holds no other values) and
  - (Y is not live) and (has no next use after execution)
  - then return register of Y for L.
  - update the address descriptor of Y

- 2. Failing (1) return an empty register

- 3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.

- 4. Else select memory location X as L

Example

Code sequence

The assignment

d := (a - b) + (a - c) + (a - c)

translated into the following three- address code sequence:

t = a - b u = a – c v = t + u d = v + u

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	R0 contains d	d in R0
	MOV R0, d		d in R0 and memory

Fig. 9.10. Code sequence.

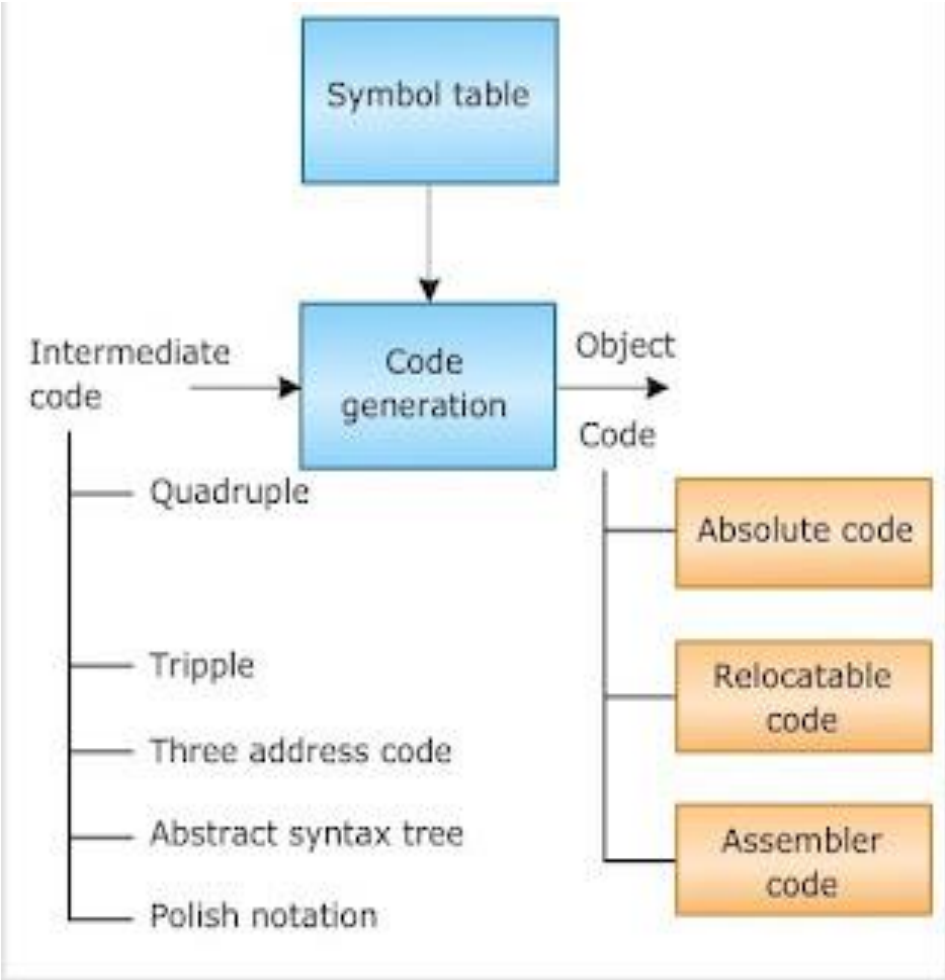
Object Code Forms

The output generated by code generator is an object code or machine code.
This code typically appears in the below forms

1. Absolute code
2. Relocatable Machine Code
3. Assembly Code

1. Absolute code:

- Absolute code is a machine code that consists of reference to actual addresses within programs address space
- The generated code can be placed directly in the memory, and execution begins quickly.
- With the help of absolute code generation, the compilation and execution of small program is done rapidly



Object Code Forms

2. Relocatable Machine Code:

- The compilation of subprograms can be carried out separately whenever relocatable machine language program is generated as output.
- A set of relocatable object modules can be linked together and can be loaded for execution with the aid of a linking loader
- The prime advantage of producing relocatable machine code is that one gains more flexibility in the process of compiling subroutines independently and calling other previously compiled projects from an object module
- When the target machine does not handle relocation automatically, the compiler must give explicit relocation data to the loader to link the independently compiled program segments

3. Assembly code:

- The code generation process becomes little easier when an assembly language program is produced as output.
- In the process of generation of code, the symbolic instructions can be generated, and assembler's macro facilities can be used.
- But the generation of assembler code as an output slows down the code generation process, since assembling, linking and loading are required.

Peephole optimization

- A simple but effective technique for locally improving the target code is peephole optimization, which 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.
- It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

characteristic of peephole optimizations:

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

Eliminating Redundant Loads and Stores

If we see the instruction sequence

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

- we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been
- loaded into register R0. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction.
- The two instructions have to be in the same basic block for this transformation to be safe.

Eliminating Unreachable Code

An unlabeled instruction immediately following an unconditional jump may be removed.

```
        if debug == 1 goto L1
        goto L2
L1:     print debugging information
L2:
```

```
        if debug != 1 goto L2
        print debugging information
L2:
```

If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
        if 0 != 1 goto L2
        print debugging information
L2:
```

Flow-of-Control Optimizations

- Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

```
        goto L1
        ...
   L1: goto L2
```

by the sequence

```
        goto L2
        ...
   L1: goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement `L1: goto L2` provided it is preceded by an unconditional jump.

Similarly, the sequence

```
        if a < b goto L1
        ...
   L1: goto L2
```

can be replaced by the sequence

```
        if a < b goto L2
        ...
   L1: goto L2
```

```
        goto L1
        . . .
   L1: if a < b goto L2
   L3:
```

may be replaced by the sequence

```
        if a < b goto L2
        goto L3
        . . .
   L3:
```

Algebraic Simplification and Reduction in Strength

- These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

x = x + 0

or

x = x * 1 in the peephole.

Reduction in Strength

- Replace expensive operations by cheaper ones
- Replace X^2 by X*X
- Replace multiplication by left shift
- Replace division by right shift
- Use faster machine instructions
- Replace **Add #1,R** by **Inc R**

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.
- These modes can also be used in code for statements like x = x+1.

Register Allocation and Assignment

- Instructions involving only register operands are faster than those involving memory operands.
- Therefore, efficient utilization of registers is vitally important in generating good code.
- Presents various strategies for deciding at each point in a program what values should reside in registers (register allocation) and in which register each value should reside (register assignment).
- Assign base addresses to one group of registers, arithmetic computations to another, the top of the stack to a fixed register.
- This approach has the advantage that it simplifies the design of a code generator.
- Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers.

Global Register Allocation

- The code generation algorithm used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block.
- To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally).
- Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.
- This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation.
- With early C compilers, a programmer could do some register allocation explicitly by using register declarations to keep certain values in registers for the duration of a procedure.
- Judicious use of register declarations did speed up many programs, but programmers were encouraged to first profile their programs to determine the program's hotspots before doing their own register allocation.

Usage Counts

- Assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register.
- However, if we use the approach to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block.
- Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block.
- Thus, if x is allocated a register, we count a savings of two for each block in loop L for which x is live on exit and in which x is assigned a value.
- On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units.
- Similarly, for each exit block B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two.

Usage Counts

- However, on the assumption that the loop is iterated many times, we may neglect these debits since they occur only once each time we enter the loop. Thus, an approximate formula for the benefit to be realized from allocating a register for x within loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B) \qquad (8.1)$$

- where use(x, B) is the number of times x is used in B prior to any definition of x; live(x, B) is 1 if x is live on exit from B and is assigned a value in B, and live(x, B) is 0 otherwise.

Register Assignment for Outer Loops

- Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops.
- If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1 - L2.
- However, if we choose to allocate x a register in L2 but not L1, we must load x on entrance to L2 and store x on exit from L2.

Example 8.17: Consider the basic blocks in the inner loop depicted in Fig. 8.17, where jump and conditional jump statements have been omitted. Assume registers R0, R1, and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig. 8.17 for convenience, immediately above and below each block, respectively. There are some subtle points about live variables that we address in the next chapter. For example, notice that both e and f are live at the end of B_1, but of these, only e is live on entry to B_2 and only f on entry to B_3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

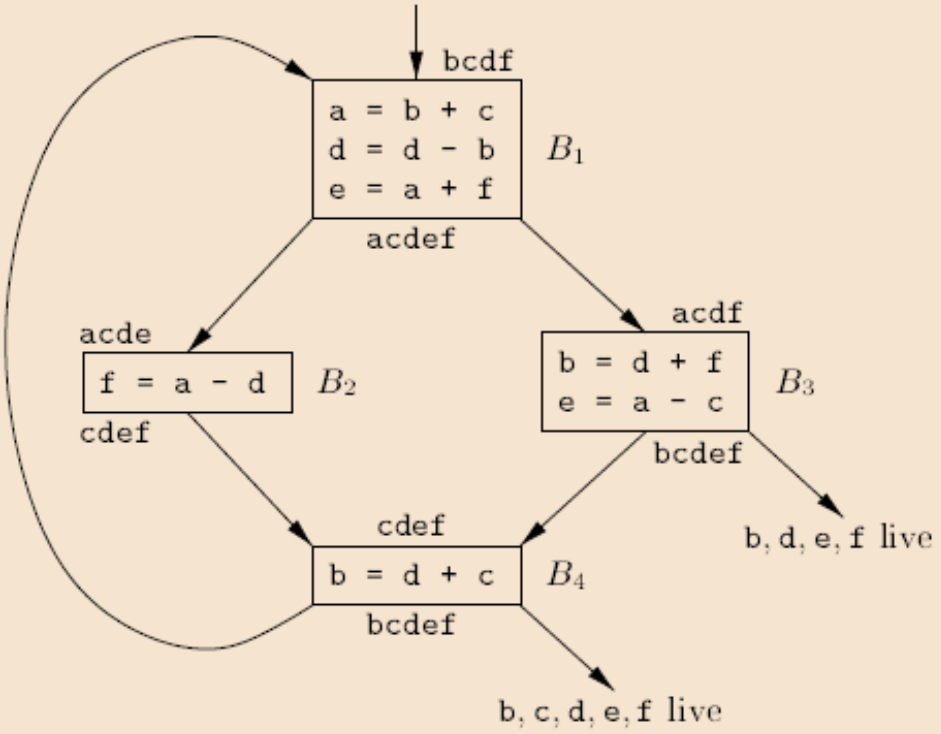


Figure 8.17: Flow graph of an inner loop

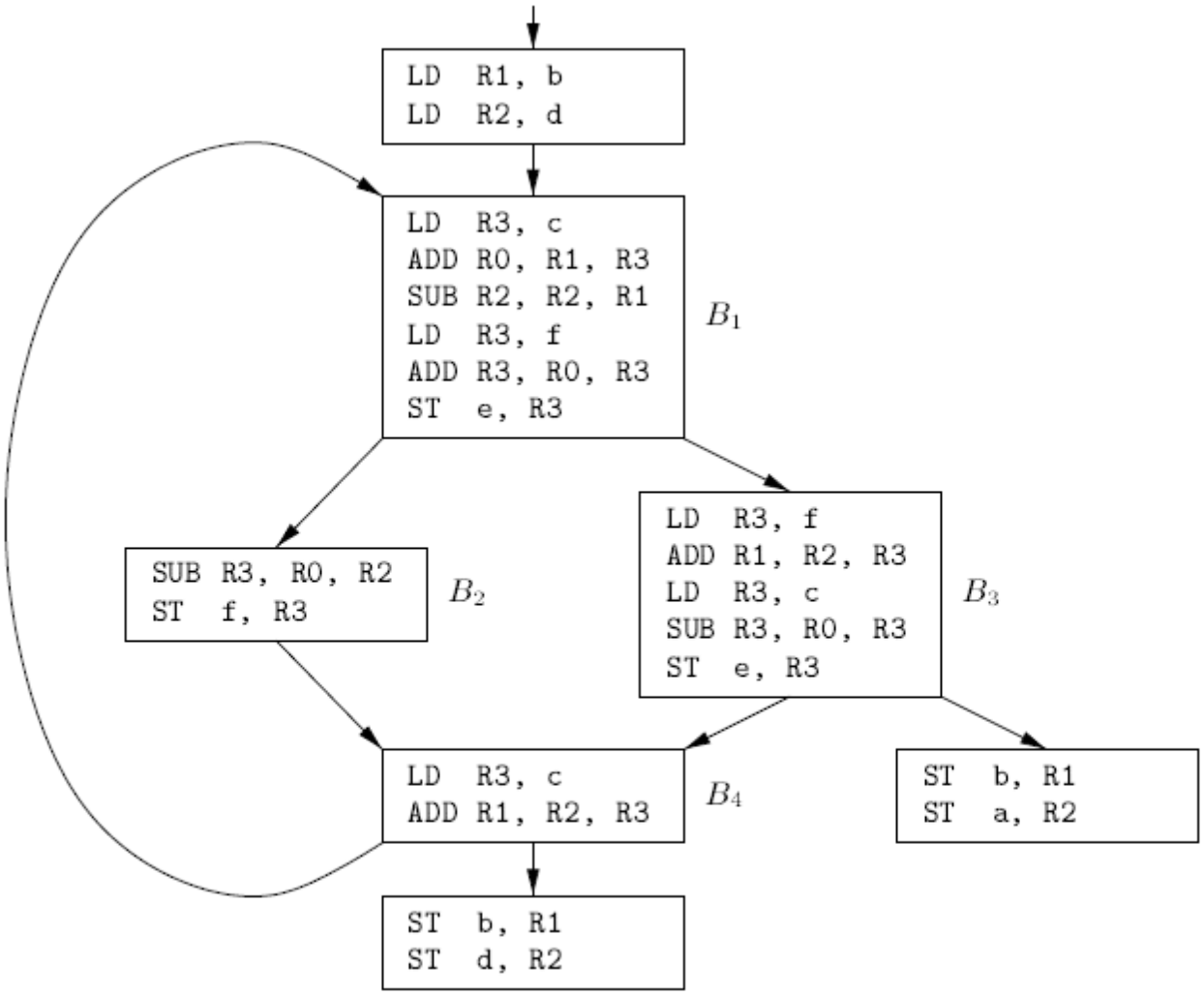


Figure 8.18: Code sequence using global register assignment

Register Allocation by Graph Coloring

- When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (spilled) into a memory location in order to free up a register.
- Graph coloring is a simple, systematic technique for allocating registers and managing register spills.
- In the method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and the three-address instructions become machine-language instructions.
- If access to variables requires instructions that use stack pointers, display pointers, base registers, or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose.
- Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction.

Register Allocation by Graph Coloring

- If access is more complex, the access must be broken into several machine instructions, and a temporary symbolic register (or several) may need to be created.
- Once the instructions have been selected, a second pass assigns physical registers to symbolic ones.
- The goal is to find an assignment that minimizes the cost of spills.
- In the second pass, for each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.
- For example, a register interference graph would have nodes for names a and d. In block B1, a is live at the second statement, which defines d; therefore, in the graph there would be an edge between the nodes for a and d.
- An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers.
- A graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color.

Register Allocation by Graph Coloring

- A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.
- Although the problem of determining whether a graph is k-colorable is NP complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice.
- Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G0.
- A k-coloring of G0 can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.
- By repeatedly eliminating nodes having fewer than k edges from the register interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible.
- At this point a node is spilled by introducing code to store and reload the register.