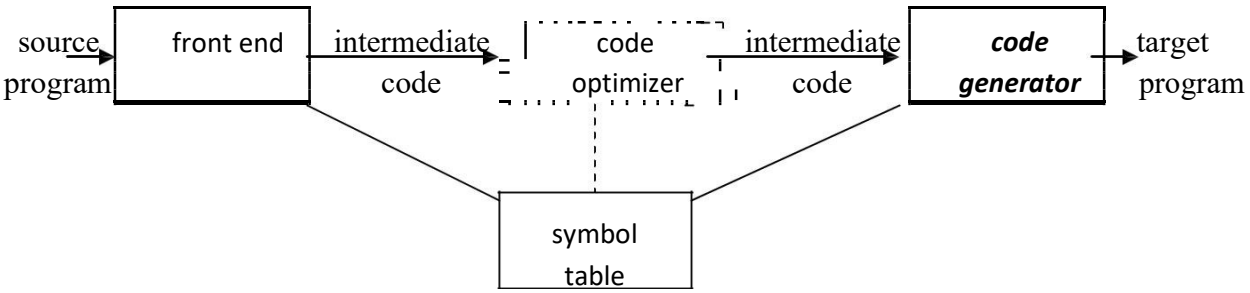
Unit-4
CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :
  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such as syntax trees and dags.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
     - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled

separately. c. Assembly language

- Code generation is made easier.

3. Memory management:

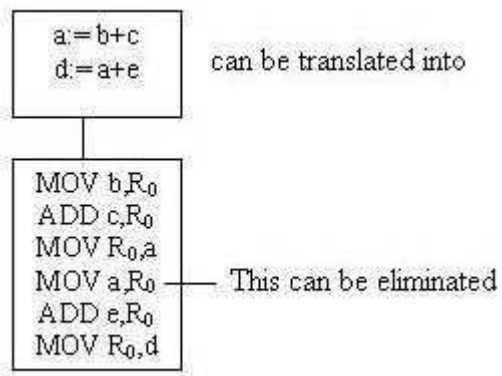
- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,

  j : **goto** *i* generates jump instruction as follows :

  - if *i* < *j*, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
  - if *i* > *j*, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j.* When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :
  - ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

¾ ***Register assignment*** – the specific register that a variable will reside in is picked

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register
  pair y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, $R_0, R_1, \ldots, R_{n-1}$.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

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

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ;
  MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

  i) MOV b, R_0
  ADD c, R_0 cost = 6
  MOV R_0, a

  ii) MOV b, a
  ADD c, a cost = 6

  iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c
  : MOV *R_1, *R_0
  ADD *R_2, *R_0 cost = 2

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

RUN-TIME STORAGE MANAGEMENT

- Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.
- The two standard storage allocation strategies are:
  1. Static allocation
  2. Stack allocation
- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.
- The following three-address statements are associated with the run-time allocation and deallocation of activation records:
  1. Call,
  2. Return,
  3. Halt, and
  4. Action, a placeholder for other statements.
-
  1. Code
  2. Static data
  3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* + 20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,
callee.static_area – Address of the activation record
callee.code_area – Address of the first instruction for called procedure
#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :
GOTO **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV *#stackstart* , SP /* initializes stack */

Code for the first procedure

HALT /* terminate execution */

Implementation of Call statement:

ADD *#caller.recordsize*, SP /* increment stack pointer */

MOV *#here* + 16, *SP /*Save return address */

GOTO *callee.code_area*

where,
caller.recordsize – size of the activation record
#here + 16 – address of the instruction following the **GOTO**

Implementation of Return statement:

GOTO *0 (SP) /*return to the caller */

SUB *#caller.recordsize*, SP /* decrement SP and restore to previous value */

BASIC BLOCKS AND FLOW GRAPHS

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 any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
  t_1 : = a * a
  t_2 : = a * b
  t_3 : = 2 * t_2
  t_4 : = t_1 + t_3
  t_5 : = b * b
  t_6 : = t_4 + t_5

Basic Block Construction:

Algorithm: 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 of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. 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.

- Consider the following source code for 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
```

The three-address code for the above source program is given as :

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

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

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

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

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

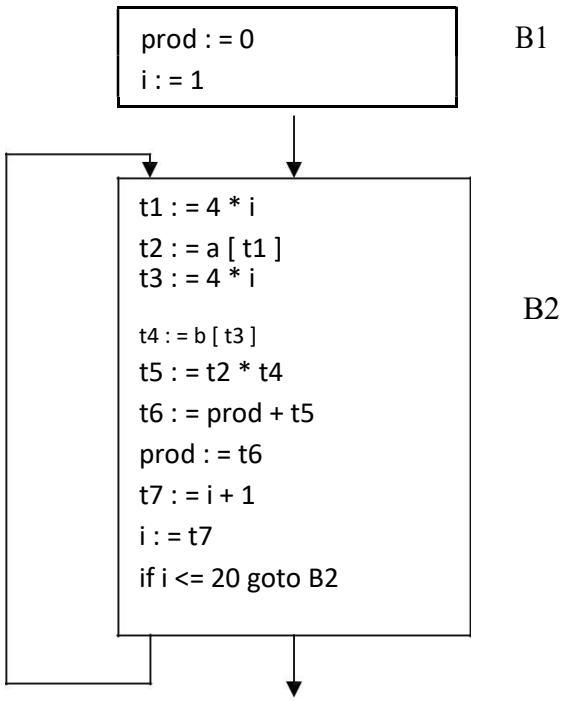
We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is t_1 and neither **b** nor **c** is t_2.

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.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



- B_1 is the *initial* node. B_2 immediately follows B_1, so there is an edge from B_1 to B_2. The target of jump from last statement of B_1 is the first statement B_2, so there is an edge from B_1 (last statement) to B_2 (first statement).
- B_1 is the *predecessor* of B_2, and B_2 is a *successor* of B_1.

Loops

- A loop is a collection of nodes in a flow graph such that
  1. All nodes in the collection are *strongly connected*.
  2. The collection of nodes has a unique *entry*.
- A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

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.

Symbol Table:

Names	Liveliness	Next-use
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.
- For example: consider the three-address statement **a := b+c**

  It can have the following sequence of codes:

```
ADD Rj, Ri      Cost = 1    // if Ri contains b and Rj contains c
        (or)
ADD c, Ri       Cost = 2    // if c is in a memory location
        (or)
MOV c, Rj       Cost = 3    // move c from memory to Rj and add
ADD Rj, Ri
```

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.
2. Consult the address descriptor for y to determine y‟, the current location of y. Prefer the register for y‟ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.
3. Generate the instruction **OP z' , L** where z‟ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

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

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

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements **a : = b [i]** and **a [i] : = b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments **a : = *p** and ***p : = a**

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
    1. Leaves are labeled by unique identifiers, either variable names or constants.
    2. Interior nodes are labeled by an operator symbol.
    3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub-expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

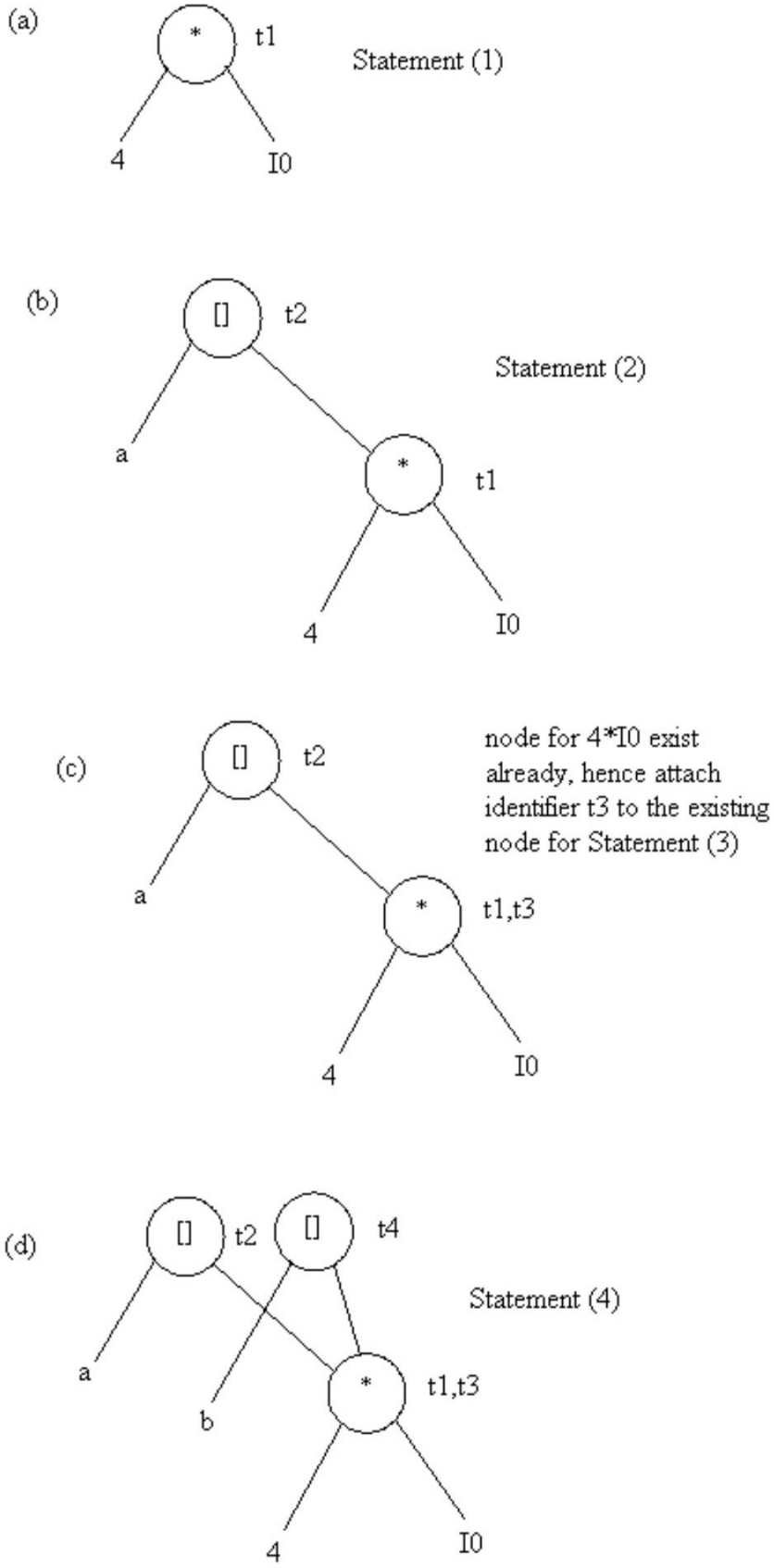
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

identifiers for the node n found in step 2 and set node(x) to n.

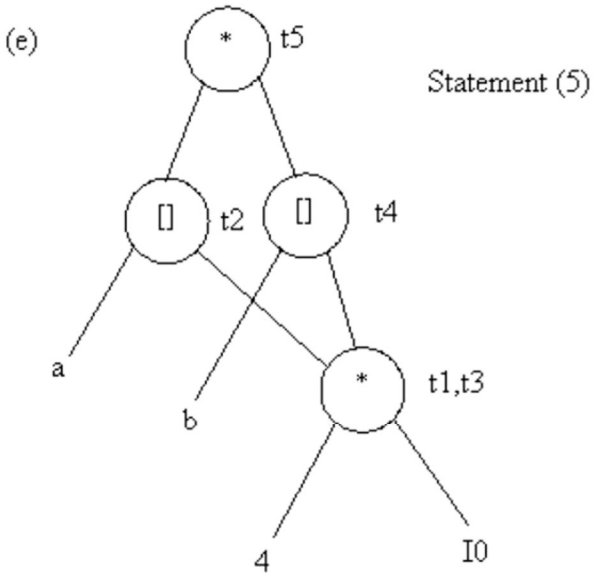
Example: Consider the block of three- address statements:

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

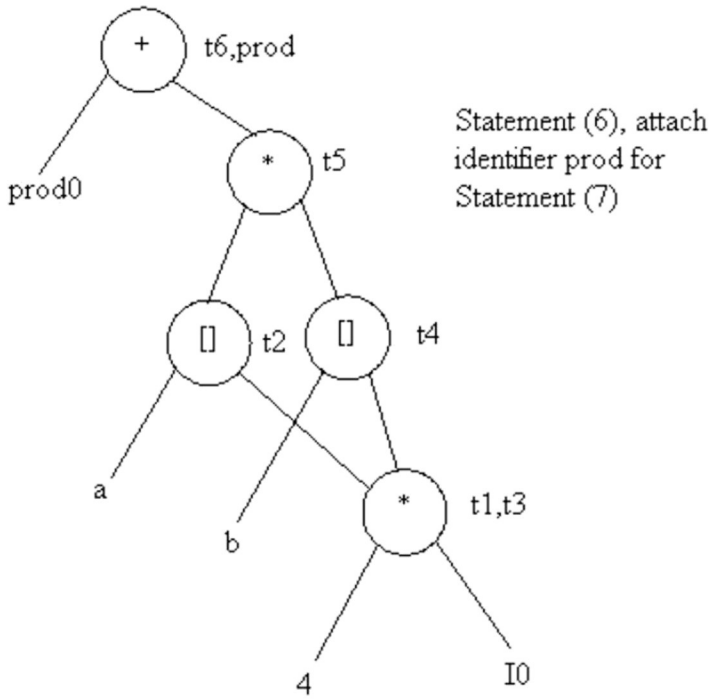
Stages in DAG Construction



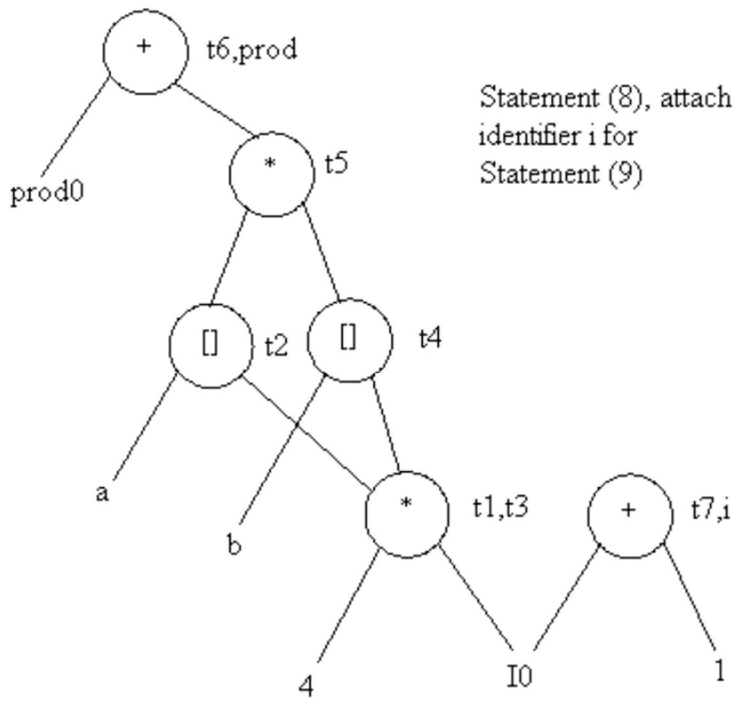
(e)



(f)

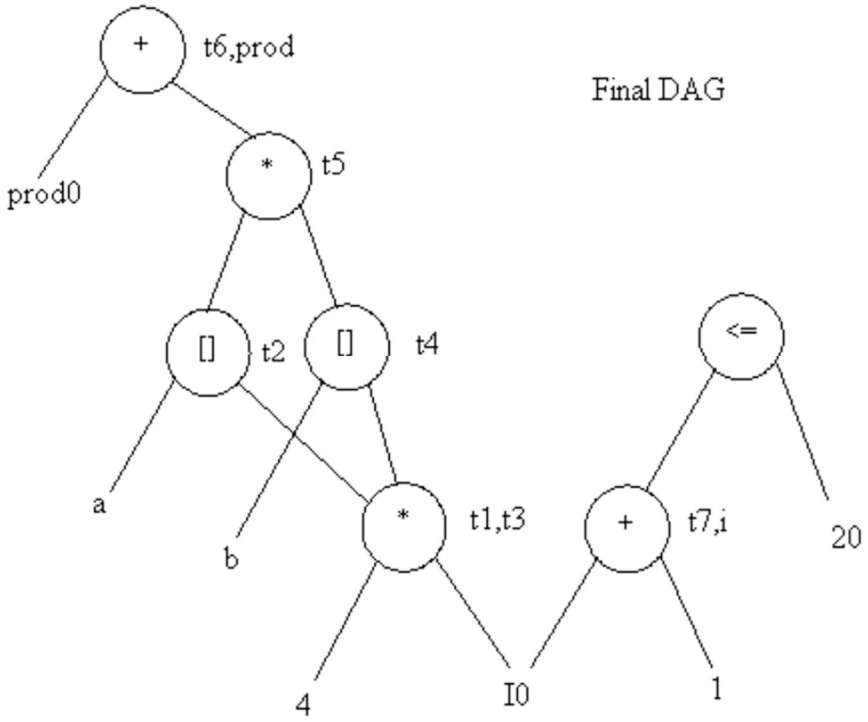


(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

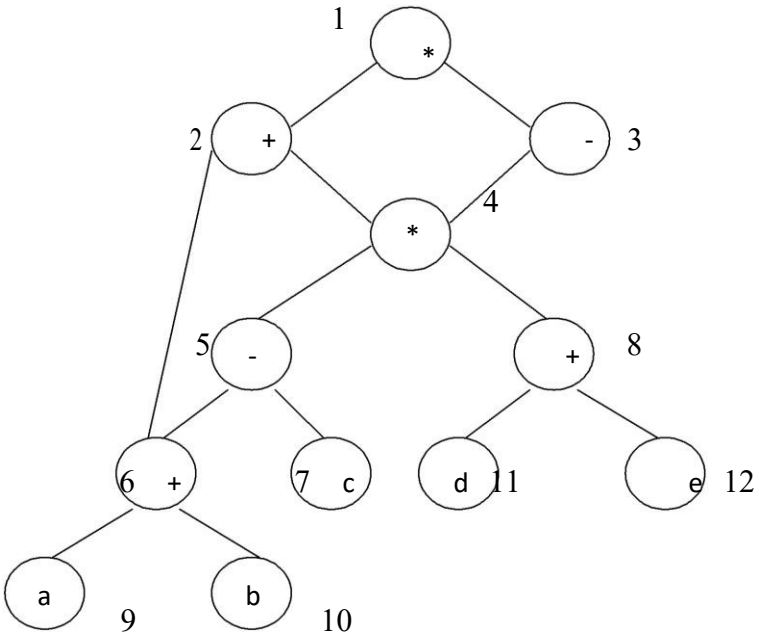
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1)  while unlisted interior nodes remain do begin
2)      select an unlisted node n, all of whose parents have been listed;
3)      list n;
4)      while the leftmost child m of n has no unlisted parents and is not a leaf do
            begin
5)              list m;
6)              n : = m
            end
        end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

$t_8 := d + e$

$t_6 := a + b$

$t_5 := t_6 - c$

$t_4 := t_5 * t_8$

$t_3 := t4 - e$

$t_2 := t_6 + t_4$

$t_1 := t_2 * t_3$

This will yield an optimal code for the DAG on machine whatever be the n umber of registers.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:
    - Redundant-instructions elimination
    - Flow-of-control optimizations
    - Algebraic simplifications
    - Use of machine idioms
    - Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For ex ample, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0

….

If ( debug ) {

Print debugging information

}
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2

  goto L2

L1: print debugging information

L2:                                   ……(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2

Print debugging information

L2:                                   ……(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: ……(c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or th e target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

….

L1: gotoL2

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

If a < b goto L2

….

L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

……..

L1: if a < b goto L2

L3: ……..(1)

- May be replaced by

If a < b goto L2

goto L3

…….

L3: ……(2)

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0

Or

x := x * 1

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. 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 these 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 i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -