UNIT-5

CODE GENERATOR

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three-address statement.

Example:

Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R_0 ADD y, R_0

Register and Address Descriptors:

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

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

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

Generating Code for Assignment Statements:

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

1. t:= a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

Code sequence for the example is as follows:

Statement	**Code Generated**	**Register descriptor Register empty**	**Address descriptor**
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 R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Design Issues

In the code generation phase, various issues can arises:

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

1. Input to the code generator

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:
a) **Assembly language:** It allows subprogram to be separately compiled.
b) **Relocatable machine language:** It makes the process of code generation easier.
c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:
The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

1. MOV b, R0 R0→b
2. ADD c, R0 R0 c + R0
3. MOV R0, a a → R0
4. MOV a, R0 R0→ a
5. ADD e, R0 R0 → e + R0
6. MOV R0, d d → R0

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:
Register allocation: In register allocation, we select the set of variables that will reside in register.
Register assignment: In Register assignment, we pick the register that contains variable.
Certain machine requires even-odd pairs of registers for some operands and result.

For example:
Consider the following division instruction of the form:

1. D x, y

Where,
x is the dividend even register in even/odd register pair
y is the divisor
Even register is used to hold the reminder.
Old register is used to hold the quotient.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

Target Machine

- The target computer is a type of byte-addressable machine. It has 4 bytes to a word.
- The target machine has n general purpose registers, R0, R1,...., Rn-1. It also has two-address instructions of the form:

1. op source, destination

Where, op is used as an op-code and source and destination are used as a data field.

- It has the following op-codes:
  ADD (add source to destination)
  SUB (subtract source from destination)
  MOV (move source to destination)
- The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

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

- Here, cost 1 means that it occupies only one word of memory.
- Each instruction has a cost of 1 plus added costs for the source and destination.
- Instruction cost = 1 + cost is used for source and destination mode.

Example:

1. Move register to memory R0 → M

1. MOV R0, M
2. cost = 1+1+1 (since address of memory location M is in word following the instruction)

2. Indirect indexed mode:

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

3. Literal Mode:

1. MOV #1, R0
2. cost = 1+1+1 = 3 (one word **for** constant 1 and one **for** instruction)

RUN-TIME STORAGE MANAGEMENT

The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure.

We can describe address in the target code using the following ways:

1. Static allocation
2. Stack allocation

In static allocation, the position of an activation record is fixed in memory at compile time.
In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

For the run-time allocation and deallocation of activation records the following three-address statements are associated:

1. Call
2. Return
3. Halt
4. Action, a placeholder for other statements

We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack

Static allocation:

1. Implementation of call statement:

The following code is needed to implement static allocation:

1. MOV #here + 20, callee.static_area /*it saves return address*/</p>
2. GOTO callee.code_area /* It transfers control to the target code for the called procedure*/

Where,
callee.static_area shows the address of the activation record.
callee.code_area shows the address of the first instruction for called procedure.
#here + 20 literal are used to return address of the instruction following GOTO.

2. Implementation of return statement:

The following code is needed to implement return from procedure callee:

1. GOTO * callee.static_area

It is used to transfer the control to the address that is saved at the beginning of the activation record.

3. Implementation of action statement:
The ACTION instruction is used to implement action statement.

4. Implementation of halt statement:
The HALT statement is the final instruction that is used to return the control to the operating system.

Stack allocation
Using the relative address, static allocation can become stack allocation for storage in activation records.
In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.
The following code is needed to implement stack allocation:

1. Initialization of stack:

1. MOV #stackstart , SP /*initializes stack*/
2. HALT /*terminate execution*/

2. Implementation of Call statement:

1. ADD #caller.recordsize, SP/* increment stack pointer */
2. MOV #here + 16, *SP /*Save return address */
3. GOTO callee.code_area

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

3. Implementation of Return statement:

1. GOTO *0 (SP) /*return to the caller */
2. SUB #caller.recordsize, SP /*decrement SP and restore to previous value */

Basic Block

Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block).

The following sequence of three address statements forms a basic block:

1. t1:= x * x
2. t2:= x * y
3. t3:= 2 * t2
4. t4:= t1 + t3
5. t5:= y * y
6. t6:= t4 + t5

Basic block construction:

Algorithm: Partition into basic blocks
Input: It contains the sequence of three address statements
Output: it contains a list of basic blocks with each three address statement in exactly one block
Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.

Consider the following source code for dot product of two vectors a and b of length 10:

1. begin
2. prod :=0;
3. i:=1;
4. do begin
5. prod :=prod+ a[i] * b[i];
6. i :=i+1;
7. end
8. while i <= 10
9. end

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

B1

1. (1) prod := 0
2. (2) i := 1

B2

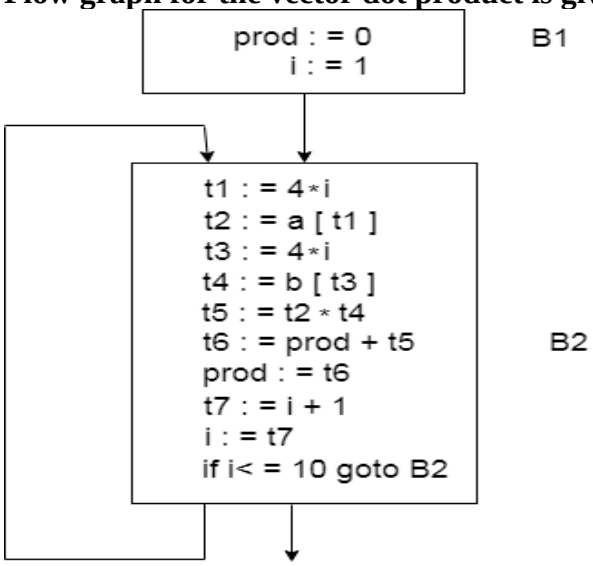
1. (3) t1 := 4* i
2. (4) t2 := a[t1]
3. (5) t3 := 4* i
4. (6) t4 := b[t3]
5. (7) t5 := t2*t4
6. (8) t6 := prod+t5
7. (9) prod := t6
8. (10) t7 := i+1
9. (11) i := t7
10. (12) if i<=10 goto (3)

Basic block B1 contains the statement (1) to (2)
Basic block B2 contains the statement (3) to (12)

Flow Graph

Flow graph is a directed graph. It contains the flow of control information for the set of basic block.
A control flow graph is used to depict that how the program control is being parsed among the blocks.
It is useful in the loop optimization.

Flow graph for the vector dot product is given as follows:



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

Optimization of Basic Blocks:

Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

There are two type of basic block optimization. These are as follows:

1. Structure-Preserving Transformations
2. Algebraic Transformations

1. Structure preserving transformations:

The primary Structure-Preserving Transformation on basic blocks is as follows:

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

(a) Common sub-expression elimination:

In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.

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

In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:

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

(b) Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
- Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

(c) Renaming temporary variables

A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

(d) Interchange of statement

Suppose a block has the following two adjacent statements:

1. t1 : = b + c
2. t2 : = x + y

These two statements can be interchanged without affecting the value of block when value of t1 does not affect the value of t2.

2. Algebraic transformations:

- In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expression.
- Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression 5*2.7 would be replaced by13.5.
- Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc.
- Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments

1. a:= b + c
2. e:= c +d +b

The following intermediate code may be generated:

1. a:= b + c
2. t:= c +d
3. e:= t + b

Machine-Independent Optimization

- Machine independent optimization attempts to improve the intermediate code to get a better target code. The part of the code which is transformed here does not involve any absolute memory location or any CPU registers.
- The process of intermediate code generation introduces much inefficiency like: using variable instead of constants, extra copies of variable, repeated evaluation of expression. Through the code optimization, you can remove such efficiencies and improves code.
- It can change the structure of program sometimes of beyond recognition like: unrolls loops, inline functions, eliminates some variables that are programmer defined.

Code Optimization can perform in the following different ways:

(1) Compile Time Evaluation:

(a) z = 5*(45.0/5.0)*r
Perform 5*(45.0/5.0)*r at compile time.

(b) x = 5.7
y = x/3.6
Evaluate x/3.6 as 5.7/3.6 at compile time.

(2) Variable Propagation:

Before Optimization the code is:

1. c = a * b
2. x = a
3. till
4. d = x * b + 4

After Optimization the code is:

```
1. c = a * b
2. x = a
3. till
4.     d = a * b + 4
```

Here, after variable propagation a*b and x*b identified as common sub expression.

(3) Dead code elimination:

Before elimination the code is:

```
1. c = a * b
2. x = b
3. till
4. d = a * b + 4
```

After elimination the code is:

```
1. c = a * b
2. till
3. d = a * b + 4
```

Here, x= b is a dead state because it will never subsequently used in the program. So, we can eliminate this state.

(4) Code Motion:

- It reduces the evaluation frequency of expression.
- It brings loop invariant statements out of the loop.

```
1. do
2. {
3.    item = 10;
4.    valuevalue = value + item;
5. } while(value<100);
6.
7.
8. //This code can be further optimized as
9.
10. item = 10;
11. do
12. {
13.    valuevalue = value + item;
14. } while(value<100);
```

(5) Induction Variable and Strength Reduction:

- Strength reduction is used to replace the high strength operator by the low strength.
- An induction variable is used in loop for the following kind of assignment like i = i + constant.

Before reduction the code is:

```
1. i = 1;
2. while(i<10)
3. {
4.     y = i * 4;
5. }
```

After Reduction the code is:

```
1. i = 1
2. t = 4
3. {
4.    while( t<40)
5.   y = t;
6.   t = t + 4;
7. }
```

Loop Optimization

Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.

If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.
For loop optimization the following three techniques are important:

1. Code motion
2. Induction-variable elimination
3. Strength reduction

1.Code Motion:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

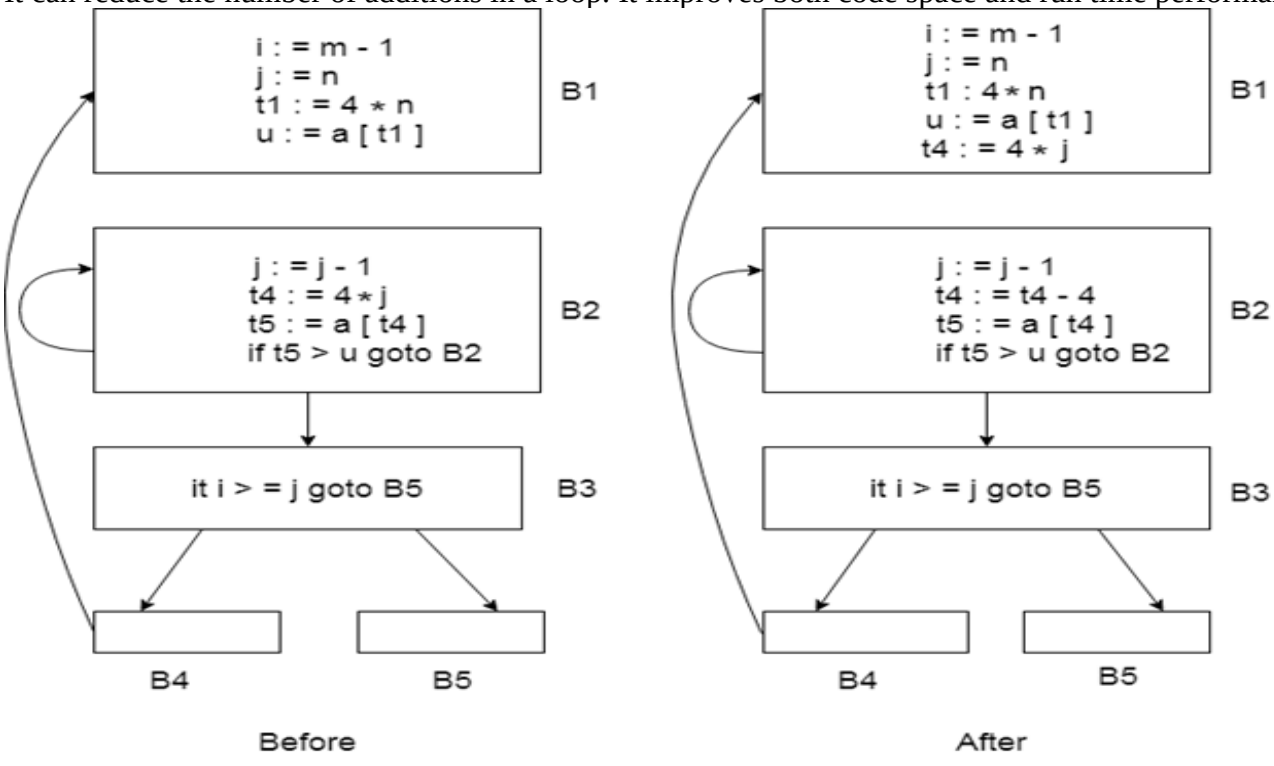
For example

In the while statement, the limit-2 equation is a loop invariant equation.

```
1. while (i<=limit-2)     /*statement does not change limit*/
2. After code motion the result is as follows:
3.          a= limit-2;
4.          while(i<=a)    /*statement does not change limit or a*/
```

2.Induction-Variable Elimination

Induction variable elimination is used to replace variable from inner loop.
It can reduce the number of additions in a loop. It improves both code space and run time performance.



In this figure, we can replace the assignment t4:=4*j by t4:=t4-4. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation t4=4*j on entry to the block B2.

3.Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

```
1. while (i<10)
2.       {
3. j= 3 * i+1;
4. a[j]=a[j]-2;
5. i=i+2;
6.       }
```

After strength reduction the code will be:

```
1. s= 3*i+1;
2.     while (i<10)
3.      {
4.         j=s;
5.         a[j]= a[j]-2;
6.         i=i+2;
7.         s=s+6;
8.      }
```

In the above code, it is cheaper to compute s=s+6 than j=3 *i

DAG representation for basic blocks

A DAG for basic block is a directed acyclic graph with the following labels on nodes:

1. The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph is labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.

- DAGs are a type of data structure. It is used to implement transformations on basic blocks.
- DAG provides a good way to determine the common sub-expression.
- It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Algorithm for construction of DAG
Input: It contains a basic block
Output: It contains the following information:

- Each node contains a label. For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed values.

1. Case (i) x:= y OP z
2. Case (ii) x:= OP y

3. Case (iii) x:= y

Method:

Step 1:

If y operand is undefined then create node(y).

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

Step 2:

For case(i), create node(OP) whose right child is node(z) and left child is node(y).

For case(ii), check whether there is node(OP) with one child node(y).

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

Output:

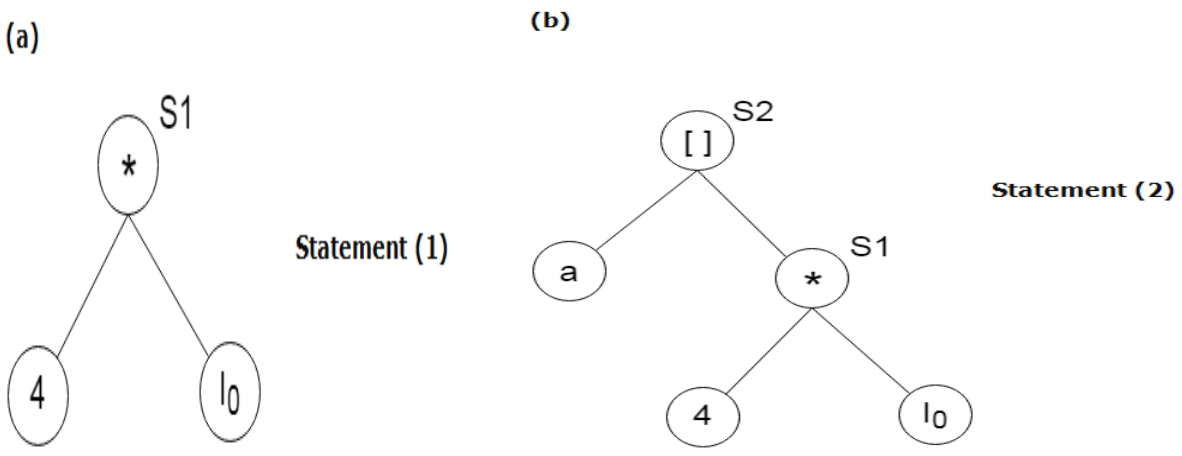
For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

Example:

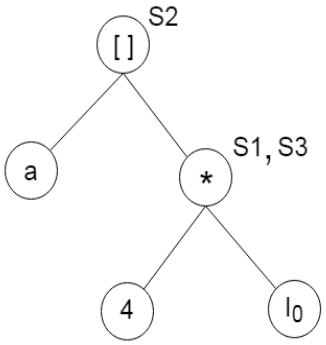
Consider the following three address statement:

1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. **if** i<= 20 **goto** (1)

Stages in DAG Construction:

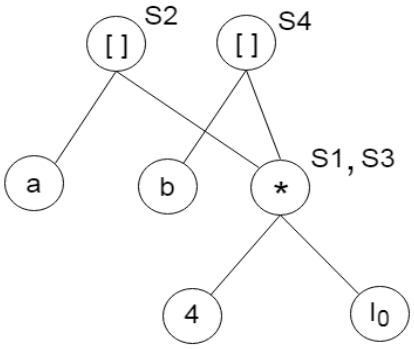


(c)



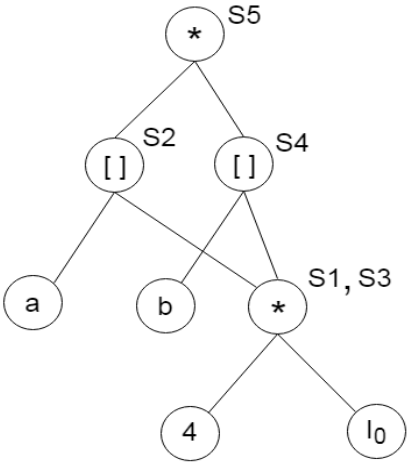
4 * I0 node exist already hence attach identifier S3 to the existing node for statement (3)

(d)



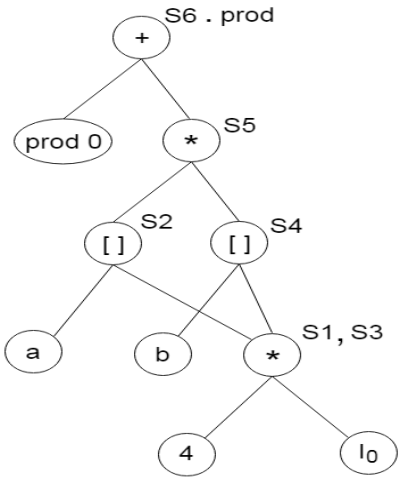
Statement (4)

(e)



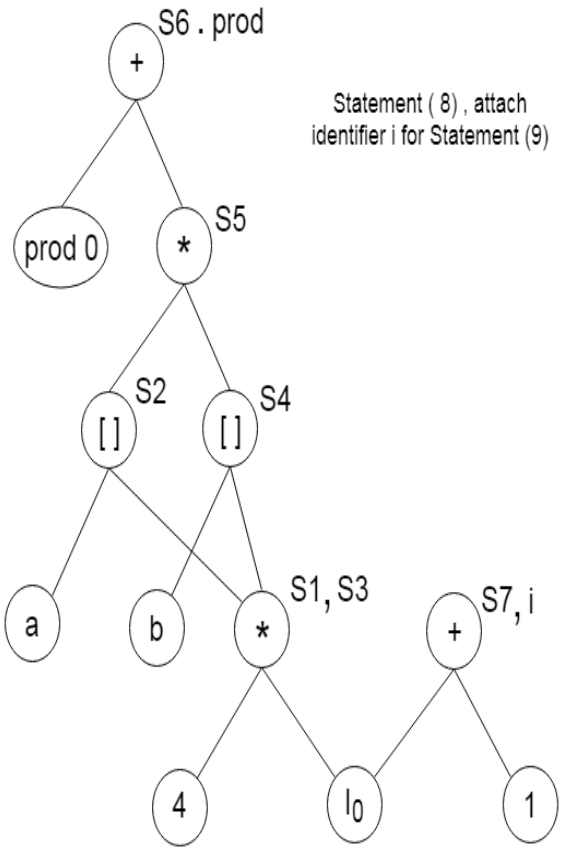
Statement (5)

(f)



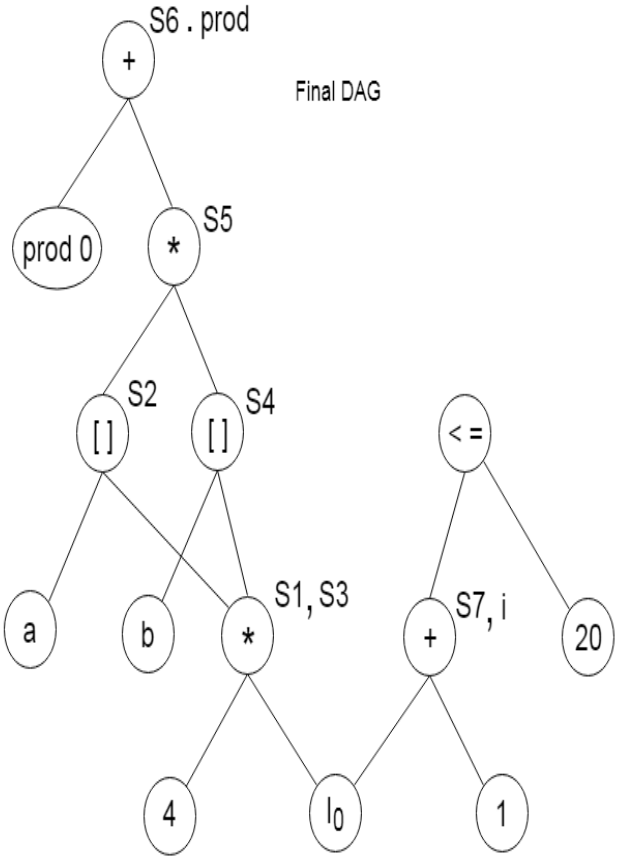
Statement (6), attach identifier prod for Statement (7)

(g)



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

(h)



Final DAG

Global data flow analysis

- To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
- For this kind of optimization user defined chaining is one particular problem.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
1. x = a + b;
2.   x = 6 * 3
```

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

```
1. a = 1;
2.   b = 2;
3.   c = 3;
4.   if (....) x = a + 5;
5.   else x = b + 4;
6.   c = x + 1;
```

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

- Which variables are guaranteed to have constant values?
- Which variables will be used before being redefined?

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.