Code Generation

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur

(www.witsolapur.org)

Code Generation

- Issues in design of a code generator and target machine,
- Run time storage management,
- Basic blocks and flow graphs,
- Next use information
- Simple code generator
- Register Allocation and Assignment
- DAG Representation of Basic Blocks
- Code generation from DAG

Introduction

- 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.

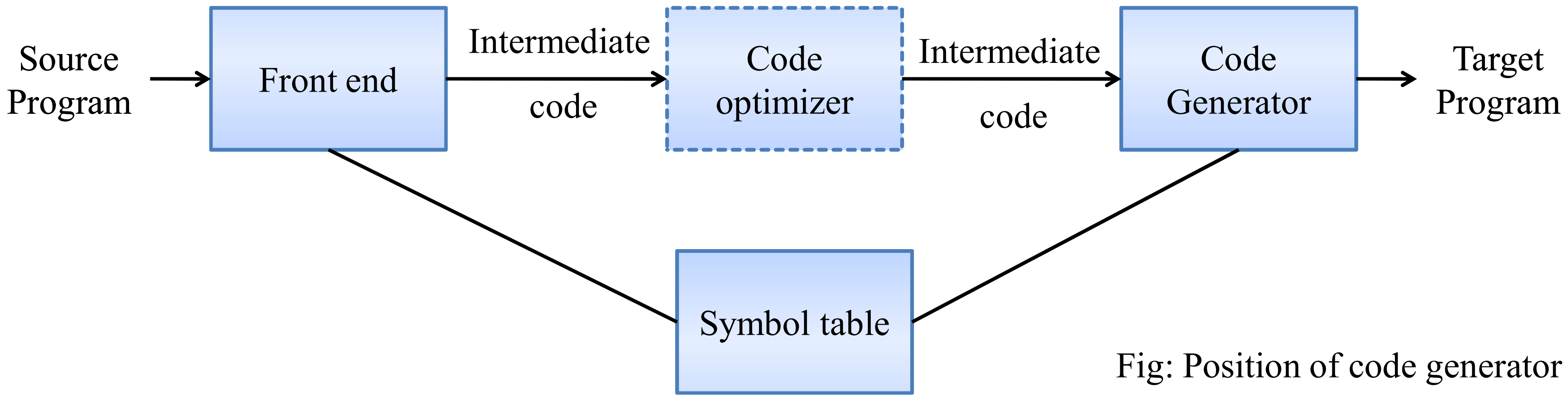


Fig: Position of code generator

Issues in 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**

Issues in design of a code generator

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 :
  - ✓ Linear representation such as postfix notation
  - ✓ Three address representation such as quadruples
  - ✓ Virtual machine representation such as stack machine code
  - ✓ Graphical representations such as syntax trees and DAGs.

Introduction

1. **Input to code generator**
    - 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.

Introduction

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.

Introduction

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.

Introduction

3. Memory management

- Labels in three-address statements have to be converted to addresses of instructions.

  For example, labels refers to the quadruple number in quadruple array

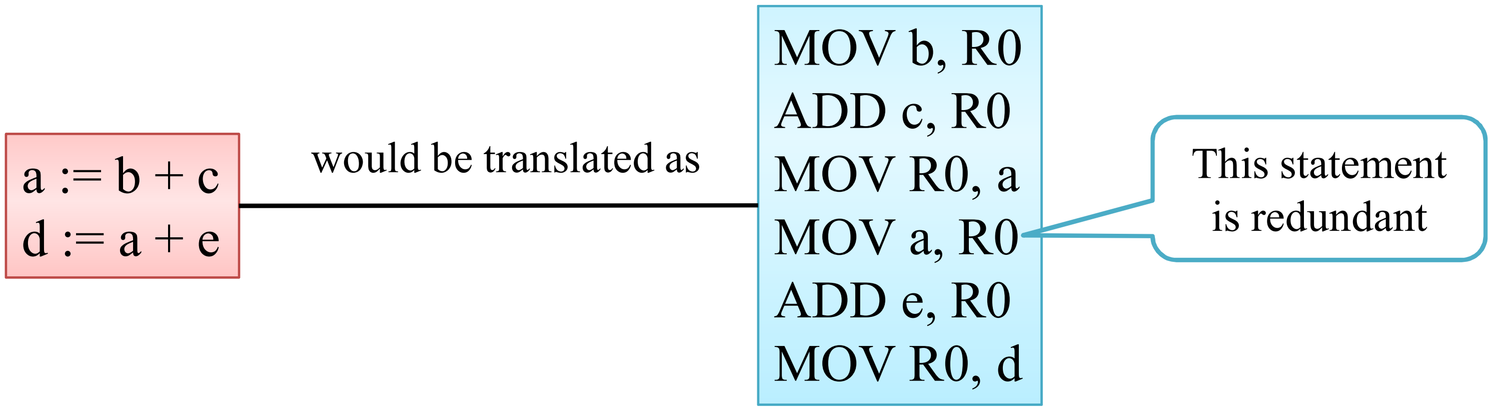
  j : goto i generates jump instruction as follows :

  - ✓ 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 isprocessed, the machine locations for all instructions that forward jumps to i are filled.

Introduction

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.



```
a := b + c
d := a + e
```

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

Introduction

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.
  - ✓ Register assignment – the specific register that a variable will reside in is picked.

Introduction

5. Register allocation

- Certain machine requires even-odd register pairs for some operands and results.

  For example , consider the multiplication instruction of the form :

  M x, y

  where, x – the multiplicand in even register of even/odd register pair

  y – multiplier

  the multiplicand is taken from the odd register of the pair

  product occupies the entire even/odd register pair

Introduction

5. Register allocation

- 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 in even register of even/odd register pair

  y – divisor

  even register holds the remainder

  odd register holds the quotient

Introduction

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.

```
a+b-(c+d)*e
```

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

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

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

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```

Cost I > Cost II

The 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, R0, R1, . . . , Rn-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 Target Machine

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

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 R0, M stores contents of Register R0 into memory location M;
  MOV 4(R0), M stores the value contents(4+contents(R0)) into M.

The Target Machine

- **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 Target Machine

- **Instruction costs :**
  - The three-address statement a : = b + c can be implemented by many different instruction sequences :

```
i) MOV b, R0
   ADD c, R0     cost = 6
   MOV R0, a
```

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

```
iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :
    MOV *R1, *R0
    ADD *R2, *R0         cost = 2
```

  - In order to generate good code for target machine, we must utilize its ddressing 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:
  - **Static allocation**
  - **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.

Run-Time Storage Management

- 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.
- We assume that the run-time memory is divided into areas for:
  1. Code
  2. Static data
  3. Stack

Run-Time Storage Management

- **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.

Run-Time Storage Management

- **Static allocation**
  - **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.

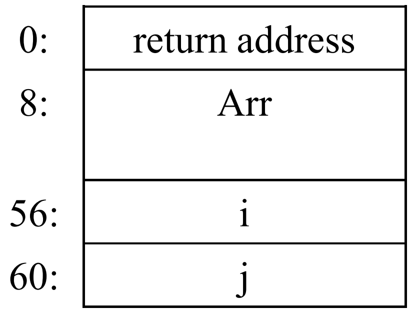
Run-Time Storage Management

- **Static allocation**

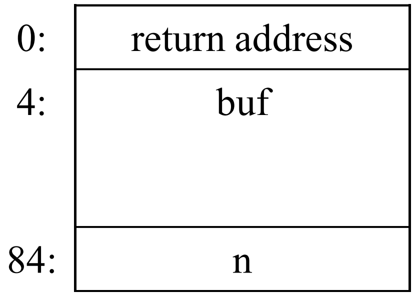
Three address code

```
/*code for c*/
action1
call p
action2
halt
-----------
/*code for p*/
action1
return
```

Activation record for c
(64 bytes)



Activation record for p
(88bytes)



```
                         /*code for c */
100: ACTION1
120: MOV #140, 364       /*save return address 140*/
132: GOTO 200            /*call p*/
140: ACTION2
160: HALT
      …

                    /*code for p*/
200: ACTION3
220: GOTO 364       /*return to address saved in location 364*/

                    /*300-363 hold activation record for c*/
300:                /*return address*/
304:                /* local data c*/

                    /*364-451 hold activation record for c*/
364:                /*return address*/
368:                /* local data c*/
```

Run-Time Storage Management

- **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.

Run-Time Storage Management

- **Stack allocation**

  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 */
```

Run-Time Storage Management

- **Stack allocation**

The codes needed to implement stack allocation are as follows:

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

Run-Time Storage Management

- **Stack allocation**

  The codes needed to implement stack allocation are as follows:

  Implementation of Return statement:

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

Run-Time Storage Management

- **Stack allocation**

  The codes needed to implement stack allocation are as follows:

  Implementation of Return statement:

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

Run-Time Storage Management

- **Stack allocation**

Three address code

```
/*code for s*/
action1
call q
action2
halt
```

```
/*code for p*/
action3
return
```

```
/*code for q*/
action4
call p
action5
call q
action6
call q
return
```

```
                        /*code for s */
100: MOV #600, SP   /*initialize the stack*/
108: ACTION1
128: ADD #ssize, SP  /*call sequence begin*/
136: MOV #150, *SP  /*push return address */
144: GOTO 300        /*call q*/
152: SUB #ssize, SP  /*restore SP*/
160: ACTION2
180:HALT
     …

                        /*code for p*/
200: ACTION3
220: GOTO *0(SP)     /*return */
```

```
                        /*code for q*/
300:ACTION4            /*conditional jump to 456*/
320: ADD #qsize, SP
328: MOV #344, *SP  /*push return address*/

336: GOTO 200          /*call p*/
344: SUB #qsize, SP
352: ACTION5
372: ADD #qsize, SP
380: MOV #396, *SP  /*push return address*/
388: GOTO 300          /*call q*/
396: SUB #qsize, SP
404: ACTION6
424: ADD #qsize, SP
432: MOV #448, *SP  /*push return address*/
440: GOTO 300          /*call q*/
448: SUB #qsize, SP
456: GOTO *0(SP)     /*return*/


600:                    /*stack starts here*/
```

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:

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

Basic Blocks and Flow Graphs

- **Basic Blocks**
  - **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.

Basic Blocks and Flow Graphs

- **Basic Blocks**
  - Example 1- 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
```

Three address code

```
(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)
```

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

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

```
(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 2: Statement (3) to (12)

Basic Blocks and Flow Graphs

- **Basic Blocks**
  - Example 2

```
        MOV 1,R0
        MOV n,R1
        JMP L2
L1:     MUL 2,R0 SUB 1,R1
L2:     JMPNZ R1,L1
```

```
        MOV 1,R0
        MOV n,R1
        JMP L2
```

```
L1:     MUL 2,R0 SUB 1,R1
```

```
L2:     JMPNZ R1,L1
```

Basic Blocks and Flow Graphs

- **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**
      - Common subexpression elimination
      - Dead-code elimination
      - Renaming temporary variables
      - Interchange of statements
    - ✓ **Algebraic transformations**

Basic Blocks and Flow Graphs

- **Transformations on Basic Blocks:**
  - **Structure-preserving transformations**

    1. Common subexpression elimination

$$\begin{array}{lcl} a := b + c & & a := b + c \\ b := a - d & & b := a - d \\ c := b + c & \Longrightarrow & c := b + c \\ d := a - d & & d := b \end{array}$$

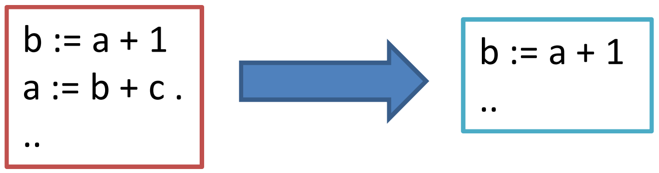
Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

Basic Blocks and Flow Graphs

- **Transformations on Basic Blocks:**
  - **Structure-preserving transformations**

    2. 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.



Assuming a is dead (not used)

Basic Blocks and Flow Graphs

- **Transformations on Basic Blocks:**
  - **Structure-preserving transformations**

    3. 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.

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

$\Rightarrow$

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

Basic Blocks and Flow Graphs

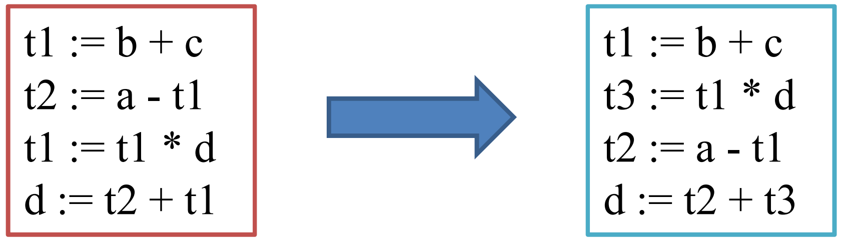
- **Transformations on Basic Blocks:**
  - **Structure-preserving transformations**

    4. 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.



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

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

Basic Blocks and Flow Graphs

- **Transformations on Basic Blocks:**
  - **Algebraic transformations**

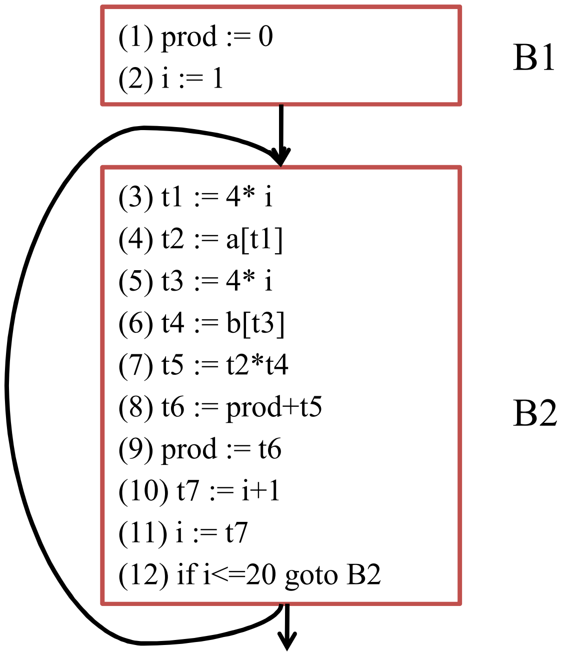
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

$$\begin{tikzcd}
\boxed{\begin{array}{l}\text{t1 := a - a}\\ \text{t2 := b + t1}\end{array}} \arrow[r] & \boxed{\begin{array}{l}\text{t1 := 0}\\ \text{t2 := b}\end{array}} \\
\boxed{\text{t1 := 0 t2 := b}} \arrow[r] & \boxed{\text{t3 := t2 << 1}} \\
\boxed{\text{t3 := t2 ** 2}} \arrow[r] & \boxed{\text{t3 := t2 * t2}} \\
\boxed{\begin{array}{l}\text{t1 := t1 + 0}\\ \text{t2 := t2 * 1}\end{array}} \arrow[r] & \text{Do not remove the statements}
\end{tikzcd}$$

Basic Blocks and Flow Graphs

- **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.

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



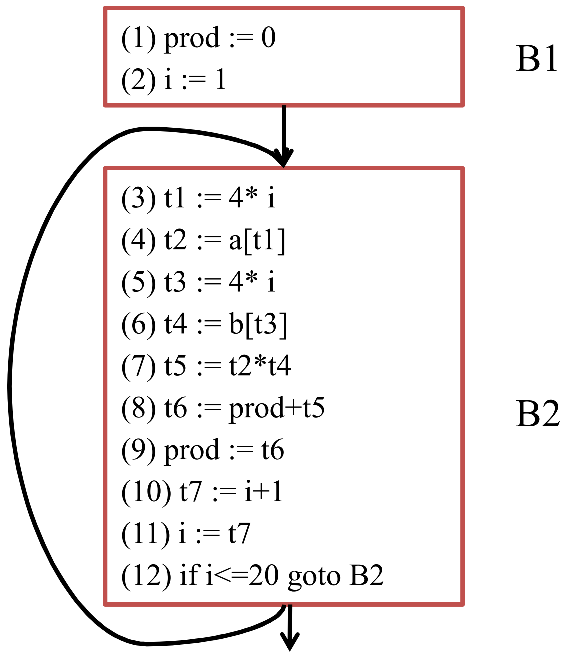
```
(1) prod := 0
(2) i := 1
```

```
(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 B2
```

Basic Blocks and Flow Graphs

- **Flow Graphs**

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

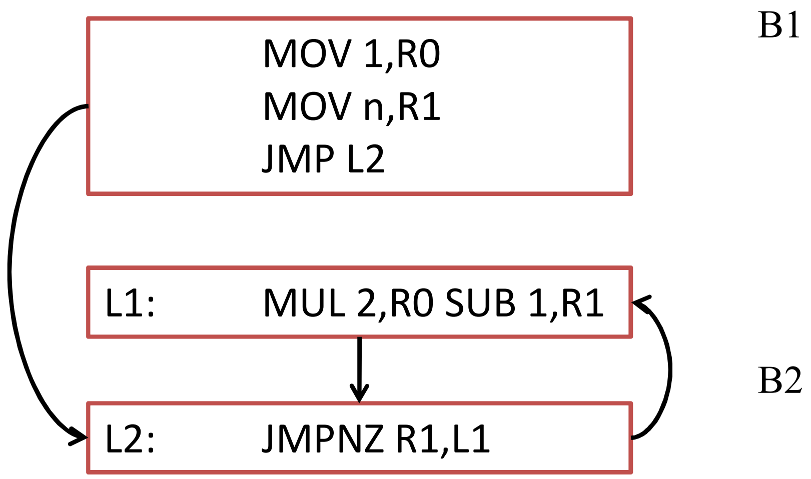


- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.

Basic Blocks and Flow Graphs

- **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.

```
        MOV 1,R0
        MOV n,R1
        JMP L2
L1:     MUL 2,R0 SUB 1,R1
L2:     JMPNZ R1,L1
```



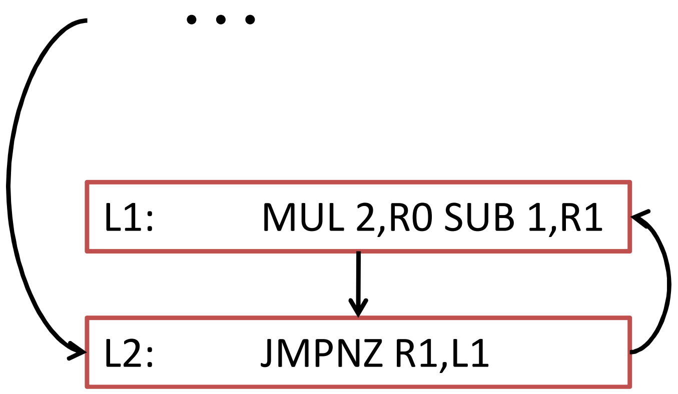
```
        MOV 1,R0
        MOV n,R1
        JMP L2
```

```
L1:     MUL 2,R0 SUB 1,R1
```

```
L2:     JMPNZ R1,L1
```

Basic Blocks and Flow Graphs

- **Flow Graphs**
  - **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

- for register and temporary allocation
- remove variables from registers if not used
- statement X = Y op Z defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

Next-use information

Suppose we are scanning i : X := Y op Z in backward scan

1. attach to statement i, information in symbol table about X, Y, Z
2. set X to “not live” and “no next use” in symbol table
3. set Y and Z to be “live” and next use as i in symbol table

Next-use information

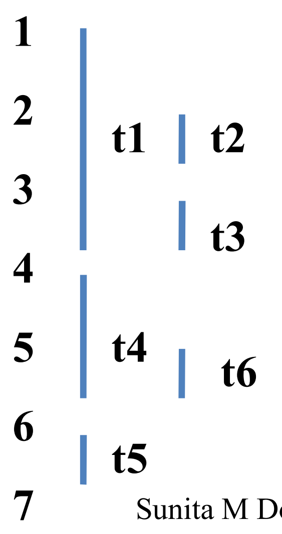
Example

```
1: t1 = a * a
2: t2 = a * b
3: t3 = 2 * t2
4: t4 = t1 + t3
5: t5 = b * b
6: t6 = t4 + t5
7: X = t6
```

```
7: no temporary is live
6: t6:use(7), t4, t5 not live
5: t5:use(6)
4: t4:use(6), t1, t3 not live
3: t3:use(4), t2 not live
2: t2:use(3)
1: t1:use(4)
```

Symbol Table

Names	Liveliness	Next-use
t1	dead	Use in 3
t2	dead	Use in 4
t3	dead	Use in 4
t4	dead	Use in 6
t5	dead	Use in 6
t6	dead	Use in 7



```
1: t1 = a * a
2: t2 = a * b
3: t2 = 2 * t2
4: t1 = t1 + t2
5: t2 = b * b
6: t1 = t1 + t2
7: X = t1
```

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

A Simple Code Generator

- **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 Simple Code Generator

- **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:

A Simple Code Generator

- **A code-generation algorithm:**

```
For each statement x:= y op z
1.  Set location L = getreg( y, z )
2.  If y is not in L, then generate MOV y’,L
            y’ denotes one of the locations where the value of y is available
                    (choose register if possible)
3.  Generate OP z’,L
            z’ is one of the locations of z;
            Update register/address descriptor of x to include L
4.  If y and/or z have no next use and are stored in register, update register descriptors to
    remove y and/or z

Finally, we store all names that are live on exit and not in their memory locations
```

A Simple Code Generator

- **A code-generation algorithm:**

To compute L=getreg(y, z)

1. If y is stored in a register R, R only holds the value y, and y has no next use, then return R;
   - ✓ Update address descriptor: value y no longer in R
2. Else, return a new empty register if available
3. Else, find an occupied register R;
   - ✓ Store contents in memory (if not there) by generating MOV R,M
   - ✓ Return register R
4. Return the memory location of x if no proper register is found or x is not used in the block

A Simple Code Generator

- **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.

A Simple Code Generator

- **Generating Code for Assignment Statements:**
  - Code sequence for d : = (a-b) + (a-c) + (a-c) is

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

A Simple Code Generator

- **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(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

A Simple Code Generator

- **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 *Rp, a	2
*p : = a	MOV a, *Rp	2

A Simple Code Generator

- **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, R0
	ADD z, R0
	MOV R0,x
	CJ< z

A Simple Code Generator

- **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 *Rp, a	2
*p : = a	MOV a, *Rp	2

Register Allocation and Assignment

- The discussed code-generation algorithm is sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- Global register allocation assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings
- Usage counts: ∑(use(x,B) + 2*(live(x,B)), where
  - use(x,B) is the number of times x is used in block B
  - live(x,B)=1 if x is live on exit from B

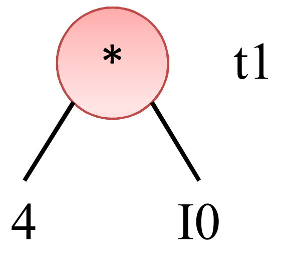
The DAG Representation of Basic Blocks

- 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.

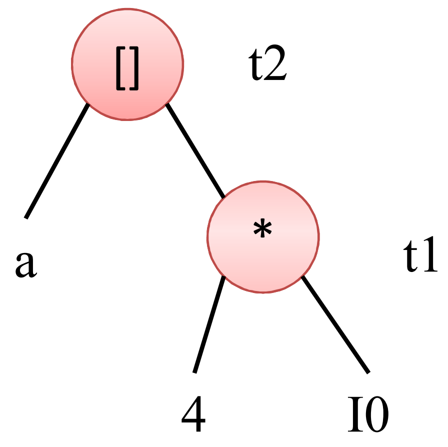
The DAG Representation of Basic Blocks

- Example

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



Statement 1

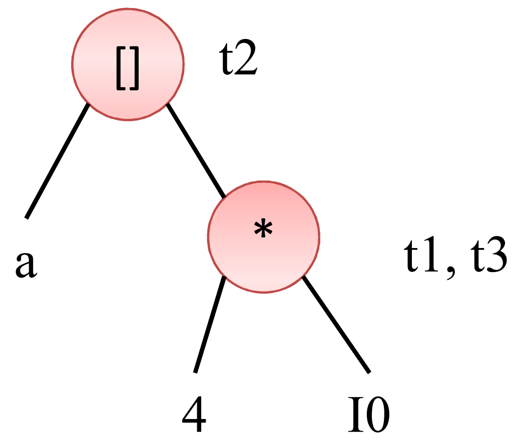


Statement 2

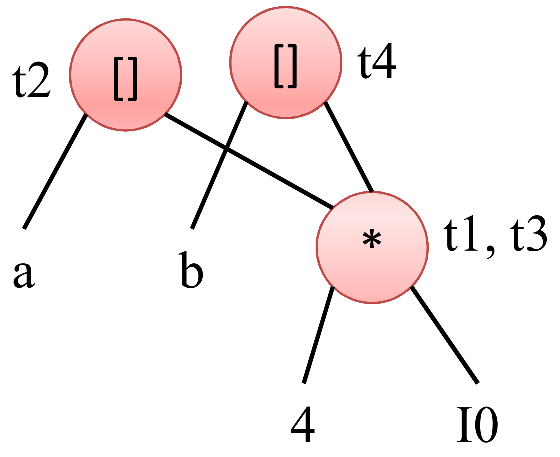
The DAG Representation of Basic Blocks

- Example

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



Statement 3

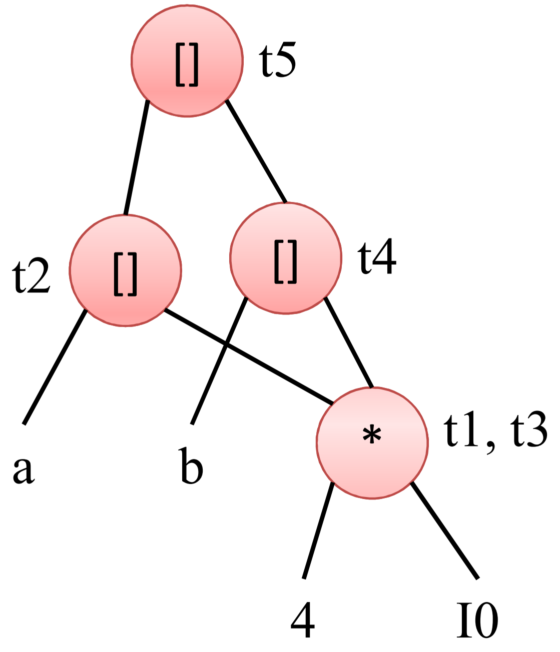


Statement 4

The DAG Representation of Basic Blocks

- Example

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

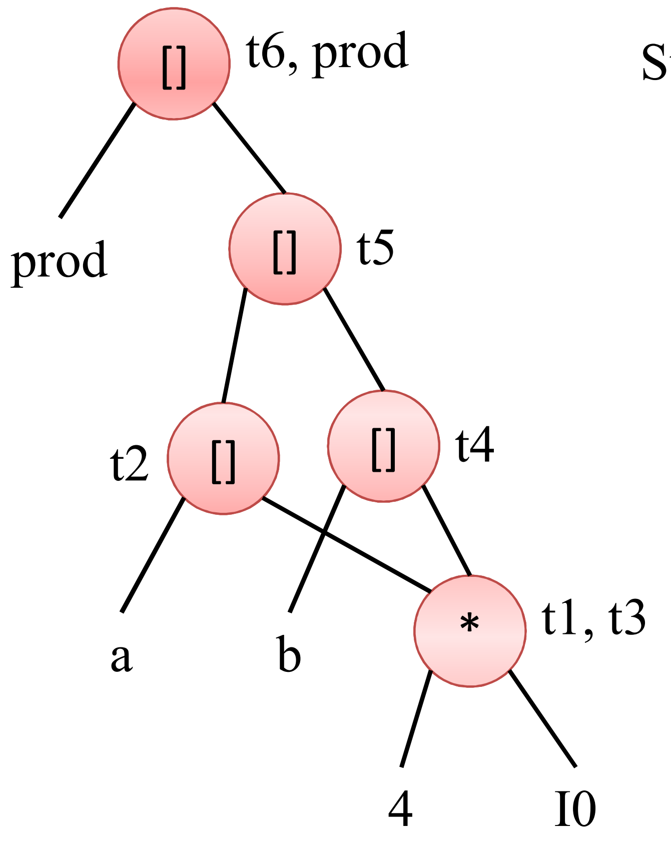


Statement 5

The DAG Representation of Basic Blocks

- Example

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

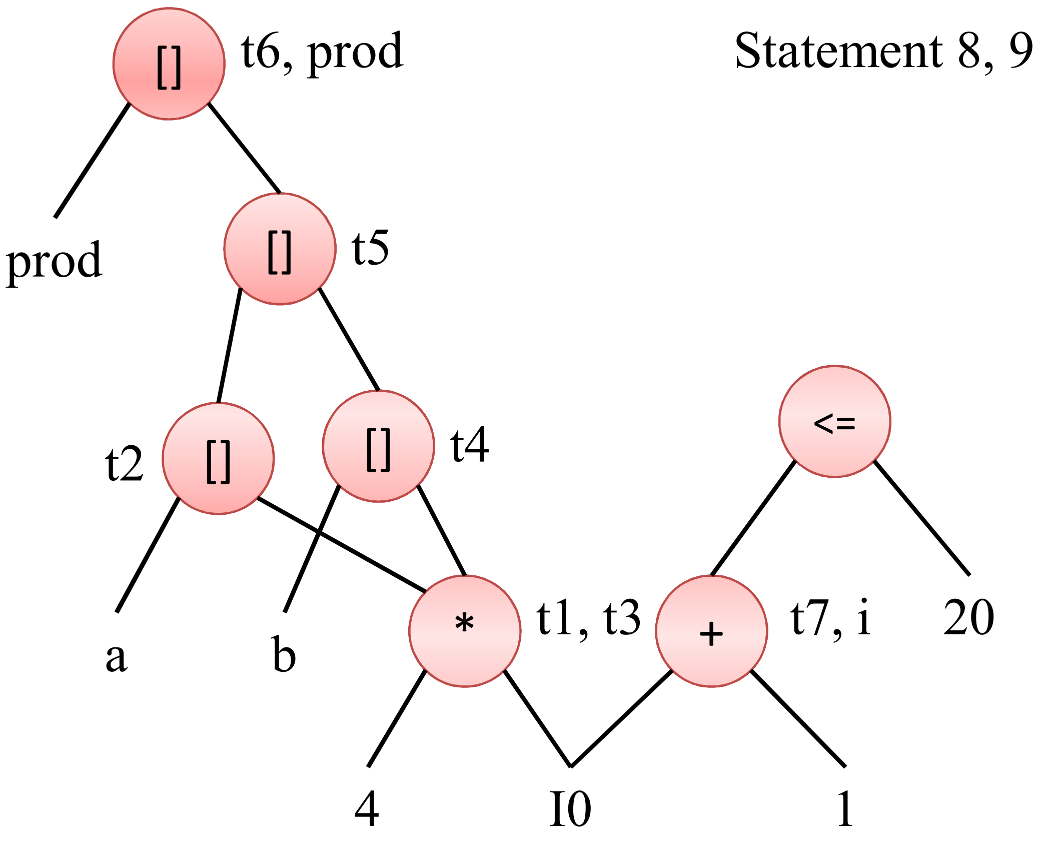


Statement 6, 7

The DAG Representation of Basic Blocks

- Example

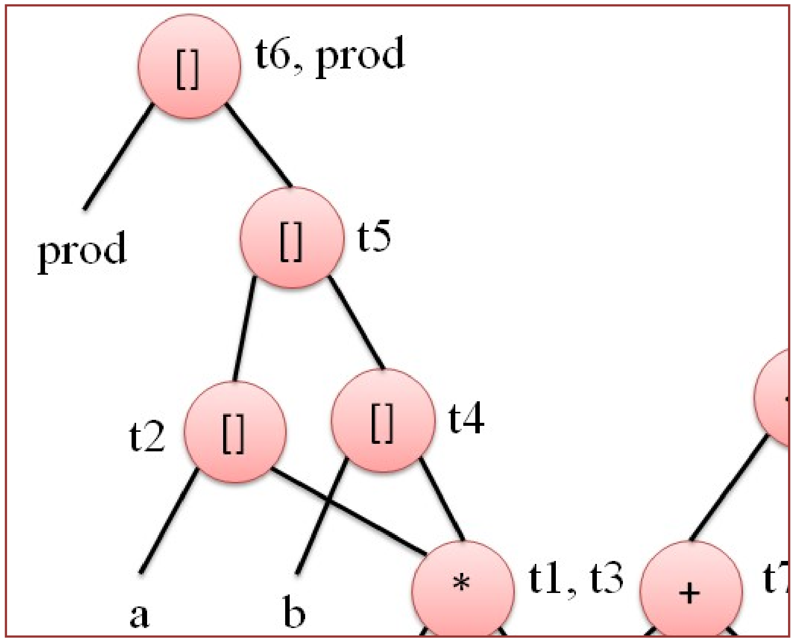
```
1. t1 := 4 * i
2. t2 := a[t1]
3. t3 := 4 * i
4. t4 := b[t3]
5. t5 := t2 * t4
6. t6 := prod + t5
7. prod := t6
8. t7 := i + 1
9. i := t7
10. if i <= 20 goto (1)
```



The DAG Representation of Basic Blocks

- Example

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



```
S1 = 4 * i
S2 = addr(A)-4
S3 = S2[S1]
S4 = 4 * i
S5 = addr(B)-4
S6 = S5[S4]
S7 = S3 * S6
S8 = prod+S7
prod = S8
S9 = I+1
I = S9
If I <= 20 goto (1)
```

```
S1 = 4 * i
S2 = addr(A)-4
S3 = S2[S1]

S5 = addr(B)-4
S6 = S5[S4]
S7 = S3 * S6

prod = prod + S7

I = I + 1
If I <= 20 goto (1)
```

The DAG Representation of Basic Blocks

- **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.

The DAG Representation of Basic Blocks

- **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

The DAG Representation of Basic Blocks

- **Algorithm for construction of DAG**

  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.

Code Generation from DAG

- 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.

Code Generation from DAG

- **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:

```
t1 : = a + b
t2 : = c + d
t3 : = e – t2
t4 : = t1 – t3
```

Generated code sequence for basic block:

```
MOV a , R0
ADD b , R0
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

```
t2 : = c + d
t3 : = e – t2
t1 : = a + b
t4 : = t1 – t3
```

Revised code sequence:

```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

In this order, two instructions MOV R0 , t1 and MOV t1 , R1 have been saved

Code Generation from DAG

- **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.

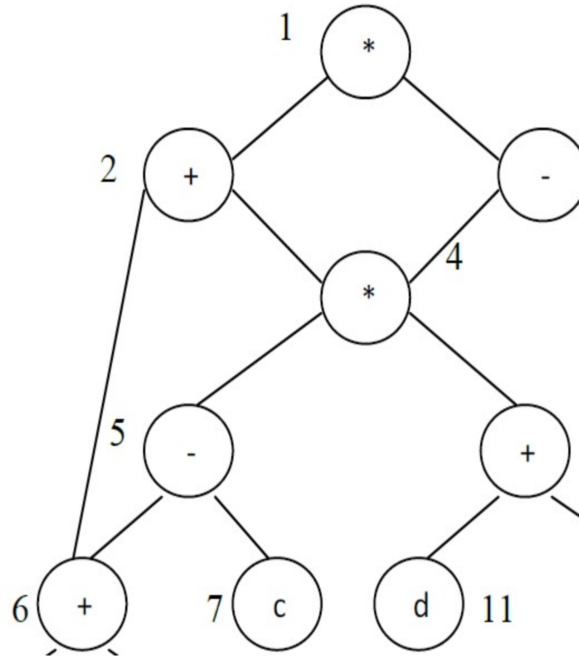
  The algorithm shown below produces the ordering in reverse.

  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
```

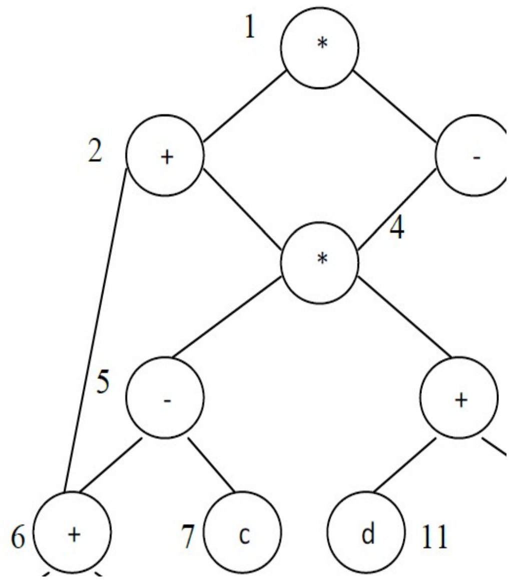
Code Generation from DAG

- **A Heuristic ordering for Dags**
  - 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 Generation from DAG

- **A Heuristic ordering for Dags**
  - Example: Consider the DAG shown below:



Code sequence:
t8 : = d + e
t6 : = a + b
t5 : = t6 – c
t4 : = t5 * t8
t3 : = t4 – e
t2 : = t6 + t4
t1 : = t2 * t3

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

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!