

# UNIT-III

## INTERMEDIATE CODE GENERATION



### Need for Intermediate code

- Eliminates the need of new compiler for new machine.
- Easy to apply the source code modifications on the intermediate code.

# REPRESENTATION OF INTERMEDIATE CODE

- **Three address codes**

- implemented as quadruples, triples, indirect triples, tree or DAG

## **Rules**

- LHS is the target and the RHS has at most two sources and one operator
- RHS sources can be either variables or constants

# Example for Three address code

**Expression:**  $a = b * c - d$

Three Address code:

- $t1 = b * c$
- $t2 = t1 - d$
- $a = t2$

# Example for Three address code

**Expression:  $a = b * -c + b * -c$**

Three Address code:

- $t1 = \text{uminus } c$
- $t2 = b * t1$
- $t3 = \text{uminus } c$
- $t4 = b * t3$
- $t5 = t2 + t4$
- $a = t5$

# Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.

Eg:  $a := b * -c + b * -c$

	<i>op</i>	<i>arg1</i>	<i>arg2</i>	<i>result</i>
(0)	<u>uminus</u>	c		t <sub>1</sub>
(1)	*	b	t <sub>1</sub>	t <sub>2</sub>
(2)	<u>uminus</u>	c		t <sub>3</sub>
(3)	*	b	t <sub>3</sub>	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	:=	t <sub>5</sub>		a

# Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.

The results of respective sub-expressions are denoted by the position of expression.

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	<u>uminus</u>	c	
(1)	*	b	(0)
(2)	<u>uminus</u>	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

# INDIRECT TRIPLES

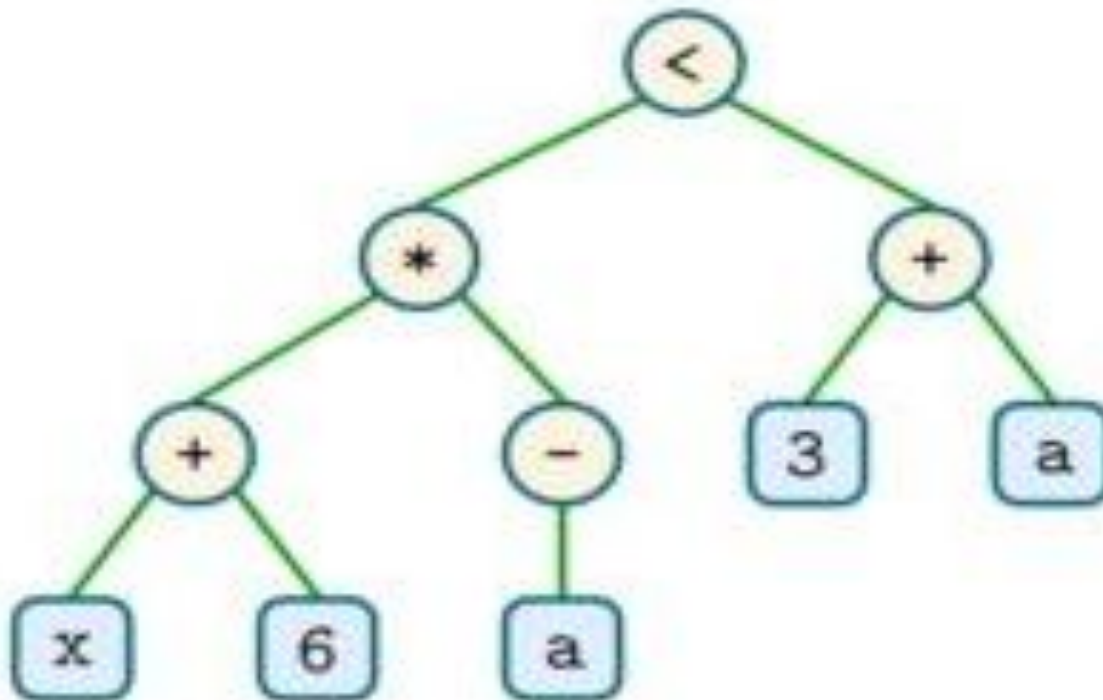
- It uses pointers instead of position to store results.
- Eg:  $a := b * -c + b * -c$

	<u>statement</u>
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(14)	<u>uminus</u>	c	
(15)	*	b	(14)
(16)	<u>uminus</u>	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

# SYNTAX TREES

- Operands as leaves and operators as roots.
- Eg:  $((x+6)*-a)<(3+a)$





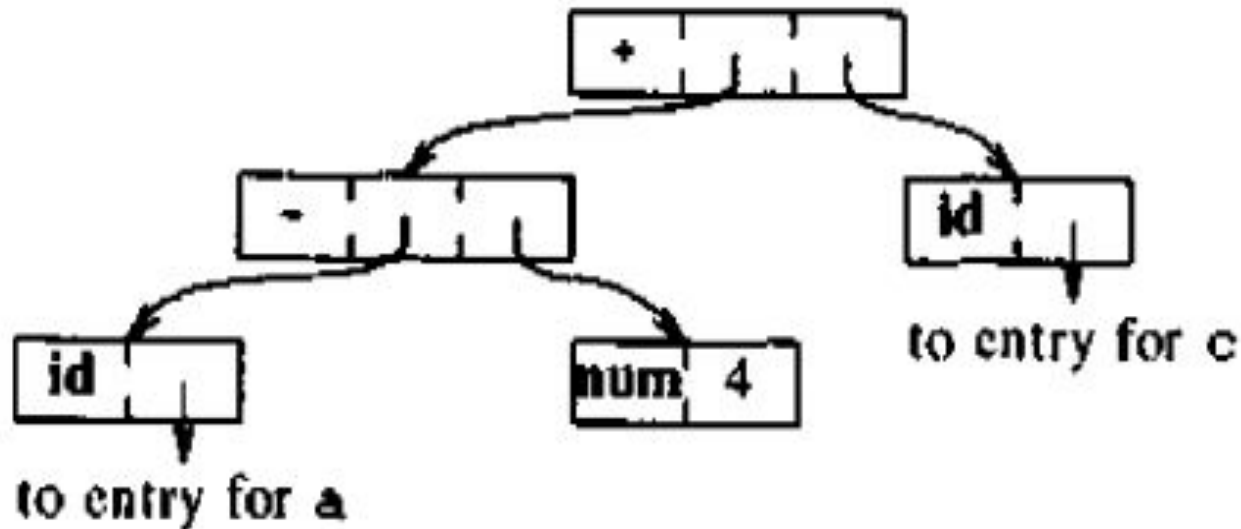
# Construction of Syntax trees

## Functions:

1. `mknode(op, left, right)`
2. `mkleaf(id, entry)`
3. `mkleaf(num, val)`

# Example: a-4+c

1. P1= mkleaf(id,entry a)
2. p2=mkleaf(num,4)
3. p3=mknnode('-',p1,p2)
4. p4=mkleaf(id,entry c)
5. p5=mknnode('+',p3,p4)



# SYNTAX DIRECTED DEFINITION (SDD)

- Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- The process of syntax directed translation is two-fold:
  1. Construction of syntax tree
  2. Computing values of attributes at each node by visiting the nodes of syntax tree.

# SYNTAX DIRECTED TRANSLATIONS (SDT)

- Uses semantic actions specified by the productions.
- Actions are computations done from left to right.
- **Semantic action:** code fragments embedded within production bodies.

**Eg:**  $E \rightarrow E_1 + T \{print\ '+'\}$

## SDD

- Declarative statements
- Specifies the computations to be done
- More readable

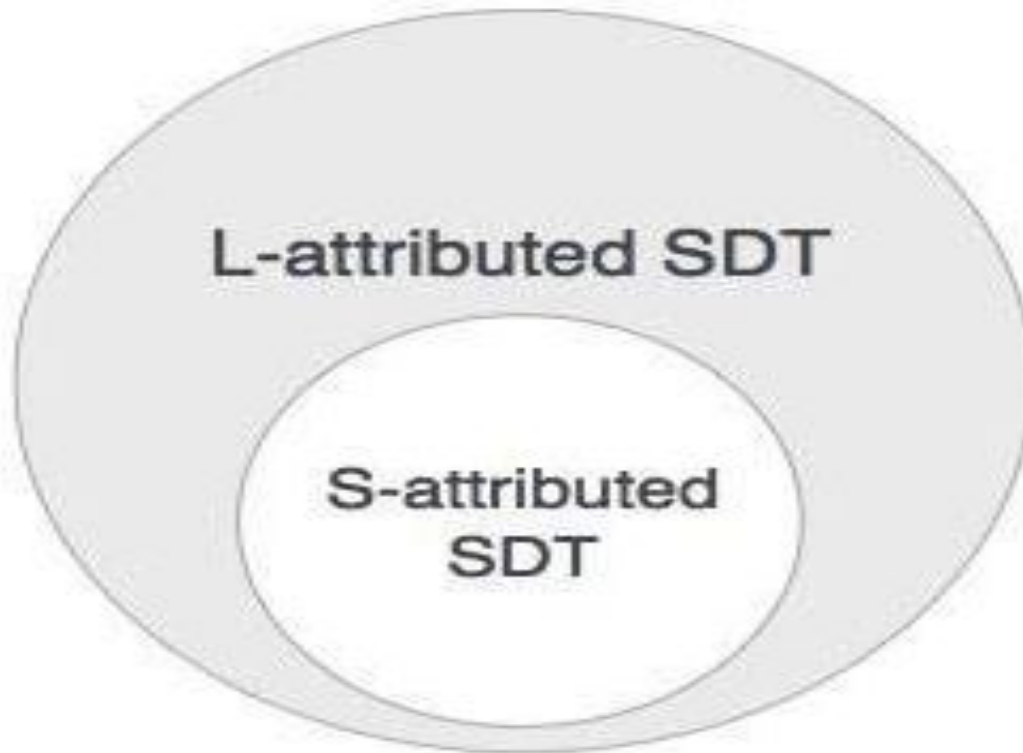
## SDT

- Procedural statements
- Specifies computations and the time at which computations must be done.
- Efficiently parsable

# Types of Translation

- **L-attributed (Left  $\square$  Right)**
  1. Performed during parsing
  2. No explicit tree construction
- **S-attributed (synthesized attributed)**
  1. bottom up parsing.
- An **attribute grammar** is a formal way to define attributes for the productions of a grammar.
- It associates **attributes** with values.
- The **evaluation** occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or **compiler**.

# Relation between S and L Attribute



# SYNTHESIED AND INHERITED ATTRIBUTES

- **Synthesized attributes:** computed from children

**EG:**  $S \square XYZ$  . S gets its value only from its children.

- **Inherited attributes:** computed from parents and siblings.

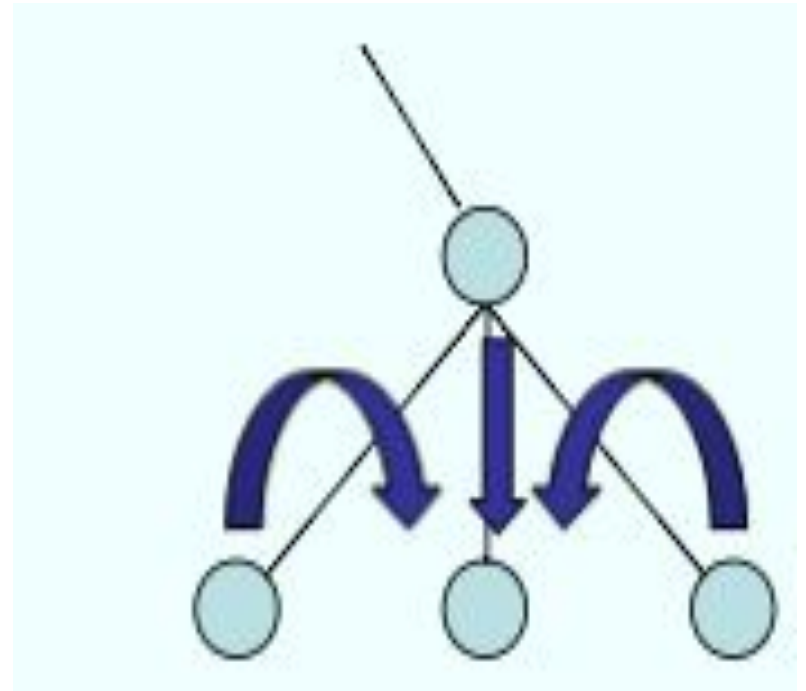
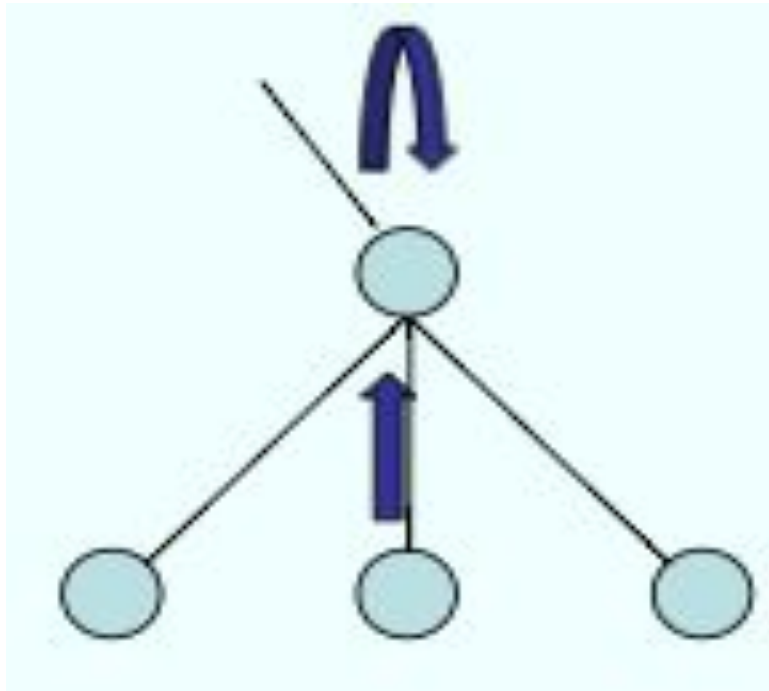
**EG:**  $S \square ABC$  . A can get its value from S, A or C.



# SYNTHESIZED VS INHERITED

SYNTHESIZED

INHERITED



# Evaluation of Attribute grammar

1. A grammar is S-attributed if every attribute is synthesized.
2. A grammar is L-attributes if every attribute is
  - 2.1 Synthesized or
  - 2.2 inherited, with restrictions that the evaluation rule uses only
    - a) attributes synthesized from parent
    - b) attributes from left siblings .

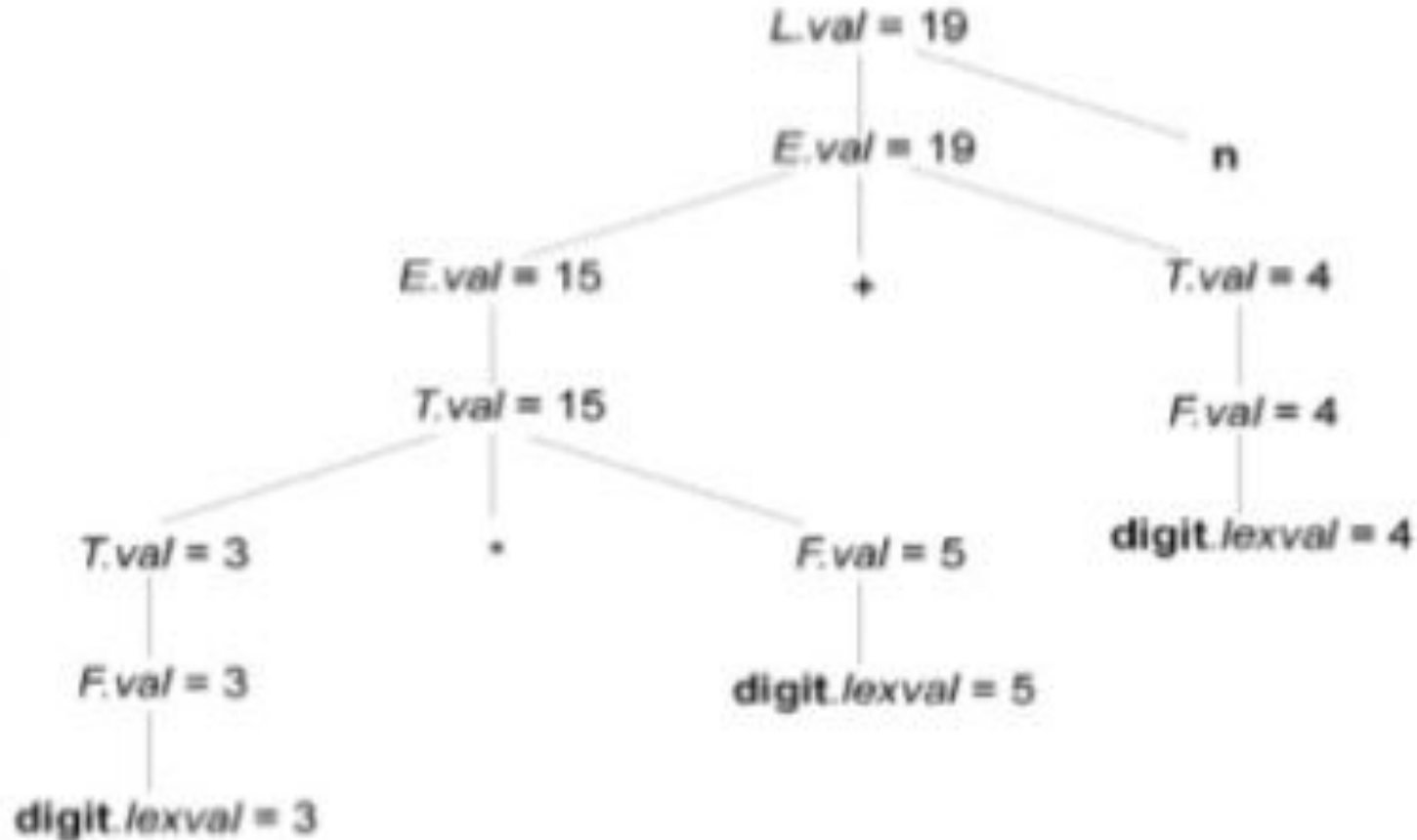
# S-ATTRIBUTED DEFINITIONS

- Contains only synthesized attributes
- Attribute values for the non-terminal at head is computed from the attribute values of the symbols at the body of the production.
- evaluated in bottom up order of nodes of the parse tree.
- post order traversal of the parse tree .

# Example

Production	Semantic rules
$L \rightarrow E_n$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

# Eg: Parse tree for $3*5+4n$



# L-ATTRIBUTED DEFINITIONS

- Attribute evaluation go from left to right and vice versa.
- Attributes may either be synthesized or inherited.
- If the attributes are **inherited**, it must be computed from:
  1. Inherited attribute associated with the production head.
  2. If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
  3. Either by inherited or synthesized attribute associated with the attribute under consideration with no cycles.

# EXAMPLE

P1:  $S \rightarrow MN \{S.val = M.val + N.val\}$

P2:  $M \rightarrow PQ \{M.val = P.val * Q.val \text{ and } P.val = Q.val\}$

Which of the following is true?

- A. Both P1 and P2 are S attributed.
- B. P1 is S attributed and P2 is L-attributed.
- C. P1 is L attributed but P2 is not L-attributed.

Ans: C

## **L attributed definitions**

- **Order of evaluation is important**
- **Construct a dependency graph using topological sorting without cycles.**



# Example

Production	Semantic Rule
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \mathbf{int}$	$T.type := \text{'integer'}$
$T \rightarrow \mathbf{real}$	$T.type := \text{'real'}$
$L \rightarrow L_1, \mathbf{id}$	$L_1.in := L.in; \text{addtype}(\mathbf{id}.entry, L.in)$
$L \rightarrow \mathbf{id}$	$\text{addtype}(\mathbf{id}.entry, L.in)$

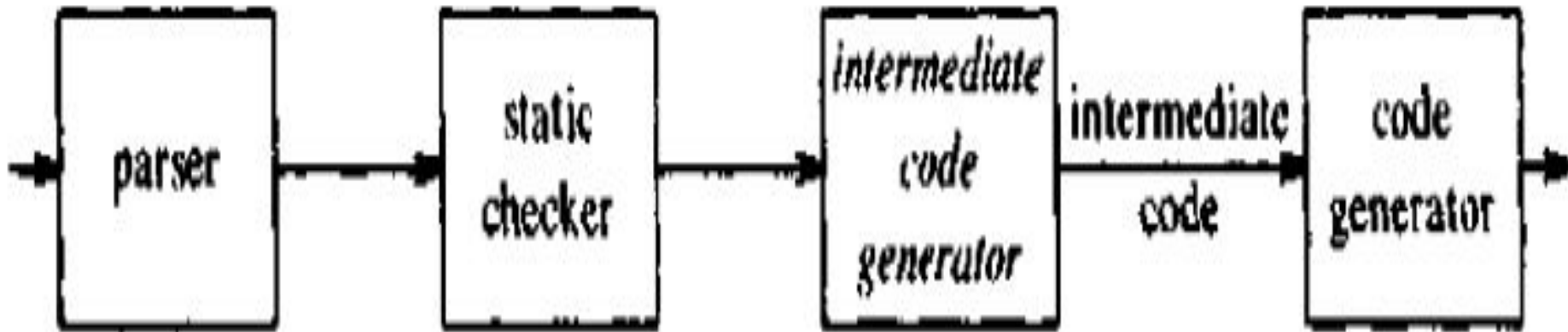
## Translation Scheme

$D \rightarrow T \{ L.in := T.type \} L$   
 $T \rightarrow \mathbf{int} \{ T.type := \text{'integer'} \}$   
 $T \rightarrow \mathbf{real} \{ T.type := \text{'real'} \}$   
 $L \rightarrow \{ L_1.in := L.in \} L_1, \mathbf{id} \{ \text{addtype}(\mathbf{id}.entry, L.in) \}$   
 $L \rightarrow \mathbf{id} \{ \text{addtype}(\mathbf{id}.entry, L.in) \}$

# INTERMEDIATE LANGUAGES

## Advantages:

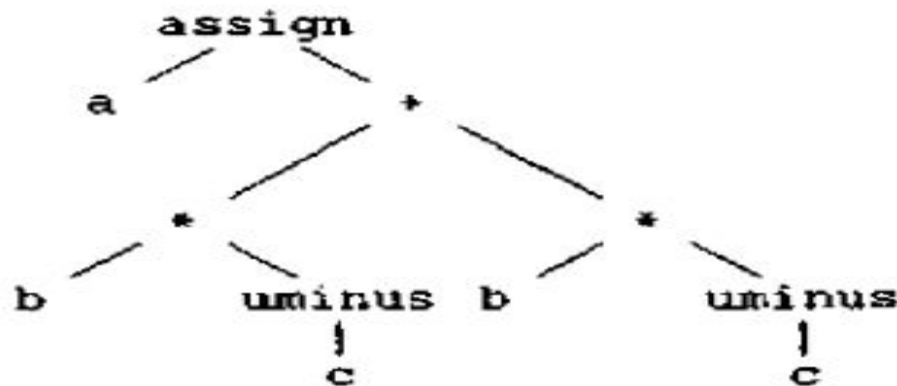
1. Portable front end and back end
2. Facilitates the use of machine independent code optimizer



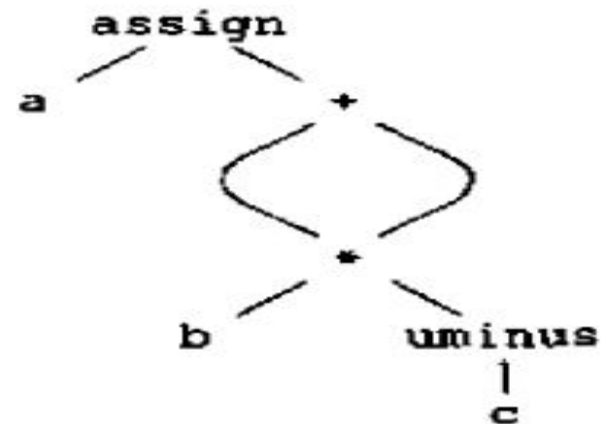
# REPRESENTATIONS OF INTERMEDIATE LANGUAGES

- **Syntax trees:**

1. Natural hierarchy of the statements
2. Syntax trees and DAGs
3. Eg:  $a = b * -c + b * -c$



(a) Syntax tree.



(b) Dag.

# REPRESENTATIONS OF INTERMEDIATE LANGUAGES (Contd..)

- Postfix notation:

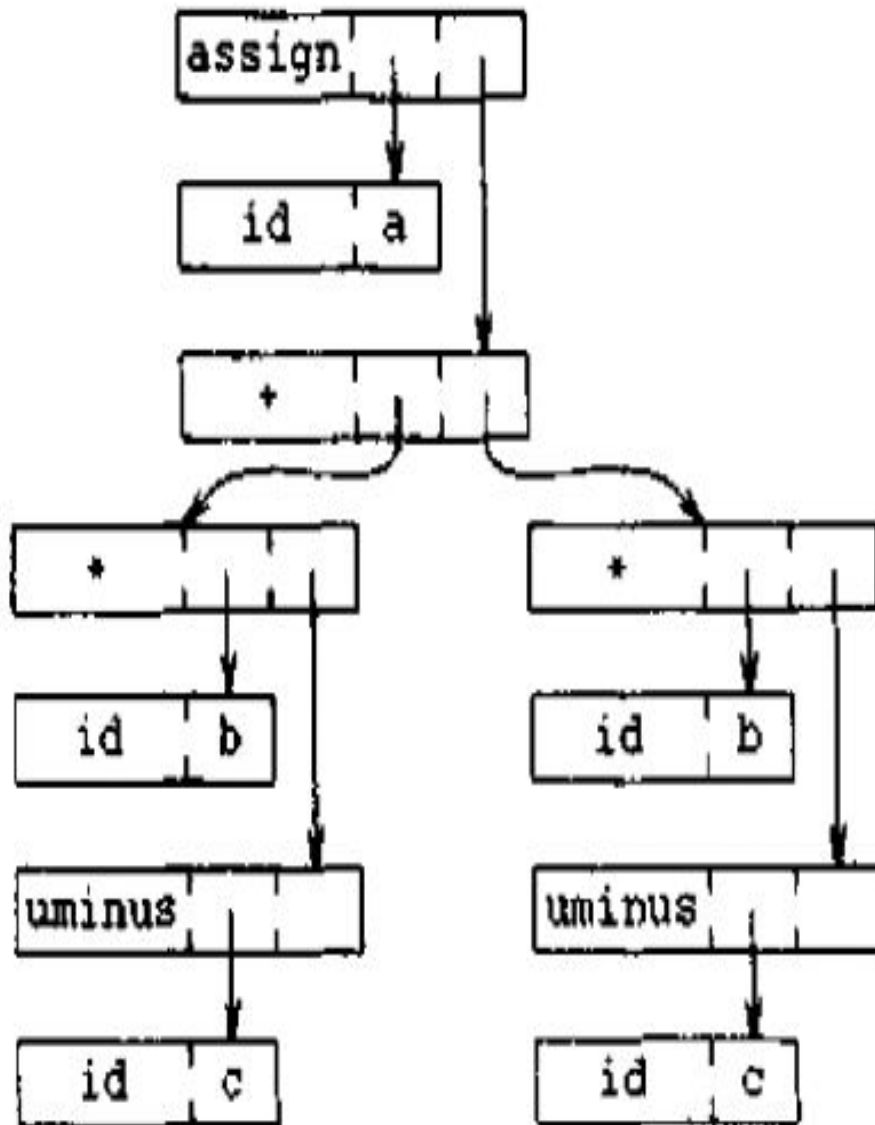
1. Linearized representation
2. Operators will follow the operands.

Eg:     a b c uminus \* b c uminus \* + assign

# SDD for creating Syntax Trees

PRODUCTION	SEMANTIC RULE
$S \rightarrow \text{id} := E$	$S.nptr := mknode('assign', mkleaf(\text{id}, \text{id.place}), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknode('+', E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknode('*', E_1.nptr, E_2.nptr)$
$E \rightarrow - E_1$	$E.nptr := mkunode('uminus', E_1.nptr)$
$E \rightarrow ( E_1 )$	$E.nptr := E_1.nptr$
$E \rightarrow \text{id}$	$E.nptr := mkleaf(\text{id}, \text{id.place})$

# REPRESENTATIONS OF SYNTAX TREES



0	id	b	
1	id	c	
2	uminus		
3	+	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11			

# THREE ADDRESS CODES

- $x = y \text{ operator } z$

$x, y, z$  are names, constants or compiler generated temporary values.

Eg:  $a = b * -c + b * -c$  (for syntax tree)

$t1 = -c$

$t2 = b * t1$

$t3 = -c$

$t4 = b * t1$

$t5 = t2 + t4$

$a = t5$

Eg:  $a = b * -c + b * -c$  (for DAG)

$t1 = -c$

$t2 = b * t1$

$t3 = t2 + t2$

$a = t3$



# TYPES OF THREE ADDRESS CODES

- **Assignment Statements:**  $x = a \text{ operator } b$
- **Assignment Statement s:**  $x = \text{operator } a$
- **Copy Statements:**  $a = b$
- **Unconditional goto:** Jump A
- **Conditional jumps:**  
if  $x \text{ relationaloperator } y$  goto A
- **Procedure calls:** param  $x$  , call  $p, n$ , return  $a$ .
- **Indexed Statement:**  $x = y[i]$  and  $x[i] = y$
- **Address and Pointers:**  $a = \&b$  and  $a = *b$

# Assignment statements

Form	Intermediate Code
$S \sqsupset id := E$	<pre> { <math>p := lookup(id.name)</math>;   if <math>p \neq nil</math> then emit (<math>p := E.place</math>)   else error } </pre>
$E \sqsupset E1 + E2$	<pre> { <math>E.place := newtmp</math>;   emit(<math>E.place := E1.place +</math>     <math>E2.place</math>) } </pre>
$E \sqsupset -E1$	<pre> { <math>E.place := newtmp</math>;   emit(<math>E.place := 'uminus' E1.place</math>) } </pre>
$E \sqsupset (E1)$	<pre> { <math>E.place := E1.place</math>; } </pre>
$E \sqsupset id$	<pre> { <math>p := lookup(id.name)</math>;   if <math>p \neq nil</math> then <math>E.place := p</math>   else error } </pre>

# BOOLEAN EXPRESSIONS

- **Two approaches:**
  - 1. Numerical approach:** True (1) and false (0)
  - 2. Flow of control:** Represent value of boolean by position in program

# NUMERICAL APPROACH

`if a < b then 1 else 0`



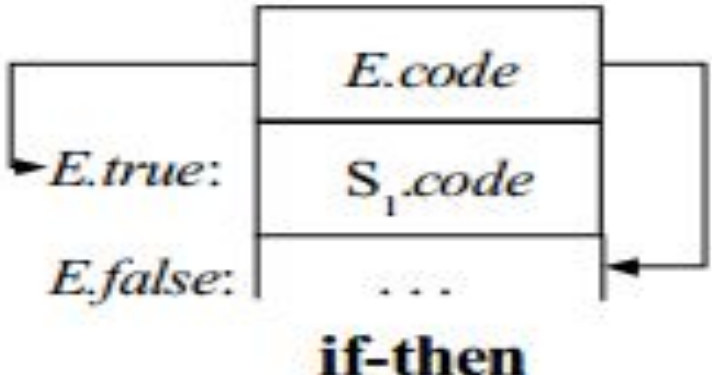
```
100:    if a < b goto 103
101:    t := 0
102:    goto 104
103:    t := 1
104:    ...
```

Statement	Intermediate Code
$E \sqsubseteq E1 \text{ or } E2$	<pre>{ E.place := newtmp;   emit(E.place ': = ' E 1 .place 'or' E 2 .place) }</pre>
$E \sqsubseteq E1 \text{ and } E2$	<pre>{ E.place := newtmp;   emit(E.place ': = ' E 1 .place 'and' E 2 .place) }</pre>
$E \sqsubseteq \text{not } E1$	<pre>{ E.place := newtmp;   emit(E.place ': = ' 'not' E1.place) }</pre>
$E \sqsubseteq (E1)$	<pre>{ E.place := E1.place; }</pre>
$E \sqsubseteq \text{true}$	<pre>{ E.place := newtmp; emit(E.place ': = 1' )}</pre>
$E \sqsubseteq \text{false}$	<pre>{ E.place := newtmp; emit(E.place ': = 0' )}</pre>

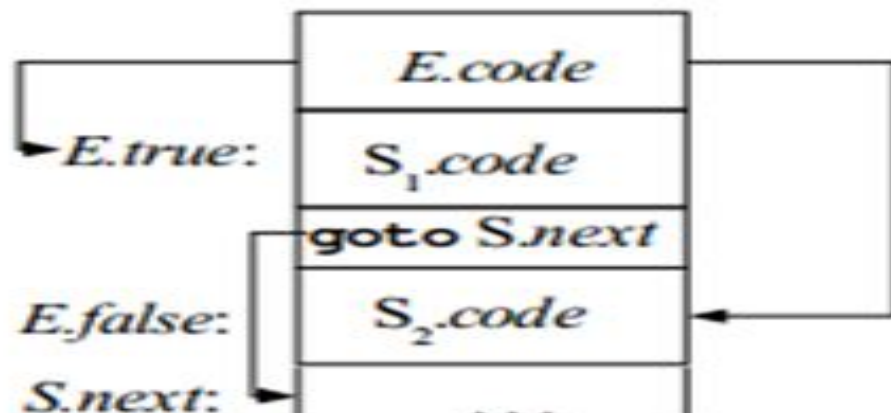
# FLOW OF CONTROL STATEMENTS

$S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S_1$

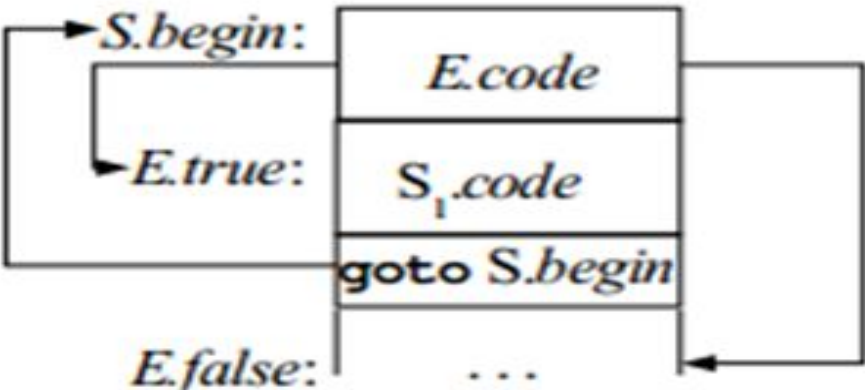
- Each boolean expression  $E$  gets two label attributes: TRUE AND FALSE
- Inherited attribute  $S.next$  is address of next instruction after  $S$
- Inherited attribute  $S.begin$  is address of first instruction of  $S$

Grammar	Intermediate Code
<p><math>S \rightarrow \text{if } E \text{ then } S_1</math></p>	<pre> { E.true := newlabel;   E.false := S.next;   S1.next := S.next;   S.code := E.code    gen (E.true ':' ) S1.code } </pre>
	 <p style="text-align: center;"><b>if-then</b></p>

<i>Grammar</i>	<i>Intermediate Code</i>
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	<pre> { <i>E.true</i> := newlabel;   <i>E.false</i> := newlabel;   <i>S1.next</i> := <i>S.next</i>;   <i>S2.next</i> := <i>S.next</i>;   <i>S.code</i> := <i>E.code</i>    gen (<i>E.true</i> ':' ) <i>S1.code</i>        gen ('goto' <i>S.next</i> )    gen (<i>E.false</i> ':' ) <i>S2.code</i> } </pre>





Grammar	Intermediate Code
<p><math>S \rightarrow \text{while } E \text{ do}</math>  <math>S1</math></p>	<pre> { <i>S.begin</i> := newlabel;   <i>E.true</i> := newlabel;   <i>E.false</i> := <i>S.next</i>;   <i>S1.next</i> := <i>S.begin</i>;   <i>S.code</i> := gen (<i>S.begin</i> ':' )    <i>E.code</i>        gen (<i>E.true</i> ':' )    <i>S1.code</i>        gen ('goto' <i>S.begin</i>) }</pre>
	 <p style="text-align: center;"><b>while-do</b></p>

# Control flow for Boolean Expressions

$E \sqsubseteq E1 \text{ or } E2$	<pre>{ E1 .true := E.true;   E1 .false := newlabel   E2 .true := E.true;   E2 .false := E.false;   E.code := E1 .code    gen (E1 .false ':' )      E2 .code }</pre>
$E \sqsubseteq E1$ and $E2$	<pre>{ <i>E1.true := newlabel;</i>   <i>E1.false := E.false</i>   <i>E2.true := E.true;</i>   <i>E2.false := E.false;</i>   <i>E.code := E1.code    gen (E1.true ':' )   </i>   <i>E2.code }</i></pre>

# Control flow for Boolean Expressions

$E \sqsubseteq \text{not } E1$	<pre>{ E1.true := E.false;   E1.false := E.true   E.code := E1.code; }</pre>
$E \sqsubseteq (E1)$	<pre>{<i>E1.true := E.true;</i> <i>E1.false := E.false</i> <i>E.code := E1.code;</i>}</pre>
$E \sqsubseteq \text{true}$	<pre>{ <i>E.code := gen ( 'goto' )    E.true</i> }</pre>
$E \sqsubseteq \text{false}$	<pre>{ <i>E.code := gen ( 'goto' )    E.false</i> }</pre>

# DECLARATIONS

GRAMMAR	DESCRIPTION
$D \rightarrow T \text{ id} \mid D \mid \in$	Declares sequence of variables with T as type and id as identifier.
$T \rightarrow B \mid C \mid \text{record } \{ \{ 'D' \} \}$	Declares basic, array or record types
$B \rightarrow \text{int} \mid \text{float}$	Declares datatype
$C \rightarrow \in \mid [\text{num}] C$	Declares array

# DECLARATIONS

$T \rightarrow B$   
 $C$                        $\{ t = B.type; w = B.width; \}$

$B \rightarrow \text{int}$                        $\{ B.type = \text{integer}; B.width = 4; \}$

$B \rightarrow \text{float}$                        $\{ B.type = \text{float}; B.width = 8; \}$

$C \rightarrow \epsilon$                        $\{ C.type = t; C.width = w; \}$

$C \rightarrow [\text{num}] C_1$                        $\{ \text{array}(\text{num.value}, C_1.type);$   
    $C.width = \text{num.value} \times C_1.width; \}$

# CASE STATEMENTS

code to evaluate E into t

goto test

L1: code for S1

goto next

L2: code for S2

goto next

.....

Ln: code for Sn

goto next

test:

# BACKPATCHING

- This way to implement boolean expressions and flow of control statements in one pass.
- **makelist(i)**: create a newlist containing only i, return a pointer to the list.
- **merge(p1,p2)**: merge lists pointed to by p1 and p2 and return a pointer to the concatenated list
- **Backpatch (p, i)**: insert i as the target label for the statements in the list pointed to by p.
- **code** is generated as **quadruples** into an array
- **labels** are **indices** into this array

# Procedure for Backpatching

- Insert a marker (sentinal )non terminal M into the grammar to pick up index of next quadruple.
- attributes truelist and falselist are used to generate jump code for boolean expressions.
- incomplete jumps are placed on lists pointed to by E.truelist and E.falselist .



# BACKPATCHING

$E \sqsubseteq E1 \text{ and } M E2$	<pre>backpatch(E1 .truelist, M.quad) E.truelist = E2.truelist E.falselist = merge(E1 .falselist, E2.falselist)</pre>
	<ul style="list-style-type: none"><li>• if E1 is false then E is also false . Place statements in E1 .falselist in E.falselist</li><li>• if E1 is true then E2 must be tested. So target of E1.truelist is beginning of E2</li><li>• target is obtained by marker M</li><li>• attribute M.quad records the number of the first statement of E2 .code</li><li>• M.quad = nextquad. Contains index of next quadruple.</li><li>• This value is backpatched to E1.truelist.</li></ul>

# BACKPATCHING

$E \sqsubseteq E1 \text{ or } M E2$	<pre>backpatch(E1 .falselist, M.quad) E.truelist = merge(E1 .truelist, E2 .truelist) E.falselist = E2 .falselist</pre>
$E \sqsubseteq \text{not } E1$	<pre>E.truelist = E1 falselist E.falselist = E1 .truelist</pre>
$E \sqsubseteq (E1)$	<pre>E.truelist = E1 .truelist E.falselist = E1 .falselist</pre>
$E \sqsubseteq \text{id1 relop id2}$	<pre>E.truelist = makelist(nextquad) E.falselist = makelist(nextquad+ 1) emit(if id1 relop id2 goto --- ) emit(goto ---)</pre>

# PROCEDURE CALLS

The translation for a call includes

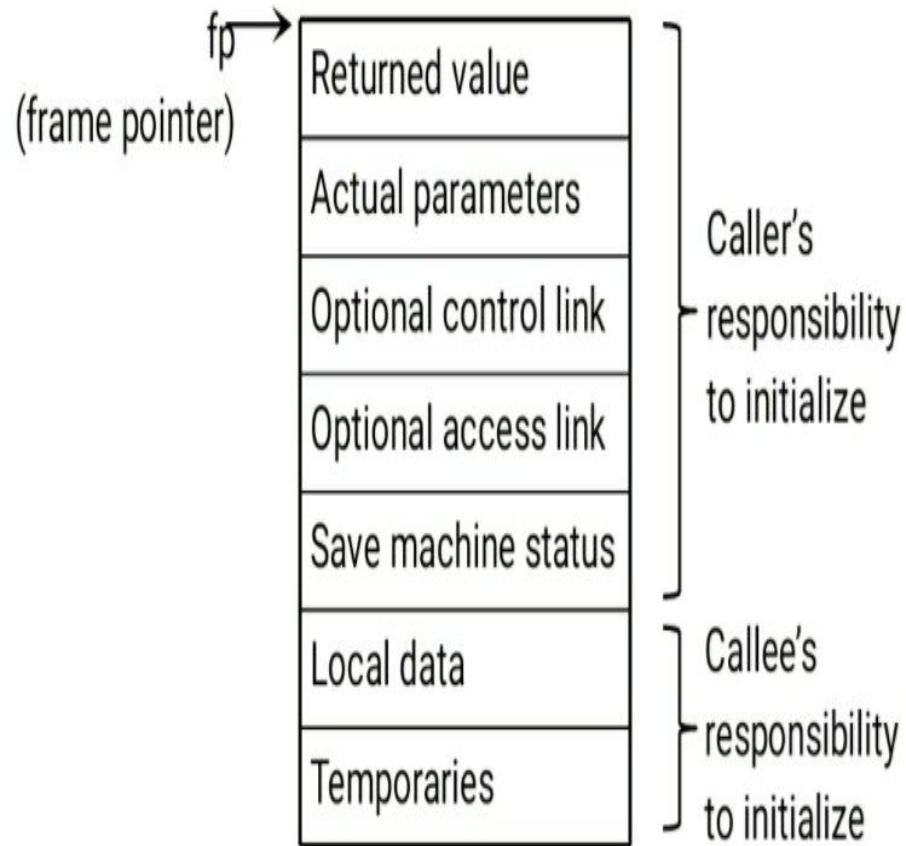
- calling sequence
- sequence of actions on entry to and exit from each procedure

**Activation Record** or **stack frame** is a block of memory used for managing the information needed by a single execution of a procedure.

The following actions take place when a procedure call occurs:

1. Memory allocation for **activation record** of the called procedure.

# STRUCTURE OF ACTIVATION RECORD



- Temporary values
- Local data of a procedure.
- machine state :PC, register values to be restored.
- Access links (optional): non-local data from other activation records.
- Actual parameters
- return value

# ACTIONS DURING PROCEDURE CALLS

- Memory allocation for the activation record of the called procedure.
- Evaluation of arguments of the called procedure.
- Setting Environment pointers: to access data .
- Save the state of the calling procedure.
- Jump to the beginning of the code for the called procedure.

# SDD FOR PROCEDURE CALLS

(1)  $S \rightarrow \text{call id ( Elist )}$

```
{ for each item p on queue do  
    emit ( ' param' p );  
    emit ( 'call' id.place)  
}
```

- Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement.

(2)  $\text{Elist} \rightarrow \text{Elist} , E$

```
{ append E.place to the end of queue }
```

(3)  $\text{Elist} \rightarrow E$

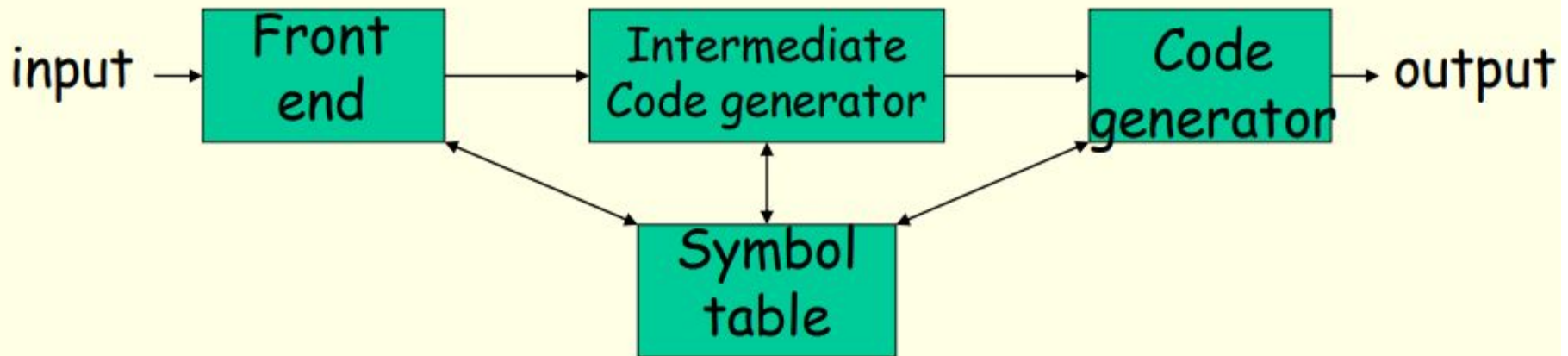
```
{ initialize queue to contain only E.place }
```

# CODE GENERATION

## Properties of Object Code

- exact meaning of the source code.
- efficient in CPU usage and memory management.

Code optimization is after code generation.



# ISSUES IN CODE GENERATION

The following are the issues in code generation phase:

1. Input to code generator
2. Target Program
3. Memory Management
4. Instruction Selection
5. Register Allocation Issues: Code Generation
6. Choice of evaluation Issues: Code Generation
7. Approaches to code generation Issues: Code Generation



# 1. Input to Code Generator

- **Input:** Intermediate Code
- Intermediate codes may be represented in postfix notation, quadruples, and syntax trees or DAGs.
- Also they must be error free from semantic errors.

## 2. Target Program

- **Output:** target program.
- Target may be absolute machine language, relocatable machine language, or assembly language.
- **Absolute machine language:** placed in a location in memory and immediately executed.
- **Relocatable machine language:** subroutine can be compiled separately. Object modules can link together and loaded by loader.
- **Assembly language:** easy code generation.

# 3. Memory Management

- Mapping names in the source program to addresses of data objects is done by the front end and the code generator.
- A name in a three-address statement refers to a symbol table entry for three-address the name.
- From the symbol table information, a relative address can be determined .

## 4. Instruction Selection

- Selecting best instructions will improve the efficiency of the program.

### **Example :**

a := b + c

d := a + e

would be translated into

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

- So, Here the fourth statement is redundant, so we can eliminate that statement.

# 5. Register Allocation Issues: Code Generation

- Use of registers in the instructions makes the computations faster.
- During register allocation, only some set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, the specific register is picked to access the variable.
- As the number of variables increase, the optimal assignment of registers to variables becomes difficult (NP complete).

## 6. Choice of evaluation Issues: Code Generation

- The order of computations affects the efficiency of the target code.
- Choose the order which demands fewer use of registers, which is a NP complete problem.

# 7. Approaches to code generation

## Issues: Code Generation

- Code generator must generate the correct code.
- Desirable properties of target program (Design goals of code generator):
  - ✓ Correct
  - ✓ Easily implementable
  - ✓ Testable
  - ✓ Maintainable

$S \rightarrow aAS \text{ \{print 3\}}$

$S \rightarrow a \text{ \{print 1\}}$

$A \rightarrow SbA \text{ \{print 3\}}$

$A \rightarrow SS \text{ \{print 2\}}$

$A \rightarrow ba \text{ \{print 2\}}$

For generating aabbbaa what will be the output printed?



# TARGET MACHINE

- Code generation demands familiarity with target machine and instruction set.

## **Familiarity with target machine:**

- byte-addressable machine with 4 bytes to a word.
- $N$  general-purpose registers,  $R_0, R_1, \dots, R_{n-1}$ .
- **General Form:** Opcode source destination
- **Opcode:** Add, Sub, Mov

# Instruction Costs

- Instruction cost = 1+cost for source and destination address modes.
- This cost depends on the length of the instruction.
- Address modes involving
  - ✓ registers have cost zero.
  - ✓ memory location or literal have cost one.
- Reducing instruction length will minimize space utilization and decreases the fetching time of the operands.

# Instruction Costs

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

# Cost Computation

**a := b + c**

1. MOV b, R0 // Cost 2

ADD c, R0 // Cost 2

MOV R0, a //Cost 2

Total cost = 6

2. MOV b, a //Cost 3

ADD c, a //Cost 3

Total cost = 6

3. MOV \*R1, \*R0

ADD \*R2, \*R0

cost = 2

Assuming R0, R1 and R2 contain the addresses of a, b, and c

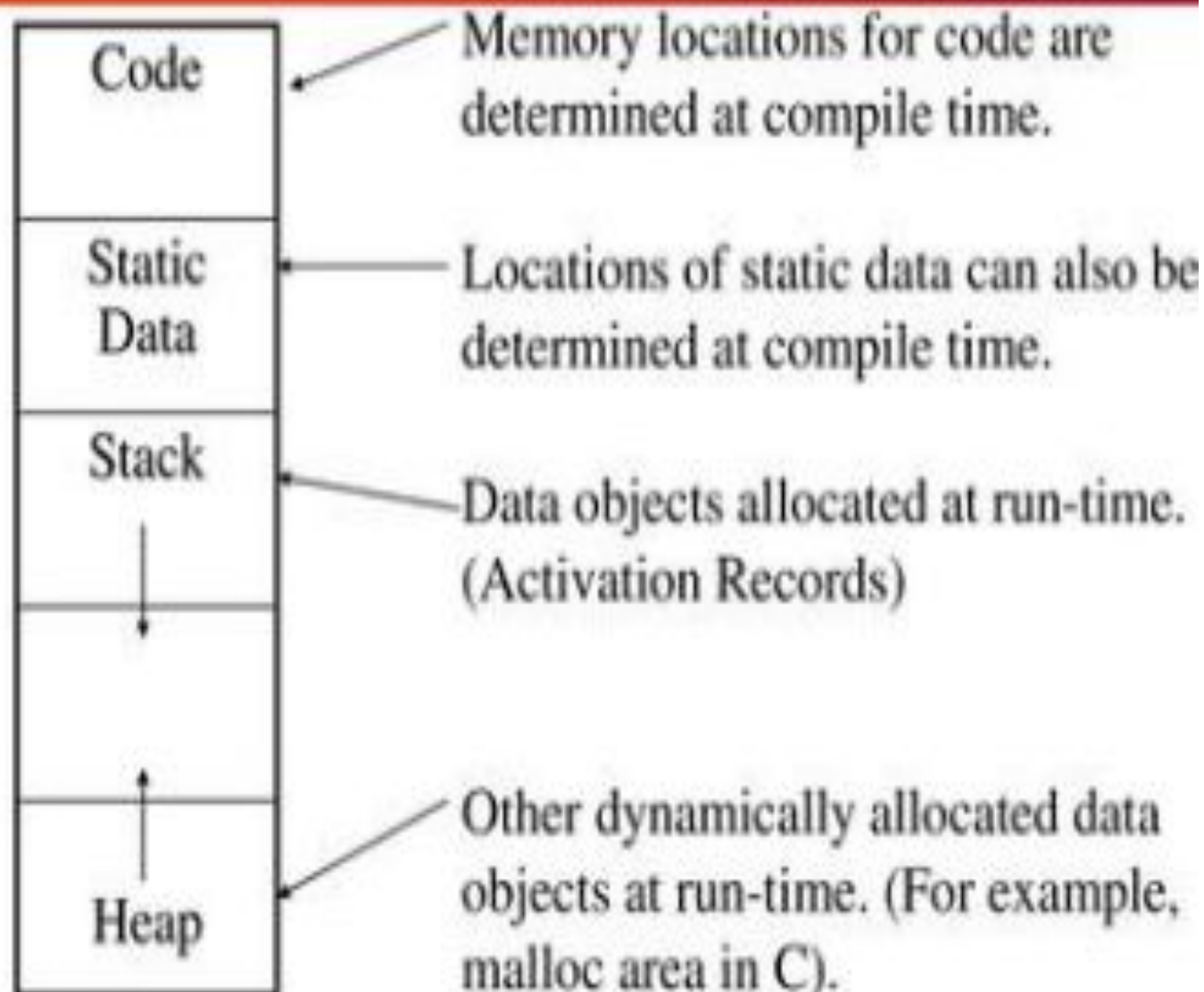
# RUNTIME STORAGE MANAGEMENT

**Activation Record:** Information needed during activation of procedure.

## **Storage Allocation Strategies:**

- **Static allocation:** memory of an activation record is fixed at compile time.
- **Stack allocation:** new activation record is pushed onto the stack for execution. It will be popped after the execution is over.

Run time memory is divided into: Code, Static data and Stack



# RUNTIME STORAGE MANAGEMENT

- The compiler must implement various abstractions in the source language definition such as
  1. Names used in a program
  2. Define the scope of variables
  3. Data types
  4. Operators
  5. Procedures
  6. Parameters
  7. and Flow of control constructs.
- The compiler must co-operate with operating system and other systems software to support the implementation of these abstractions on the target machine.
- This can be done by the compiler by creating run-time environment.

# **Why runtime environment is needed?**

1. Managing the processor stack.
2. Layout and allocation of memory for various variables used in the source program.
3. Instructions to copy the actual parameters on top of the stack when a function is called.
4. Allocating and de-allocating the memory dynamically with the help of operating system.
5. The mechanism used by the target program to access variables.
6. The mechanism to pass parameters.
7. The interfaces to the operating system, I/O devices and other programs.



# Why runtime environment is needed?

- At run time, we need a system to map NAMES (in the source program) to STORAGE on the machine.
- Allocation and deallocation of memory is handled by a RUN- TIME SUPPORT SYSTEM typically linked and loaded along Run-Time Environments
- TIME SUPPORT SYSTEM typically linked and loaded along with the compiled target code.
- One of the primary responsibilities of the run-time system is to manage ACTIVATIONS of procedures

# STORAGE ALLOCATION STRATEGIES

The various storage allocation strategies to allocate storage in different data areas of memory are:

- 1. Static Allocation:** Storage is allocated for all data objects at compile time Stack Allocation
- 2. Stack Allocation:** The storage is managed as a stack
- 3. Heap Allocation:** It is one of Dynamic Storage Allocation. The storage is allocated and deallocated at runtime from a data area known as heap.

# STATIC ALLOCATION

- Statically allocated names are bound to relocatable storage at compile time.
- Storage bindings of statically allocated names never change.
- The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required
- The required number of bytes (possibly aligned) is set aside for the name.
- The relocatable address of the storage is fixed at compile time

# **Actions during static allocation**

1. Call
2. Return
3. Halt
4. Action

## **Implementation of call statement:**

callee.static\_area - Address of the activation record

callee.code\_area - Address of the first instruction for called procedure

- 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

# 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.

# LIMITATIONS OF STATIC ALLOCATION

- The size required must be known at compile time.
- Recursive procedures cannot be implemented statically.
- No data structure can be created dynamically as all data is static.



# STACK ALLOCATION

- In a stack-based allocation, the previous restrictions are lifted (Pascal, C, etc)
- procedures are allowed to be called recursively
- Need to hold multiple activation records for the same procedure
- Created as required and placed on the stack
- Each record will maintain a pointer to the record that activated it
- On completion, the current record will be deleted from the stack and control is passed to the calling record
- Dynamic memory allocation is allowed
- Pointers to data locations are allowed

# STACK ALLOCATION

- Storage is organized as a stack.
- Activation records are pushed and popped.
- Locals and parameters are contained in the activation records for the call.
- This means locals are bound to fresh storage on every call.
- We just need a `stack_top` pointer.
- To allocate a new activation record, we just increase `stack_top`.
- To deallocate an existing activation record, we just decrease `stack_top`.

# STACK ALLOCATION

- When relative address is used in static allocation , then it becomes stack allocation.
- Activation records are accessed as offsets from the value in this register.
- **Initialization of stack:**

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

```
//Code for the first procedure
```

```
HALT
```

### **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:** It is a piece of straight line code, without any jumps in or out of the middle of a block.

## **Algorithm to find Basic Block:**

1. Identify the header of basic block. Potential candidates for headers are:

- First statement of a program.
- Statements that are target of any branch (conditional/unconditional).
- Statements that immediately follow any branch statement.

2. Header statements and the statements following them form a basic block.

3. A basic block does not include any header statement of any other basic block

# Example: Matrix Multiplication

```
1)    i = 1
2)    j = 1
3)    t1 = 10 * i
4)    t2 = t1 + j
5)    t3 = 8 * t2
6)    t4 = t3 - 88
7)    a[t4] = 0.0
8)    j = j + 1
9)    if j <= 10 goto (3)
10)   i = i + 1
11)   if i <= 10 goto (2)
12)   i = 1
13)   t5 = i - 1
14)   t6 = 88 * t5
15)   a[t6] = 1.0
16)   i = i + 1
17)   if i <= 10 goto (13)
```

# Basic Blocks

Block	Code
B1	1) i=1
B2	2) j=1
B3	3) t1 = 10 * i 4) t2 = t1 + j 5) t3 = 8 * t2 6) t4 = t3 - 88 7) a[t4] = 0.0 8) j = j + 1 9) if j <= 10 goto (3)
B4	10) i = i + 1 11) if i <= 10 goto (2)
B5	12) i = 1
B6	13) t5 = i - 1 14) t6 = 88 * t5 15) a[t6] = 1.0 16) i = i + 1 17) if i <= 10 goto (13)

# TRANSFORMATIONS ON BASIC BLOCKS

- For improving the quality of code.
- **Types:**
  - ✓ Structure preserving transformations
  - ✓ Algebraic transformations

## **Structure preserving Transformations**

1. Common Sub expression Elimination
2. Dead code Elimination
3. Renaming temporary variables
4. Interchange of two independent adjacent statements.



# Structure preserving Transformations

## Common sub expression elimination:

**Example:**  $a=b+c$ ;  $b=a-d$ ;  $c=b+c$ ;  $d=a-d$

Transform into:  $a=b+c$ ;  $b=a-d$ ;  $c=b+c$ ;  $d=a$

## Dead code Elimination:

A variable is said to be dead, if it is not subsequently used. That variable can be removed from basic block.

**Example:** If  $x$  is not used elsewhere in the program, the  $x=y-8$  can be eliminated.

# Structure preserving Transformations

- **Renaming of temporary variables**

Consider  $t := a+b$  where  $t$  is a temporary. If we change 't' to 'u', then the value of basic block is not changed.

- **Interchange of statement**

Consider the two adjacent statements:

$a=b+c$

$U= i+j$

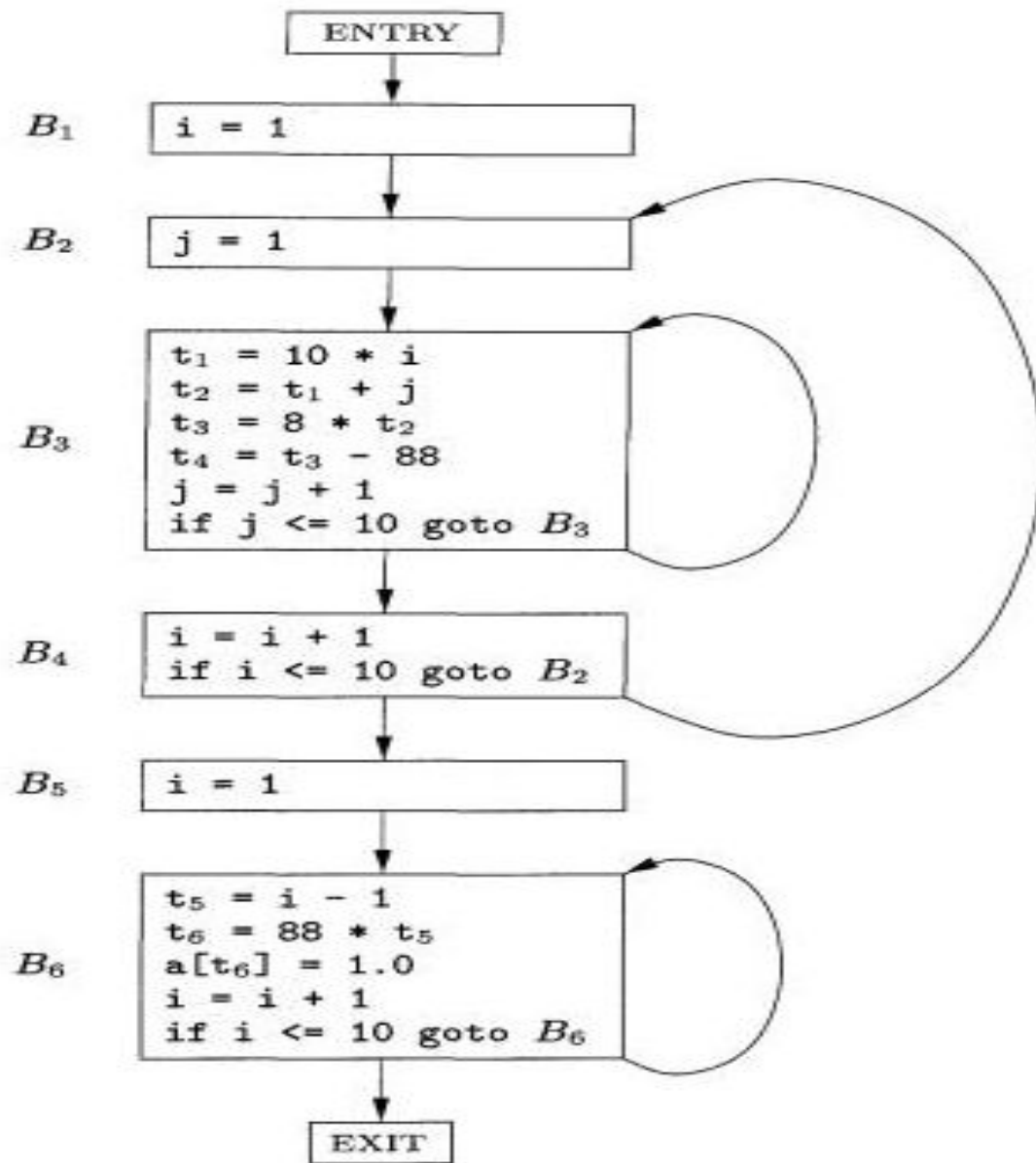
These two statements can be interchanged, if  $b$  and  $c$  don't affect  $u$ ;  $i$  and  $j$  don't affect  $a$ .

# Algebraic transformations

- Countless algebraic transformation can used to change the set of expression algebraic computed by the basic block into an algebraically equivalent set.
- The useful ones are those that simplify expressions or replace expensive operation by cheaper one.
- Example:
  - i)  $x = x + 0$  or  $x = x * 1$  can be eliminated.
  - ii) Use  $x = x * x$  instead of  $\text{pow}(x, 2)$

# FLOW GRAPHS

- A graph representation of three-address statements called a flow graph.
- Representation of basic blocks.
- Understanding code-generation algorithms.
- Nodes in the flow graph represent computations,
- Edges represent the flow of control.



# Rules for flow graph

- Directed edges will be drawn from block B1 to B2, only if B2 follows B1 during execution.
- Two cases:
  1. There is a conditional or unconditional jump from B1 to first statement of B2.
  2. B2 immediately follows B1 and B1 don't end in an unconditional jump.
- B1 is predecessor of B2 or B2 is successor of B1.

# Loops in Flow graph

Loops could be identified through the following:

- All nodes are strongly connected (i.e.) it must be possible to reach from a statement to another inside the same loop.
- Collection of nodes with unique entry (i.e.) the only way to reach a node of the loop from a node outside the loop is through its entry point.
- A loop which contains no other loop is called inner loop.

# NEXT USE INFORMATION

- In every basic block we need to find whether the value contained in the variable will be used again later in the block.
- If a variable has no next-use , then the register allocated to the variable can be reused.
- **Live-on-exit** : If the value contained in the variable has a use outside the block.
- Global data-flow analysis is used to find Live on exit.



# Example

1. `x=i;`
  2. `y=x + a; // y uses x value.`
- The next-use information is collected by backward scanning of the code in that specific block.
  - Next use and Live to exit information must be collected from symbol table before register reuse.

# Next Use Algorithm

Scan backwards. Consider the statement

**i)  $x=y \text{ op } z$ .**

1. Attach to the statement i, the live to exit and next use information of x, y and z currently found in symbol table .
2. In symbol table set x as “not live ” and “no next use”.
3. Set y and z as “live ” and “next use of y and z as i”.

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

# Example for Next Use

	<u>Defs</u>	<u>Next-Uses</u>
104: $y := a + 5$	...	...
105: ...	104	{106, 109, 112}
106: $b := y * b$	105	...
107: ...	106	{109, 110}
108: ...	107	...
109: $x := b * y$	108	...
110: $b := b - x$	...	...
111: ...	...	...
112: $c := y + b$		

# Storage for Temporary Names

- Two temporaries can be loaded to same location if they are not live simultaneously.

t1 := a \* a

t2 := a \* b

t3 := 4 \* t2

t4 := t1 + t3

t5 := b \* b

t6 := t4 + t5

t1 := a \* a

t2 := a \* b

t2 := 4 \* t2

t1 := t1 + t2

t2 := b \* b

t1 := t1 + t2

# SIMPLE CODE GENERATOR

- Generates target code
- Variables can be left in registers only
  - i) If their register is needed for another computation.
  - ii) Just before a procedure call, jump.
- Code generator algorithm stores everything while moving across the boundaries of basic blocks.

# Simple code generator

- **Register descriptor:** Keeps track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- **Address descriptor:** Stores the location where the current value of the name can be found at run time.

# 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  $x := y \text{ op } z$ , perform the following actions:

1. Invoke `getreg()` to determine the location  $L$  which stores the result of  $y \text{ op } z$ .
2. Consult the address descriptor for  $y$  to determine the current location of  $y'$ . If  $y$  is present both in memory and registers, give more priority for the location in register.

# CODE GENERATION ALGORITHM

- Generate the instruction  $OP\ z', L$  where  $z'$  is a current location of  $z$ . 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.
- 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

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

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

Statements	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 R0
$d := v + u$	ADD R1, R0  MOV R0, d	R0 contains d	d in R0 d in R0 and memory

# Generating Code for Indexed Assignments

$a := b[i]$  and  $a[i] := b$

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

# Generating Code for Pointer Assignments

$a := *p$  and  $*p := a$

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

# DAG REPRESENTATION OF BASIC BLOCKS

- DAGs are useful for implementing transformations on basic blocks.
- how the value computed by a statement is used in subsequent statements.
- determining common sub - expressions.

A DAG for a basic block has the following labels on nodes:

1. Leaves are labelled by unique identifiers, either variable names or constants.
2. Interior nodes are labelled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

# ALGORITHM FOR DAG

Input: A basic block

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

1. Create a label for each node.

Leaves=identifier; Interior nodes=operators.

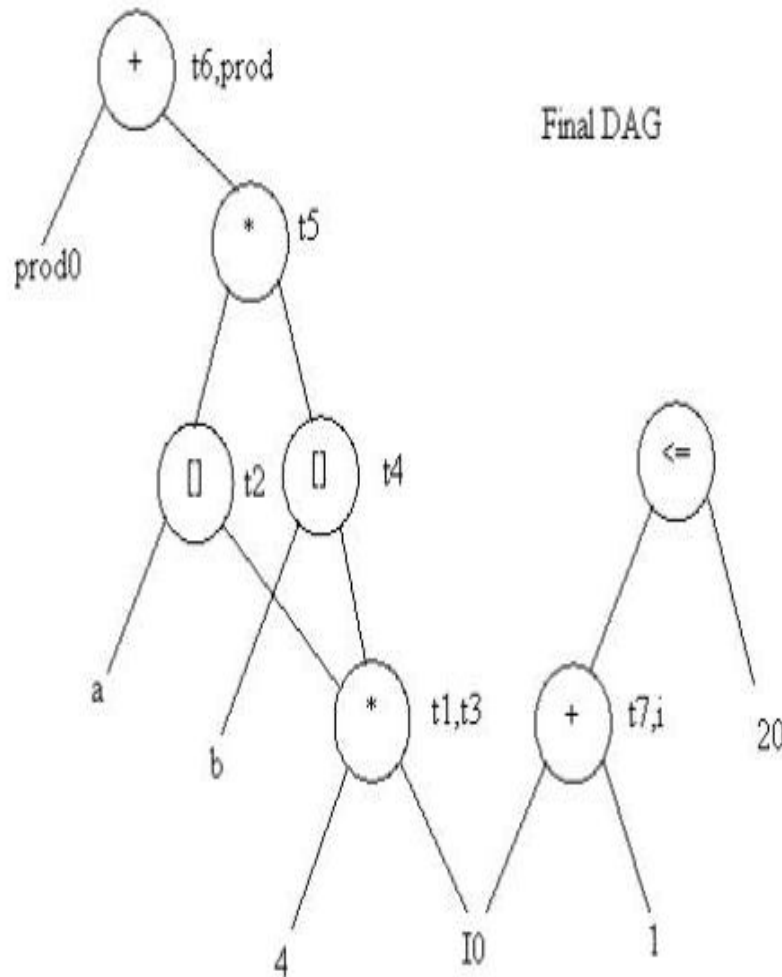
2. For each node a list which hold the computed values.

(i)  $x := y \text{ OP } z$

(ii)  $x := \text{OP } y$

(iii)  $x := y$

# DAG



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

# APPLICATIONS OF DAG

1. Automatically detect common sub expressions.
2. Find the identifiers have their values used in the block.
3. The statements computed values that could be used outside the block.



# PEEPHOLE OPTIMIZATIONS

- The peephole is a small, moving window on the target program.
- **Peephole Optimization** : A method to improve the performance of the target program by examining a short sequence of target instructions (peephole) by replacing the instructions by a shorter or faster sequence.
- Each improvement may spawn over many other further improvements.

# Characteristics of Peephole Optimization

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

# 1. Elimination of redundant instruction

Consider,

MOV a, R0

MOV R0,a

Copying done twice.

## 2. Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed.
- Eliminate jumps over jumps.
- This operation can be repeated to eliminate a sequence of instructions.

# Unreachable Code

If debug =1 goto L1 goto L2

L1: print debugging information L2: ..... (a)

Can be replaced by

If debug =1 goto L1

L1 .....

If debug !=1 goto L2

L2.....

This can eliminate potential unreachable statements.

# Flow of Control Optimizations

Goto L1

...

L1: Goto L2

Can be replaced by

**goto L2**

....

**L1: goto L2**

# Algebraic Simplifications

Eliminate

1)  $x = x * 1$

2)  $x = x + 0$

## Reduction in strength

- 1) Replace expensive operations by least cost operations.

Eg  $x^2$  is cheaper than  $x * x$  because the latter requires fetching of  $x$  twice.

- **Use of Machine Idioms:**
- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Example: The machine architecture may support auto increment and decrement operations.
- These modes improves the quality of code when pushing or popping a stack.
- `i=i+1` could be replaced as `i++`

# CROSS COMPILERS

- Runs on one machine and produces object code for another machine.
- The cross-compiler is used to implement the compiler, which can work on three languages:
  1. The source language
  2. The object language
  3. The language in which it is written.
- If a compiler has been implemented in its own language, then this arrangement is called a "bootstrap" arrangement.



**Which of the following statements about peephole optimization is False?**

- A.** It is applied to a small part of the code
- B.** It can be used to optimize intermediate code
- C.** To get the best out of this, it has to be applied repeatedly
- D.** It can be applied to the portion of the code that is not contiguous

In compiler optimization, operator strength reduction uses mathematical identities to replace slow math operations with faster operations. Which of the following code replacements is an illustration of operator strength reduction ?

Replace  $P + P$  by  $2 * P$  or Replace  $3 + 4$  by  $7$ .

Replace  $P * 32$  by  $P \ll 5$

Replace  $P * 0$  by  $0$

Replace  $(P \ll 4) - P$  by  $P * 15$