

## CONTINUATION-4— RUN-TIME ENVIRONMENTS

### SOURCE LANGUAGE ISSUES

#### Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;  
  var i : integer;  
  begin  
    for i := 1 to 9 do read(a[i])  
  end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

#### Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

#### Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

### The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

```
var i : integer ;
```

or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

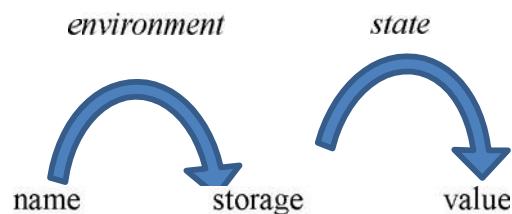
The portion of the program to which a declaration applies is called the *scope* of that declaration.

### Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. “Data object” corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.

The term *state* refers to a function that maps a storage location to the value held there.

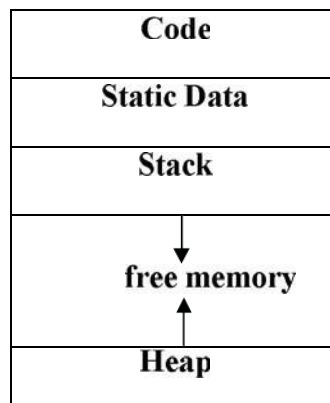


When an *environment* associates storage location  $s$  with a name  $x$ , we say that  $x$  is *bound* to  $s$ . This association is referred to as a *binding* of  $x$ .

## STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

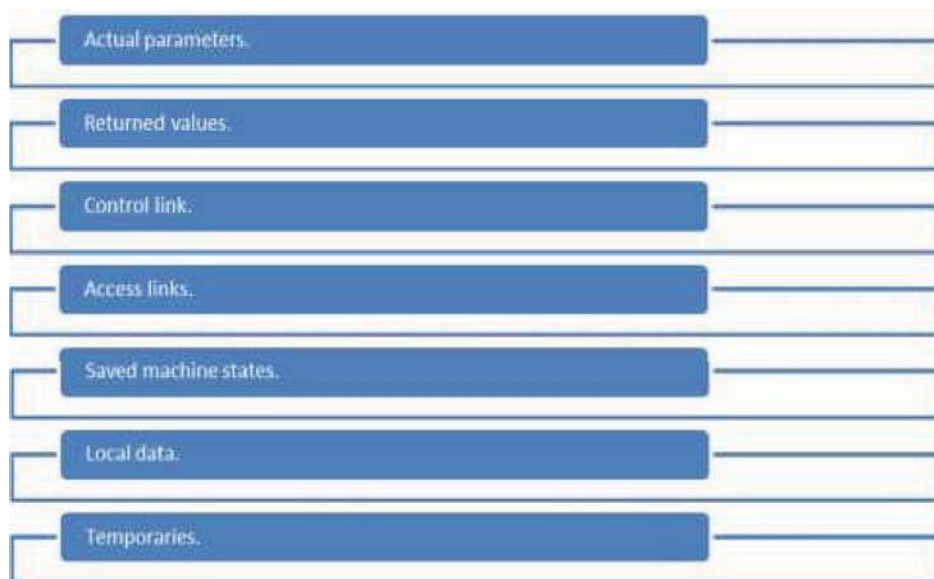
### Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

### Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack*.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.



- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

## **STORAGE ALLOCATION STRATEGIES**

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

## **STATIC ALLOCATION**

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

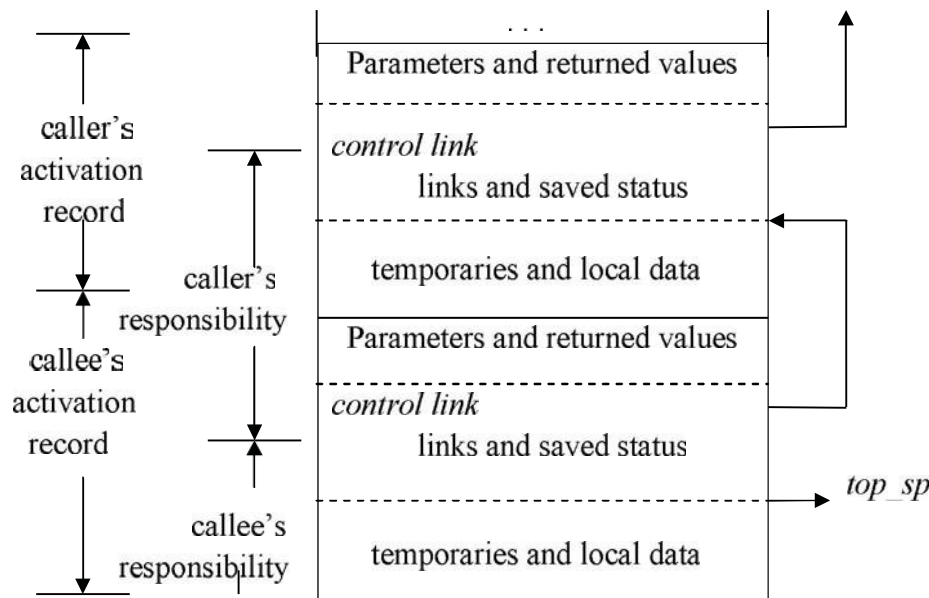
## **STACK ALLOCATION OF SPACE**

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

## **Calling sequences:**

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

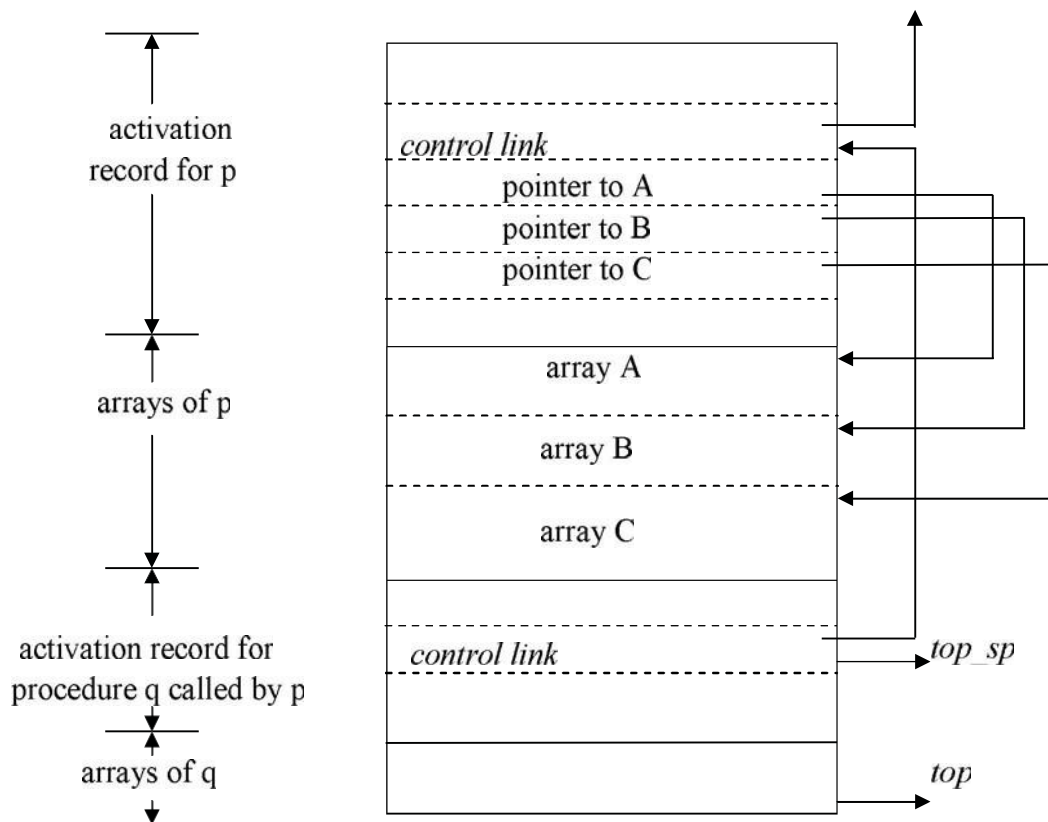


**Division of tasks between caller and callee**

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top\_sp* into the callee's activation record. The caller then increments the *top\_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top\_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top\_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top\_sp*; the caller therefore may use that value.

### Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



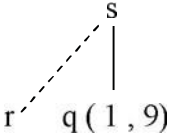
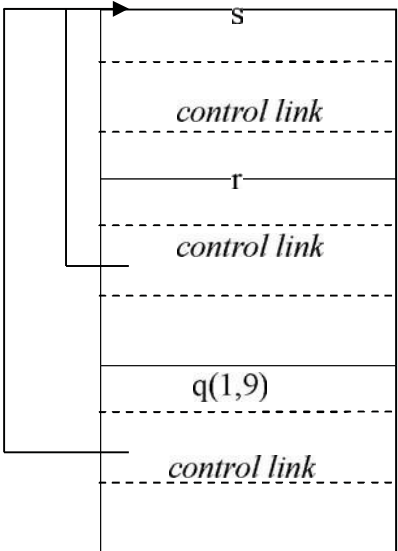
### Access to dynamically allocated arrays

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

## HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
  2. A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
  - Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

## UNIT 5 CODE GENERATION

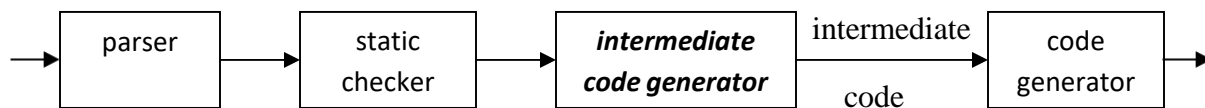
### INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

**Benefits of using a machine-independent intermediate form are:**

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

#### *Position of intermediate code generator*



### INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

- Syntax tree
- Postfix notation
- Three address code

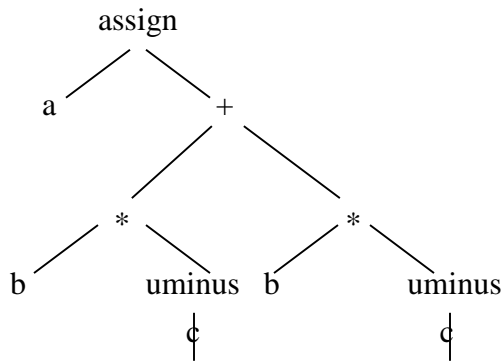
The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

#### **Graphical Representations:**

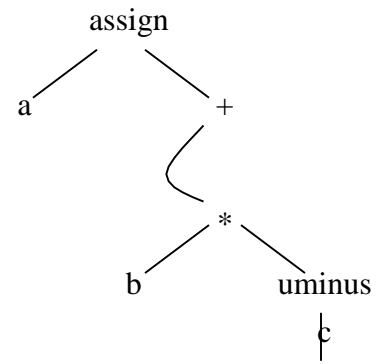
##### **Syntax tree:**

A syntax tree depicts the natural hierarchical structure of a source program. **Adag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b \* - c + b \* - c** are as follows:





(a) Syntax tree



(b) Dag

### Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

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

### Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and \* are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input  $a : = b * - c + b * - c$ .

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

### Syntax-directed definition to produce syntax trees for assignment statements

## ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

$$P \rightarrow M D$$
$$M \rightarrow \varepsilon$$
$$D \rightarrow D ; D \mid \textbf{id} : T \mid \textbf{proc id} ; N D ; S$$
$$N \rightarrow \varepsilon$$

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

### Translation scheme to produce three-address code for assignments

$S \rightarrow \text{id} : = E$	$\{ p := \text{lookup}(\text{id.name});$ $\quad \textbf{if } p \neq \text{nil} \textbf{ then}$ $\quad \text{emit}( p := E.\text{place})$ $\quad \textbf{else error } \}$
$E \rightarrow E_1 + E_2$	$\{ E.\text{place} := \text{newtemp};$ $\quad \text{emit}( E.\text{place} := E_1.\text{place} + E_2.\text{place} ) \}$
$E \rightarrow E_1 * E_2$	$\{ E.\text{place} := \text{newtemp};$ $\quad \text{emit}( E.\text{place} := E_1.\text{place} * E_2.\text{place} ) \}$
$E \rightarrow -E_1$	$\{ E.\text{place} := \text{newtemp};$ $\quad \text{emit}( E.\text{place} := \text{'uminus'} E_1.\text{place} ) \}$
$E \rightarrow ( E_1 )$	$\{ E.\text{place} := E_1.\text{place} \}$

```

E → id          { p := lookup (id.name);
                  if p ≠ nil then
                      E.place := p
                  else error }

```

## Reusing Temporary Names

- The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
- Temporaries can be reused by changing *newtemp*. The code generated by the rules for  $E \rightarrow E_1 + E_2$  has the general form:
 

```

      evaluate E1 into t1
      evaluate E2 into t2
      t := t1 + t2
      
```
- The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- Keep a count *c*, initialized to zero. Whenever a temporary name is used as an operand, decrement *c* by 1. Whenever a new temporary name is generated, use \$*c* and increase *c* by 1.
- For example, consider the assignment  $x := a * b + c * d - e * f$

### Three-address code with stack temporaries

<i>statement</i>	<i>value of c</i>
	0
\$0 := a * b	1
\$1 := c * d	2
\$0 := \$0 + \$1	1
\$1 := e * f	2
\$0 := \$0 - \$1	1
x := \$0	0

## Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *sw*, then the *i*th element of array *A* begins in location

$$base + (i - low) \times w$$

where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of *A*[*low*].

The expression can be partially evaluated at compile time if it is rewritten as

$$ixw + (base - lowxw)$$

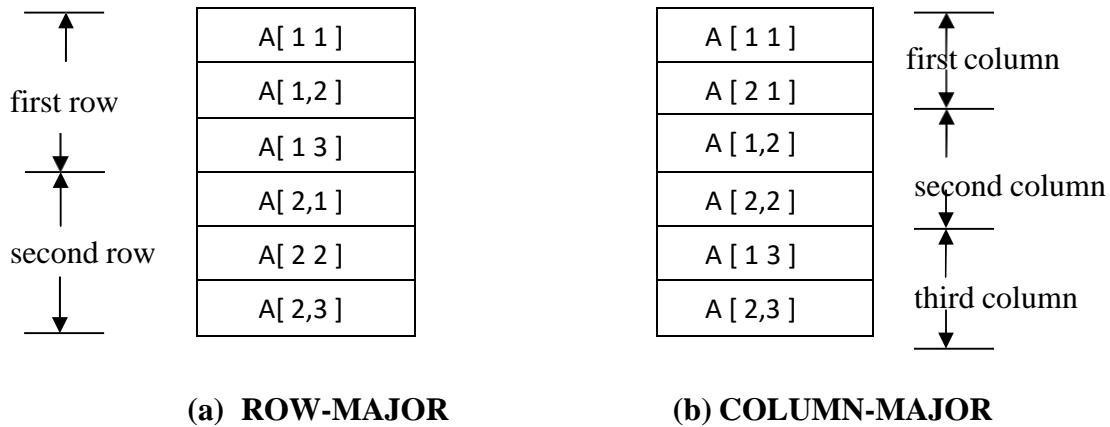
The subexpression  $c = base - lowxw$  can be evaluated when the declaration of the array is seen. We assume that  $c$  is saved in the symbol table entry for  $A$ , so the relative address of  $A[i]$  is obtained by simply adding  $ixw$  to  $c$ .

### Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

### Layouts for a 2 x 3 array



In the case of row-major form, the relative address of  $A[i_1, i_2]$  can be calculated by the formula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

where,  $low_1$  and  $low_2$  are the lower bounds on the values of  $i_1$  and  $i_2$  and  $n_2$  is the number of values that  $i_2$  can take. That is, if  $high_2$  is the upper bound on the value of  $i_2$ , then  $n_2 = high_2 - low_2 + 1$ .

Assuming that  $i_1$  and  $i_2$  are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

### Generalized formula:

The expression generalizes to the following expression for the relative address of  $A[i_1, i_2, \dots, i_k]$

$$((\dots ((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w + base - ((\dots ((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) \times w$$

for all  $j$ ,  $n_j = high_j - low_j + 1$



## The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

- (1)  $S \rightarrow L := E$
- (2)  $E \rightarrow E + E$
- (3)  $E \rightarrow (E)$
- (4)  $E \rightarrow L$
- (5)  $L \rightarrow Elist$
- (6)  $L \rightarrow id$
- (7)  $Elist \rightarrow Elist, E$
- (8)  $Elist \rightarrow id[E$

We generate a normal assignment if  $L$  is a simple name, and an indexed assignment into the location denoted by  $L$  otherwise :

- (1)  $S \rightarrow L := E \{ \text{if } L.offset = \text{null} \text{ then } /* L \text{ is a simple id} */$   
 $\quad \text{emit}(L.place \text{ ' := ' } E.place) ;$   
 $\quad \text{else}$   
 $\quad \text{emit}(L.place \text{ '[' } L.offset \text{ ' ]' ' := ' } E.place) \}$
- (2)  $E \rightarrow E_1 + E_2 \quad \{ E.place := \text{newtemp};$   
 $\quad \text{emit}(E.place \text{ ' := ' } E_1.place \text{ ' + ' } E_2.place) \}$
- (3)  $E \rightarrow ( E_1 ) \{ E.place := E_1.place \}$

When an array reference  $L$  is reduced to  $E$ , we want the  $r$ -value of  $L$ . Therefore we use indexing to obtain the contents of the location  $L.place[L.offset]$  :

- (4)  $E \rightarrow L \{ \text{if } L.offset = \text{null} \text{ then } /* L \text{ is a simple id} */$   
 $\quad E.place := L.place$   
 $\quad \text{else begin}$   
 $\quad \quad E.place := \text{newtemp};$   
 $\quad \quad \text{emit}(E.place \text{ ' := ' } L.place \text{ '[' } L.offset \text{ ' ]' })$   
 $\quad \text{end} \}$
- (5)  $L \rightarrow Elist$   $\{ L.place := \text{newtemp};$   
 $\quad L.offset := \text{newtemp};$   
 $\quad \text{emit}(L.place \text{ ' := ' } c(Elist.array));$   
 $\quad \text{emit}(L.offset \text{ ' := ' } Elist.place \text{ '*' width}(Elist.array)) \}$
- (6)  $L \rightarrow id \{ L.place := id.place;$   
 $\quad L.offset := \text{null} \}$
- (7)  $Elist \rightarrow Elist_1, E \{ t := \text{newtemp};$   
 $\quad m := Elist_1.ndim + 1;$   
 $\quad \text{emit}(t \text{ ' := ' } Elist_1.place \text{ '*' limit}(Elist_1.array, m));$   
 $\quad \text{emit}(t \text{ ' := ' } t \text{ ' + ' } E.place);$   
 $\quad Elist.array := Elist_1.array;$

*Elist.place* := *t*;  
*Elist.ndim* := *m*}

(8) *Elist*  $\sqsubseteq$  **id**[*E*{*Elist.array* := **id.place**;

*Elist.place* := *E.place*;  
*Elist.ndim* := 1 }

### Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production  $E \sqsubseteq E + E$  is :

$E \sqsubseteq E + E \{ E.type :=$   

**if** *E*<sub>1</sub>.*type* = *integer* **and**  
*E*<sub>2</sub>.*type* = *integer* **then** *integer*  
**else** *real* }

The entire semantic rule for  $E \sqsubseteq E + E$  and most of the other productions must be modified to generate, when necessary, three-address statements of the form *x* := *int* to *real* *y*, whose effect is to convert integer *y* to a real of equal value, called *x*.

#### Semantic action for $E \Rightarrow E_1 + E_2$

```

E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
    emit( E.place ':' = ' E1.place 'int + ' E2.place );
    E.type := integer
end
else if E1.type = real and E2.type = real then begin
    emit( E.place ':' = ' E1.place 'real + ' E2.place );
    E.type := real
end
else if E1.type = integer and E2.type = real then begin
    u := newtemp;
    emit( u ':' = "int to real" ' E1.place );
    emit( E.place ':' = ' u 'real + ' E2.place );
    E.type := real
end
else if E1.type = real and E2.type = integer then begin
    u := newtemp;
    emit( u ':' = "int to real" ' E2.place );
    emit( E.place ':' = ' E1.place 'real + ' u );
    E.type := real
end
else
    E.type := type_error;

```

For example, for the input  $x := y + i * j$  assuming  $x$  and  $y$  have type *real*, and  $i$  and  $j$  have type *integer*, the output would look like

```
t1 := i int * j
t3 := int to real t1
t2 := y real + t3
x := t2
```

## BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and**, **or**, and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form  $E_1 \text{ relop } E_2$ , where  $E_1$  and  $E_2$  are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid ( E ) \mid \text{id relop id} \mid \text{true} \mid \text{false}$$

### Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false *numerically* and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by *flow of control*, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

### Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for **a or b and not c** is the three-address sequence
 

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

- A relational expression such as  $a < b$  is equivalent to the conditional statement
 

```
if a < b then 1 else 0
```

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```

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

```

### Translation scheme using a numerical representation for booleans

$E \sqcap E_1 \text{ or } E_2$	{ $E.place := newtemp;$ $emit( E.place ':=' E_1.place \text{ 'or' } E_2.place )$ }
$E \sqcap E_1 \text{ and } E_2$	{ $E.place := newtemp;$ $emit( E.place ':=' E_1.place \text{ 'and' } E_2.place )$ }
$E \sqcap \text{not} E_1$	{ $E.place := newtemp;$ $emit( E.place ':=' \text{ 'not' } E_1.place )$ }
$E \sqcap ( E_1 )$	{ $E.place := E_1.place$ }
$E \sqcap \text{id}_1 \text{ relop id}_2$	{ $E.place := newtemp;$ $emit( \text{ 'if' id}_1.place \text{ relop op id}_2.place \text{ 'goto' nextstat} + 3 );$ $emit( E.place ':=' \text{ '0' } );$ $emit( \text{ 'goto' nextstat} + 2 );$ $emit( E.place ':=' \text{ '1' } )$ }
$E \sqcap \text{true}$	{ $E.place := newtemp;$ $emit( E.place ':=' \text{ '1' } )$ }
$E \sqcap \text{false}$	{ $E.place := newtemp;$ $emit( E.place ':=' \text{ '0' } )$ }

### Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “**short-circuit**” or “**jumping**” code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and**, **or**, and **not** if we represent the value of an expression by a position in the code sequence.

### Translation of $a < b \text{ or } c < d \text{ and } e < f$

100 : if a < b goto 103	107 : t <sub>2</sub> := 1
101 : t <sub>1</sub> := 0	108 : if e < f goto 111
102 : goto 104	109 : t <sub>3</sub> := 0
103 : t <sub>1</sub> := 1	110 : goto 112
104 : if c < d goto 107	111 : t <sub>3</sub> := 1
105 : t <sub>2</sub> := 0	112 : t <sub>4</sub> := t <sub>2</sub> and t <sub>3</sub>
106 : goto 108	113 : t <sub>5</sub> := t <sub>1</sub> or t <sub>4</sub>



## Flow-of-Control Statements

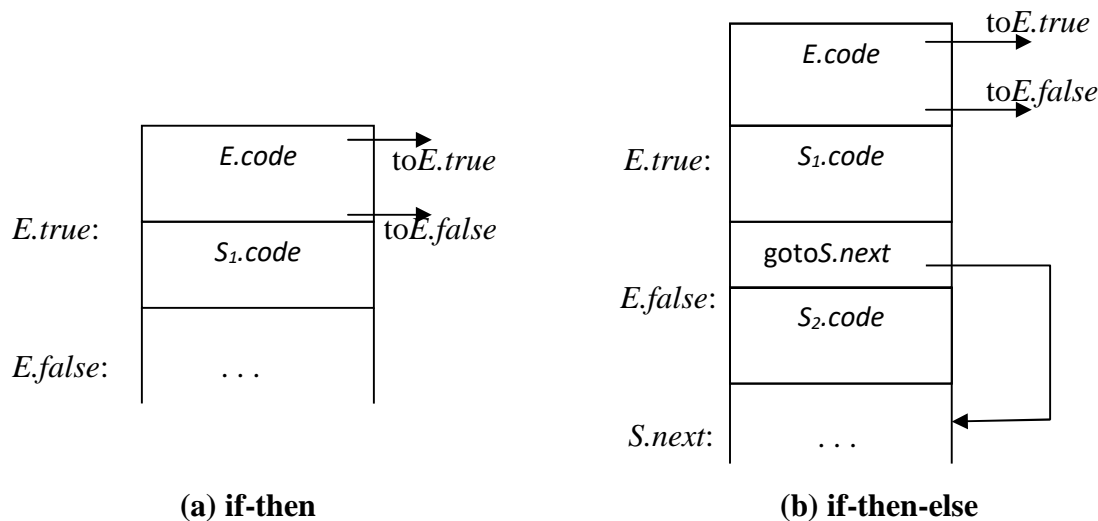
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

$$\begin{aligned}
 S \rightarrow & \text{if } E \text{ then } S_1 \\
 & | \text{if } E \text{ then } S_1 \text{ else } S_2 \\
 & | \text{while } E \text{ do } S_1
 \end{aligned}$$

In each of these productions,  $E$  is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- $E.true$  is the label to which control flows if  $E$  is true, and  $E.false$  is the label to which control flows if  $E$  is false.
- The semantic rules for translating a flow-of-control statement  $S$  allow control to flow from the translation  $S.code$  to the three-address instruction immediately following  $S.code$ .
- $S.next$  is a label that is attached to the first three-address instruction to be executed after the code for  $S$ .

### Code for if-then , if-then-else, and while-do statements



### Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{if } E \text{ then } S_1$	$E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel \text{gen}(E.true \text{ ':' }) \parallel S_1.code$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.true := \text{newlabel};$ $E.false := \text{newlabel};$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel \text{gen}(E.true \text{ ':' }) \parallel S_1.code \parallel$ $\text{gen}(\text{'goto' } S.next) \parallel$ $\text{gen}(E.false \text{ ':' }) \parallel S_2.code$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := \text{newlabel};$ $E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := \text{gen}(S.begin \text{ ':' }) \parallel E.code \parallel$ $\text{gen}(E.true \text{ ':' }) \parallel S_1.code \parallel$ $\text{gen}(\text{'goto' } S.begin)$

### Control-Flow Translation of Boolean Expressions:

#### Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := \text{newlabel};$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel \text{gen}(E_1.false \text{ ':' }) \parallel E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E.true := \text{newlabel};$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel \text{gen}(E_1.true \text{ ':' }) \parallel E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow ( E_1 )$	$E_1.true := E.true;$

	$E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow id_1 \text{ relop } id_2$	$E.code := gen('if' id_1.place \text{ relop } id_2.place$ $\quad 'goto' E.true) // gen('goto' E.false)$
$E \rightarrow true$	$E.code := gen('goto' E.true)$
$E \rightarrow false$	$E.code := gen('goto' E.false)$

## CASE STATEMENTS

The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

### Switch-statement syntax

**switch***expression*

**begin**

**case***value:statement*

**case***value:statement*

...

**case***value:statement*

**default** :*statement*

**end**

There is a selector expression, which is to be evaluated, followed by constant values that the expression might take, including a default “value” which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases is large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say  $i_{min}$  to  $i_{max}$ , and the number of different values is a reasonable fraction of  $i_{max} - i_{min}$ , then we can construct an array of labels, with the label of the statement for value  $j$  in the entry of the table with offset  $j - i_{min}$  and the label for the default in entries not filled otherwise. To perform switch,

evaluate the expression to obtain the value of  $j$ , check the value is within range and transfer to the table entry at offset  $j - i_{\min}$ .

### Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```

switch  $E$ 
begin
    case  $V_1 : S_1$ 
    case  $V_2 : S_2$ 
        . . .
    case  $V_{n-1} : S_{n-1}$ 
    default :  $S_n$ 
end

```

This case statement is translated into intermediate code that has the following form :

#### Translation of a case statement

```

                                code to evaluate  $E$  into  $t$ 
                                goto test
L1 :                          code for  $S_1$ 
                                goto next
L2 :                          code for  $S_2$ 
                                goto next
                                . . .
Ln-1 :                        code for  $S_{n-1}$ 
                                goto next
Ln :                          code for  $S_n$ 
                                goto next
test :                          if  $t = V_1$  goto L1
                                if  $t = V_2$  goto L2
                                . . .
                                if  $t = V_{n-1}$  goto Ln-1
                                goto Ln
next :

```

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next**, and a new temporary **t** are generated.
- As expression  $E$  is parsed, the code to evaluate  $E$  into  $t$  is generated. After processing  $E$ , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label  $L_i$  is created and entered into the symbol table. A pointer to this symbol-table entry and the value  $V_i$  of case constant are placed on a stack (used only to store cases).



- Each statement  $\text{case } V_i : S_i$  is processed by emitting the newly created label  $L_i$ , followed by the code for  $S_i$ , followed by the jump **goto next**.
- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

```

case  $V_1$   $L_1$ 
case  $V_2$   $L_2$ 
...
case  $V_{n-1}$   $L_{n-1}$ 
case  $t$   $L_n$ 
label next

```

where  $t$  is the name holding the value of the selector expression  $E$ , and  $L_n$  is the label for the default statement.

## BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels **backpatching**.

To manipulate lists of labels, we use three functions :

1. *makelist( $i$ )* creates a new list containing only  $i$ , an index into the array of quadruples; *makelist* returns a pointer to the list it has made.
2. *merge( $p_1, p_2$ )* concatenates the lists pointed to by  $p_1$  and  $p_2$ , and returns a pointer to the concatenated list.
3. *backpatch( $p, i$ )* inserts  $i$  as the target label for each of the statements on the list pointed to by  $p$ .

## Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

- (1)  $S \rightarrow \text{if } E \text{ then } S$
- (2)  $\quad \quad \quad | \text{if } E \text{ then } S \text{ else } S$
- (3)  $\quad \quad \quad | \text{while } E \text{ do } S$
- (4)  $| \text{begin } L \text{ end}$
- (5)  $\quad \quad \quad | A$
- (6)  $L \rightarrow L ; S$
- (7)  $\quad \quad \quad | S$

Here  $S$  denotes a statement,  $L$  a statement list,  $A$  an assignment statement, and  $E$  a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

## Scheme to implement the Translation:

The nonterminal  $E$  has two attributes  $E.truelist$  and  $E.falselist$ .  $L$  and  $S$  also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes  $L.nextlist$  and  $S.nextlist$ .  $S.nextlist$  is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement  $S$  in execution order, and  $L.nextlist$  is defined similarly.

The semantic rules for the revised grammar are as follows:

- (1)  $S \rightarrow \text{if } E \text{ then } M_1 S_1 \text{ Nelse } M_2 S_2$   
 $\quad \quad \quad \{ \text{backpatch}(E.truelist, M_1.quad);$   
 $\quad \quad \quad \text{backpatch}(E.falselist, M_2.quad);$   
 $\quad \quad \quad S.nextlist := \text{merge}(S_1.nextlist, \text{merge}(N.nextlist, S_2.nextlist)) \}$

We backpatch the jumps when  $E$  is true to the quadruple  $M_1.quad$ , which is the beginning of the code for  $S_1$ . Similarly, we backpatch jumps when  $E$  is false to go to the beginning of the code for  $S_2$ . The list  $S.nextlist$  includes all jumps out of  $S_1$  and  $S_2$ , as well as the jump generated by  $N$ .

- (2)  $N \rightarrow \epsilon \{ N.nextlist := \text{makelist}(\text{nextquad});$   
 $\quad \quad \quad \text{emit}(\text{'goto\_'}) \}$
- (3)  $M \rightarrow \epsilon \{ M.quad := \text{nextquad} \}$
- (4)  $S \rightarrow \text{if } E \text{ then } M S_1$   $\quad \quad \quad \{ \text{backpatch}(E.truelist, M.quad);$   
 $\quad \quad \quad S.nextlist := \text{merge}(E.falselist, S_1.nextlist) \}$
- (5)  $S \rightarrow \text{while } M_1 \text{ Edo } M_2 S_1$   $\quad \quad \quad \{ \text{backpatch}(S_1.nextlist, M_1.quad);$   
 $\quad \quad \quad \text{backpatch}(E.truelist, M_2.quad);$   
 $\quad \quad \quad S.nextlist := E.falselist$   
 $\quad \quad \quad \text{emit}(\text{'goto' } M_1.quad) \}$
- (6)  $S \rightarrow \text{begin } L \text{ end} \{ S.nextlist := L.nextlist \}$

(7)  $S \rightarrow A\{S.nextlist: = \mathbf{nil}\}$

The assignment  $S.nextlist: = \mathbf{nil}$  initializes  $S.nextlist$  to an empty list.

(8)  $L \rightarrow LI; M\ S\{\text{backpatch}(L_{1}.nextlist, M.quad);$   
 $L.nextlist: = S.nextlist\}$

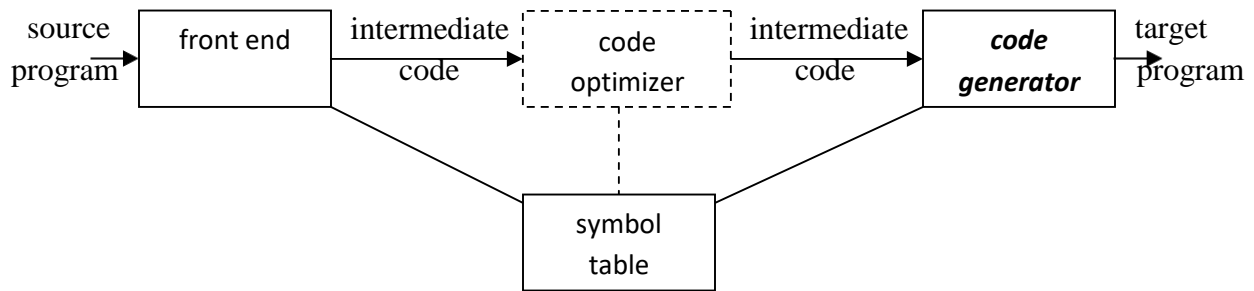
The statement following  $L_1$  in order of execution is the beginning of  $S$ . Thus the  $L_1.nextlist$  list is backpatched to the beginning of the code for  $S$ , which is given by  $M.quad$ .

(9)  $L \rightarrow S\{L.nextlist: = S.nextlist\}$

## UNIT---5---CODE GENERATION

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

### Position of code generator



### ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

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

#### 1. Input to code generator:

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

#### 2. Target program:

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

- b. Relocatable machine language
  - It allows subprograms to be compiled separately.
- c. Assembly language
  - Code generation is made easier.

### 3. Memory management:

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

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

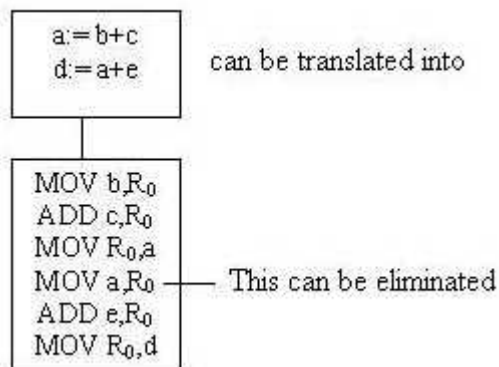
For example,

*j:goto* generates jump instruction as follows :

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

### 4. Instruction selection:

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



### 5. Register allocation

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

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

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

D    x, y

where, x – dividend even register in even/odd register pair

y – divisor

even register holds the remainder

odd register holds the quotient

## 6. Evaluation order

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

## TARGET MACHINE

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

*op    source, destination*

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

- It has the following op-codes :

MOV (move *source* to *destination*)

ADD (add *source* to *destination*)

SUB (subtract *source* from *destination*)

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

### Address modes with their assembly-language forms

MODE	FORM	ADDRESS	ADDED COST
<i>absolute</i> M		M	1
<i>register</i> R		R	0
<i>indexed</i>	$c(R)c + contents(R)$		1
<i>indirect register</i> *	$contents(R)$	0	
<i>indirect indexed</i> *	$c(R)contents(c + contents(R))$		1
<i>literal</i> # <i>c</i>	$c1$		

- For example :  $\text{MOV } R_0, M$  stores contents of Register  $R_0$  into memory location  $M$  ;  
 $\text{MOV } 4(R_0), M$  stores the value  $\text{contents}(4 + \text{contents}(R_0))$  into  $M$ .

### Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.  
 For example :  $\text{MOV } R_0, R_1$  copies the contents of register  $R_0$  into  $R_1$ . It has cost one, since it occupies only one word of memory.
- The three-address statement  $a := b + c$  can be implemented by many different instruction sequences :

i)  $\text{MOV } b, R_0$

$\text{ADD } c, R_0$  cost = 6

$\text{MOV } R_0, a$

ii)  $\text{MOV } b, a$

$\text{ADD } c, a$  cost = 6

iii) Assuming  $R_0, R_1$  and  $R_2$  contain the addresses of  $a, b$ , and  $c$  :

$\text{MOV } *R_1, *R_0$

$\text{ADD } *R_2, *R_0$  cost = 2

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

### RUN-TIME STORAGE MANAGEMENT

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

## **Static allocation**

### **Implementation of call statement:**

The codes needed to implement static allocation are as follows:

**MOV***#here+ 20, callee.static\_area* /\*It saves return address\*/

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

where,

*callee.static\_area*– Address of the activation record

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

*#here+ 20* – Literal return address which is the address of the instruction following GOTO.

### **Implementation of return statement:**

A return from procedure *callee* is implemented by :

**GOTO***\*callee.static\_area*

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

### **Implementation of action statement:**

The instruction ACTION is used to implement action statement.

### **Implementation of halt statement:**

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

## **Stack allocation**

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

The codes needed to implement stack allocation are as follows:

### **Initialization of stack:**

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

Code for the first procedure

**HALT** /\* terminate execution \*/

### **Implementation of Call statement:**

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

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

**GOTO***callee.code\_area*



where,

*caller.recordsize* – size of the activation record

*#here* + 16 – address of the instruction following the **GOTO**

### Implementation of Return statement:

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

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

## BASIC BLOCKS AND FLOW GRAPHS

### Basic Blocks

- *Abasic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:  
     $t_1 := a * a$   
     $t_2 := a * b$   
     $t_3 := 2 * t_2$   
     $t_4 := t_1 + t_3$   
     $t_5 := b * b$   
     $t_6 := t_4 + t_5$

### Basic Block Construction:

**Algorithm:** Partition into basic blocks

**Input:** A sequence of three-address statements

**Output:** A list of basic blocks with each three-address statement in exactly one block

**Method:**

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
  - a. The first statement is a leader.
  - b. Any statement that is the target of a conditional or unconditional goto is a leader.
  - c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

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

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

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

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

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

## Transformations on Basic Blocks:

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

- Structure-preserving transformations
- Algebraic transformations

### 1. Structure preserving transformations:

#### a) Common subexpression elimination:

$a := b + c$		$a := b + c$
$b := a - d$	$\longrightarrow$	$b := a - d$
$c := b + c$		$c := b + c$
$d := a - d$		$d := b$

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

#### b) Dead-code elimination:

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

#### c) Renaming temporary variables:

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

Such a block is called a *normal-form block*.

#### d) Interchange of statements:

Suppose a block has the following two adjacent statements:

$t1 := b + c$   
 $t2 := x + y$

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

### 2. Algebraic transformations:

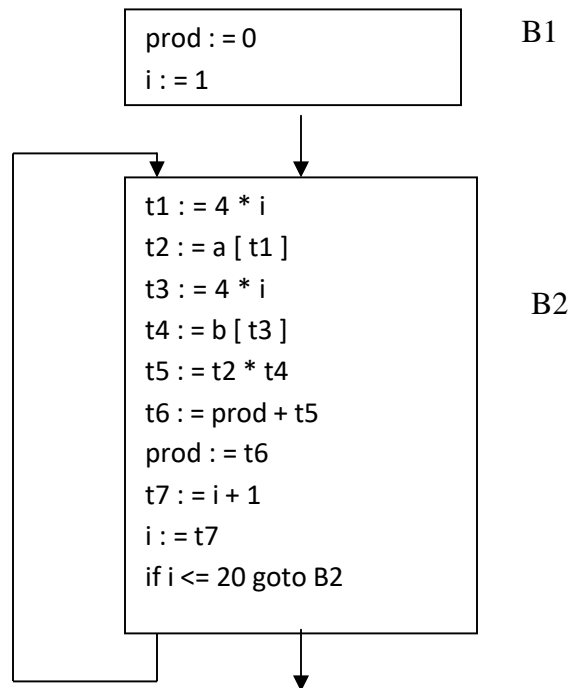
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

- $x := x + 0$  or  $x := x * 1$  can be eliminated from a basic block without changing the set of expressions it computes.
- The exponential statement  $x := y * * 2$  can be replaced by  $x := y * y$ .

## Flow Graphs

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



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

## Loops

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

## NEXT-USE INFORMATION

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

**Input:** Basic block B of three-address statements

**Output:** At each statement  $i: x = y \text{ op } z$ , we attach to  $i$  the liveness and next-uses of  $x$ ,  $y$  and  $z$ .

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

1. Attach to statement  $i$  the information currently found in the symbol table regarding the next-use and liveness of  $x$ ,  $y$  and  $z$ .
2. In the symbol table, set  $x$  to “not live” and “no next use”.
3. In the symbol table, set  $y$  and  $z$  to “live”, and next-uses of  $y$  and  $z$  to  $i$ .

### Symbol Table:

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

### A SIMPLE CODE GENERATOR

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

ADD  $R_j, R_i$                       Cost = 1     // if  $R_i$  contains  $b$  and  $R_j$  contains  $c$

(or)

ADD  $c, R_i$                       Cost = 2     // if  $c$  is in a memory location

(or)

MOV  $c, R_j$                       Cost = 3     // move  $c$  from memory to  $R_j$  and add

ADD  $R_j, R_i$

### Register and Address Descriptors:

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

### A code-generation algorithm:

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

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

### Generating Code for Assignment Statements:

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

$t := a - b$

$u := a - c$

$v := t + u$

$d := v + u$

with  $d$  live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
$t := a - b$	MOV a, 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

The table shows the code sequences generated for the indexed assignment statements

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

Statements	Code Generated	Cost
a := b[i]	MOV b(R <sub>i</sub> ), R	2
a[i] := b	MOV b, a(R <sub>i</sub> )	3

## Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments

**a := \*p** and **\*p := a**

Statements	Code Generated	Cost
a := *p	MOV *R <sub>p</sub> , a	2
*p := a	MOV a, *R <sub>p</sub>	2

## Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z      /* jump to z if condition code is negative */
x := y + z if x < 0 goto z	MOV y, R <sub>0</sub> ADD z, R <sub>0</sub> MOV R <sub>0</sub> , x CJ< z

## THE DAG REPRESENTATION FOR BASIC BLOCKS

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

## Algorithm for construction of DAG

**Input:** A basic block

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

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

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

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

Case (iii)  $x := y$

**Method:**

**Step 1:** If  $y$  is undefined then create  $\text{node}(y)$ .

If  $z$  is undefined, create  $\text{node}(z)$  for case(i).

**Step 2:** For the case(i), create a  $\text{node}(\text{OP})$  whose left child is  $\text{node}(y)$  and right child is

$\text{node}(z)$ . ( Checking for common sub expression). Let  $n$  be this node.

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

For case(iii),  $\text{node } n$  will be  $\text{node}(y)$ .

**Step 3:** Delete  $x$  from the list of identifiers for  $\text{node}(x)$ . Append  $x$  to the list of attached

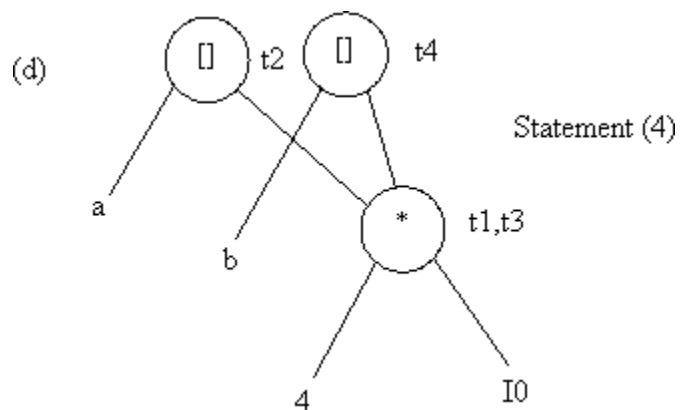
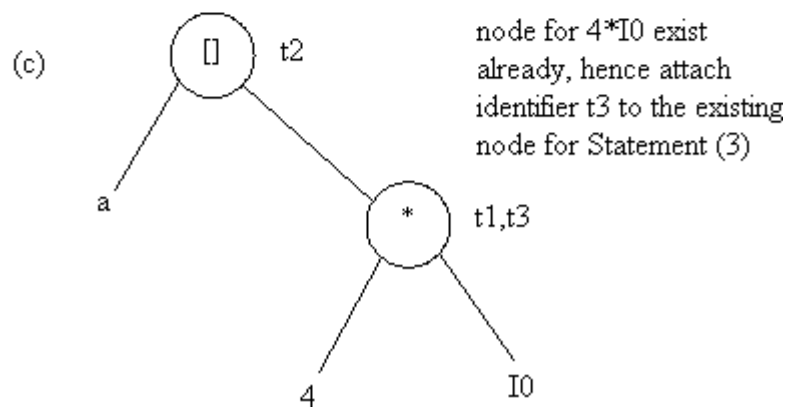
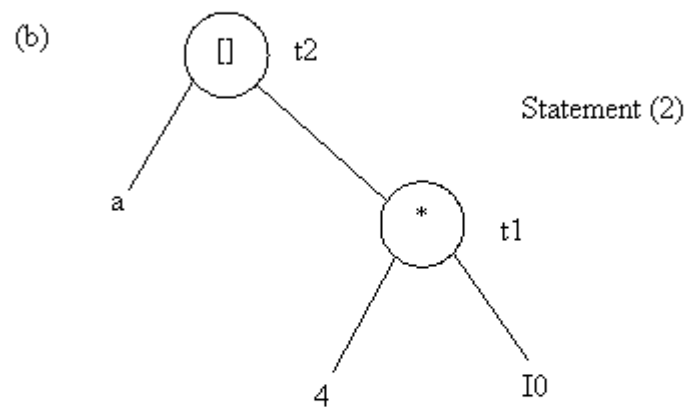
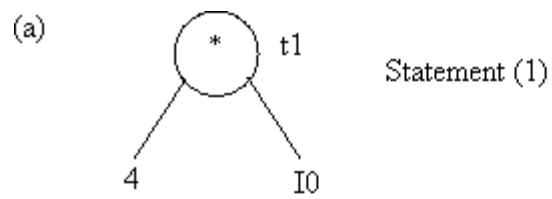
identifiers for the node  $n$  found in step 2 and set  $\text{node}(x)$  to  $n$ .

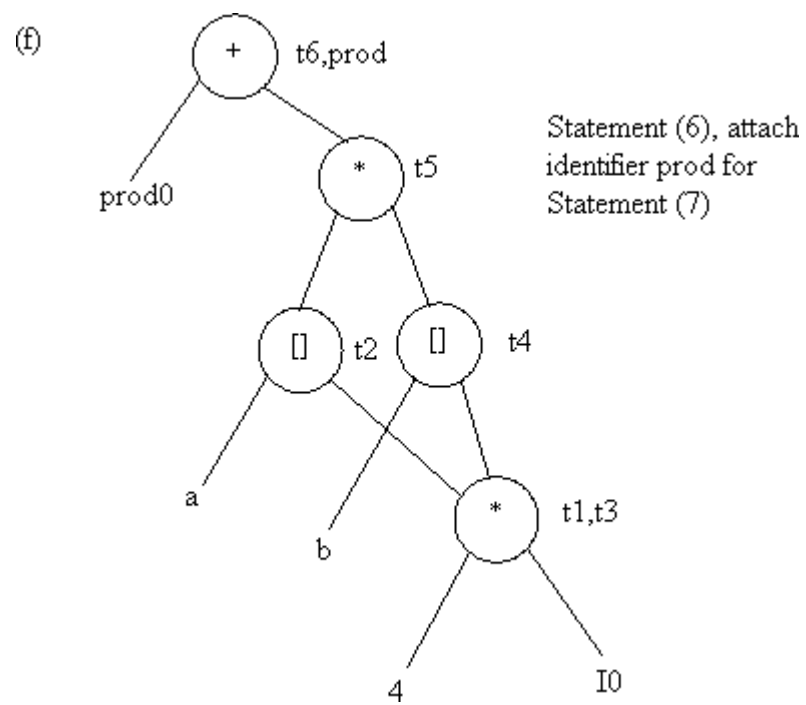
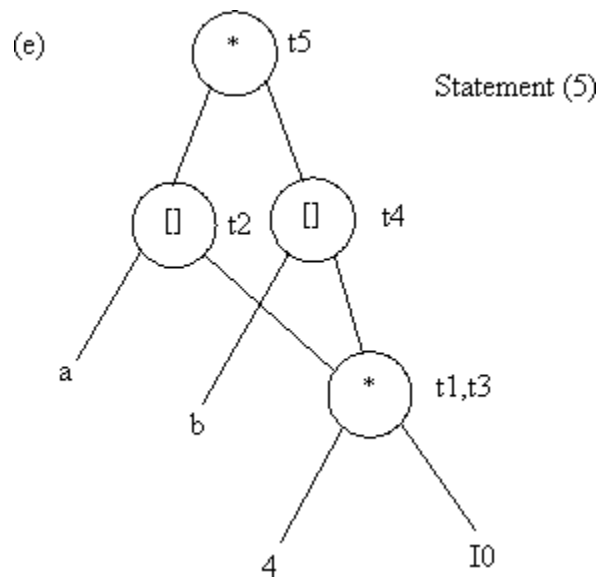
**Example:** Consider the block of three- address statements:

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

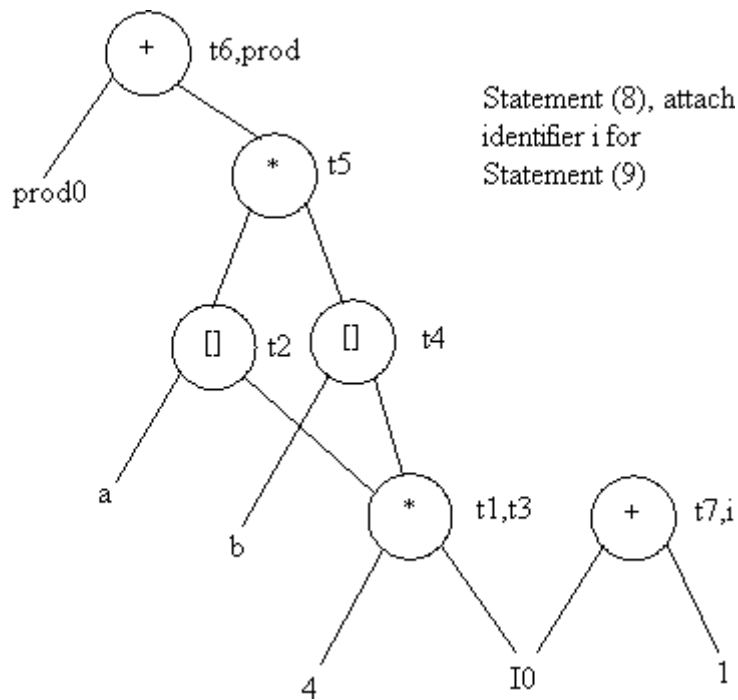


## Stages in DAG Construction

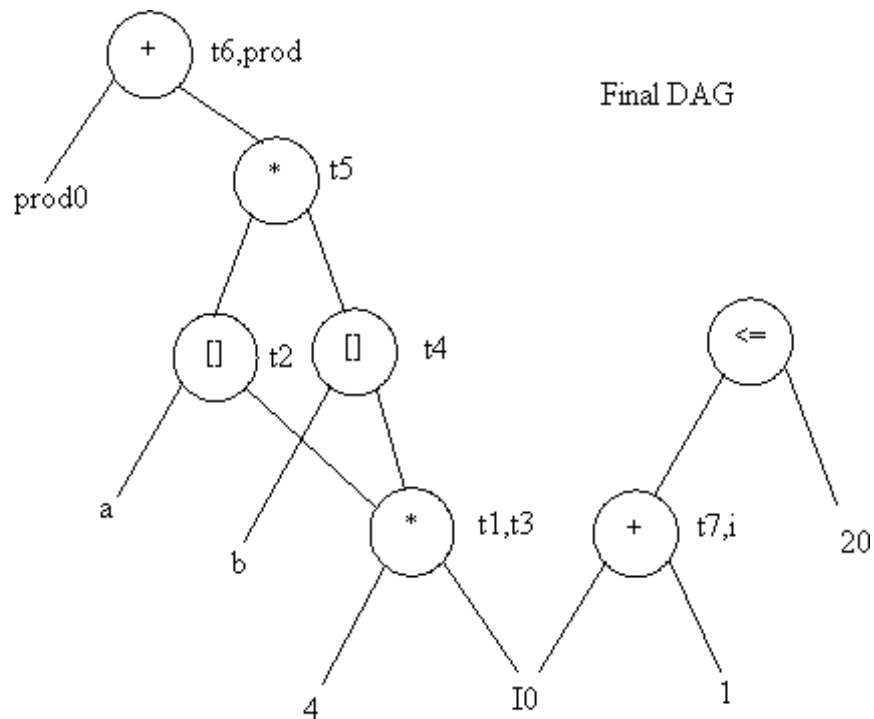




(g)



(h)



### Application of DAGs:

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

## GENERATING CODE FROM DAGs

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

### Rearranging the order

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

For example, consider the following basic block:

```
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 t<sub>1</sub> occurs immediately before t<sub>4</sub>.

```
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 R<sub>0</sub> , t<sub>1</sub>** and **MOV t<sub>1</sub> , R<sub>1</sub>** have been saved.

## A Heuristic ordering for Dags

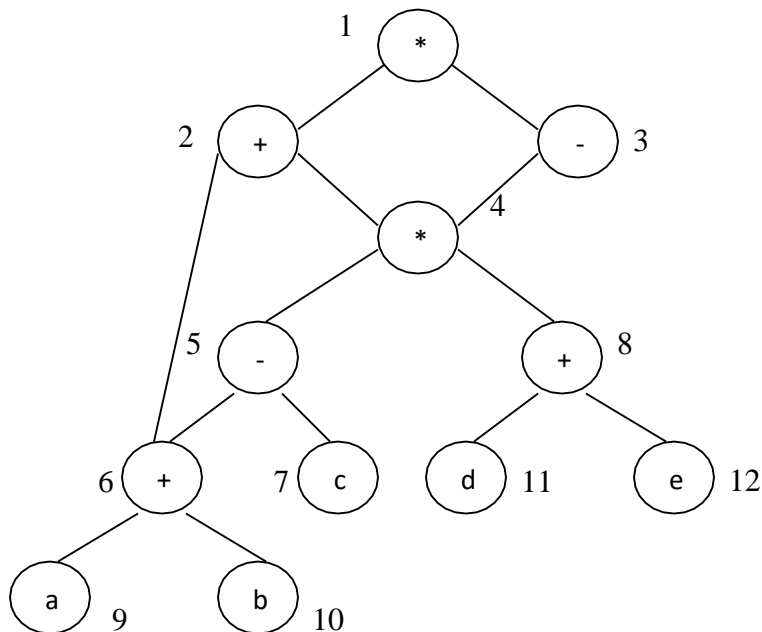
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

### Algorithm:

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

**Example:** Consider the DAG shown below:



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

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

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

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

**Code sequence:** $t_8 := d + e$  $t_6 := a + b$  $t_5 := t_6 - c$  $t_4 := t_5 * t_8$  $t_3 := t_4 - e$  $t_2 := t_6 + t_4$  $t_1 := t_2 * t_3$ 

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

## OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

### Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

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

#### ➤ Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

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

The 2<sup>nd</sup> and 4<sup>th</sup> statements compute the same expression: b+c and a-d

Basic block can be transformed to

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

### ➤ **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

### ➤ **Renaming of temporary variables:**

- A statement  $t := b + c$  where  $t$  is a temporary name can be changed to  $u := b + c$  where  $u$  is another temporary name, and change all uses of  $t$  to  $u$ .
- In this we can transform a basic block to its equivalent block called normal-form block.

### ➤ **Interchange of two independent adjacent statements:**

- Two statements

$t_1 := b + c$

$t_2 := x + y$

can be interchanged or reordered in its computation in the basic block when value of  $t_1$  does not affect the value of  $t_2$ .

### **Algebraic Transformations:**

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression  $2 * 3.14$  would be replaced by  $6.28$ .
- The relational operators  $<=$ ,  $>=$ ,  $<$ ,  $>$ ,  $+$  and  $=$  sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

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

the following intermediate code may be generated:

$a := b + c$   
 $t := c + d$   
 $e := t + b$

- Example:

$x := x + 0$  can be removed

$x := y ** 2$  can be replaced by a cheaper statement  $x := y * y$



- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate  $x*y-x*z$  as  $x*(y-z)$  but it may not evaluate  $a+(b-c)$  as  $(a+b)-c$ .

## LOOPS IN FLOW GRAPH

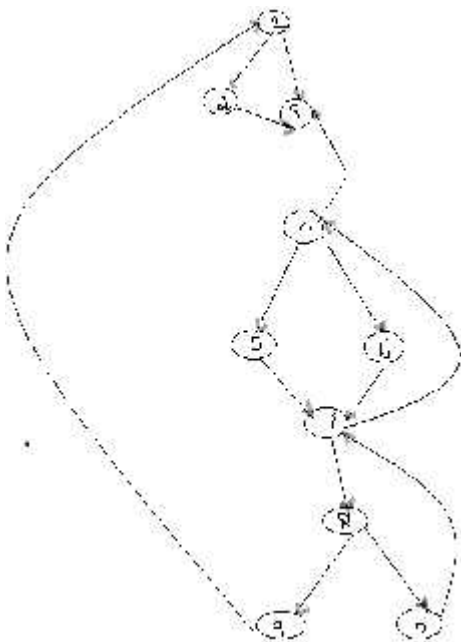
A graph representation of three-address statements, called **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

### Dominators:

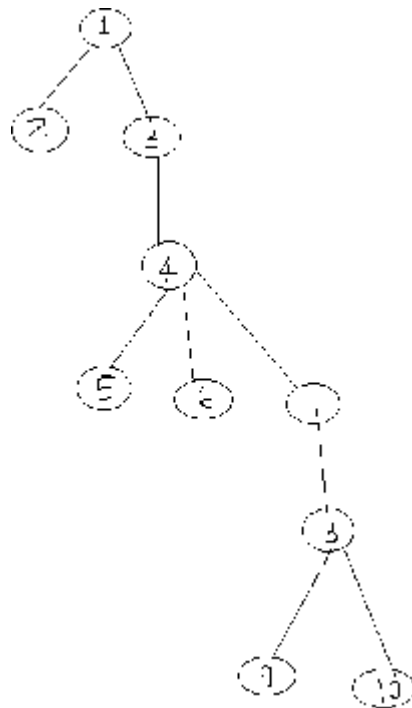
In a flow graph, a node  $d$  dominates node  $n$ , if every path from initial node of the flow graph to  $n$  goes through  $d$ . This will be denoted by  $d \text{ dom } n$ . Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

- \*In the flow graph below,
- \*Initial node,node1 dominates every node.
- \*node 2 dominates itself
- \*node 3 dominates all but 1 and 2.
- \*node 4 dominates all but 1,2 and 3.
- \*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
- \*node 7 dominates 7,8 ,9 and 10.
- \*node 8 dominates 8,9 and 10.
- \*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node  $d$  dominates only its descendants in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of  $n$  on any path from the initial node to  $n$ .
- In terms of the dom relation, the immediate dominator  $m$  has the property is  $d \neq n$  and  $d \text{ dom } n$ , then  $d \text{ dom } m$ .



$$D(1)=\{1\}$$

$$D(2)=\{1,2\}$$

$$D(3)=\{1,3\}$$

$$D(4)=\{1,3,4\}$$

$$D(5)=\{1,3,4,5\}$$

$$D(6)=\{1,3,4,6\}$$

$$D(7)=\{1,3,4,7\}$$

$$D(8)=\{1,3,4,7,8\}$$

$$D(9)=\{1,3,4,7,8,9\}$$

$$D(10)=\{1,3,4,7,8,10\}$$

## Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
  - The properties of loops are
    - ✓ A loop must have a single entry point, called the header. This entry point dominates all nodes in the loop, or it would not be the sole entry to the loop.
    - ✓ There must be at least one way to iterate the loop (i.e.) at least one path back to the header.
  - One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If  $a \rightarrow b$  is an edge,  $b$  is the head and  $a$  is the tail. These types of edges are called as back edges.
- ✓ Example:

In the above graph,

$7 \rightarrow 4$      4 DOM 7

$10 \rightarrow 7$      7 DOM 10

$4 \rightarrow 3$

$8 \rightarrow 3$

$9 \rightarrow 1$

- The above edges will form loop in flow graph.
- Given a back edge  $n \rightarrow d$ , we define the natural loop of the edge to be  $d$  plus the set of nodes that can reach  $n$  without going through  $d$ . Node  $d$  is the header of the loop.

**Algorithm:** Constructing the natural loop of a back edge.

**Input:** A flow graph  $G$  and a back edge  $n \rightarrow d$ .

**Output:** The set loop consisting of all nodes in the natural loop  $n \rightarrow d$ .

**Method:** Beginning with node  $n$ , we consider each node  $m \neq d$  that we know is in loop, to make sure that  $m$ 's predecessors are also placed in loop. Each node in loop, except for  $d$ , is placed once on stack, so its predecessors will be examined. Note that because  $d$  is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach  $n$  without going through  $d$ .

**Procedure** insert( $m$ );  
**if**  $m$  is not in *loop* **then begin**  
     $loop := loop \cup \{m\}$ ;  
    push  $m$  on *stack*  
**end;**

$stack := \text{empty};$

```

loop: = {d};
insert(n);
while stack is not empty do begin
    pop m, the first element of stack, off stack;
    foreach predecessor p of m do insert(p)
end

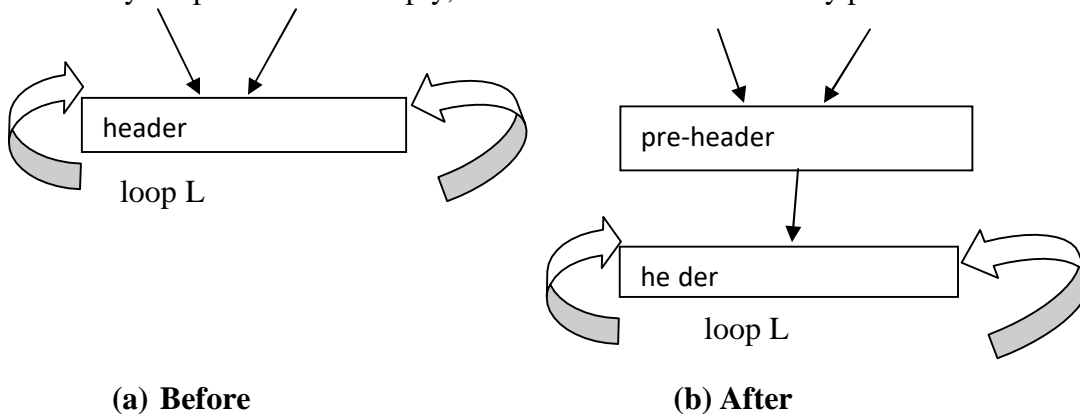
```

### Inner loop:

- If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjoint or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

### Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheader.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



### Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

•**Definition:**

A flow graph  $G$  is reducible if and only if we can partition the edges into two disjoint groups, *forward edges* and *back edges*, with the following properties.

- ✓The forward edges from an acyclic graph in which every node can be reached from initial node of  $G$ .
- ✓The back edges consist only of edges where heads dominate their tails.
- ✓Example: The above flow graph is reducible.
- If we know the relation DOM for a flow graph, we can find and remove all the back edges.
- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges  $4 \rightarrow 3$ ,  $7 \rightarrow 4$ ,  $8 \rightarrow 3$ ,  $9 \rightarrow 1$  and  $10 \rightarrow 7$  whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

## PEEPHOLE OPTIMIZATION

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

## Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R<sub>0</sub>,a

(2) MOV a,R<sub>0</sub>

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R<sub>0</sub>. If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

## Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions.

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

```
#define debug 0

....

If ( debug ) {

    Print debugging information

}
```

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

```
If debug =1 goto L2

goto L2

L1: print debugging information

L2: ..... (a)
```

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

```
If debug≠1 goto L2

Print debugging information

L2: ..... (b)
```

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

If debug≠0 goto L2

Print debugging information

L2: ..... (c)

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

### **Flows-Of-Control Optimizations:**

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

goto L1

....

L1: gotoL2

by the sequence

goto L2

....

L1: goto L2

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

if a < b goto L1

....

L1: goto L2

can be replaced by

If a < b goto L2

....

L1: goto L2

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

goto L1

.....

L1: if a < b goto L2

L3: ..... (1)

- May be replaced by

If a < b goto L2

goto L3

.....

L3: ..... (2)

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

### **Algebraic Simplification:**

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

$x := x + 0$

Or

$x := x * 1$

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

### **Reduction in Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example,  $x^2$  is invariably cheaper to implement as  $x * x$  than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X * X$

### **Use of Machine Idioms:**

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $i := i + 1$ .



$i:=i+1 \rightarrow i++$

$i:=i-1 \rightarrow i--$

## INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation, compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions”, such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

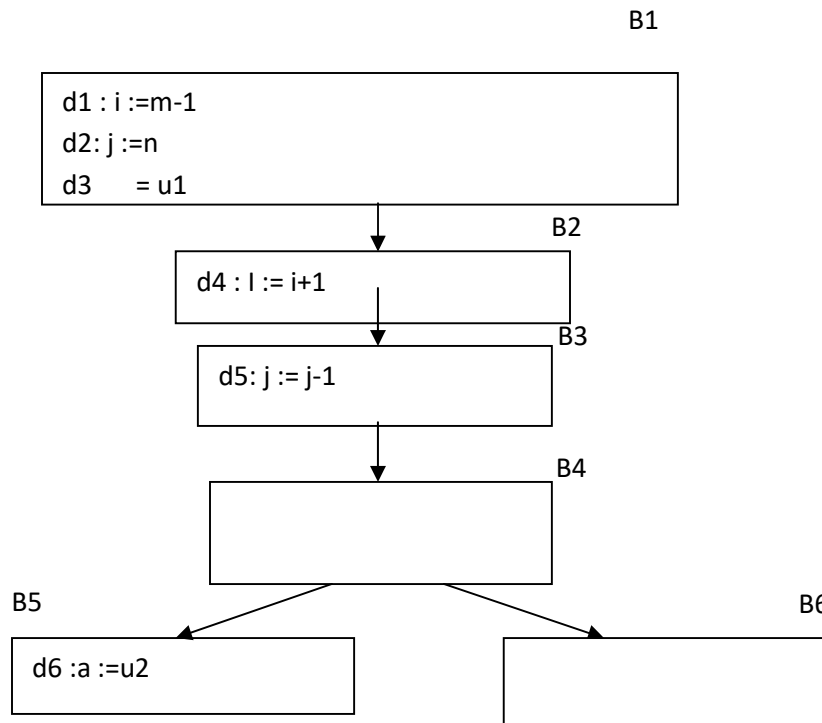
$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

This equation can be read as “ the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.
- ✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining  $\text{out}[s]$  in terms of  $\text{in}[s]$ , we need to proceed backwards and define  $\text{in}[s]$  in terms of  $\text{out}[s]$ .
- ✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write  $\text{out}[s]$  we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- ✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

### Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from  $p_1$  to  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that for each  $i$  between 1 and  $n-1$ , either
  - ✓  $P_i$  is the point immediately preceding a statement and  $p_{i+1}$  is the point immediately following that statement in the same block, or
  - ✓  $P_i$  is the end of some block and  $p_{i+1}$  is the beginning of a successor block.

### Reaching definitions:

- A definition of variable  $x$  is a statement that assigns, or may assign, a value to  $x$ . The most common forms of definition are assignments to  $x$  and statements that read a value from an i/o device and store it in  $x$ .
- These statements certainly define a value for  $x$ , and they are referred to as **unambiguous** definitions of  $x$ . There are certain kinds of statements that may define a value for  $x$ ; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of  $x$  are:
  - ✓ A call of a procedure with  $x$  as a parameter or a procedure that can access  $x$  because  $x$  is in the scope of the procedure.
  - ✓ An assignment through a pointer that could refer to  $x$ . For example, the assignment  $*q = y$  is a definition of  $x$  if it is possible that  $q$  points to  $x$ . we must assume that an assignment through a pointer is a definition of every variable.
- We say a definition  $d$  reaches a point  $p$  if there is a path from the point immediately following  $d$  to  $p$ , such that  $d$  is not “killed” along that path. Thus a point can be reached

by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.