

Acknowledgements

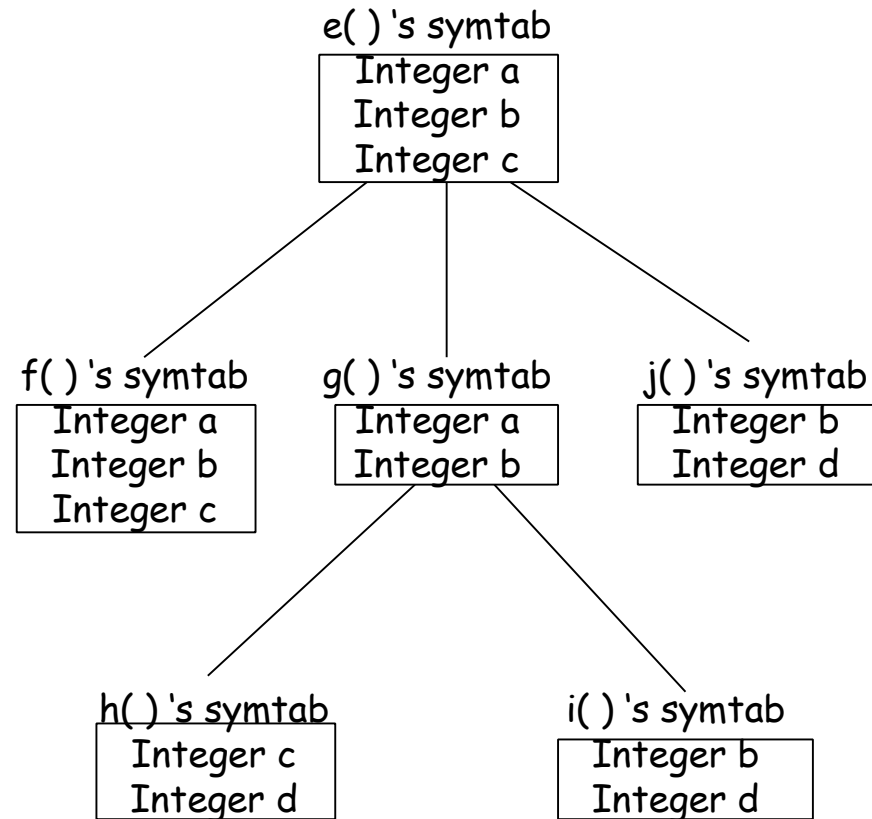
The slides for this lecture are a modified versions of the offering by **Prof. Sanjeev K Aggarwal**

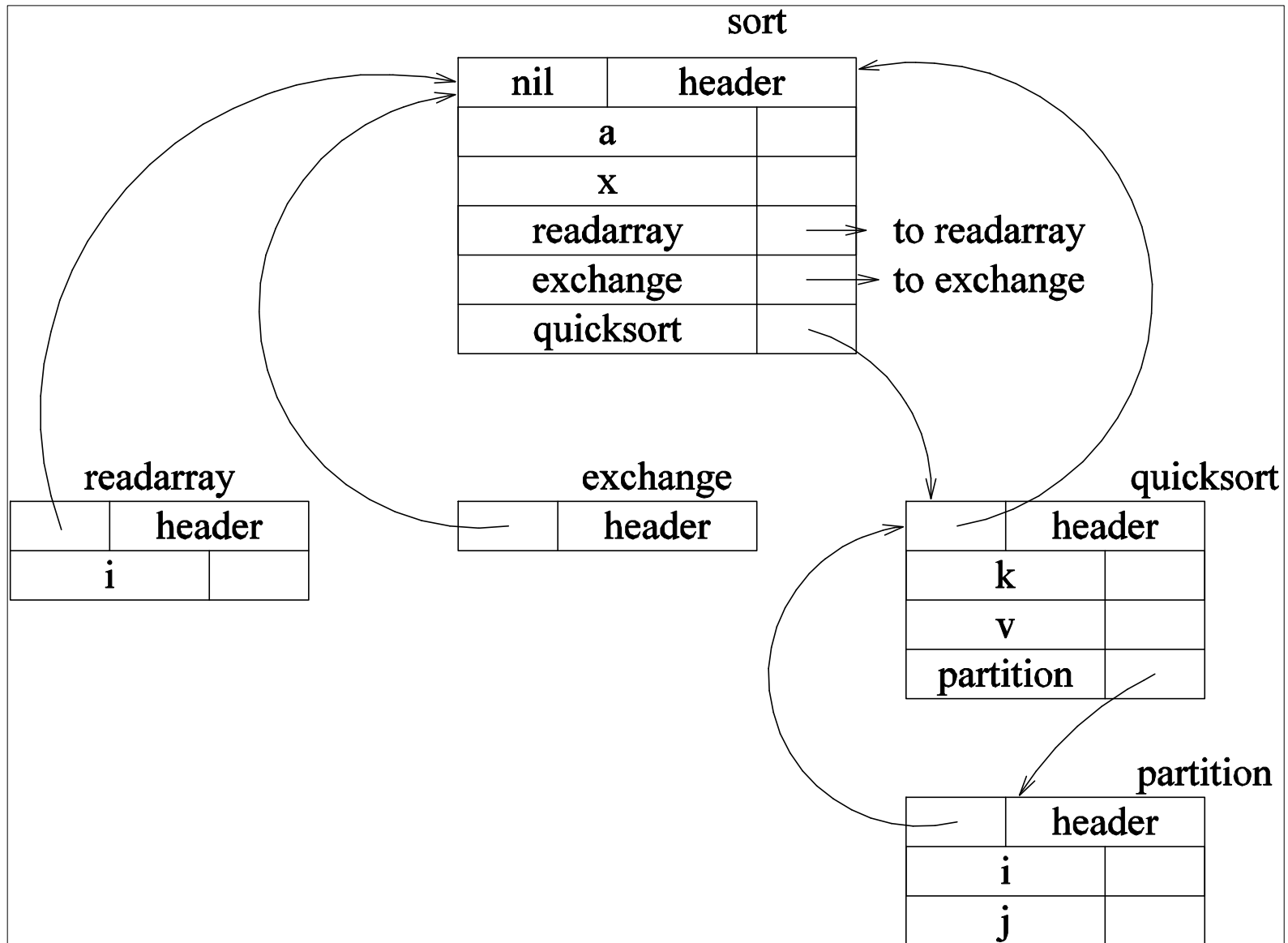
Three address code

- It is a sequence of statements of the general form $X := Y \text{ op } Z$ where
 - X , Y or Z are names, constants or compiler generated temporaries
 - op stands for any operator such as a fixed- or floating-point arithmetic operator, or a logical operator

Global Symbol table structure

- scope and visibility rules determine the structure of global symbol table
- for Algol class of languages scoping rules structure the symbol table as tree of local tables
 - global scope as root
 - tables for nested scope as children of the table for the scope they are nested in





Declarations

For each name create symbol table entry with information like type and relative address

$P \rightarrow \{\text{offset}=0\} D$

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T$ $\text{enter}(\text{id.name}, T.\text{type}, \text{offset});$
 $\text{offset} = \text{offset} + T.\text{width}$

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

$T \rightarrow \text{real}$ $T.\text{type} = \text{real}; T.\text{width} = 8$

Declarations ...

$T \rightarrow \text{array [num] of } T_1$

$T.\text{type} = \text{array}(\text{num.val}, T_1.\text{type})$

$T.\text{width} = \text{num.val} \times T_1.\text{width}$

$T \rightarrow \uparrow T_1$

$T.\text{type} = \text{pointer}(T_1.\text{type})$

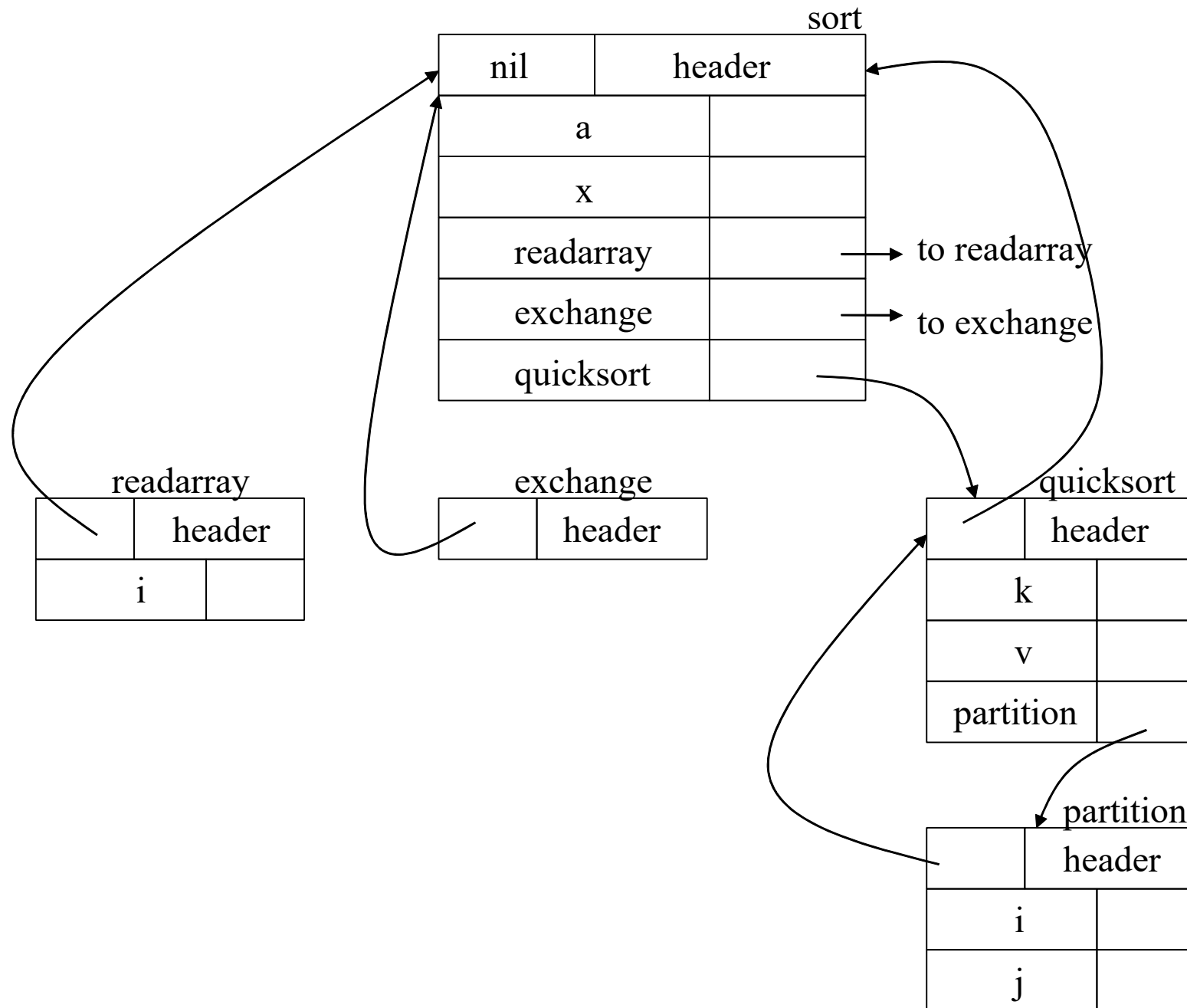
$T.\text{width} = 4$

Keeping track of local information

- when a nested procedure is seen, processing of declaration in enclosing procedure is temporarily suspended
- assume following language
$$P \rightarrow D$$
$$D \rightarrow D ; D \mid id : T \mid \text{proc } id ; D ; S$$
- a new symbol table is created when procedure declaration
$$D \rightarrow \text{proc } id ; D_1 ; S$$
is seen
- entries for D_1 are created in the new symbol table
- the name represented by id is local to the enclosing procedure

Example

```
program sort;  
  var a : array[1..n] of integer;  
      x : integer;  
  procedure readarray;  
    var i : integer;  
    .....  
  procedure exchange(i,j:integer);  
    .....  
  procedure quicksort(m,n : integer);  
    var k,v : integer;  
        function partition(x,y:integer):integer;  
          var i,j: integer;  
          .....  
        .....  
  .....  
begin{main}  
  .....  
end.
```

Creating symbol table

- `mktable (previous)`
create a new symbol table and return a pointer to the new table. The argument `previous` points to the enclosing procedure
- `enter (table, name, type, offset)`
creates a new entry
- `addwidth (table, width)`
records cumulative width of all the entries in a table
- `enterproc (table, name, newtable)`
creates a new entry for procedure name. `newtable` points to the symbol table of the new procedure

Creating symbol table ...

$P \rightarrow$ {t=mktable(nil);
 push(t,tblptr);
 push(0,offset)}

D

 {addwidth(top(tblptr),top(offset));
 pop(tblptr);
 pop(offset)}

$D \rightarrow D ; D$

Creating symbol table ...

D → proc id;

```
{t = mktable(top(tblptr));  
push(t, tblptr); push(0, offset)}
```

D₁; S

```
{t = top(tblptr);  
addwidth(t, top(offset));  
pop(tblptr); pop(offset);;  
enterproc(top(tblptr), id.name, t)}
```

D → id: T

```
{enter(top(tblptr), id.name, T.type, top(offset));  
top(offset) = top (offset) + T.width}
```

Field names in records

$T \rightarrow \text{record}$

```
{t = mktable(nil);  
  push(t, tblptr); push(0, offset)}
```

D end

```
{T.type = record(top(tblptr));  
  T.width = top(offset);  
  pop(tblptr); pop(offset)}
```

3AC generation: Two Techniques

- gen based
 - Useful for generating 3AC in linked-list
 - Each non-terminal has an attribute "code" to hold the head of the list for that segment
 - Lists are merged to finally create the whole IR
- emit based
 - Useful for 3AC in an array
 - code is emitted (immediately) --- written sequentially in an array, each write advances the current pointer

Arithmetic expression into 3AC (gen-based)

$E \rightarrow id$ $p = \text{lookup}(id.name);$
 if $p \neq \text{nil}$ then $E.place = p$
 else error
 $E.code = ''$

$E \rightarrow -E_1$ $E.place = \text{newtmp}$
 $E.code = E_1.code \parallel$
 $\text{gen}(E.place = - E_1.place)$

$E \rightarrow (E_1)$ $E.place = E_1.place$
 $E.code = E_1.code$

Arithmetic expression into 3AC (gen-based)

$S \rightarrow id = E$

$S.code = E.code \parallel$
 $gen(id.place = E.place)$

$E \rightarrow E_1 + E_2$

$E.place = newtmp()$
 $E.code = E_1.code \parallel E_2.code \parallel$
 $gen(E.place = E_1.place + E_2.place)$

$E \rightarrow E_1 * E_2$

$E.place = newtmp()$
 $E.code = E_1.code \parallel E_2.code \parallel$
 $gen(E.place = E_1.place * E_2.place)$

Example

For $a = b * -c + b * -c$
following code is generated

$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_3 = -c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

Names in the Symbol table (emit-based)

$S \rightarrow id = E$

```
{p = lookup(id.name);  
if p <> nil then emit(p = E.place)  
else error}
```

$E \rightarrow id$

```
{p = lookup(id.name);  
if p <> nil then E.place = p  
else error}
```

Boolean Expressions (emit-based)

- compute logical values
- change the flow of control
- boolean operators are: and or not

$E \rightarrow$ E or E
| E and E
| not E
| (E)
| id relop id
| true
| false

Methods of translation

- Evaluate similar to arithmetic expressions
 - Normally use 1 for true and 0 for false
- implement by flow of control
 - given expression E_1 or E_2
if E_1 evaluates to true
then E_1 or E_2 evaluates to true
without evaluating E_2

Numerical representation

- a or b and not c

$$t_1 = \text{not } c$$

$$t_2 = b \text{ and } t_1$$

$$t_3 = a \text{ or } t_2$$

- relational expression $a < b$ is equivalent to
if $a < b$ then 1 else 0

1. if $a < b$ goto 4.

2. $t = 0$

3. goto 5

4. $t = 1$

5.

Syntax directed translation of boolean expressions

$E \rightarrow E_1 \text{ or } E_2$

$E.\text{place} = \text{newtmp}()$
 $\text{emit}(E.\text{place} '=' E_1.\text{place} 'or' E_2.\text{place})$

$E \rightarrow E_1 \text{ and } E_2$

$E.\text{place} = \text{newtmp}()$
 $\text{emit}(E.\text{place} '=' E_1.\text{place} 'and' E_2.\text{place})$

$E \rightarrow \text{not } E_1$

$E.\text{place} = \text{newtmp}()$
 $\text{emit}(E.\text{place} '=' 'not' E_1.\text{place})$

$E \rightarrow (E_1) \quad E.\text{place} = E_1.\text{place}$

Syntax directed translation of boolean expressions

$E \rightarrow \text{id1 relop id2}$

$E.\text{place} = \text{newtmp}()$

$\text{emit}(\text{if id1.place relop id2.place goto nextstat}+3)$

$\text{emit}(E.\text{place} = 0)$

$\text{emit}(\text{goto nextstat}+2)$

$\text{emit}(E.\text{place} = 1)$

$E \rightarrow \text{true}$

$E.\text{place} = \text{newtmp}$

$\text{emit}(E.\text{place} = '1')$

$E \rightarrow \text{false}$

$E.\text{place} = \text{newtmp}$

$\text{emit}(E.\text{place} = '0')$

Example:

Code for $a < b$ or $c < d$ and $e < f$

100: if $a < b$ goto 103	107: $t_2 = 1$
101: $t_1 = 0$	108: if $e < f$ goto 111
102: goto 104	109: $t_3 = 0$
103: $t_1 = 1$	110: goto 112
104: if $c < d$ goto 107	111: $t_3 = 1$
105: $t_2 = 0$	112: $t_4 = t_2$ and t_3
106: goto 108	113: $t_5 = t_1$ or t_4

Addressing Array Elements

- Arrays are stored in a block of consecutive locations
- assume width of each element is w
- i th element of array A begins in location
 $\text{base} + (i - \text{low}) \times w$
where base is relative address of $A[\text{low}]$
- the expression is equivalent to
 $i \times w + (\text{base} - \text{low} \times w)$
 $\rightarrow i \times w + \text{const}$

2-dimensional array

- storage can be either row major or column major
- in case of 2-D array stored in row major form address of $A[i_1, i_2]$ can be calculated as

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + i_2 - \text{low}_2) \times w$$

where $n_2 = \text{high}_2 - \text{low}_2 + 1$

- rewriting the expression gives

$$\begin{aligned} & ((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w) \\ & \rightarrow ((i_1 \times n_2) + i_2) \times w + \text{constant} \end{aligned}$$

- this can be generalized for $A[i_1, i_2, \dots, i_k]$

Example

- Let A be a 10×20 array
therefore, $n_1 = 10$ and $n_2 = 20$
and assume $w = 4$

- code to access $A[y,z]$ is

$$t_1 = y * 20$$

$$t_1 = t_1 + z$$

$$t_2 = 4 * t_1$$

$$t_3 = A - 84 \quad \{(((low_1 \times n_2) + low_2) \times w) = (1 \times 20 + 1) \times 4 = 84\}$$

$$t_4 = t_2 + t_3$$

$$x = t_4$$

Type conversion within assignments

$E \rightarrow E_1 + E_2$

`E.place = newtmp();`

`if E1.type = integer and E2.type = integer`

`then emit(E.place '=' E1.place 'int+' E2.place);`

`E.type = integer;`

...

similar code if both E₁.type and E₂.type are real

...

`else if E1.type = int and E2.type = real`

`then`

`u = newtmp();`

`emit(u '=' inttoreal E1.place);`

`emit(E.place '=' u 'real+' E2.place);`

`E.type = real;`

...

similar code if E₁.type is real and E₂.type is integer

Example

```
real x, y;  
int i, j;  
x = y + i * j
```

generates code

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

Flow of Control

$S \rightarrow \text{while } E \text{ do } S_1$

$S.\text{begin} :$

$E.\text{code}$

if $E.\text{place} = 0$ goto $S.\text{after}$

$S_1.\text{code}$

goto $S.\text{begin}$

$S.\text{after} :$

$S.\text{begin} = \text{newlabel}$

$S.\text{after} = \text{newlabel}$

$S.\text{code} = \text{gen}(S.\text{begin}:)$

||

$E.\text{code} ||$

$\text{gen}(\text{if } E.\text{place} = 0 \text{ goto } S.\text{after}) ||$

$S_1.\text{code} ||$

$\text{gen}(\text{goto } S.\text{begin}) ||$

$\text{gen}(S.\text{after}:)$

Flow of Control ...

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

$E.\text{code}$

if $E.\text{place} = 0$ goto $S.\text{else}$

$S_1.\text{code}$

goto $S.\text{after}$

$S.\text{else}:$

$S_2.\text{code}$

$S.\text{after}:$

$S.\text{else} = \text{newlabel}$

$S.\text{after} = \text{newlabel}$

$S.\text{code} = E.\text{code} ||$

gen(if $E.\text{place} = 0$ goto $S.\text{else}$) ||

$S_1.\text{code} ||$

gen(goto $S.\text{after}$) ||

gen($S.\text{else} :$) ||

$S_2.\text{code} ||$

gen($S.\text{after} :$)

Short Circuit Evaluation of boolean expressions

- Translate boolean expressions without:
 - generating code for boolean operators
 - evaluating the entire expression
- Flow of control statements
 - $S \rightarrow \text{if } E \text{ then } S_1$
 - | $\text{if } E \text{ then } S_1 \text{ else } S_2$
 - | $\text{while } E \text{ do } S_1$

Control flow translation of boolean expression

$E \rightarrow E_1 \text{ or } E_2$

$E_1.\text{true} = E.\text{true}$
 $E_1.\text{false} = \text{newlabel}$
 $E_2.\text{true} = E.\text{true}$
 $E_2.\text{false} = E.\text{false}$
 $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E_1.\text{false})$
 $\parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.\text{true} = \text{new label}$
 $E_1.\text{false} = E.\text{false}$
 $E_2.\text{true} = E.\text{true}$
 $E_2.\text{false} = E.\text{false}$
 $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E_1.\text{true})$
 $\parallel E_2.\text{code}$

Control flow translation of boolean expression ...

$E \rightarrow \text{not } E_1$ $E_1.\text{true} := E.\text{false}$
 $E_1.\text{false} := E.\text{true}$
 $E.\text{code} := E_1.\text{code}$

$E \rightarrow (E_1)$ $E_1.\text{true} := E.\text{true}$
 $E_1.\text{false} := E.\text{false}$
 $E.\text{code} := E_1.\text{code}$

Control flow translation of boolean expression ...

if E is of the form

$a < b$

then code is of the form

if $a < b$ goto E.true

goto E.false

$E \rightarrow id_1 \text{ relop } id_2$

$E.\text{code} = \text{gen}(\text{if } id_1 \text{ relop } id_2 \text{ goto } E.\text{true}) \parallel$
 $\text{gen}(\text{goto } E.\text{false})$

$E \rightarrow \text{true}$

$E.\text{code} = \text{gen}(\text{goto } E.\text{true})$

$E \rightarrow \text{false}$

$E.\text{code} = \text{gen}(\text{goto } E.\text{false})$

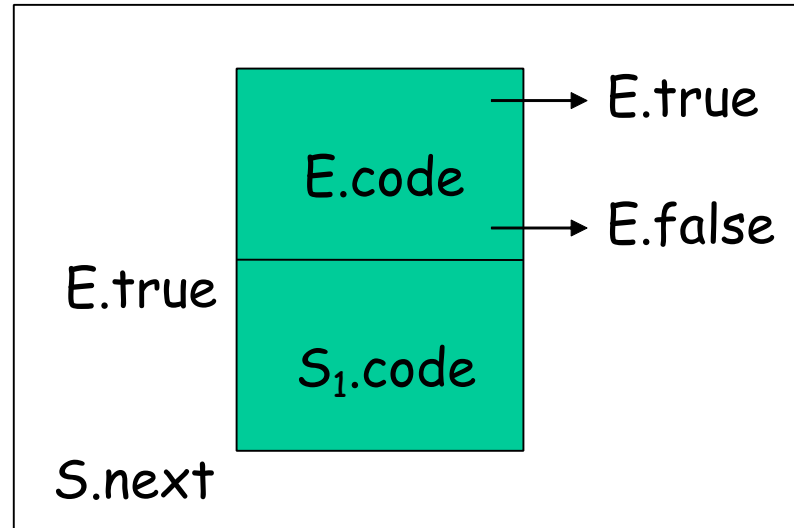
Example

Code for $a < b$ or $c < d$ and $e < f$

```
    if a < b goto Ltrue
    goto L1
L1:  if c < d goto L2
    goto Lfalse
L2:  if e < f goto Ltrue
    goto Lfalse
```

Ltrue:

Lfalse:



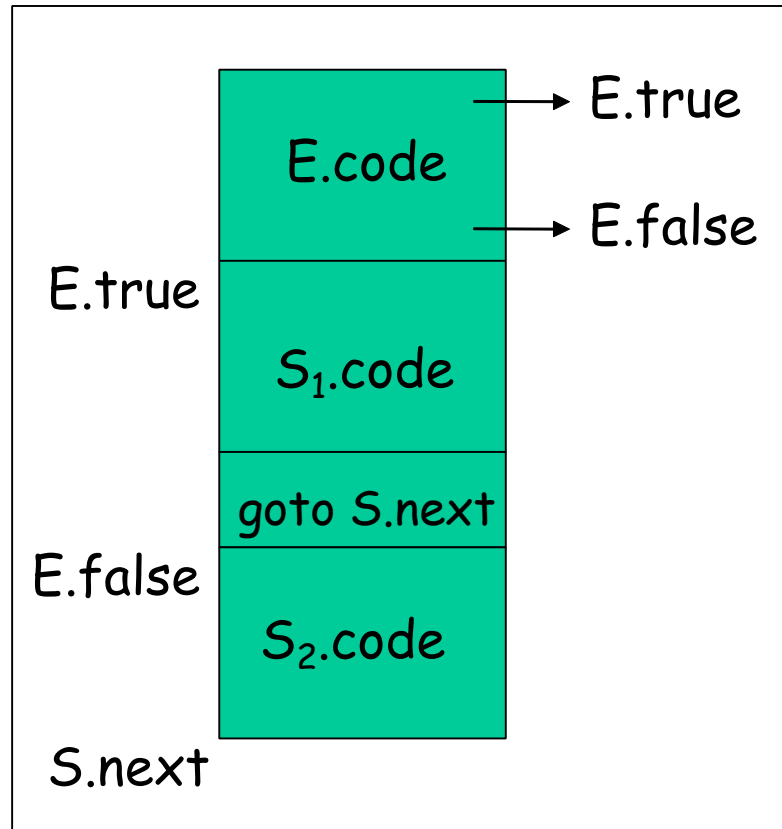
$S \rightarrow \text{if } E \text{ then } S_1$

$E.\text{true} = \text{newlabel}$

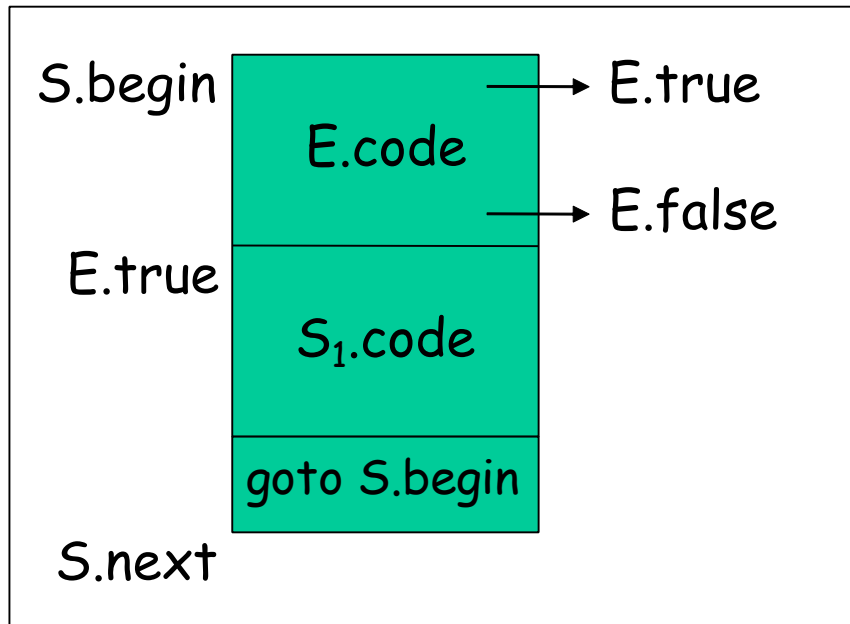
$E.\text{false} = S.\text{next}$

$S_1.\text{next} = S.\text{next}$

$S.\text{code} = E.\text{code} ||$
 $\quad \text{gen}(E.\text{true} ':') ||$
 $\quad S_1.\text{code}$



$S \rightarrow$ if E then S_1 else S_2
E.true = newlabel
E.false = newlabel
 S_1 .next = S.next
 S_2 .next = S.next
S.code = E.code ||
gen(E.true ':') ||
S₁.code ||
gen(goto S.next) ||
gen(E.false ':') ||
S₂.code



```
S → while E do S1  
    S.begin = newlabel  
    E.true = newlabel  
    E.false = S.next  
    S1.next = S.begin  
    S.code = gen(S.begin ':') ||  
             E.code ||  
             gen(E.true ':') ||  
             S1.code ||  
             gen(goto S.begin)
```

Example ...

Code for

```
while a < b do
  if c < d then
    x = y + z
  else
    x = y - z
```

```
L1:   if a < b goto L2
      goto Lnext
L2:   if c < d goto L3
      goto L4
L3:   t1 = Y + Z
      X = t1
      goto L1
L4:   t1 = Y - Z
      X = t1
      goto L1
Lnext:
```


Case Statement

- switch expression

```
begin
  case value: statement
  case value: statement
  ....
  case value: statement
  default: statement
end
```
- evaluate the expression
- find which value in the list of cases is the same as the value of the expression.
 - Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression
- execute the statement associated with the value found

Translation

```
code to evaluate E into t
if t <> V1 goto L1
code for S1
goto next
L1:
if t <> V2 goto L2
code for S2
goto next
L2:
.....
Ln-2: if t <> Vn-1 goto Ln-1
code for Sn-1
goto next
Ln-1: code for Sn
next:
```

```
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: if t = V1 goto L1
if t = V2 goto L2
....
if t = Vn-1 goto Ln-1
goto Ln
next:
```

Efficient for n-way branch

BackPatching

- way to implement boolean expressions and flow of control statements in one pass
- code is generated as quadruples into an array
- labels are indices into this array
- **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

Boolean Expressions

$$\begin{aligned} E \rightarrow & E_1 \text{ or } M E_2 \\ & | E_1 \text{ and } M E_2 \\ & | \text{ not } E_1 \\ & | (E_1) \\ & | id_1 \text{ relop } id_2 \\ & | \text{ true} \\ & | \text{ false} \\ M \rightarrow & \epsilon \end{aligned}$$

- Insert a marker 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 $E.falselist$

Boolean expressions ...

- Consider $E \rightarrow E_1$ and $M E_2$
 - if E_1 is false then E is also false so statements in E_1 .falselist become part of E .falselist
 - if E_1 is true then E_2 must be tested so target of E_1 .truelist is beginning of E_2
 - target is obtained by marker M
 - attribute M .quad records the number of the first statement of E_2 .code

$E \rightarrow E_1 \text{ or } M E_2$
 backpatch(E_1 .falselist, M .quad)
 E .truelist = merge(E_1 .truelist, E_2 .truelist)
 E .falselist = E_2 .falselist

$E \rightarrow E_1 \text{ and } M E_2$
 backpatch(E_1 .truelist, M .quad)
 E .truelist = E_2 .truelist
 E .falselist = merge(E_1 .falselist, E_2 .falselist)

$E \rightarrow \text{not } E_1$
 E .truelist = E_1 .falselist
 E .falselist = E_1 .truelist

$E \rightarrow (E_1)$
 E .truelist = E_1 .truelist
 E .falselist = E_1 .falselist

$E \rightarrow id_1 \text{ relop } id_2$
 $E.\text{truelist} = \text{makelist}(\text{nextquad})$
 $E.\text{falselist} = \text{makelist}(\text{nextquad} + 1)$
 $\text{emit}(\text{if } id_1 \text{ relop } id_2 \text{ goto } \text{---})$
 $\text{emit}(\text{goto } \text{---})$

$E \rightarrow \text{true}$
 $E.\text{truelist} = \text{makelist}(\text{nextquad})$
 $\text{emit}(\text{goto } \text{---})$

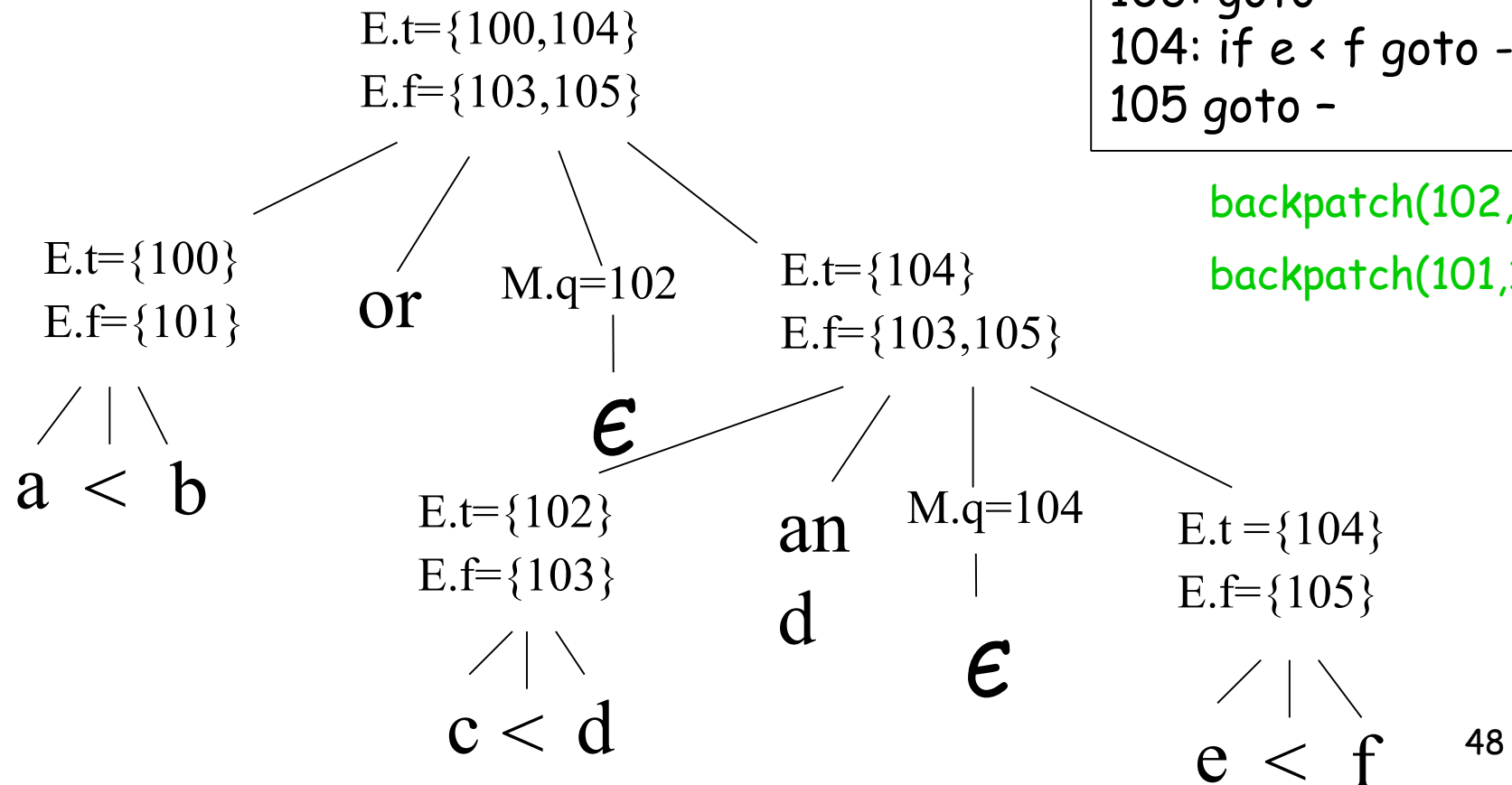
$E \rightarrow \text{false}$
 $E.\text{falselist} = \text{makelist}(\text{nextquad})$
 $\text{emit}(\text{goto } \text{---})$

$M \rightarrow \epsilon$
 $M.\text{quad} = \text{nextquad}$

Generate code for $a < b$ or $c < d$ and $e < f$

Initialize nextquad to 100

```
100: if a < b goto -  
101: goto - 102  
102: if c < d goto - 104  
103: goto -  
104: if e < f goto -  
105: goto -
```



Flow of Control Statements

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

S : Statement

A : Assignment

L : Statement list

Scheme to implement translation

- E has attributes truelist and falselist
- L and S have a list of unfilled quadruples to be filled by backpatching
- $S \rightarrow \text{while } E \text{ do } S_1$
requires labels S.begin and E.true
 - markers M_1 and M_2 record these labels
 $S \rightarrow \text{while } M_1 \text{ E do } M_2 \text{ } S_1$
 - when while. .. is reduced to S
backpatch $S_1.\text{nextlist}$ to make target of all the statements to $M_1.\text{quad}$
 - E.truelist is backpatched to go to the beginning of S_1 ($M_2.\text{quad}$)

Scheme to implement translation ...

$S \rightarrow \text{if } E \text{ then } M \ S_1$
 backpatch(E.truelist, M.quad)
 S.nextlist = merge(E.falselist, S₁.nextlist)

$S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$
 backpatch(E.truelist, M₁.quad)
 backpatch(E.falselist, M₂.quad)
 S.nextlist = merge(S₁.nextlist, N.nextlist,
 S₂.nextlist)

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$
 backpatch(S₁.nextlist, M₁.quad)
 backpatch(E.truelist, M₂.quad)
 S.nextlist = E.falselist
 emit(goto M₁.quad)

Scheme to implement translation ...

$S \rightarrow \text{begin } L \text{ end}$ $S.\text{nextlist} = L.\text{nextlist}$

$S \rightarrow A$ $S.\text{nextlist} = \text{makelist}()$

$L \rightarrow L_1 ; M S$ $\text{backpatch}(L_1.\text{nextlist}, M.\text{quad})$
 $L.\text{nextlist} = S.\text{nextlist}$

$L \rightarrow S$ $L.\text{nextlist} = S.\text{nextlist}$

$N \rightarrow \epsilon$ $N.\text{nextlist} = \text{makelist}(\text{nextquad})$
 $\text{emit}(\text{goto } \text{---})$

$M \rightarrow \epsilon$ $M.\text{quad} = \text{nextquad}$

Procedure Calls

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

$\text{Elist} \rightarrow \text{Elist} , E$

$\text{Elist} \rightarrow E$

- Calling sequence
 - allocate space for activation record
 - evaluate arguments
 - establish environment pointers
 - save status and return address
 - jump to the beginning of the procedure

Procedure Calls ...

Example

- parameters are passed by reference
- storage is statically allocated
- use param statement as place holder for the arguments
- called procedure is passed a pointer to the first parameter
- pointers to any argument can be obtained by using proper offsets

Code Generation

- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

$S \rightarrow \text{call id (Elist)}$
for each item p on queue do emit('param' p)
emit('call' id.place)

$\text{Elist} \rightarrow \text{Elist} , E$
append E.place to the end of queue

$\text{Elist} \rightarrow E$
initialize queue to contain E.place

Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation and de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.
 - If a and b are activations of two procedures then their lifetime is either non overlapping or nested
 - A procedure is recursive if an activation can begin before an earlier activation of the same procedure has ended

Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point
- Formal parameters are the one that appear in declaration. Actual Parameters are the one that appear in when a procedure is called

Activation tree

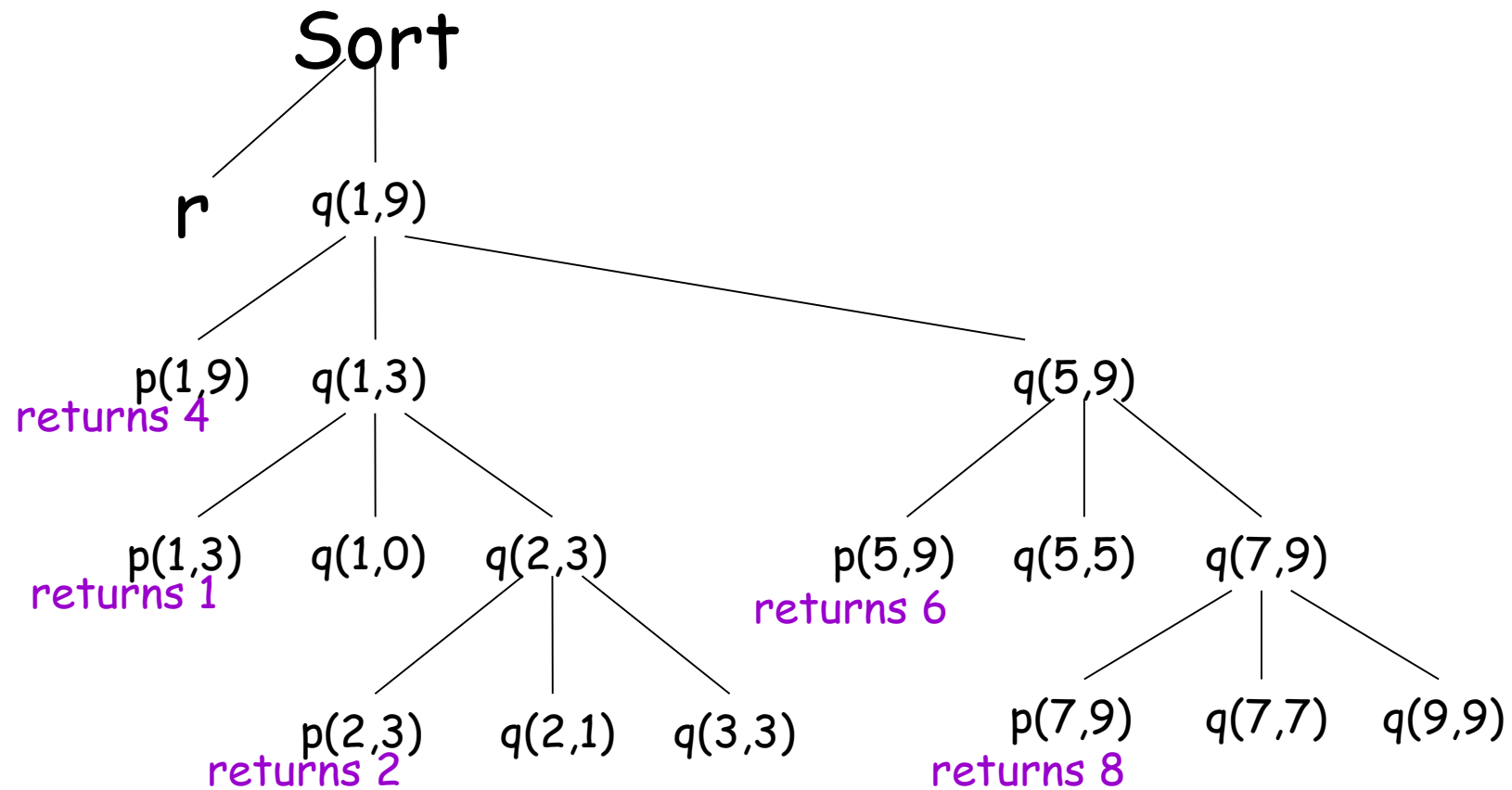
- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
 - The root represents the activation of main program
 - Each node represents an activation of procedure
 - The node **a** is parent of **b** if control flows from **a** to **b**
 - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

Example

```
program sort;  
  var a : array[0..10] of integer;  
  
  procedure readarray;  
    var i :integer;  
    :  
  function partition (y, z  
                      :integer) :integer;  
    var i, j ,x, v :integer;  
    :
```

```
    procedure quicksort (m, n  
                        :integer);  
      var i :integer;  
      :  
      i:= partition (m,n);  
      quicksort (m,i-1);  
      quicksort(i+1, n);  
      :  
begin{main}  
  readarray;  
  quicksort(1,9)  
end.
```

Activation Tree

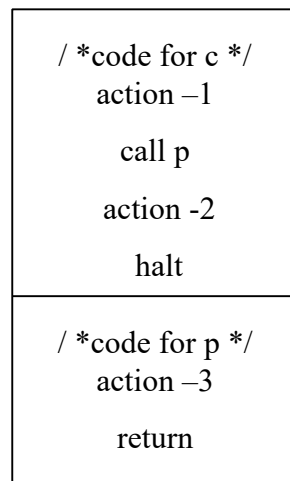


Control stack

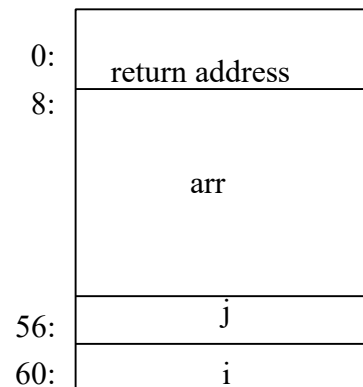
- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node n is at the top of the stack the stack contains the nodes along the path from n to the root

Run Time Storage Management

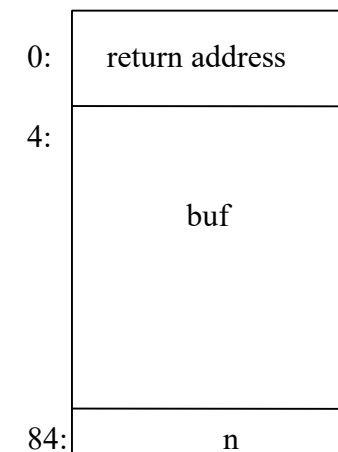
- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements:
call, return, halt and action



Three address code



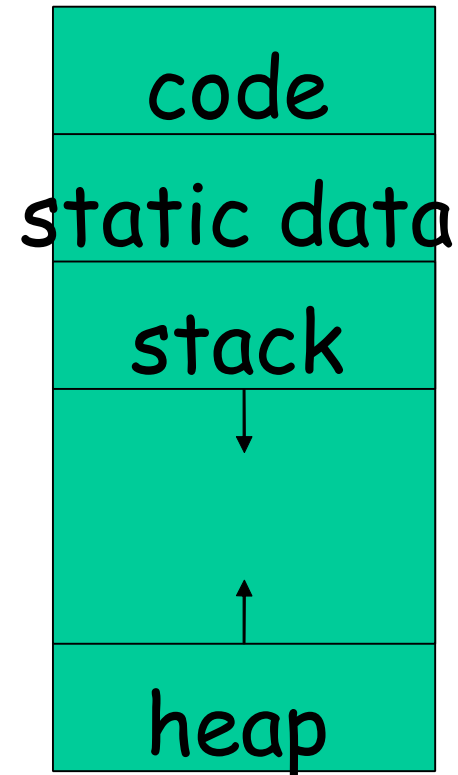
Activation record for c
(64 bytes)



Activation record for p (88 bytes)

Storage organization

- The runtime storage might be subdivided into
 - Target code
 - Data objects
 - Stack to keep track of procedure activation
 - Heap to keep all other information



Activation Record

- **temporaries:** used in expression evaluation
- **local data:** field for local data
- **saved machine status:** holds info about machine status before procedure call
- **access link :** to access non local data
- **control link :** points to activation record of caller
- **actual parameters:** field to hold actual parameters
- **returned value:** field for holding value to be returned

Temporaries
local data
machine status
Access links
Control links
Parameters
Return value

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?
- Can storage be dynamically allocated?
- Can storage be de-allocated?

Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
- Memory allocation is done as the declarations are processed
- Data may have to be aligned (in a word) padding is done to have alignment.
 - Compiler may pack the data so no padding is left
 - Additional instructions may be required to execute packed data

Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects
- Stack allocation: manages the runtime storage as a stack
- Heap allocation: allocates and de-allocates storage as needed at runtime from heap

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure

- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- Compiler decides location of each activation
- All the addresses can be filled at compile time
- Constraints
 - Size of all data objects must be known at compile time
 - Recursive procedures are not allowed
 - Data structures cannot be created dynamically

Static Allocation

- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- The instruction sequence is
MOV #here+20, callee.static-area
GOTO callee.code-area

Static Allocation ...

- `callee.static-area` and `callee.code-area` are constants referring to address of the activation record and the first address of called procedure respectively.
- `#here+20` in the move instruction is the return address; the address of the instruction following the goto instruction
- A return from procedure callee is implemented by

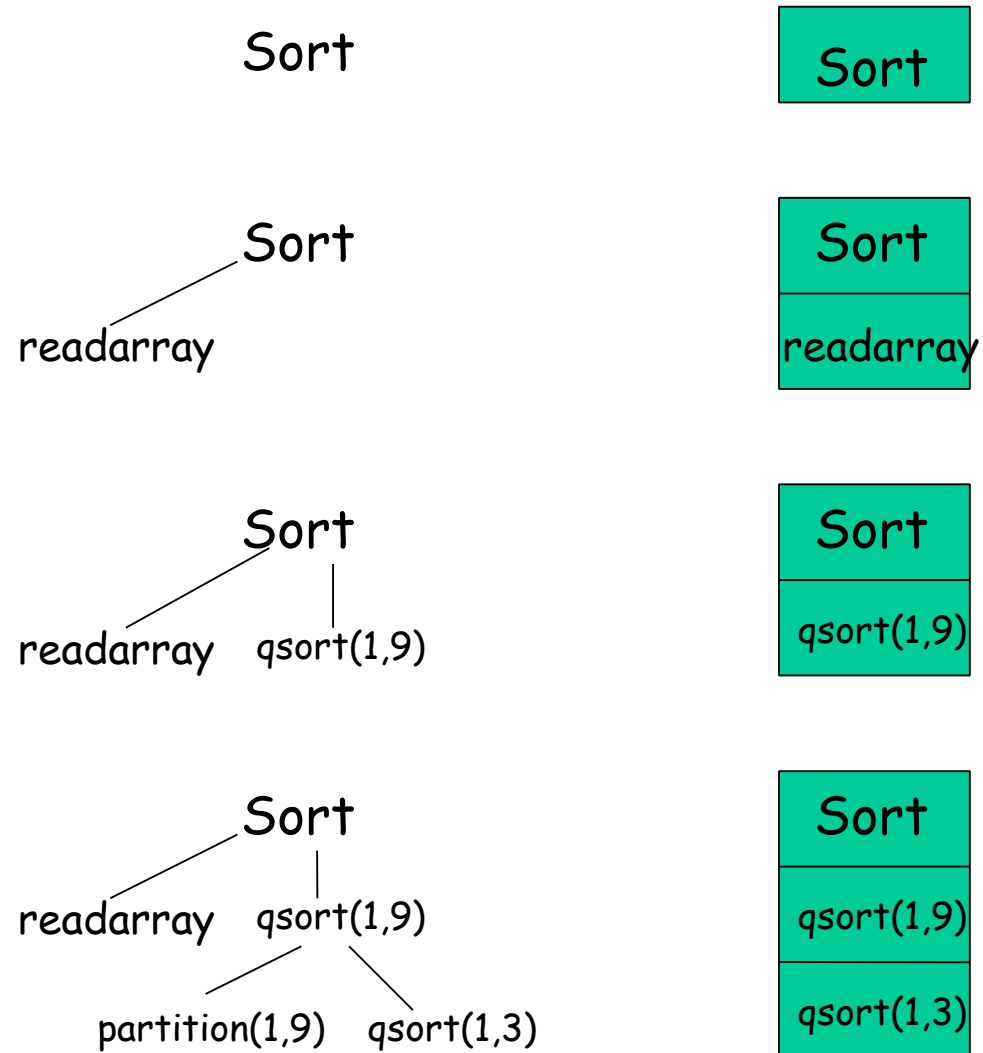
`GOTO *callee.static-area`

Example

- Assume each action block takes 20 bytes of space
- Start address of code for c and p is 100 and 200
- The activation records are statically allocated starting at addresses 300 and 364.

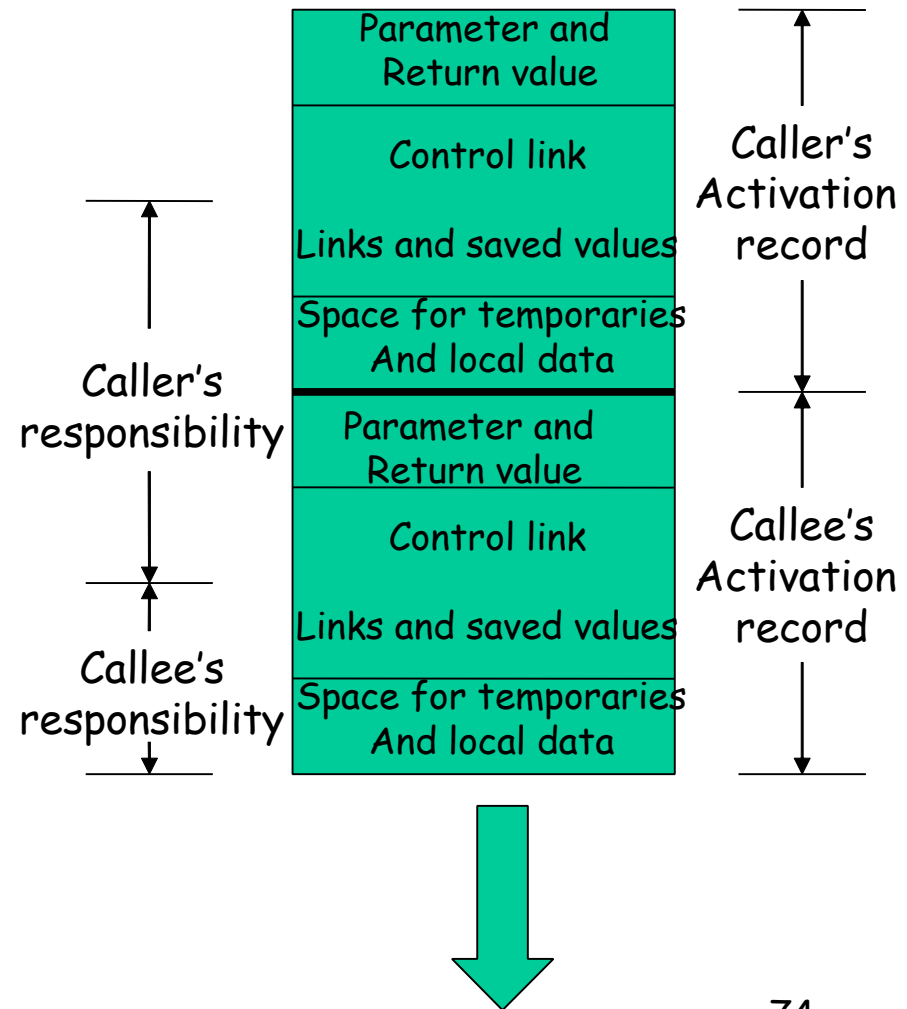
```
100: ACTION-1
120: MOV 140, 364
132: GOTO 200
140: ACTION-2
160: HALT
:
200: ACTION-3
220: GOTO *364
:
300:
304:
:
364:
368:
```


Stack Allocation



Calling Sequence

- A call sequence allocates an activation record and enters information into its field
- A return sequence restores the state of the machine so that calling procedure can continue execution



Call Sequence

- Caller evaluates the actual parameters
- Caller stores return address and other values (control link) into callee's activation record
- Callee saves register values and other status information
- Callee initializes its local data and begins execution

Return Sequence

- Callee places a return value next to activation record of caller
- Restores registers using information in status field
- Branch to return address
- Caller copies return value into its own activation record

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and words in the record are accessed with an offset from the register
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area

```
MOV #Stackstart, SP  
code for the first procedure  
HALT
```

Stack Allocation ...

- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure

ADD #caller.recordsize, SP

MOVE #here+ 16, *SP

GOTO callee.code_area

Stack Allocation ...

- The return sequence consists of two parts.
- The called procedure transfers control to the return address using

`GOTO *0(SP)`

`0(SP)` is the address of the first word in the activation record and `*0(SP)` is the return address saved there.

- The second part of the return sequence is in caller which decrements `SP`

`SUB #caller.recordsize, SP`

Example

- Consider the quicksort program
- Assume activation records for procedures *s*, *p* and *q* are *ssize*, *psize* and *qsize* respectively (determined at compile time)
- First word in each activation holds the return address
- Code for the procedures start at 100, 200 and 300 respectively, and stack starts at 600.

action-1 */* code for s * /*
call *q*

action-2
halt

action-3 */* code for p * /*
return

action-4 */* code for q * /*
call *p*

action-5
call *q*

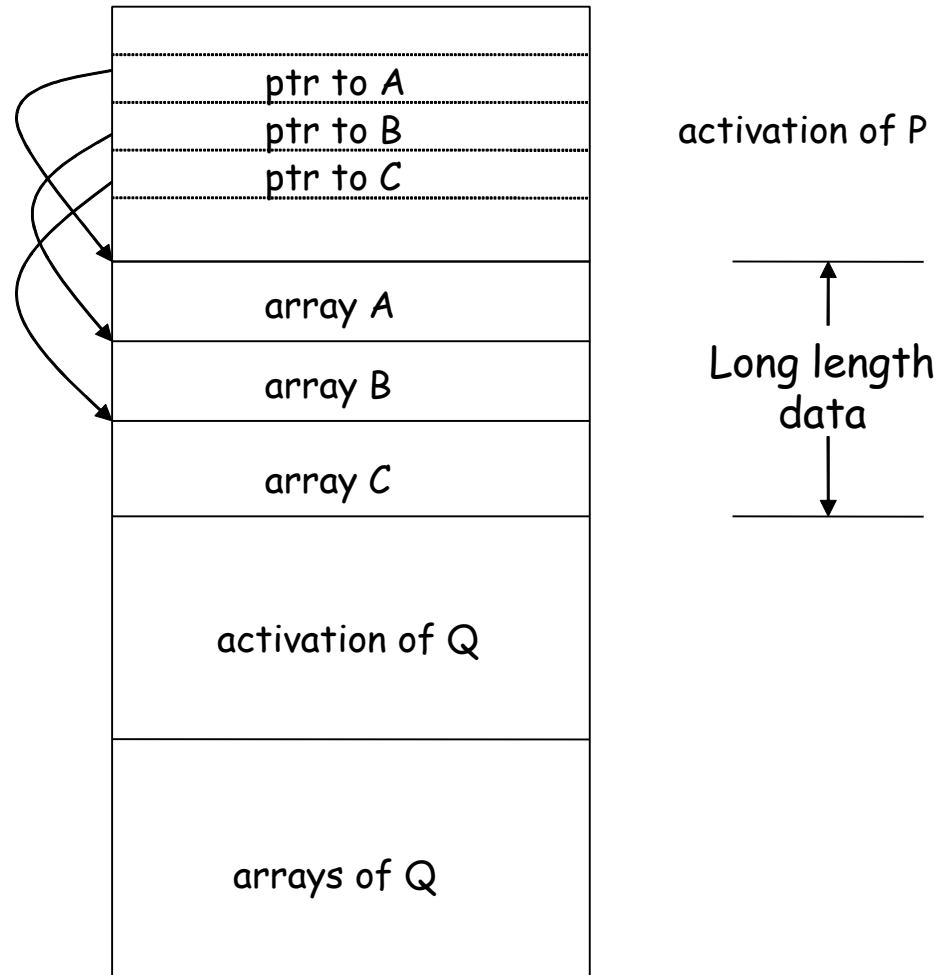
action-6
call *q*
return

100: **MOVE #600, SP**
108: action-1
128: **ADD #ssize, SP**
136: **MOVE 152, *SP**
144: **GOTO 300**
152: **SUB #ssize, SP**
160: action-2
180: **HALT**
...

200: action-3
220: **GOTO *0(SP)**
...

300: action-4
320: **ADD #qsize, SP**
328: **MOVE 344, *SP**
336: **GOTO 200**
344: **SUB #qsize, SP**
352: action-5
372: **ADD #qsize, SP**
380: **MOVE 396, *SP**
388: **GOTO 300**
396: **SUB #qsize, SP**
404: action-6
424: **ADD #qsize, SP**
432: **MOVE 448, *SP**
440: **GOTO 300**
448: **SUB #qsize, SP**
456: **GOTO *0(SP)**

Long Length Data



Dangling references

Referring to locations which have been deallocated

main()

```
{int *p;  
  p = dangle(); /* dangling reference */  
}
```

```
int *dangle();  
{  
  int i=23;  
  return &i;  
}
```

Heap Allocation

- Stack allocation cannot be used if:
 - The values of the local variables must be retained when an activation ends
 - A called activation outlives the caller
- In such a case de-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records

Heap Allocation ...

- Pieces may be de-allocated in any order
- Over time the heap will consist of alternate areas that are free and in use
- Heap manager is supposed to make use of the free space
- For efficiency reasons it may be helpful to handle small activations as a special case
- For each size of interest keep a linked list of free blocks of that size

Heap Allocation ...

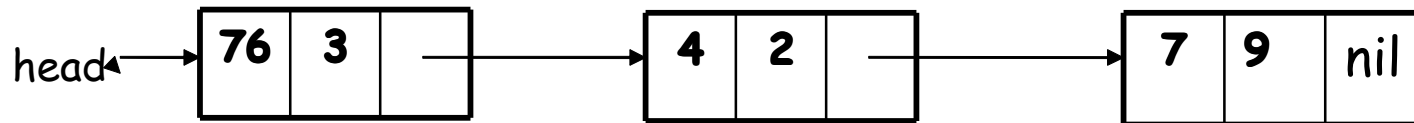
- Fill a request of size s with block of size s' where s' is the smallest size greater than or equal to s
- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory so that time taken by the manager may be negligible compared to the computation time

Language Facility for Dynamic Storage Allocation

- Storage is usually taken from heap
- Allocated data is retained until deallocated
- Allocation can be either explicit or implicit
 - Pascal : explicit allocation and de-allocation by `new()` and `dispose()`
 - Lisp : implicit allocation when `cons` is used, and de-allocation through garbage collection

Dynamic Storage Allocation

`new(p); p^.key:=k; p^.info:=i;`



Garbage : unreachable cells

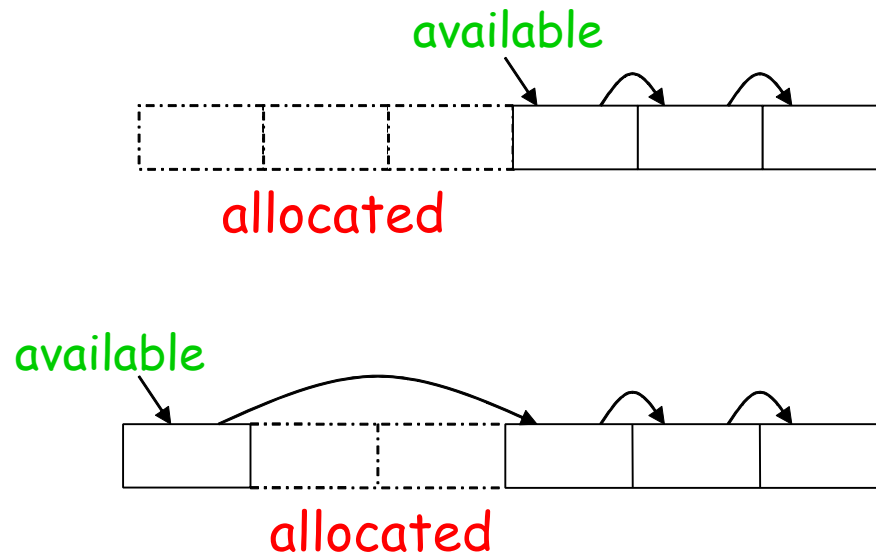
- Lisp does garbage collection
- Pascal and C do not

`head^.next := nil;`

Dangling reference

Explicit Allocation of Fixed Sized Blocks

- Link the blocks in a list
- Allocation and de-allocation can be done with very little overhead



Explicit Allocation of Fixed Sized Blocks ...

- blocks are drawn from contiguous area of storage
- An area of each block is used as pointer to the next block
- A pointer **available** points to the first block
- Allocation means removing a block from the available list
- De-allocation means putting the block in the available list
- Compiler routines need not know the type of objects to be held in the blocks
- Each block is treated as a variant record

Explicit Allocation of Variable Size Blocks

- Storage can become fragmented
- Situation may arise
 - If program allocates five blocks
 - then de-allocates second and fourth block



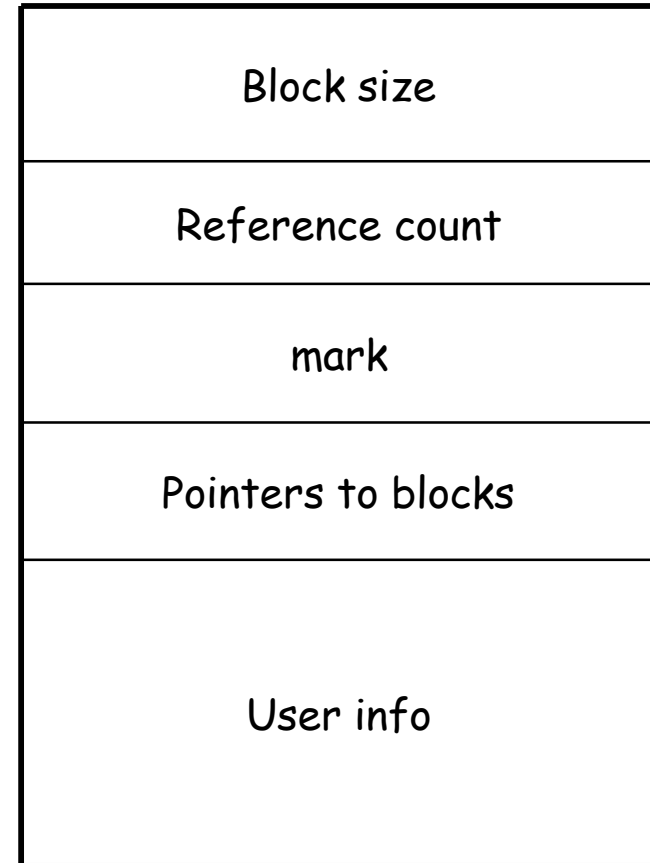
- Fragmentation is of no consequence if blocks are of fixed size
- Blocks can not be allocated even if space is available

First Fit Method

- When a block of size s is to be allocated
 - search first free block of size $f \geq s$
 - sub divide into two blocks of size s and $f-s$
 - time overhead for searching a free block
- When a block is de-allocated
 - check if it is next to a free block
 - combine with the free block to create a larger free block

Implicit De-allocation

- Requires co-operation between user program and run time system
- Run time system needs to know when a block is no longer in use
- Implemented by fixing the format of storage blocks



Recognizing Block boundaries

- If block size is fixed then position information can be used
- Otherwise keep size information to determine the block boundaries

Whether Block is in Use

- References may occur through a pointer or a sequence of pointers
- Compiler needs to know position of all the pointers in the storage
- Pointers are kept in fixed positions and user area does not contain any pointers

Reference Count

- Keep track of number of blocks which point directly to the present block
- If count drops to 0 then block can be deallocated
- Maintaining reference count is costly
 - assignment $p:=q$ leads to change in the reference counts of the blocks pointed to by both p and q
- Reference counts are used when pointers do not appear in cycles

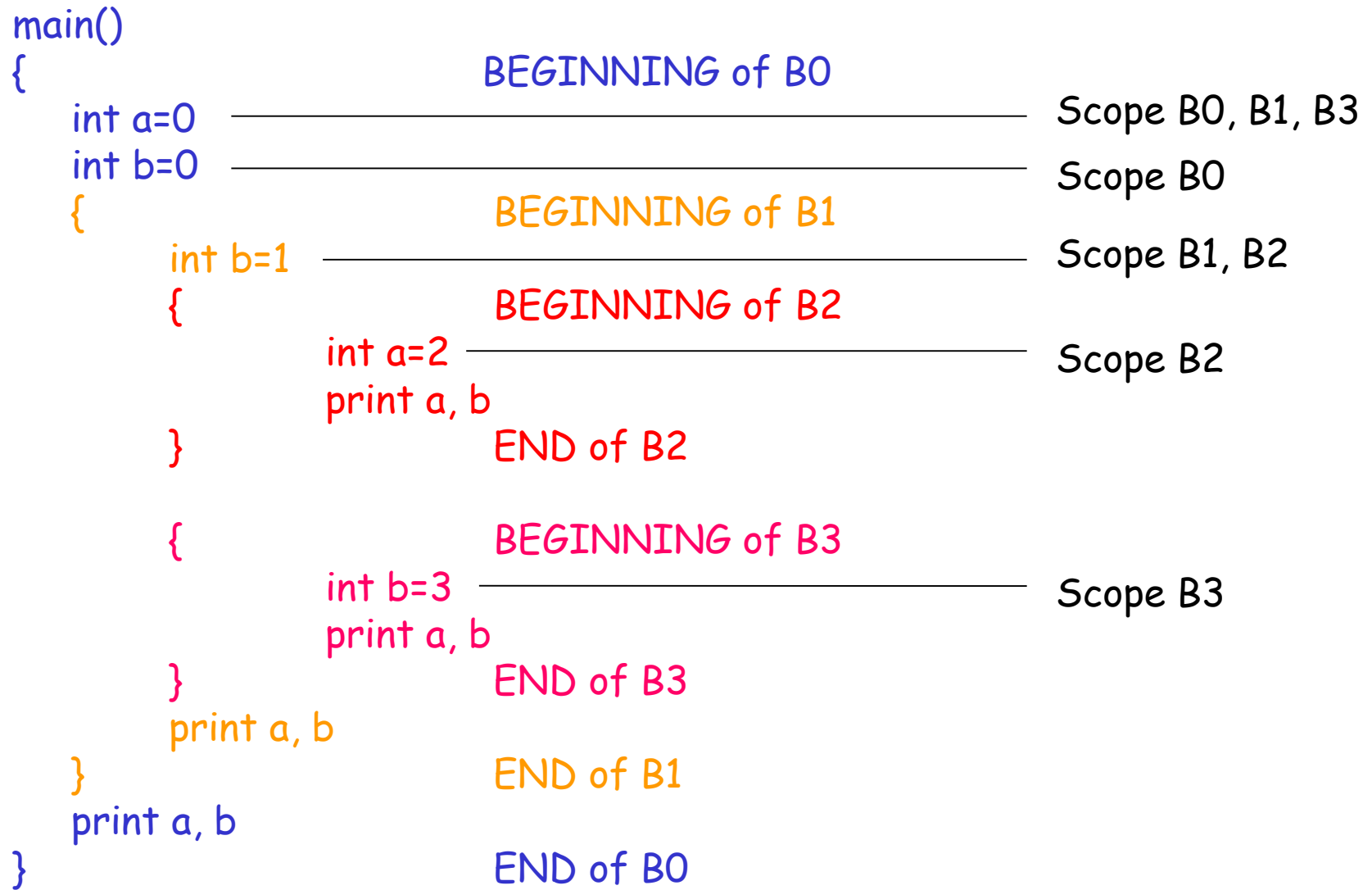
Marking Techniques

- Suspend execution of the user program
- use frozen pointers to determine which blocks are in use
- This approach requires knowledge of all the pointers
- Go through the heap marking all the blocks unused
- Then follow pointers marking a block as used that is reachable
- De-allocate a block still marked unused
- Compaction: move all used blocks to the end of heap. All the pointers must be adjusted to reflect the move

Block

- A block statement contains its own data declarations
- Blocks can be nested
- The property is referred to as block structured
- Scope of the declaration is given by most closely nested rule
 - The scope of a declaration in block B includes B
 - If a name X is not declared in B
then an occurrence of X is in the scope of declarator X in B'
such that
 - B' has a declaration of X
 - B' is most closely nested around B

Example



Blocks ...

- Blocks are simpler to handle than procedures
- Blocks can be treated as parameter less procedures
- Use stack for memory allocation
- Allocate space for complete procedure body at one time

a0
b0
b1
a2,b3

Scope of declaration

- A declaration is a syntactic construct associating information with a name
 - Explicit declaration :Pascal (Algol class of languages)
var i : integer
 - Implicit declaration: Fortran
i is assumed to be integer
- There may be independent declarations of same name in a program.
- Scope rules determine which declaration applies to a name
- Name binding

name $\xrightarrow{\text{environment}}$ storage $\xrightarrow{\text{state}}$ value

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)

Lexical scope without nested procedures

Only two scopes: locals (stack allocated) and globals (static allocated)

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically
- A non local name must be local to the top of the stack
- Stack allocation of non local has advantage:
 - Non locals have static allocations

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- Only two scopes:
 - locals (stack allocated)
 - globals (static allocated)

Scope with nested procedures

```
Program sort;  
  var a: array[1..n] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
    begin  
  
    end;  
  procedure exchange(i,j:integer)  
    begin  
  
    end;  
end;
```

```
procedure quicksort(m,n:integer);  
  var k,v : integer;  
  
  function partition(y,z:integer):  
    integer;  
    var i,j: integer;  
    begin  
  
    end;  
  begin  
  
  end;  
end.
```


What is the problem?

```
procedure p()  
  int x;  
  
  procedure q()  
  begin  
    print x;  
  end  
  
  procedure r()  
  begin  
    q();  
  end
```

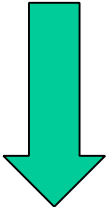
```
procedure s()  
begin  
  r();  
end  
  
procedure t()  
begin  
  s();  
end  
  
begin  
  q(); r(); s(); t();  
end
```

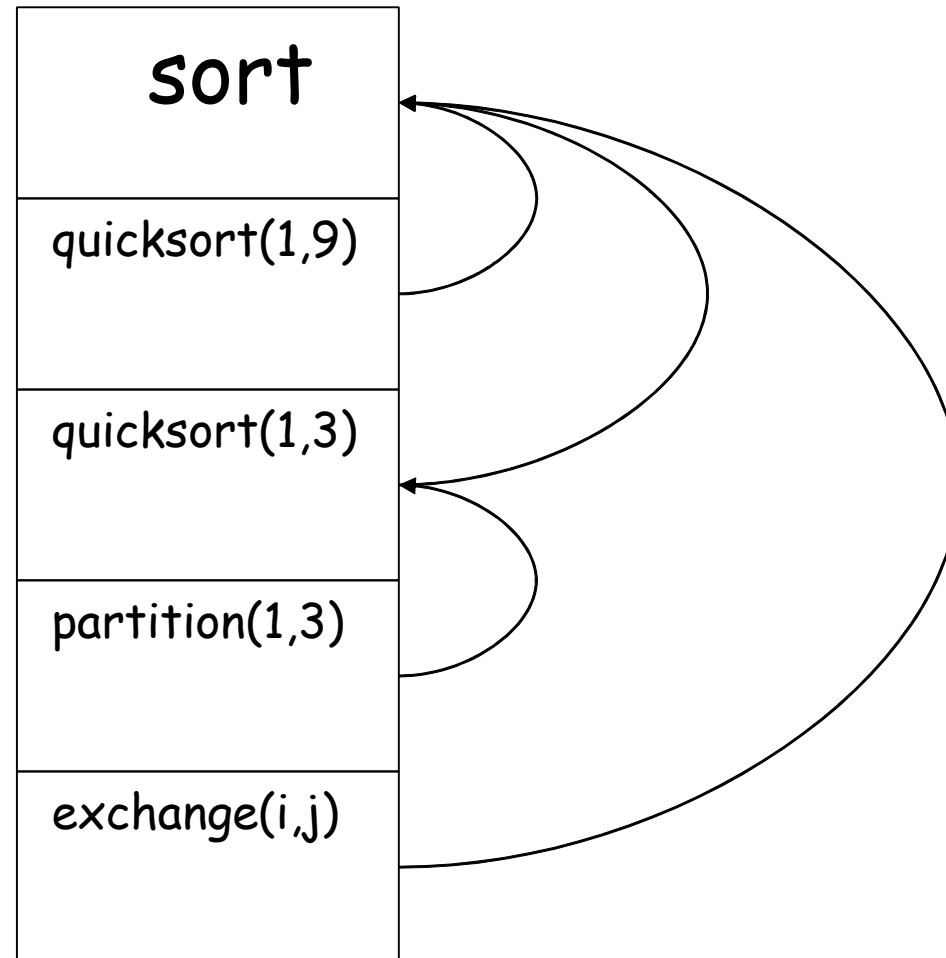
Nesting Depth

- Main procedure is at depth 1
- Add 1 to depth as we go from enclosing to enclosed procedure

Access to non-local names

- Include a field 'access link' in the activation record
- If p is nested in q then access link of p points to the access link in most recent activation of q


Stack



Access to non local names ...

- Suppose procedure p at depth np refers to a non-local a at depth na , then storage for a can be found as
 - follow $(np-na)$ access links from the record at the top of the stack
 - after following $(np-na)$ links we reach procedure for which a is local
- Therefore, address of a non local a in procedure p can be stored in symbol table as
$$(np-na, \text{offset of } a \text{ in record of activation having } a)$$

```
procedure p()
```

```
  int i;
```

```
  procedure x()
```

```
  begin
```

```
    print(i); // nx = np + 1
```

```
  end
```

```
begin
```

```
  x();
```

```
end
```

p



x

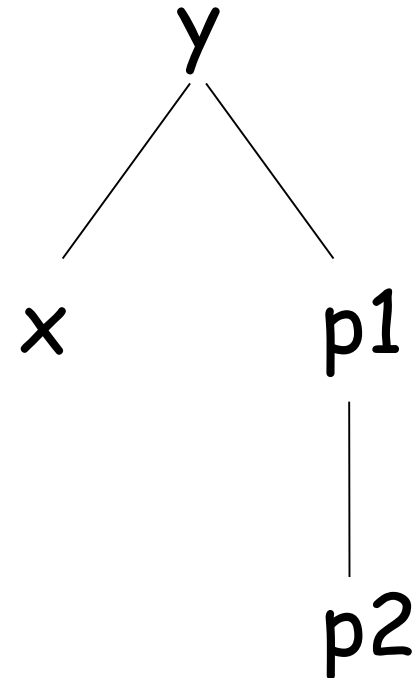
Can nx be any more deeper than np?

```
procedure y()  
  int i;
```

```
  procedure x()  
  begin  
    print(i);  
  end
```

```
  procedure p1()  
    procedure p2()  
    begin  
      x(); // np2 = nx - 1  
    end
```

```
  begin  
    x(); // np1 = nx  
  end
```

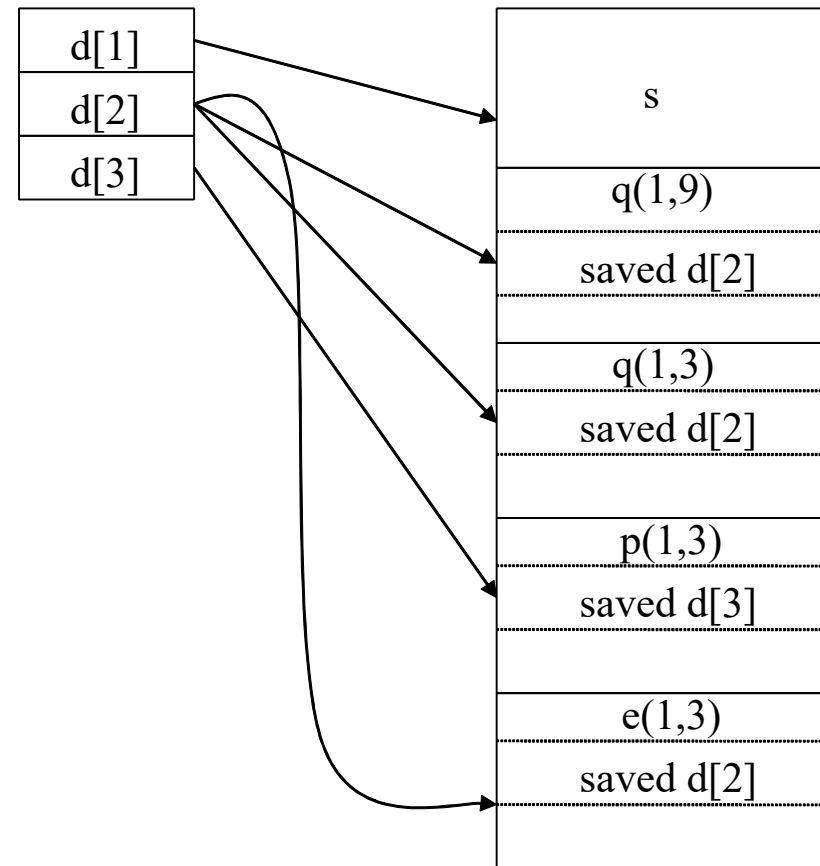


How to setup access links?

- suppose procedure p at depth n_p calls procedure x at depth n_x .
- The code for setting up access links depends upon whether the called procedure is nested within the caller.
 - $n_p < n_x$
Called procedure is nested more deeply than p .
Therefore, x must be declared in p . The access link in the called procedure must point to the access link of the activation just below it
 - $n_p \geq n_x$
From scoping rules enclosing procedure at the depth $1, 2, \dots, n_x - 1$ must be same. Follow $n_p - (n_x - 1)$ links from the caller, we reach the most recent activation of the procedure that encloses both called and calling procedure

Displays

- Faster access to non locals
- Uses an array of pointers to activation records
- Non locals at depth i is in the activation record pointed to by $d[i]$



Justification for Displays

- Suppose procedure at depth j calls procedure at depth i
- Case $j < i$ then $i = j + 1$
 - called procedure is nested within the caller
 - first j elements of display need not be changed
 - set $d[i]$ to the new activation record
- Case $j \geq i$
 - enclosing procedure at depths $1 \dots i-1$ are same and are left un-disturbed
 - old value of $d[i]$ is saved and $d[i]$ points to the new record
 - display is correct as first $i-1$ records are not disturbed 113

Dynamic Scope

- Binding of non local names to storage do not change when new activation is set up
- A non local name a in the called activation refers to same storage that it did in the calling activation

Dynamic Scoping: Example

- Consider the following program

```
program dynamic (input, output);
```

```
  var r: real;
```

```
  procedure show;
```

```
    begin write(r) end;
```

```
  procedure small;
```

```
    var r: real;
```

```
    begin r := 0.125; show end;
```

```
begin
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

```
  show; small; writeln;
```

```
end.
```

Example ...

- Output under lexical scoping

0.250 0.250

0.250 0.250

- Output under dynamic scoping

0.250 0.125

0.250 0.125

Implementing Dynamic Scope

- Deep Access
 - Dispense with access links
 - use control links to search into the stack (that contains name-value pairs)
 - term deep access comes from the fact that search may go deep into the stack
- Shallow Access
 - hold current value of each name in static memory
 - when a new activation of *p* occurs a local name *n* in *p* takes over the storage for *n*
 - previous value of *n* is saved in the activation record of *p*

Parameter Passing

- Call by value
 - actual parameters are evaluated and their rvalues are passed to the called procedure
 - used in Pascal and C
 - formal is treated just like a local name
 - caller evaluates the actual parameters and places rvalue in the storage for formals
 - call has no effect on the activation record of caller

Parameter Passing ...

- Call by reference (call by address)
 - the caller passes a pointer to each location of actual parameters
 - if actual parameter is a name then lvalue is passed
 - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

Parameter Passing ...

- Copy restore (copy-in copy-out, call by value result)
 - actual parameters are evaluated, rvalues are passed by call by value, lvalues are determined before the call
 - when control returns, the current rvalues of the formals are copied into lvalues of the locals

Parameter Passing ...

- Call by name (used in Algol)
 - names are copied
 - local names are different from names of calling procedure

```
swap(i,a[i])  
temp = i  
i = a[i]  
a[i] = temp
```