SUBROTINAS

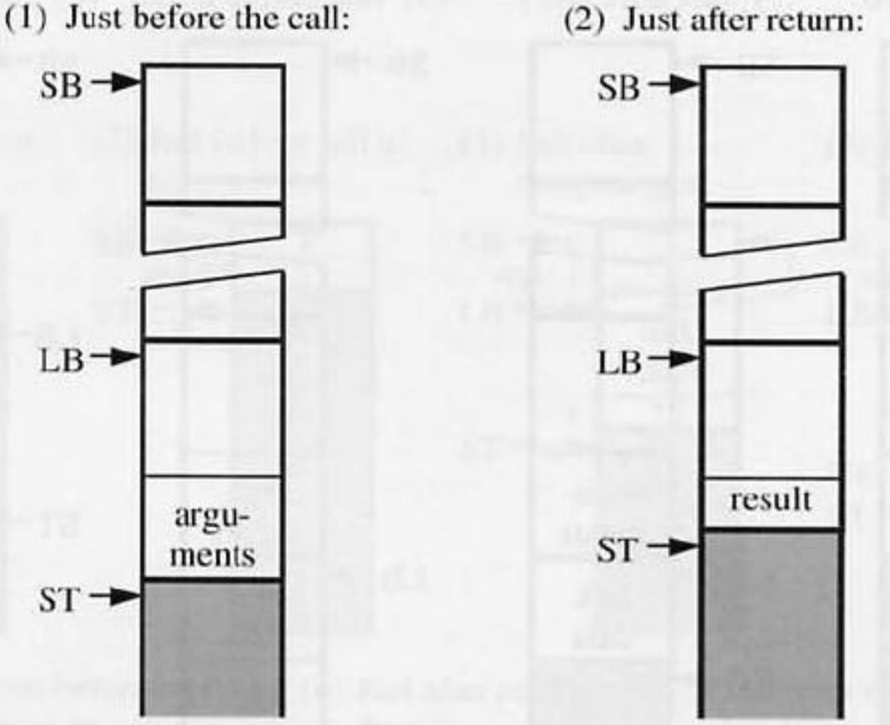
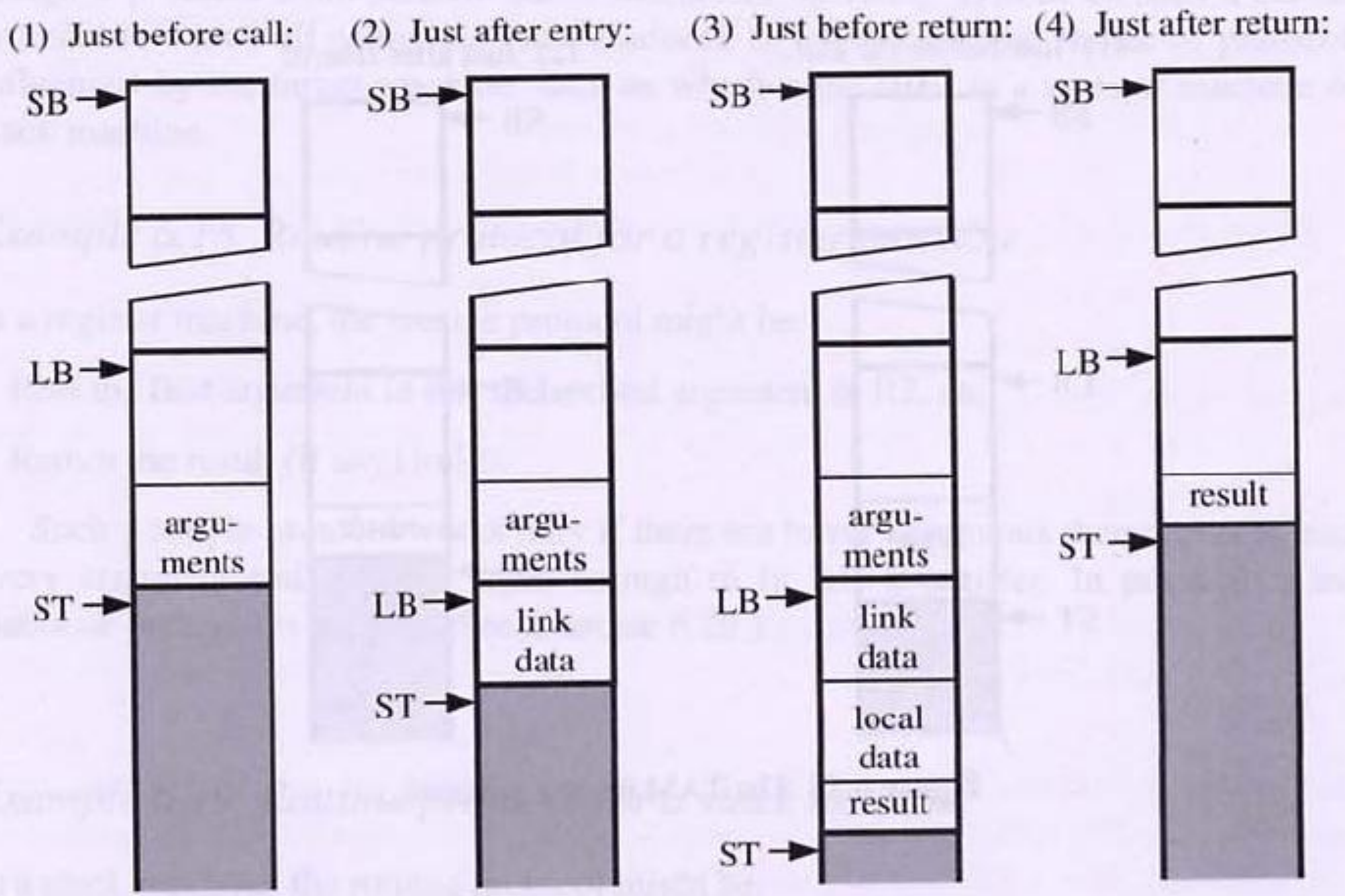
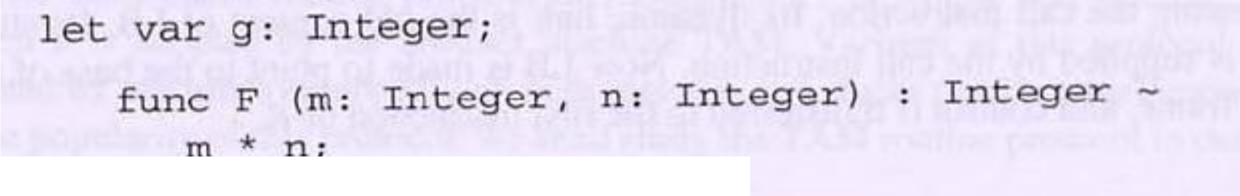
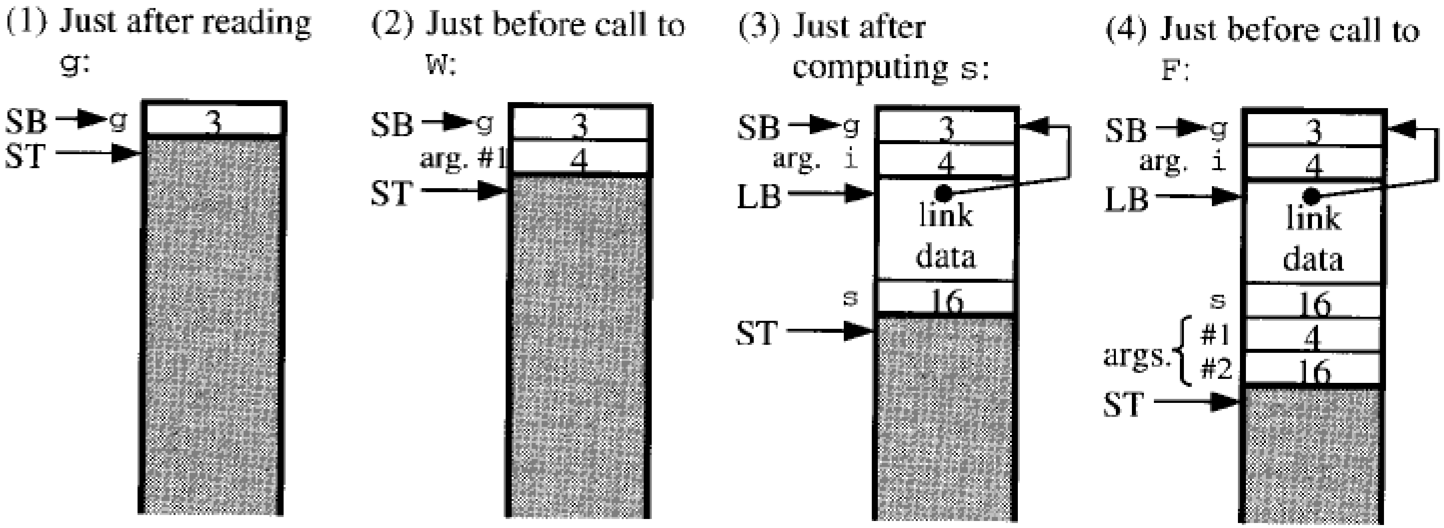


Figure 6.18 The TAM routine protocol.


(1) Just before call:
(2) Just after entry:
(3) Just before return:
(4) Just after return:
SB
LB
ST
argu- ments
link data
local data
result



```
let var g: Integer;
   func F (m: Integer, n: Integer) : Integer ~
      m * n;
   proc W (i: Integer) ~
      let const s ~ i * i
      in
         begin
         putint(F(i, s));
         putint(F(s, s))
         end
in
   begin
   getint(var g);
   W(g+1)
   end
```

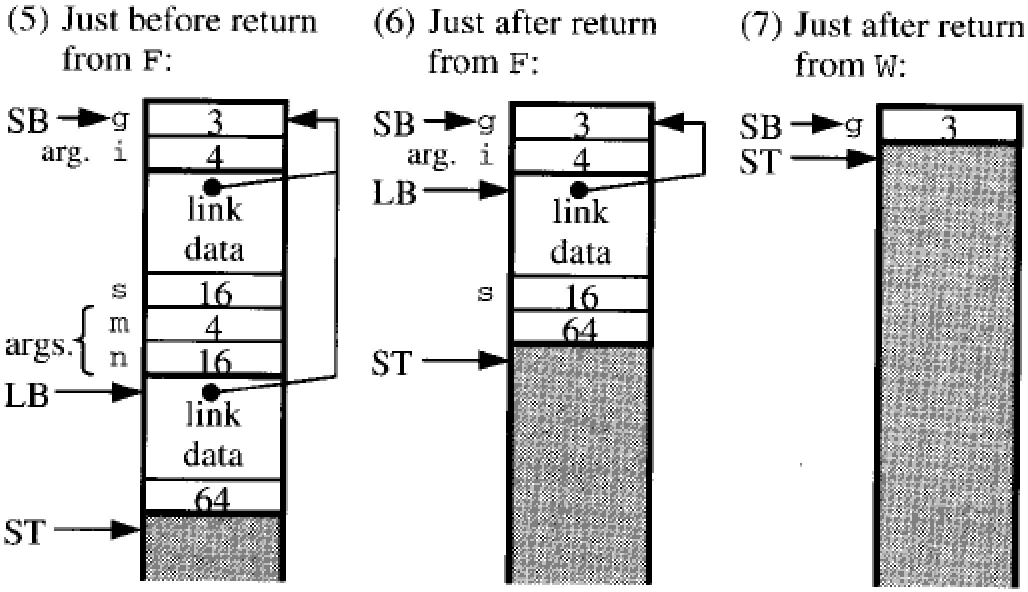


(1) Just after reading `g`:

(2) Just before call to `W`:

(3) Just after computing `s`:

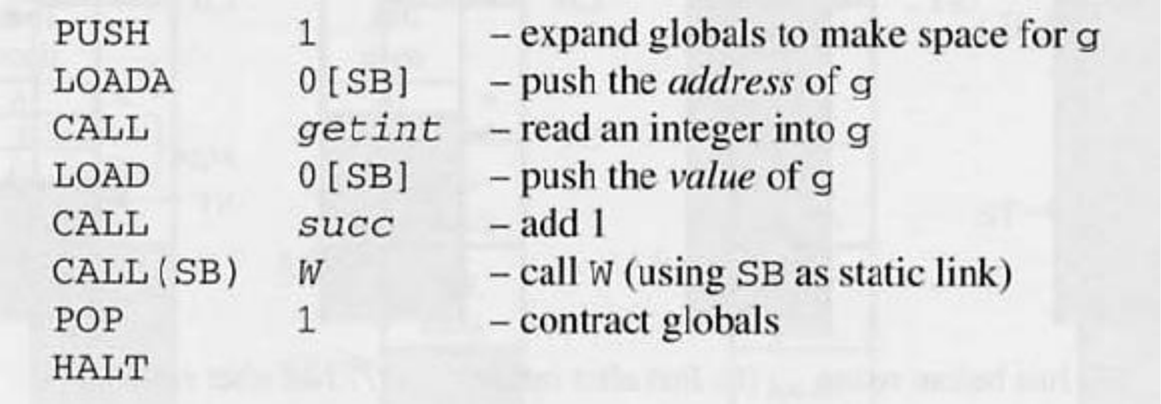
(4) Just before call to `F`:



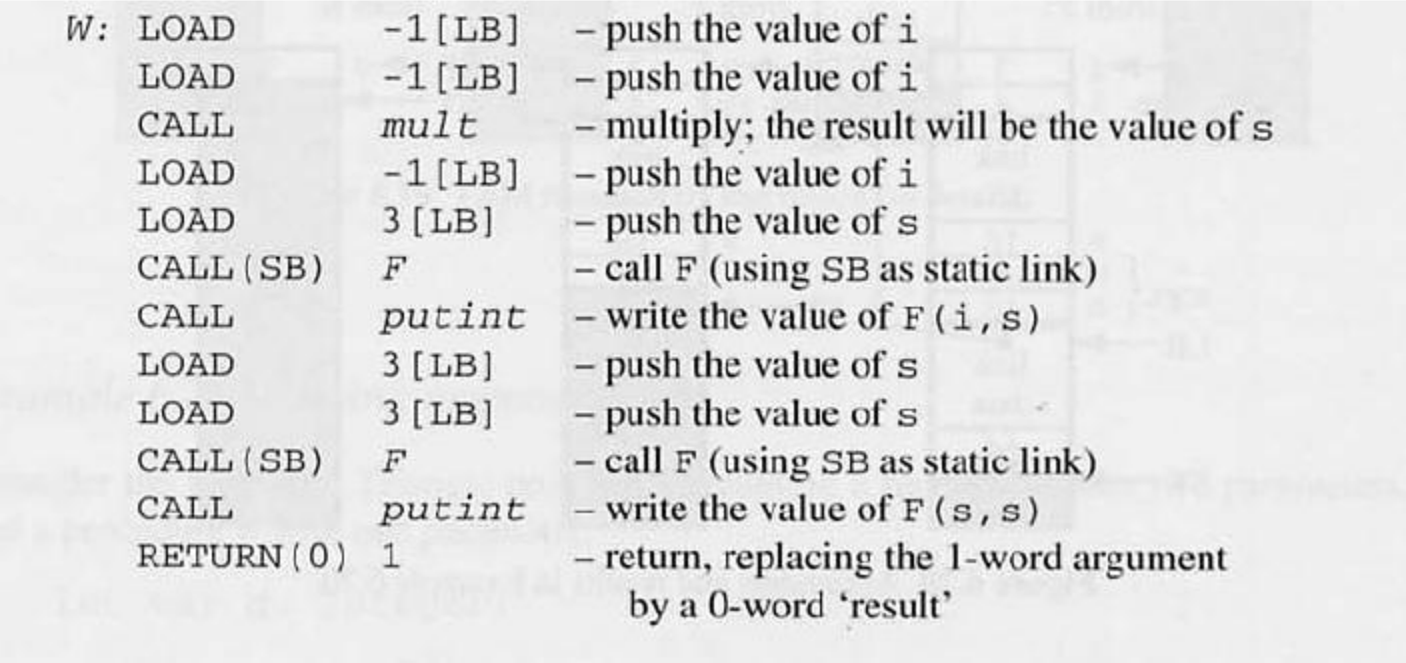
(5) Just before return from F:

(6) Just after return from F:

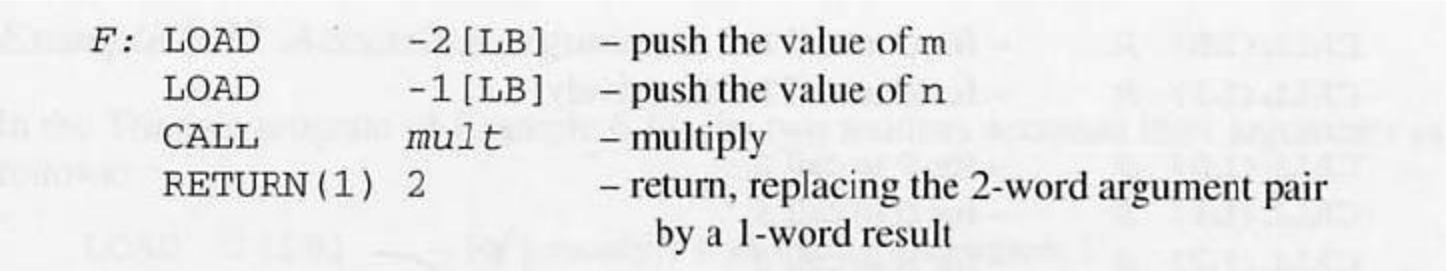
(7) Just after return from W:



```
PUSH        1          – expand globals to make space for g
LOADA       0[SB]      – push the address of g
CALL        getint     – read an integer into g
LOAD        0[SB]      – push the value of g
CALL        succ       – add 1
CALL(SB)    W          – call W (using SB as static link)
POP         1          – contract globals
HALT
```

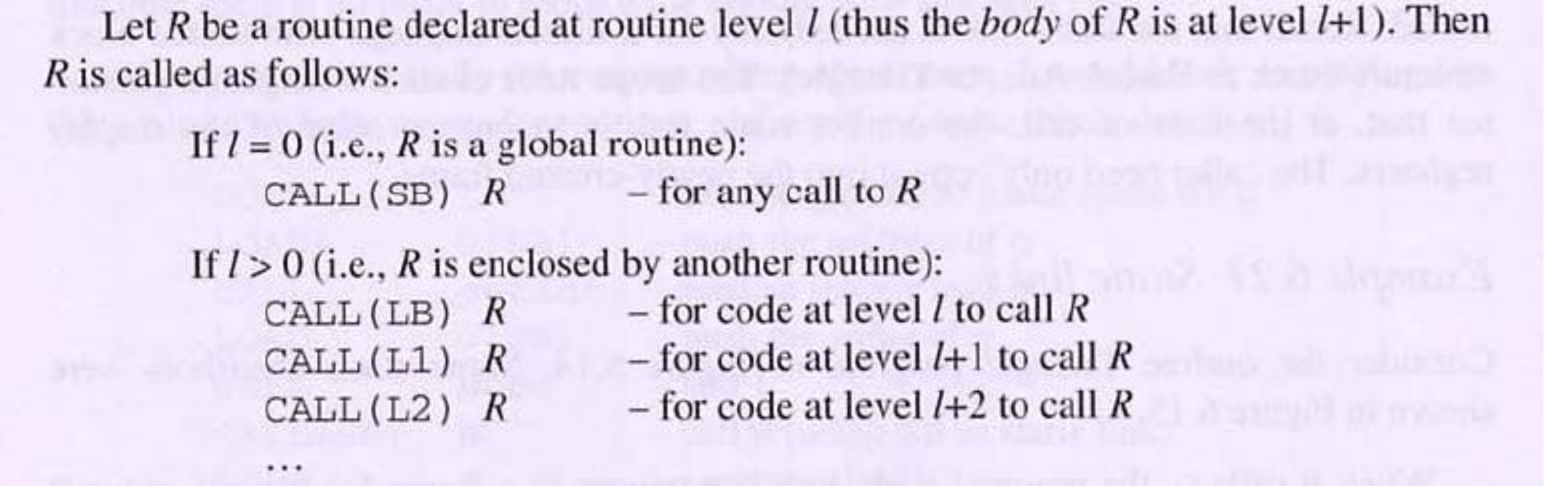


```
W: LOAD         -1[LB]    – push the value of i
   LOAD         -1[LB]    – push the value of i
   CALL         mult      – multiply; the result will be the value of s
   LOAD         -1[LB]    – push the value of i
   LOAD         3[LB]     – push the value of s
   CALL(SB)     F         – call F (using SB as static link)
   CALL         putint    – write the value of F(i,s)
   LOAD         3[LB]     – push the value of s
   LOAD         3[LB]     – push the value of s
   CALL(SB)     F         – call F (using SB as static link)
   CALL         putint    – write the value of F(s,s)
   RETURN(0)    1         – return, replacing the 1-word argument
                              by a 0-word 'result'
```



```
F: LOAD       -2[LB]   – push the value of m
   LOAD       -1[LB]   – push the value of n
   CALL       mult     – multiply
   RETURN(1)  2        – return, replacing the 2-word argument pair
                           by a 1-word result
```

LINK ESTÁTICO



Let *R* be a routine declared at routine level *l* (thus the *body* of *R* is at level *l*+1). Then *R* is called as follows:

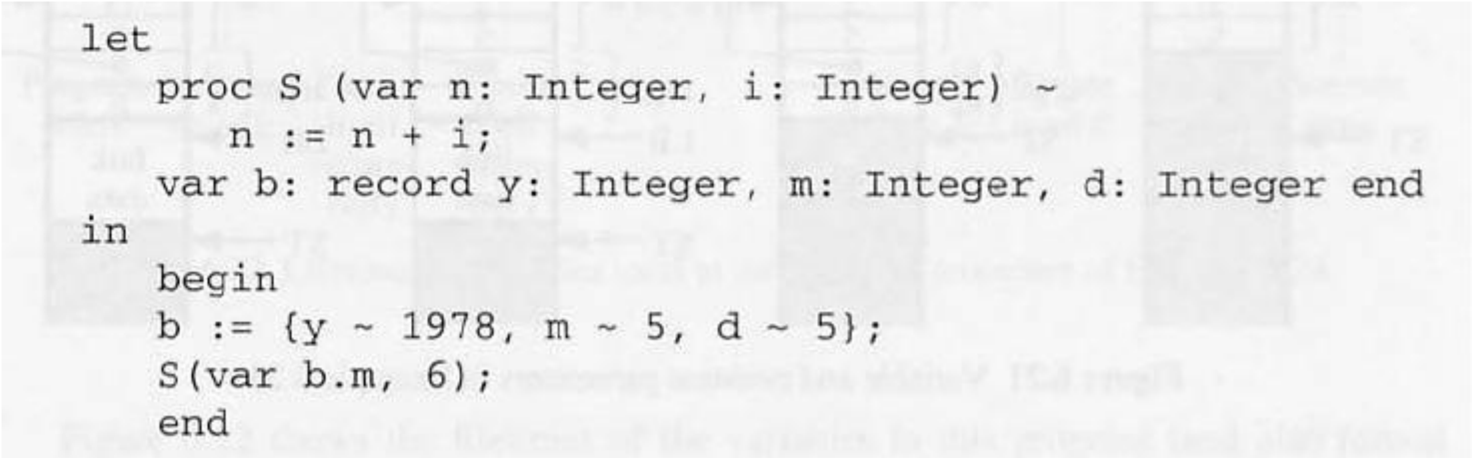
If *l* = 0 (i.e., *R* is a global routine):

`CALL(SB)` *R* – for any call to *R*

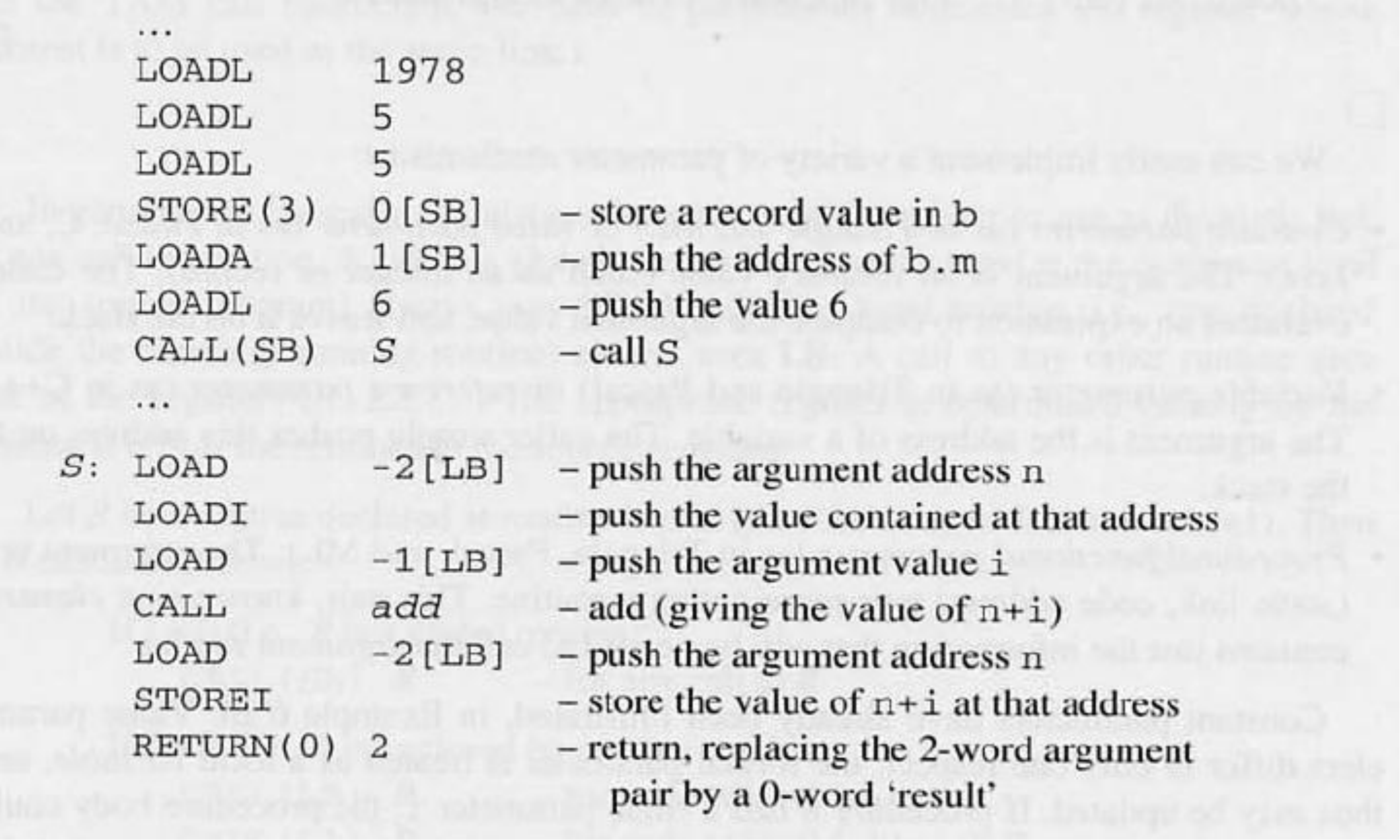
If *l* > 0 (i.e., *R* is enclosed by another routine):

`CALL(LB)` *R* – for code at level *l* to call *R*
`CALL(L1)` *R* – for code at level *l*+1 to call *R*
`CALL(L2)` *R* – for code at level *l*+2 to call *R*
...

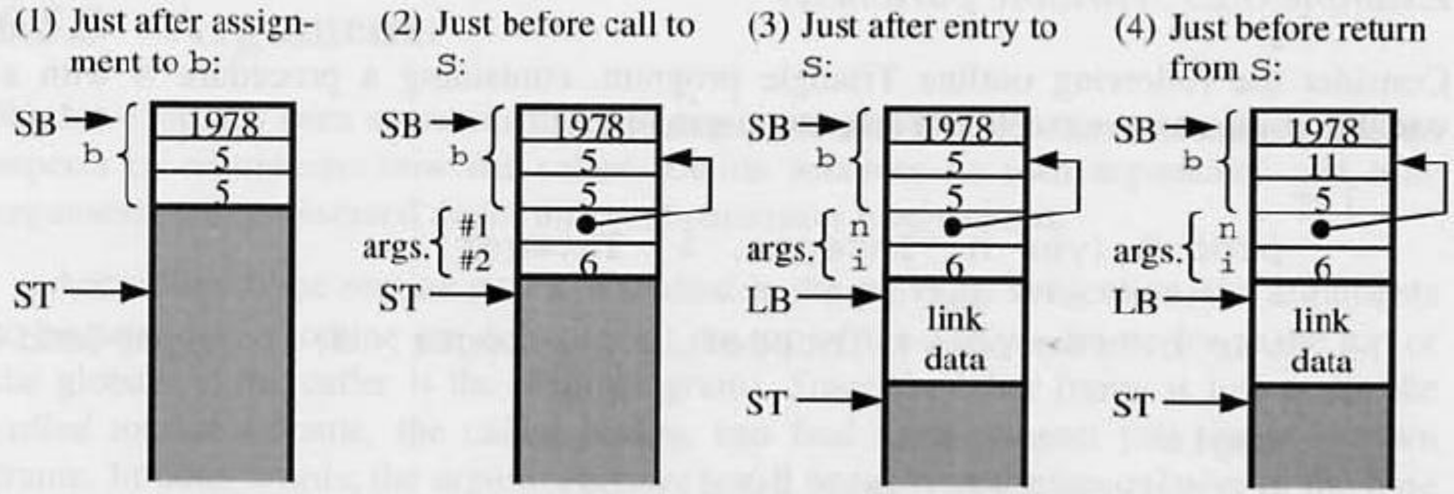
ARGUMENTOS



```
let
   proc S (var n: Integer, i: Integer) ~
      n := n + i;
   var b: record y: Integer, m: Integer, d: Integer end
in
   begin
   b := {y ~ 1978, m ~ 5, d ~ 5};
   S(var b.m, 6);
   end
```



```
     ...
     LOADL       1978
     LOADL       5
     LOADL       5
     STORE(3)    0[SB]    – store a record value in b
     LOADA       1[SB]    – push the address of b.m
     LOADL       6        – push the value 6
     CALL(SB)    S        – call S
     ...
S:   LOAD        -2[LB]   – push the argument address n
     LOADI                – push the value contained at that address
     LOAD        -1[LB]   – push the argument value i
     CALL        add      – add (giving the value of n+i)
     LOAD        -2[LB]   – push the argument address n
     STOREI               – store the value of n+i at that address
     RETURN(0)   2        – return, replacing the 2-word argument
                            pair by a 0-word 'result'
```



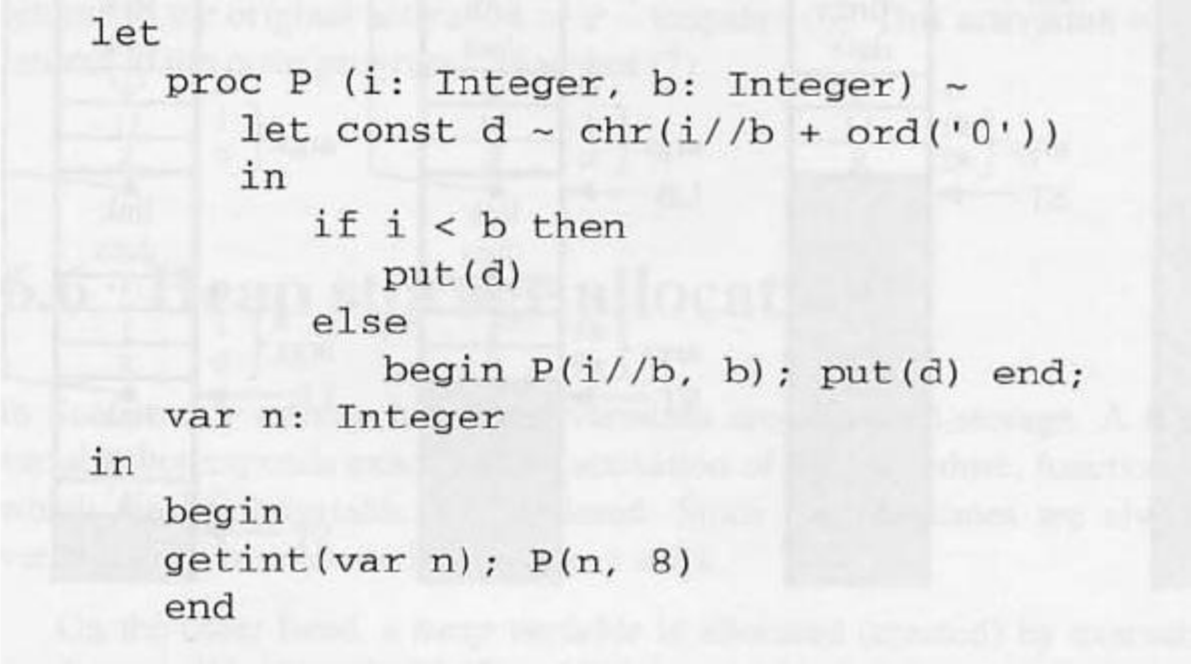
(1) Just after assignment to b:

(2) Just before call to S:

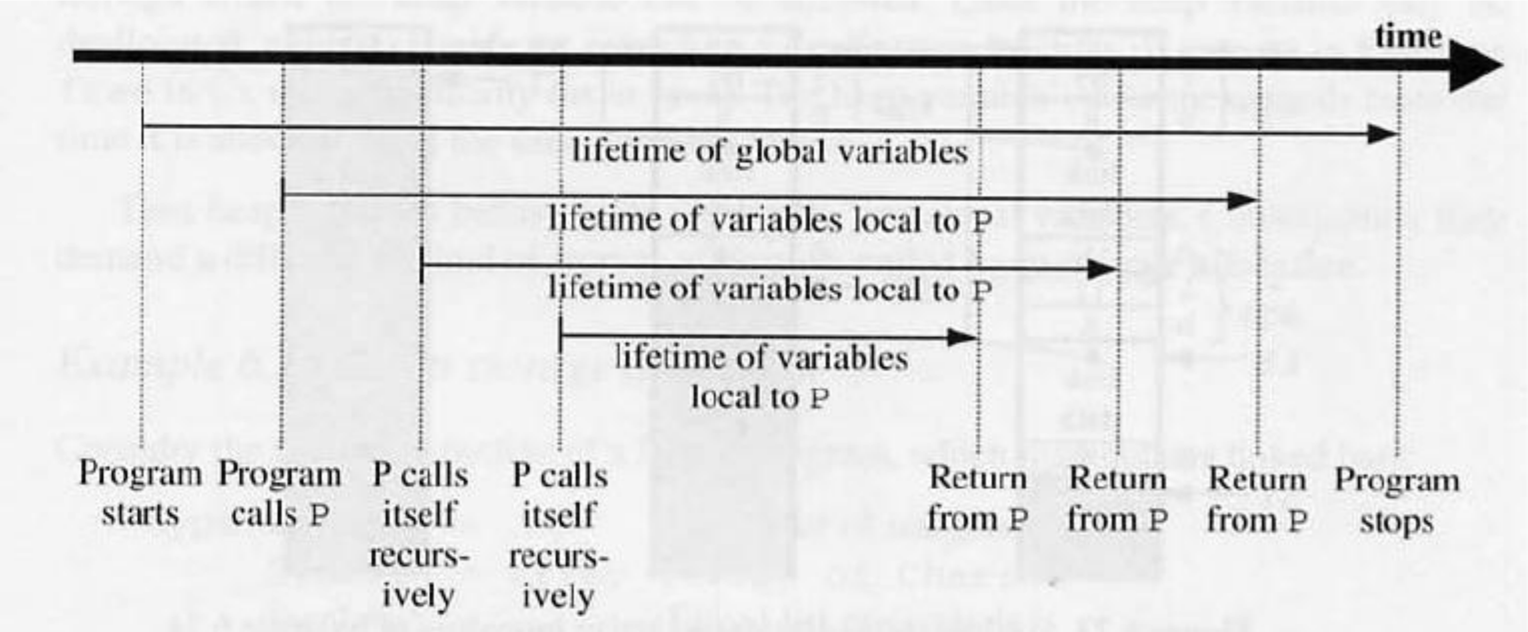
(3) Just after entry to S:

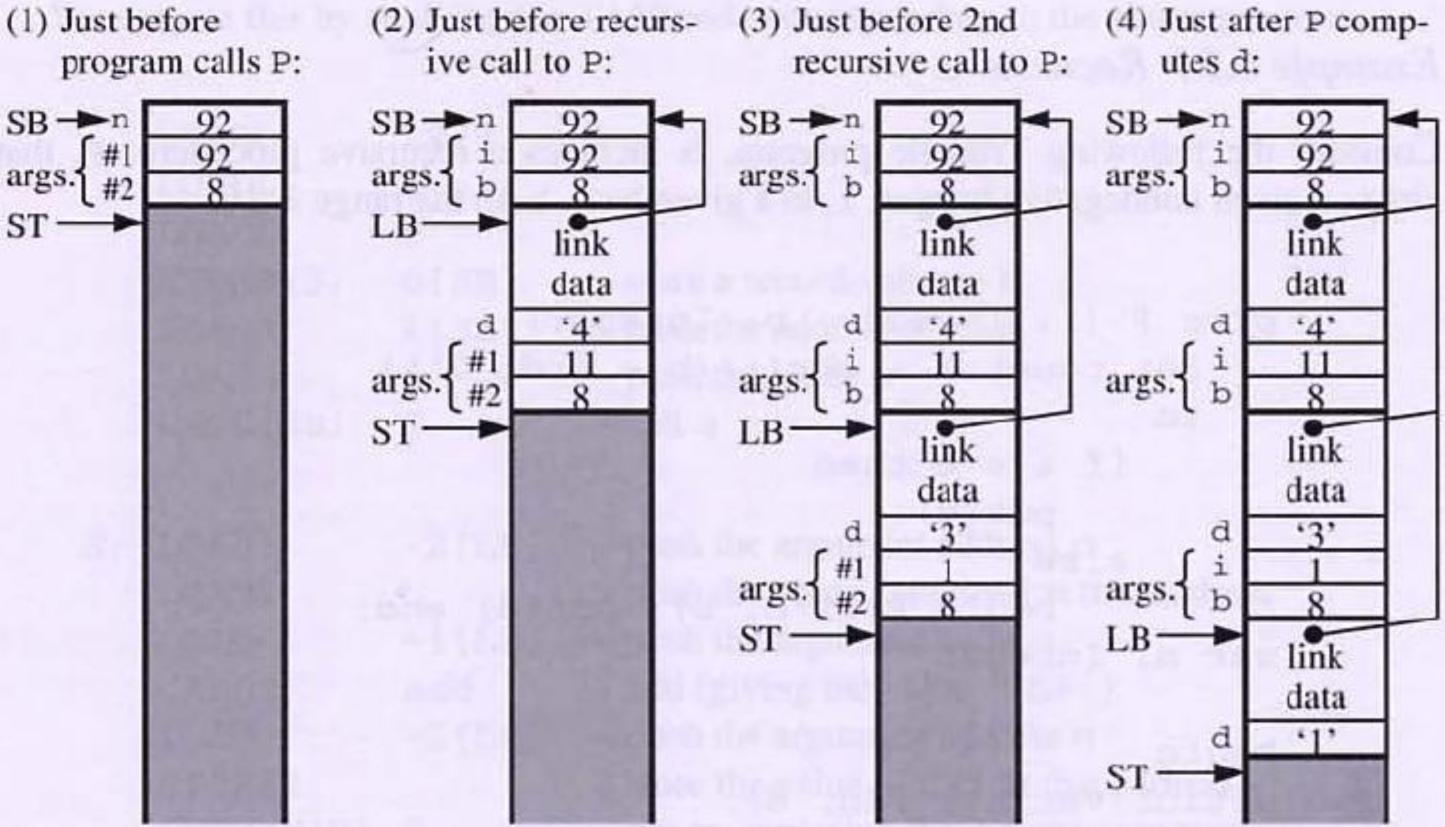
(4) Just before return from S:

RECURSÃO



```
let
   proc P (i: Integer, b: Integer) ~
      let const d ~ chr(i//b + ord('0'))
      in
         if i < b then
            put(d)
         else
            begin P(i//b, b); put(d) end;
   var n: Integer
in
   begin
   getint(var n); P(n, 8)
   end
```


time
lifetime of global variables
lifetime of variables local to P
lifetime of variables local to P
lifetime of variables local to P
Program starts
Program calls P
P calls itself recursively
P calls itself recursively
Return from P
Return from P
Return from P
Program stops

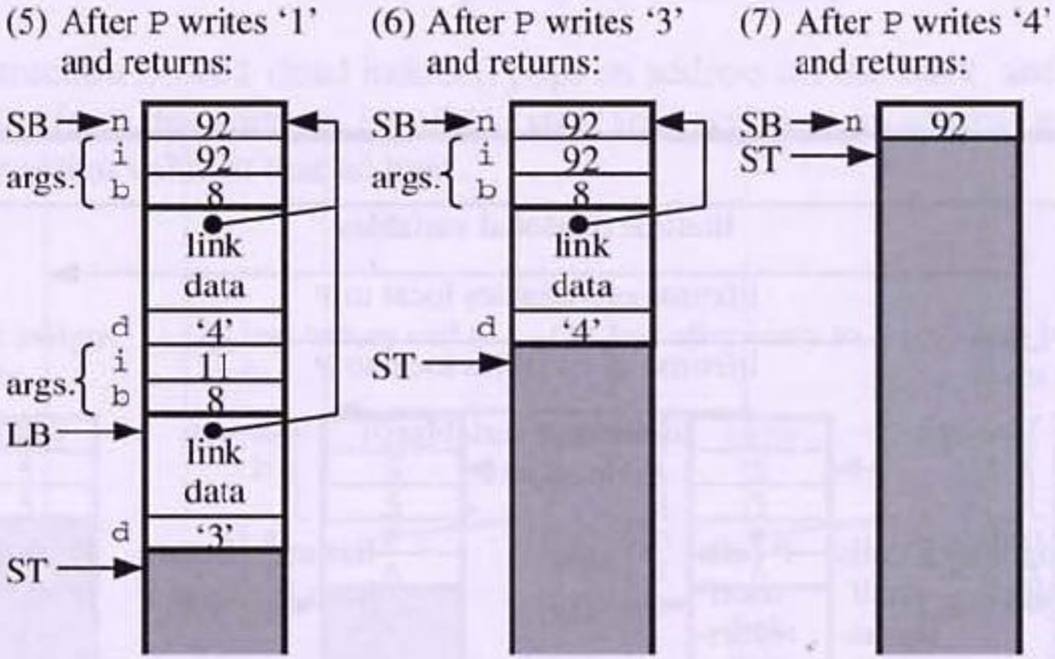


(1) Just before program calls P:

(2) Just before recursive call to P:

(3) Just before 2nd recursive call to P:

(4) Just after P computes d:

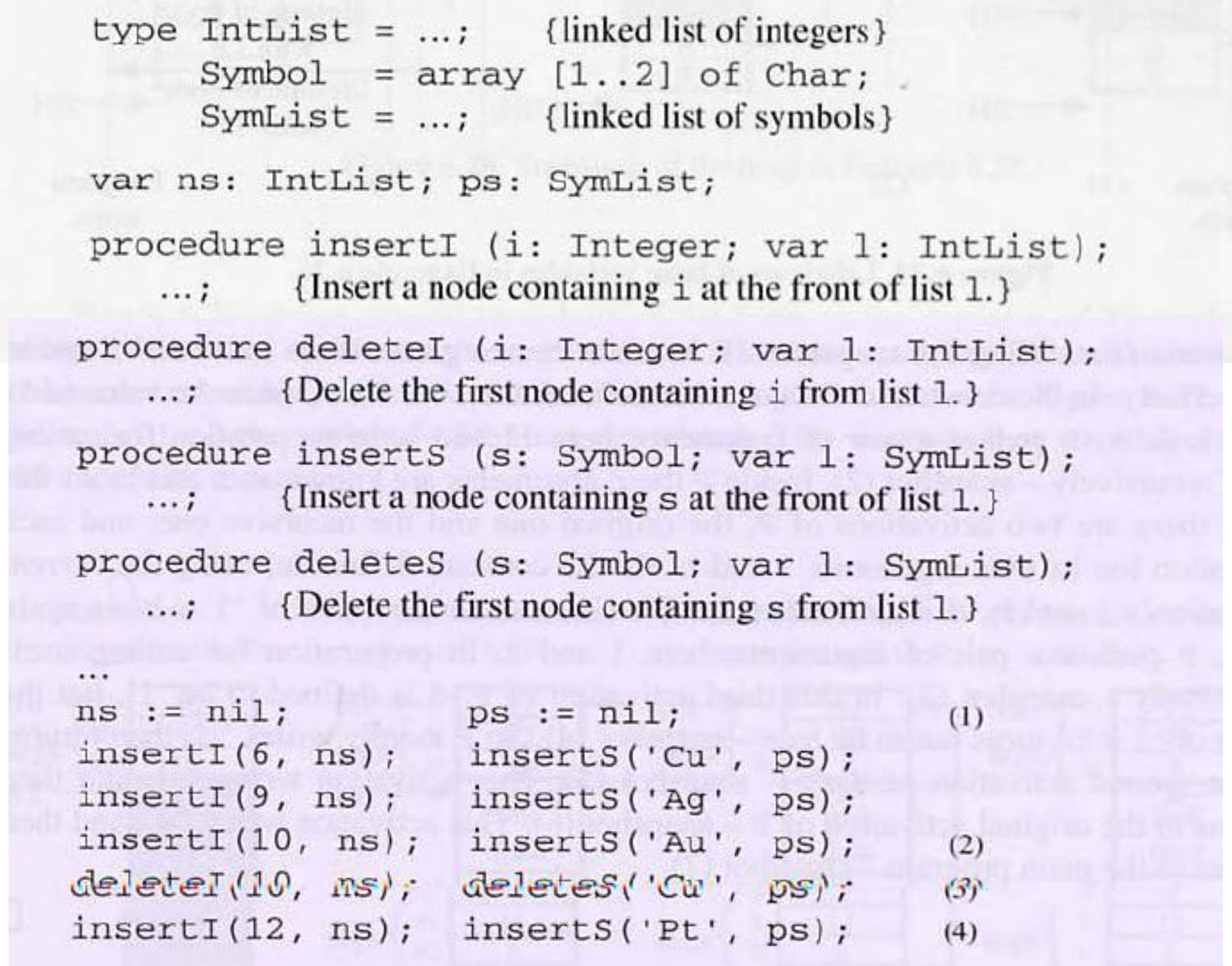


(5) After P writes '1' and returns:

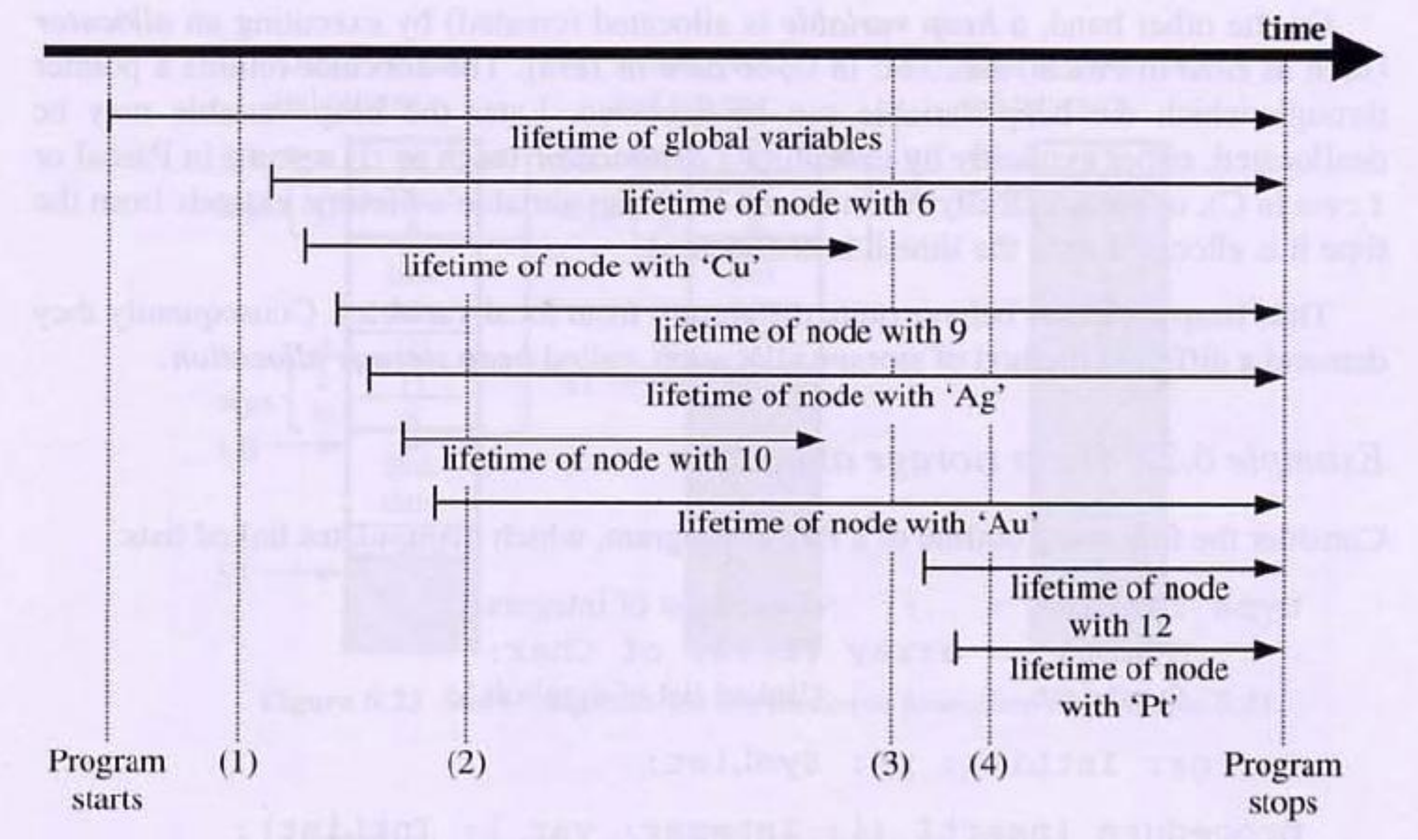
(6) After P writes '3' and returns:

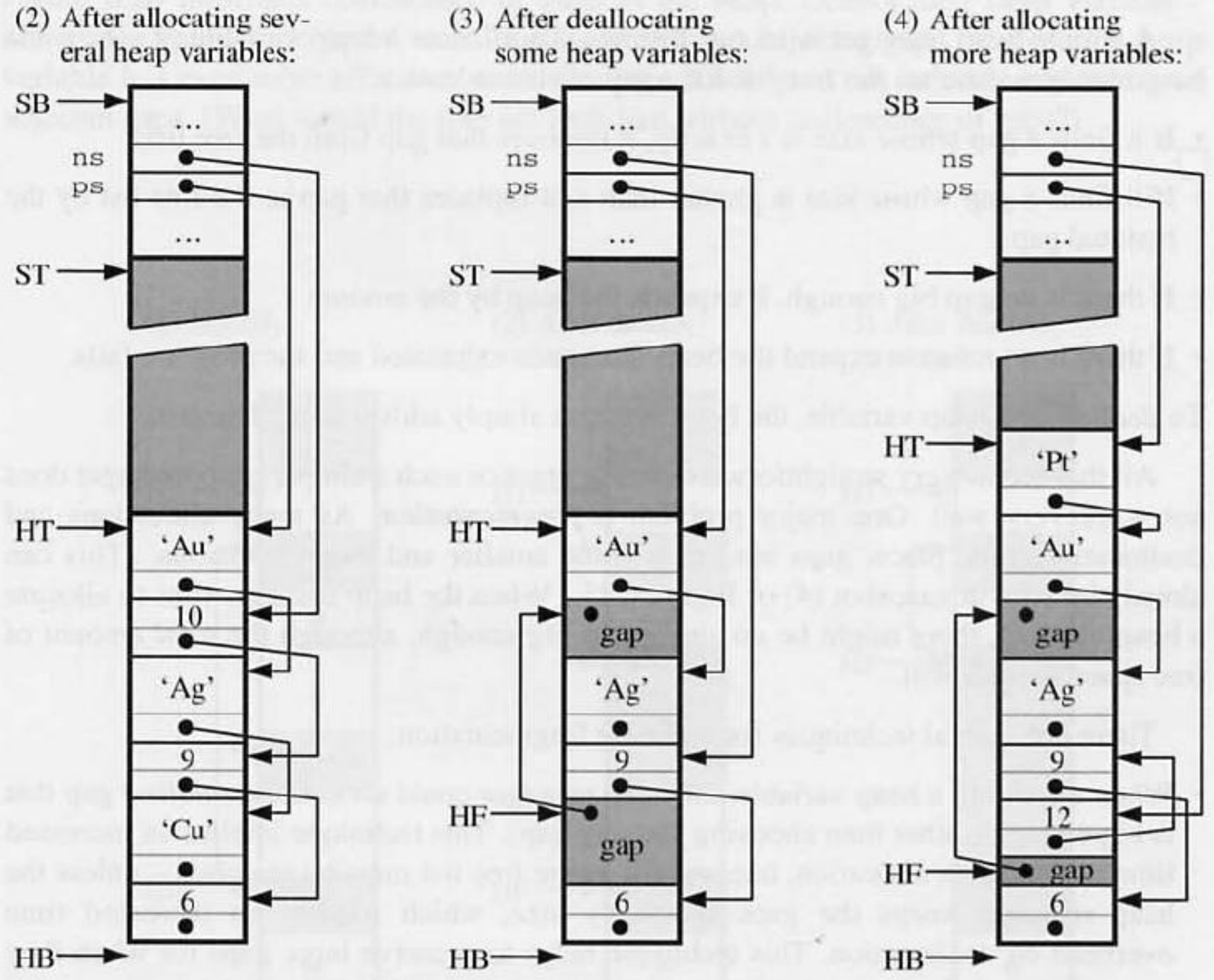
(7) After P writes '4' and returns:

HEAP



```
type IntList = ...;     {linked list of integers}
     Symbol  = array [1..2] of Char;
     SymList = ...;     {linked list of symbols}

var ns: IntList; ps: SymList;

procedure insertI (i: Integer; var l: IntList);
   ...;      {Insert a node containing i at the front of list l.}

procedure deleteI (i: Integer; var l: IntList);
   ...;      {Delete the first node containing i from list l.}

procedure insertS (s: Symbol; var l: SymList);
   ...;      {Insert a node containing s at the front of list l.}

procedure deleteS (s: Symbol; var l: SymList);
   ...;      {Delete the first node containing s from list l.}

...
ns := nil;          ps := nil;                (1)
insertI(6, ns);     insertS('Cu', ps);
insertI(9, ns);     insertS('Ag', ps);
insertI(10, ns);    insertS('Au', ps);        (2)
deleteI(10, ns);    deleteS('Cu', ps);        (3)
insertI(12, ns);    insertS('Pt', ps);        (4)
```

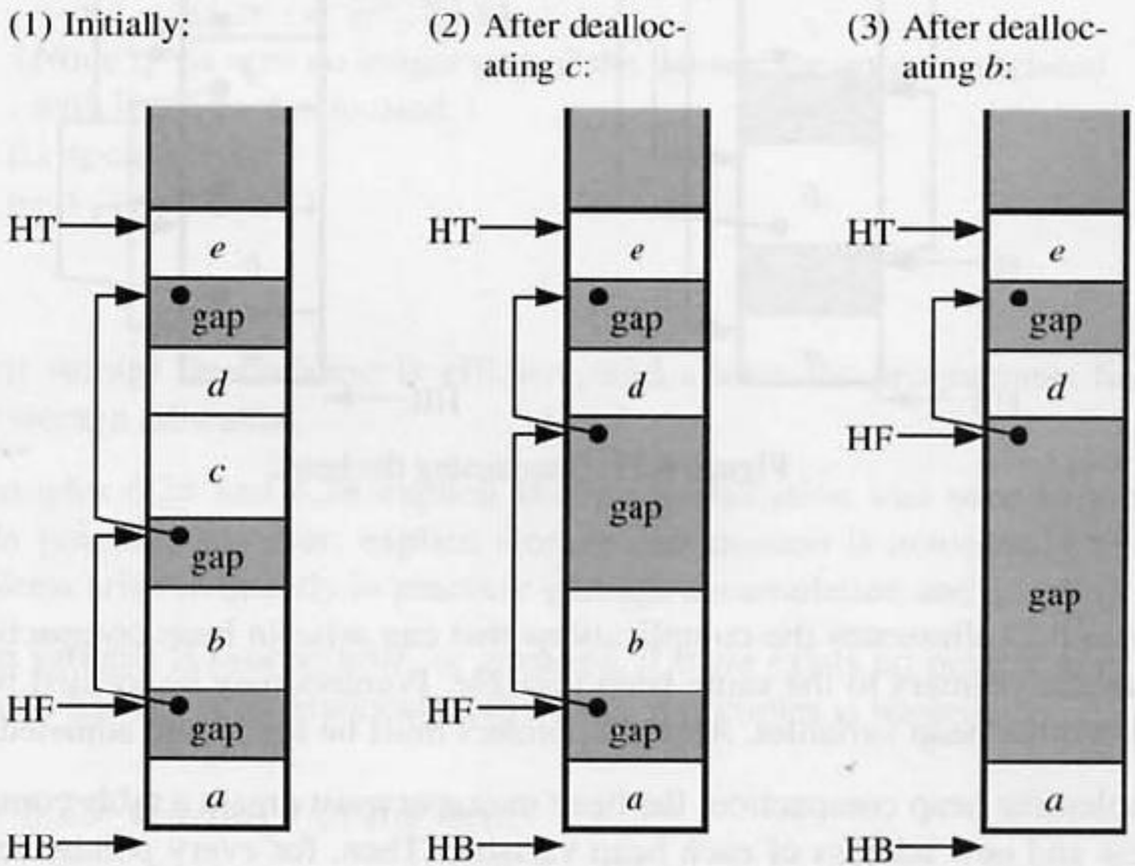

time
lifetime of global variables
lifetime of node with 6
lifetime of node with 'Cu'
lifetime of node with 9
lifetime of node with 'Ag'
lifetime of node with 10
lifetime of node with 'Au'
lifetime of node with 12
lifetime of node with 'Pt'
Program starts
(1)
(2)
(3)
(4)
Program stops



(2) After allocating several heap variables:

(3) After deallocating some heap variables:

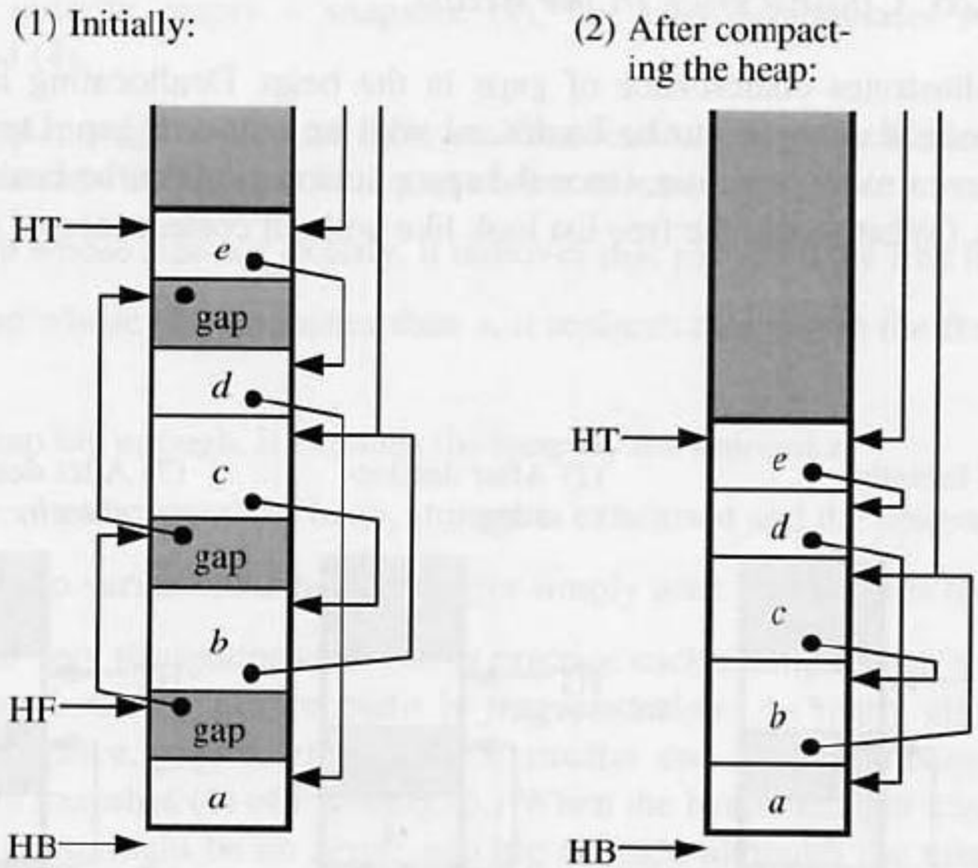
(4) After allocating more heap variables:



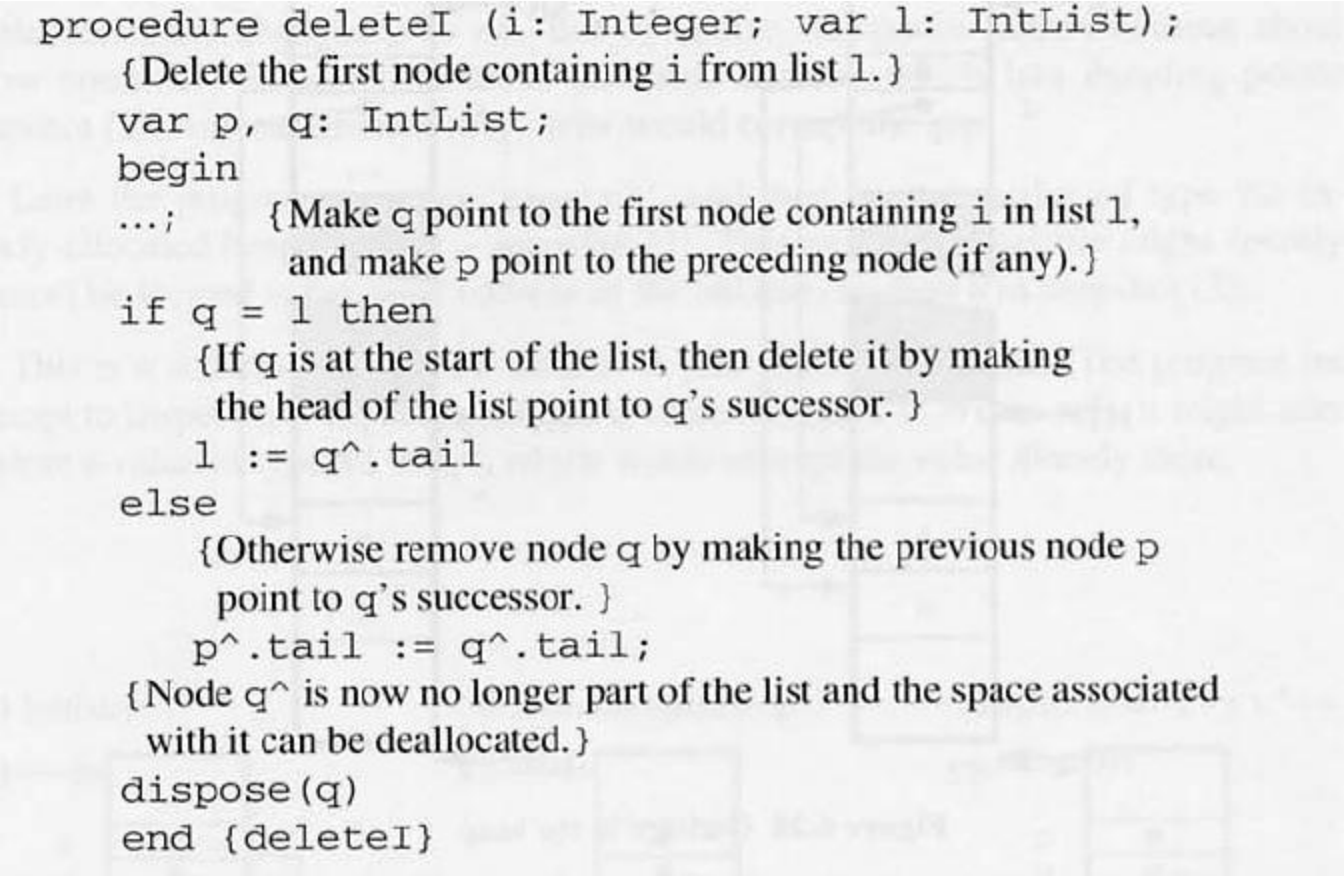
(1) Initially:

(2) After deallocating *c*:

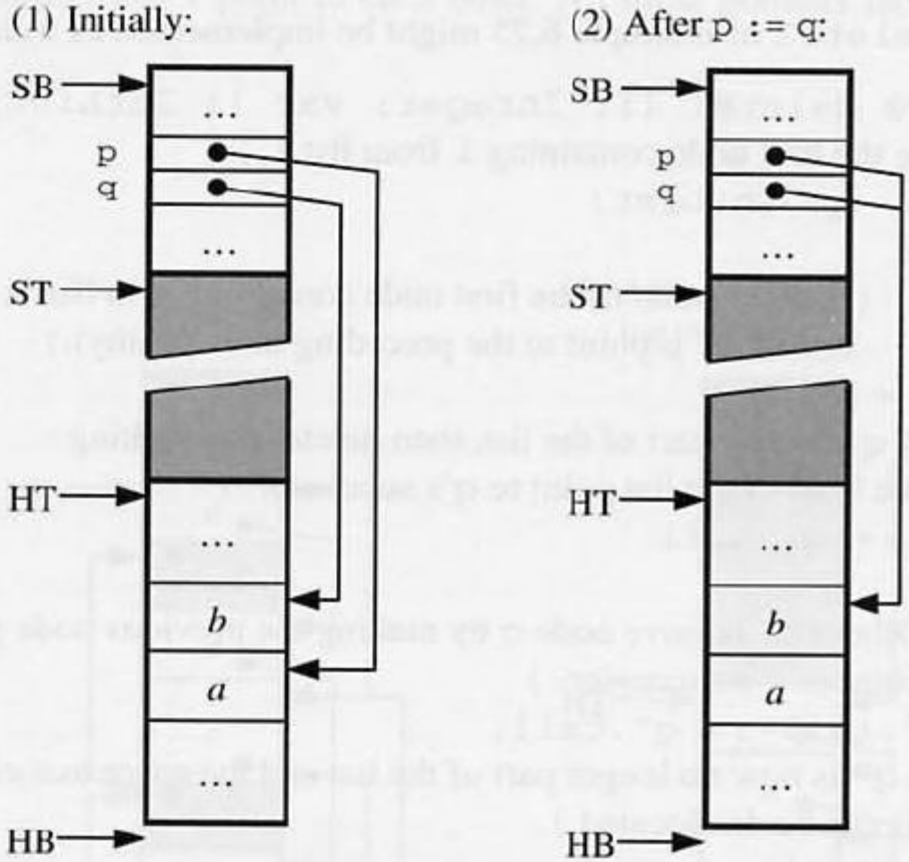
(3) After deallocating *b*:

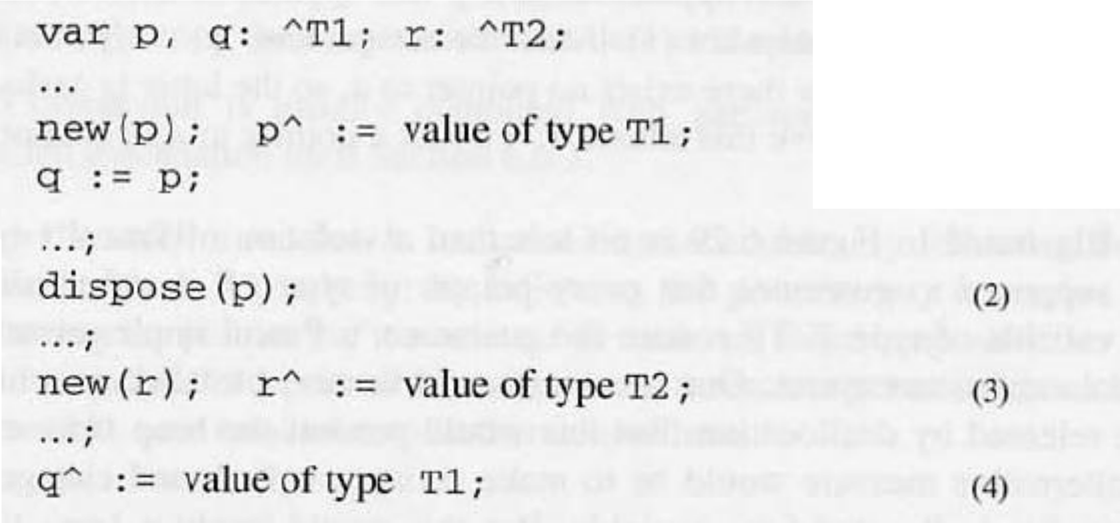


(1) Initially:
(2) After compacting the heap:
HT
HF
HB
e
gap
d
c
gap
b
gap
a
HT
HB
e
d
c
b
a

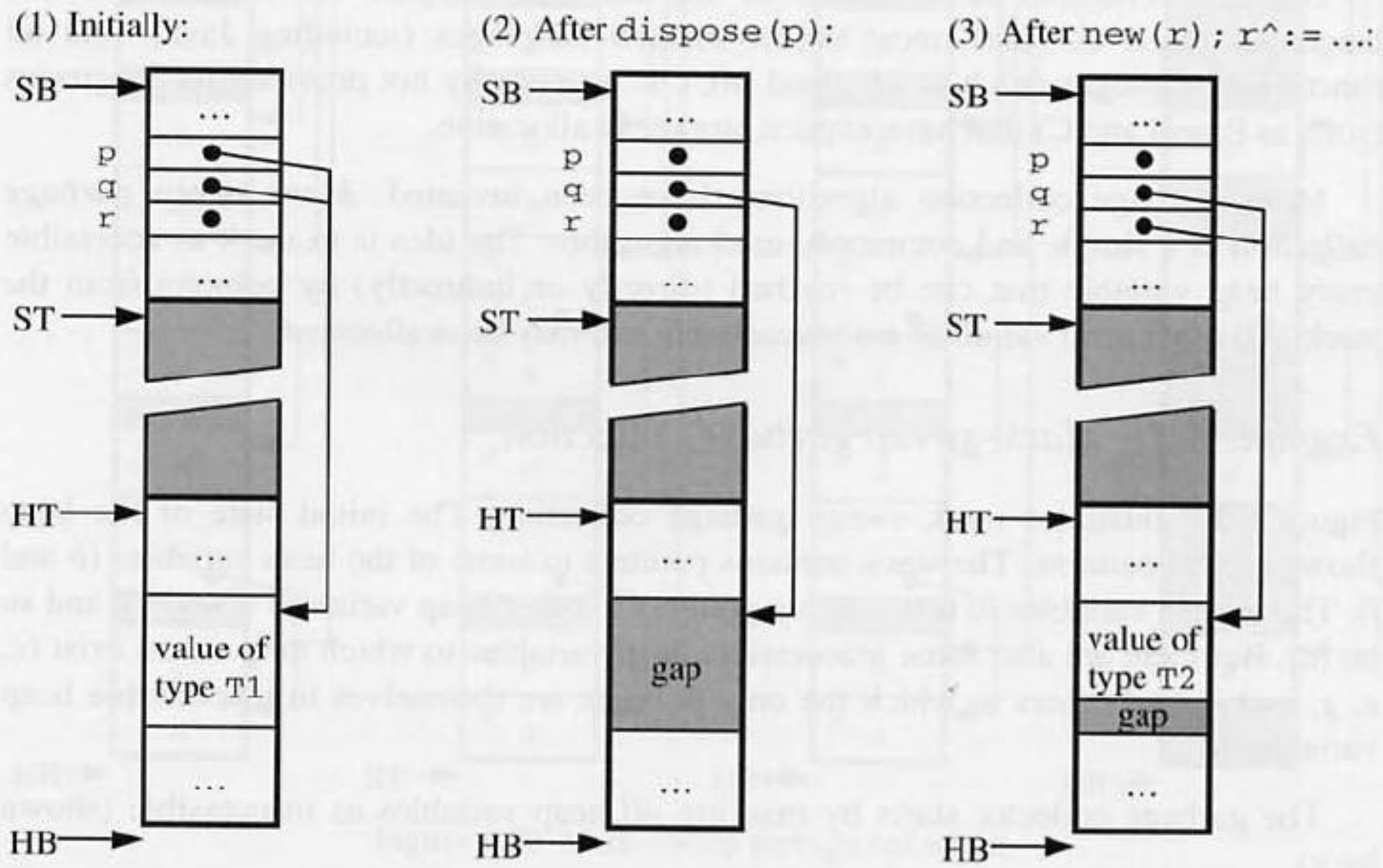


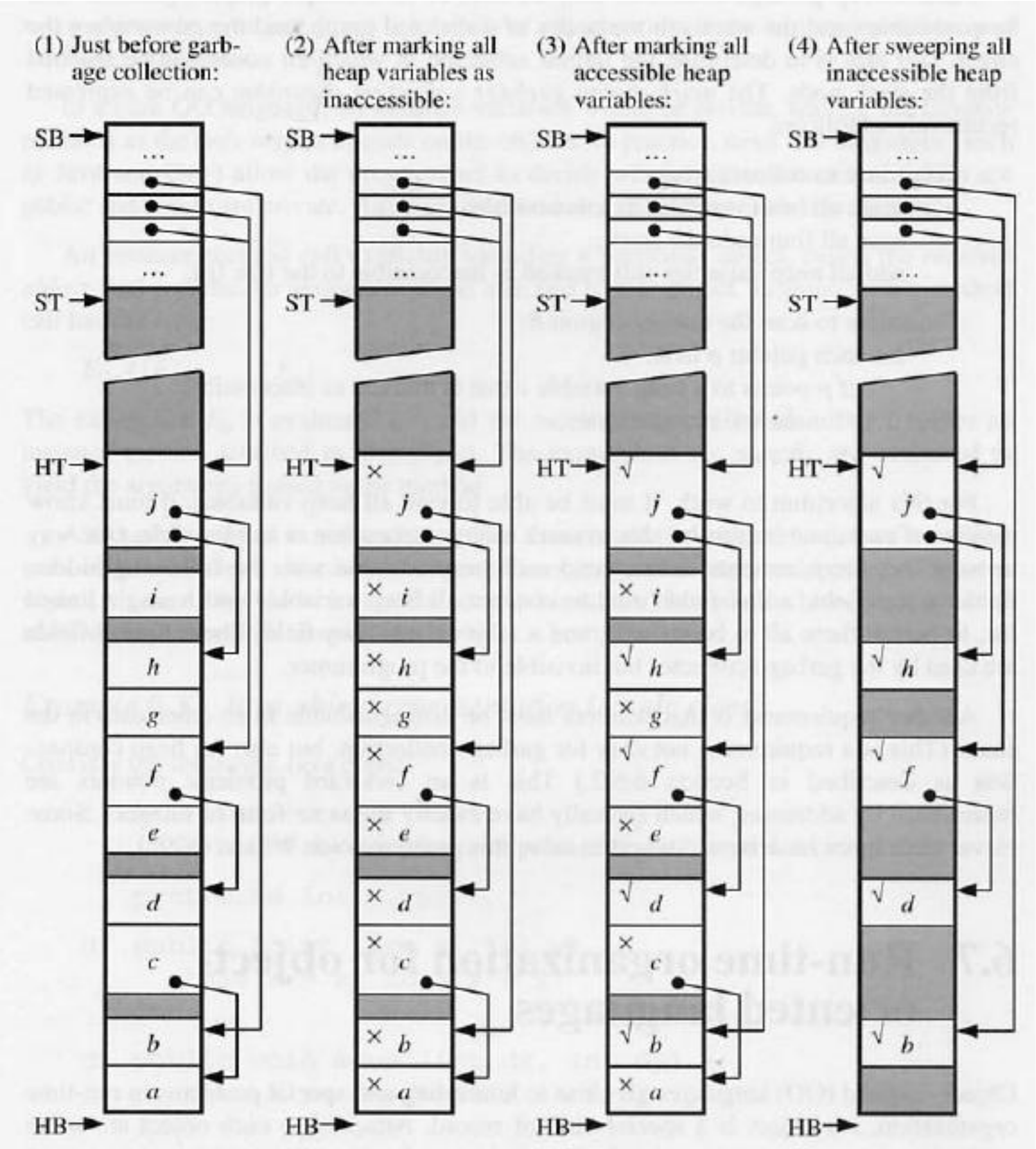
```
procedure deleteI (i: Integer; var l: IntList);
    {Delete the first node containing i from list l.}
    var p, q: IntList;
    begin
    ...;    {Make q point to the first node containing i in list l,
             and make p point to the preceding node (if any).}
    if q = l then
        {If q is at the start of the list, then delete it by making
         the head of the list point to q's successor. }
        l := q^.tail
    else
        {Otherwise remove node q by making the previous node p
         point to q's successor. }
        p^.tail := q^.tail;
     {Node q^ is now no longer part of the list and the space associated
      with it can be deallocated.}
    dispose(q)
    end {deleteI}
```

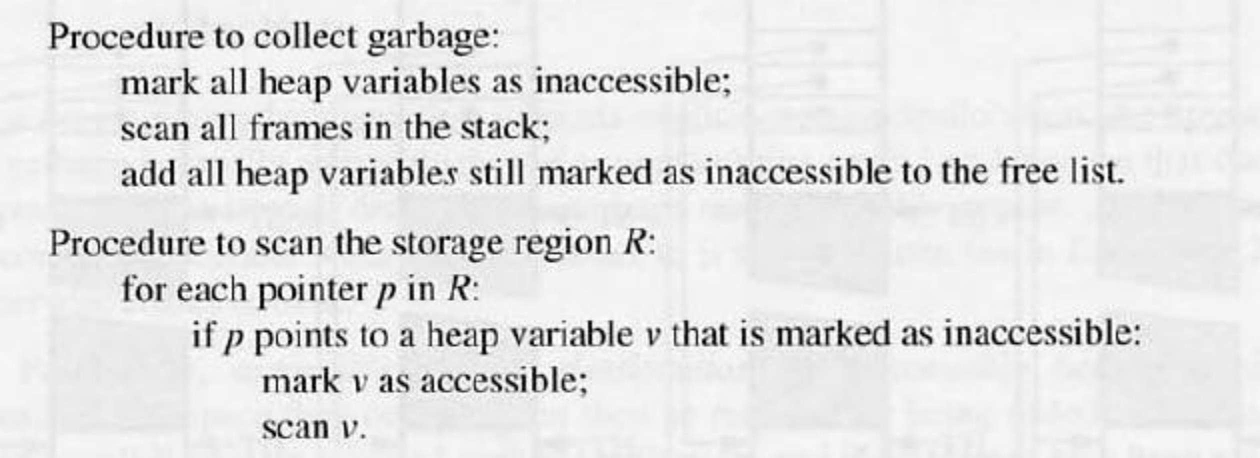

(1) Initially:
SB
p
q
…
…
ST
HT
…
b
a
…
HB
(2) After p := q:
SB
p
q
…
…
ST
HT
…
b
a
…
HB



```
var p, q: ^T1; r: ^T2;
...
new(p);  p^ := value of type T1;
q := p;
...;
dispose(p);                                    (2)
...;
new(r);  r^ := value of type T2;               (3)
...;
q^ := value of type T1;                        (4)
```


(1) Initially:
(2) After dispose(p):
(3) After new(r); r^:= …:
SB
p
q
r
ST
HT
HB
…
value of type T1
gap
value of type T2


(1) Just before garbage collection:
(2) After marking all heap variables as inaccessible:
(3) After marking all accessible heap variables:
(4) After sweeping all inaccessible heap variables:
SB
ST
HT
HB
...
j
i
h
g
f
e
d
c
b
a
×
√



```
Procedure to collect garbage:
    mark all heap variables as inaccessible;
    scan all frames in the stack;
    add all heap variables still marked as inaccessible to the free list.

Procedure to scan the storage region $R$:
    for each pointer $p$ in $R$:
        if $p$ points to a heap variable $v$ that is marked as inaccessible:
            mark $v$ as accessible;
            scan $v$.
```