Geheugenallocatie: statische arrays

Addressering van een array-component $b[i_1 , \ldots , i_k]$:
Genereer code om, at runtime, het relatieve adres *r* van de component t.o.v. het beginadres ρ(b) van de array te berekenen:

$$r = (\hat{\imath}_1 \times d_2 \times d_3 \times \ldots \times d_k + \hat{\imath}_2 \times d_3 \times d_4 \times \ldots \times d_k + \ldots + \hat{\imath}_{k-1} \times d_k + \hat{\imath}_k)$$
$$- (u_1 \times d_2 \times d_3 \times \ldots \times d_k + u_2 \times d_3 \times d_4 \times \ldots \times d_k + \ldots + u_{k-1} \times d_k + u_k)$$

Rekening houdend met de grootte *g* (in plaats van 1) van de componenten heeft dit de vorm

$$h*g - d*g \qquad (\text{waar } d = u_1 \times d^{(1)} + u_2 \times d^{(2)} + \ldots + u_k \times d^{(k)})$$

waar enkel *h* nog at runtime berekend moet worden. Om niet telkens die *d*g* te moeten aftrekken van het eerste deel zullen we werken met een fictief startadres.

Met de notatie $d^{(j)} = \prod_{l=j+1}^{k} d_l$ en met testen op de grenzen:

$$
\begin{aligned}
code_I\ [i_1, \ldots, i_k]\ arr\ \rho \quad = \quad & code_R\ i_1\ \rho;\ \mathbf{chk}\ u_1\ o_1;\ \mathbf{ixa}\ g\ d^{(1)};\\
& code_R\ i_2\ \rho;\ \mathbf{chk}\ u_2\ o_2;\ \mathbf{ixa}\ g\ d^{(2)};\\
& \quad \vdots\\
& code_R\ i_k\ \rho;\ \mathbf{chk}\ u_k\ o_k;\ \mathbf{ixa}\ g\ d^{(k)};\\
& \mathbf{dec}\ \mathrm{a}\ g\ d;
\end{aligned}
$$

where $arr = (g; u_1, o_1, \ldots, u_n, o_n)$

Aangepaste instructie om r gemakkelijk te kunnen berekenen

Instr.	Meaning	Cond.	Results
ixa *q*	*STORE*[*SP* −1] := *STORE*[*SP* −1] + *STORE*[*SP*] ∗*q* *SP* := *SP* −1	(a,i)	(a)

Idee: berekening van geïndexeerd adres. STORE[SP-1] bevat een startadres, STORE[SP] bevat de index van de geselecteerde subarray en q is de grootte van de subarray.

Geheugenallocatie voor dynamische arrays

Dynamische arrays: veralgemeen het schema tot het geval waar de array-grenzen u_i en o_i pas bepaald worden wanneer de array gecreëerd wordt: de grenzen worden gegeven d.m.v. expressies die actuele parameters bevatten (de array is ofwel een parameter ofwel een locale variable)

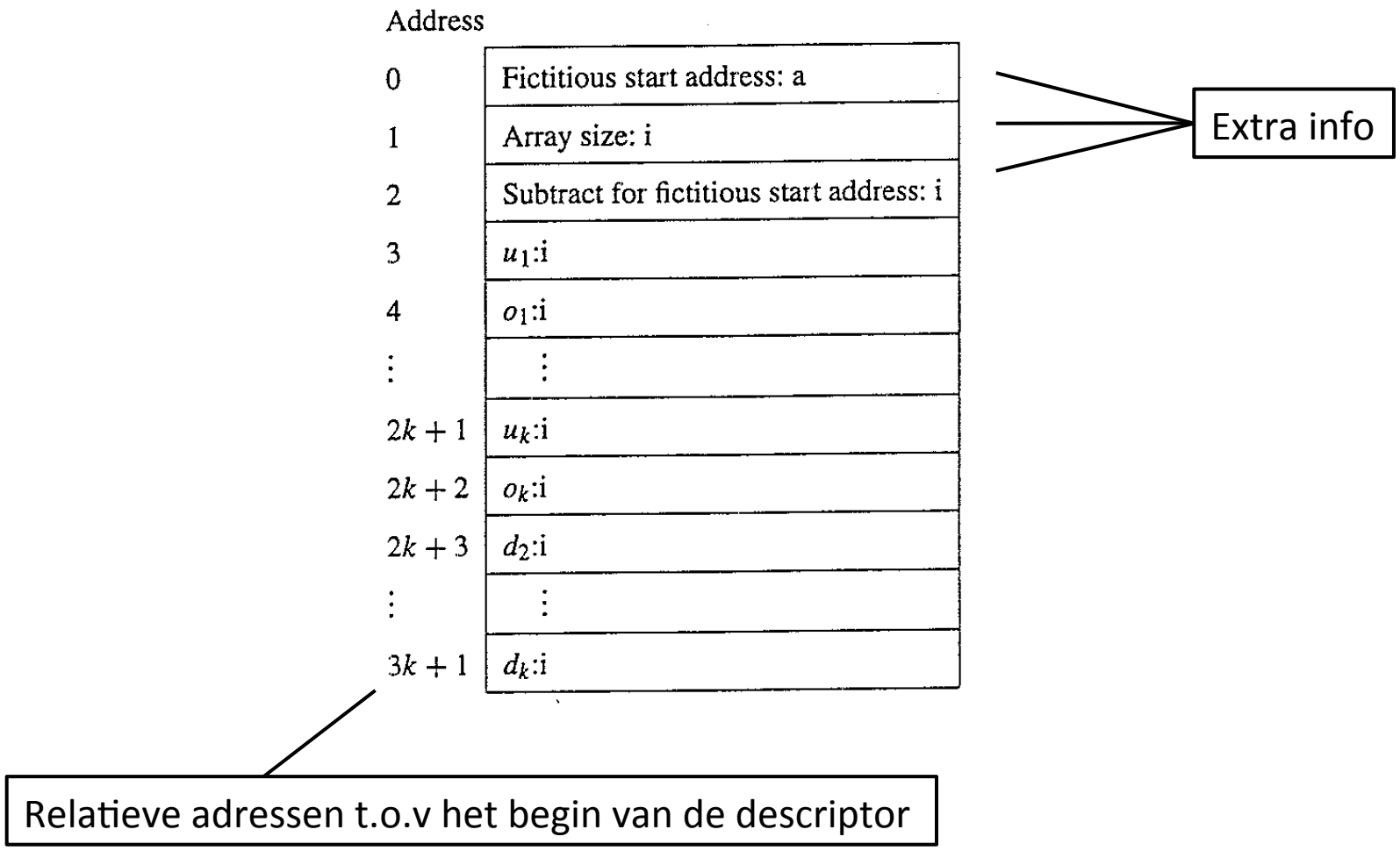
```
procedure pr(val p, q: integer)          twee parameters
        begin
            array a[p+q .. 2q] of real   lokale array-variabele
            …
        end
```

De waarden van de grenzen u_i en o_i (en dus ook de d_i) kunnen nu pas bepaald worden at runtime, als de waarden van de parameters bekend zijn, maar ze blijven wel ongewijzigd zolang de array bestaat, dus ze worden bepaald telkens bij een oproep van pr.

Geheugenallocatie voor dynamische arrays

Oplossing: de array wordt gebruikt met behulp van een array descriptor, een tabel waarin de waarden van u_i ,o_i en d_i opgeslagen worden wanneer de array gecreëerd wordt. De array-componenten zelf worden bewaard in een ander blok geheugenplaatsen. Merk op dat de grootte van de descriptor vast ligt en bekend is at compile time – het volstaat om het aantal dimensies te kennen.

Array descriptor - layout



Fictief start-adres: (start adres) – d×g (remember r = h×g – d×g)

Merk op dat de grootte van de descriptor vast ligt en bekend is at compile time: het volstaat om het aantal dimensies k te kennen.

Fictief startadres = (startadres) – d × g

L-value van a[..] was van de vorm (startadres) + r
met r = h × g – d × g
dat wordt dus nu (Fictief startadres) + h × g.

h wordt nu berekend met een Horner-schema:

$$h = (\ldots((\hat{\imath}_1 \times d_2 + \hat{\imath}_2) \times d_3 + \hat{\imath}_3) \times d_4 + \ldots) \times d_k + \hat{\imath}_k$$

De d_i zijn uiteraard te vinden in de descriptor.

Adresseren van een component

Voor het indexeren van een dynamische array b: $\rho(b)$ is nu de eerste locatie van de descriptor. We gebuiken twee nieuwe code-functies $code_{Ld}$ en $code_{ld}$.

- De functie $code_{Ld}$ genereert enkel een **ldc** instructie die het adres van de array descriptor op de top van de stack laadt.
- De code gegenereerd door $code_{ld}$ dupliceert dit adres, gebruikt het bovenste duplicaat om het fictieve start adres te laden (met een indirectie) en verwijst naar de rest van de array-descriptor indirect met behulp van het onderste duplicaat.

$code_{Ld}\ b[i_1, \ldots, i_k]\ \rho =$

ldc a $\rho(b)$; descriptor address

$code_{Id}\ [i_1, \ldots, i_k]\ g\ \rho$ (static) component size g

$code_{Id}\ [i_1, \ldots, i_k]\ g\ \rho =$

dpl i; duplicate highest stack entry

ind i; fictitious start address

ldc i 0;

$code_R\ i_1\ \rho$; **add** i; **ldd** $2k + 3$; **mul** i;

$code_R\ i_2\ \rho$; **add** i; **ldd** $2k + 4$; **mul** i;

⋮

$code_R\ i_{k-1}\ \rho$; **add** i; **ldd** $3k + 1$; **mul** i;

$code_R\ i_k\ \rho$; **add** i;

ixa g

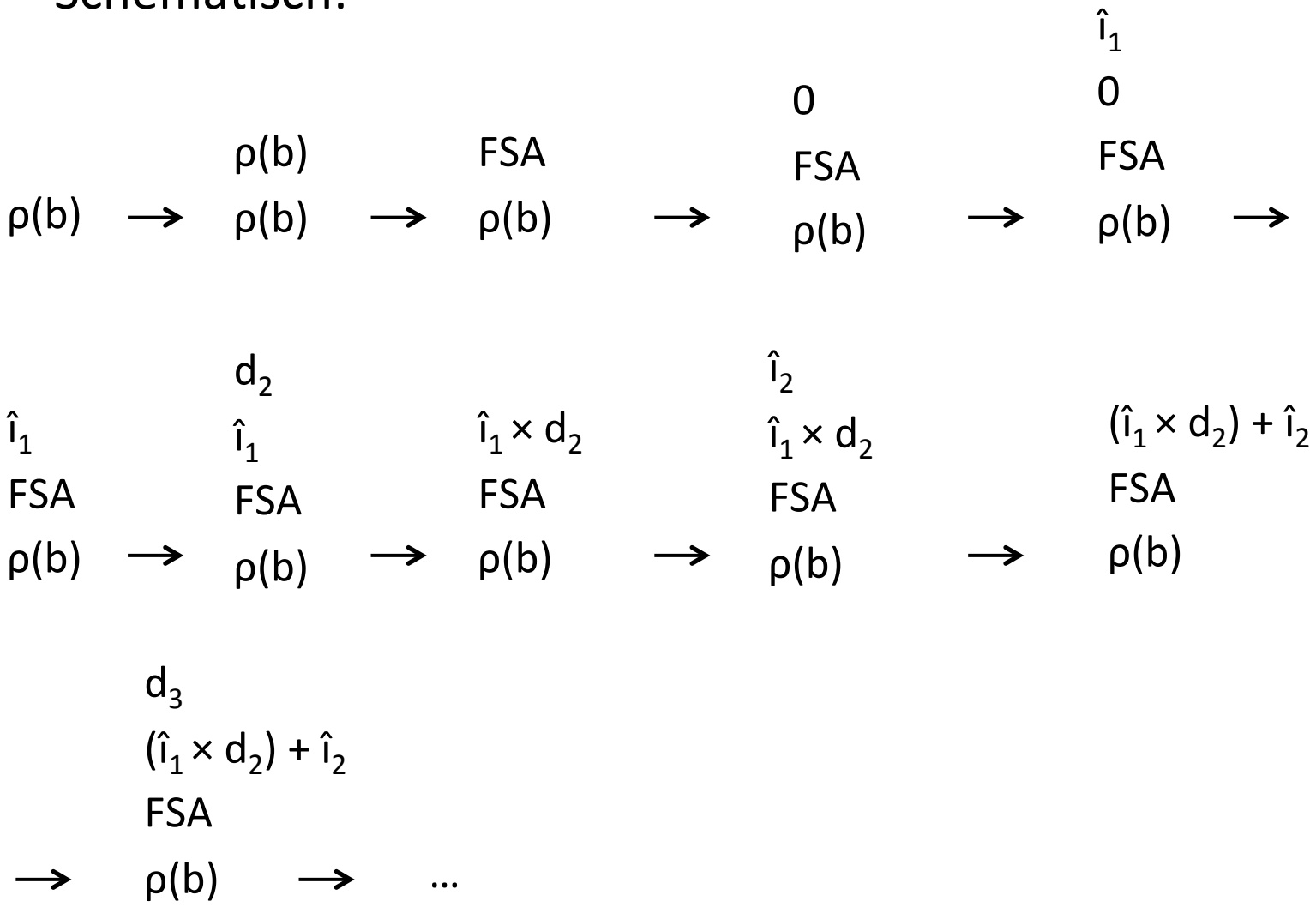
sli a

2k+3: plaats van d_2 in de descriptor

Table 2.9 Instructions for dynamic arrays.

Instr.	Meaning	Cond.	Result
dpl T	$SP := SP + 1$;	(T)	(T, T)
	$STORE[SP] := STORE[SP - 1]$		
ldd q	$SP := SP + 1$;	(a, T_1, T_2)	(a, T_1, T_2, i)
	$STORE[SP] := STORE[STORE[SP - 3] + q]$		
sli T_2	$STORE[SP - 1] := STORE[SP]$;	(T_1, T_2)	(T_2)
	$SP := SP - 1$		

Schematisch:



Geheugenallocatie voor andere datatypes

records, structs, … met velden van mogelijk verschillende types:

var v: record
c_1: t_1;
c_2: t_2;
…
c_k: t_k
end

gebruik: $v.c_2$

Vertaal de veldnamen in relatieve adressen t.o.v. het startadres van de record. Dus om te verwijzen naar het veld c_i : genereer

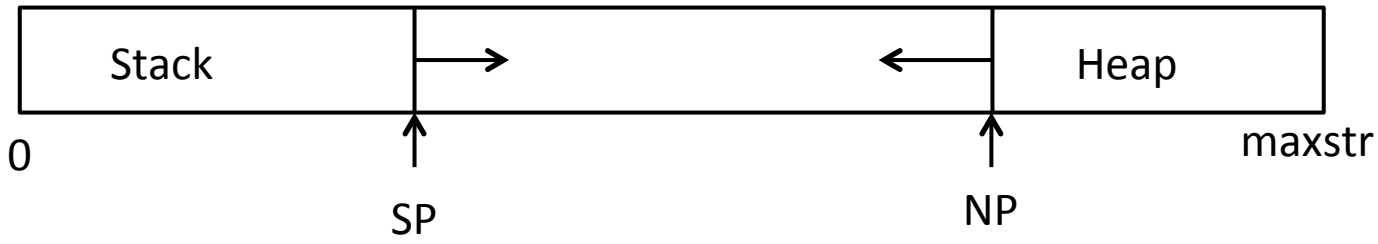
ldc a ρ(v) ; inc a ρ(c_i)

met $\rho(c_i) = \sum_{j=1}^{i-1} size(t_j)$

Pointers en dynamische geheugenallocatie

new, malloc , … : creatie van naamloze objecten, bv. voor lijststructuren.

Uitbreiding van de P-machine: heap op het eind van de geheugen-array

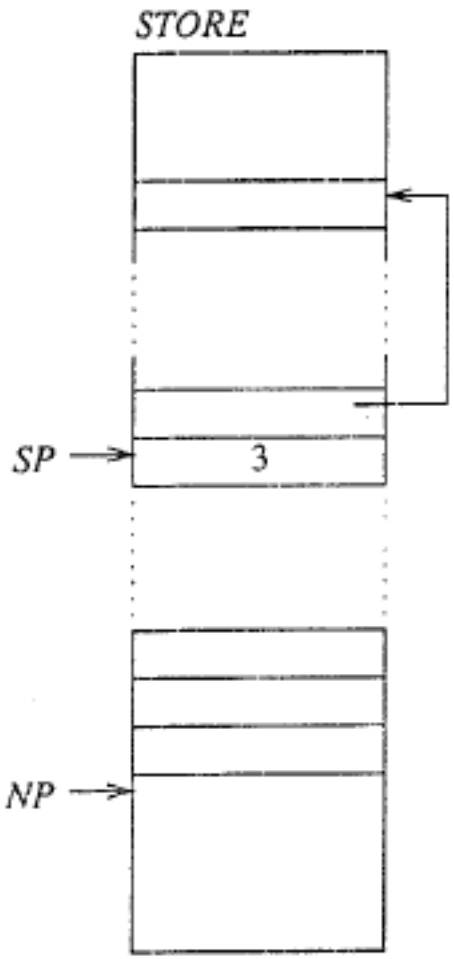


Bij het starten van een procedure kan de maximale waarde van SP berekend worden: EP (extreme stack pointer)

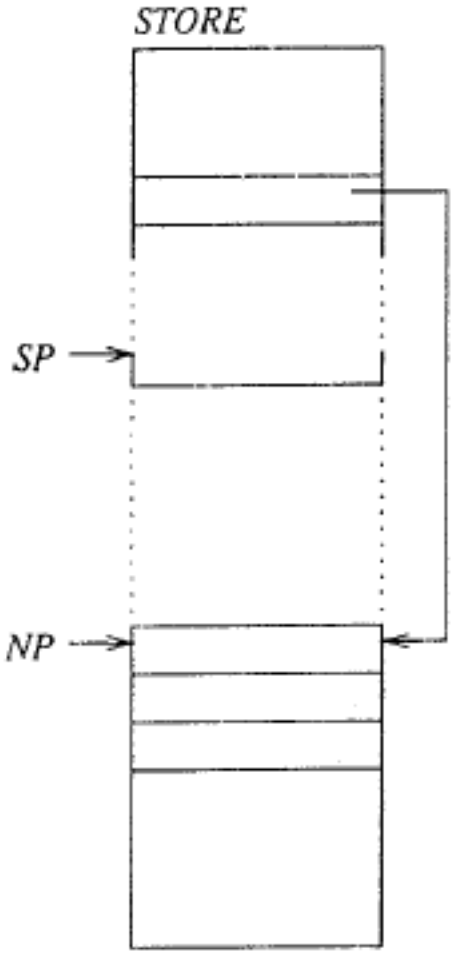
Table 2.10 The **new** instruction.

Instr.	Meaning	Cond.	Result
new	**if** $NP - STORE[SP] \leq EP$ **then** *error*('store overflow') **else** $NP := NP - STORE[SP]$; $STORE[STORE[SP - 1]] := NP$; $SP := SP - 2$ **fi**;	(a,i)	

Voor de uitvoering: bovenaan de stack staat de grootte van het te alloceren blok en het adres waarin het adres van het nieuwe blok beschikbaar komt



voor



na

Vertaling van new(x) van de source-taal:

$code(\mathbf{new}(x))\,\rho =$
$\quad \mathbf{ldc}\ \mathrm{a}\ \rho(x);\ \mathbf{ldc}\ \mathrm{i}\ size(\mathrm{t});\ \mathbf{new}$ if x is a variable of type $\uparrow t$

Voor een willekeurige combinatie van indices, dereferencing, veldnamen:

$code_L(xr)\ \rho = \mathbf{ldc}\ \mathrm{a}\ \rho(x);\ code_M(r)\ \rho$ for name x

$code_M(.xr)\ \rho = \mathbf{inc}\ \mathrm{a}\ \rho(x);\ code_M(r)\ \rho$ for name x

$code_M(\uparrow r)\ \rho = \mathbf{ind}\ \mathrm{a};\ code_M(r)\ \rho$

$code_M([i]r)\ \rho = code_{Id}[i]\ g\ \rho;\ code_M(r)\ \rho$ where g is the component size of the indexed array

$code_M(\varepsilon) = \varepsilon$

(M van modify: $code_M$ wijzigt de rest r)

Example 2.4 Suppose we have the following declaration.

```
type  t =  record
           a: array[--5.. + 5, 1..9] of integer
           b: ↑ t
           end;
var   i, j : integer;
      pt   : ↑ t;
```

Assuming that $\rho(i) = 5$, $\rho(j) = 6$ and $\rho(pt) = 7$, the variable denotation $pt \uparrow .b \uparrow .a[i + 1, j]$ is compiled as follows:

Example 2.4 Suppose we have the following declaration.

```
type  t =   record
            a: array[--5.. + 5, 1..9] of integer
            b: ↑ t
            end;
var   i, j  : integer;
      pt    : ↑ t;
```

Assuming that $\rho(i) = 5$, $\rho(j) = 6$ and $\rho(pt) = 7$, the variable denotation $pt \uparrow .b \uparrow .a[i + 1, j]$ is compiled as follows:

ldc a 7;	Load address of *pt*
ind a;	Load start address of record
inc a 99;	Compute start address of record component *b*
ind a;	Dereference pointer
inc a 0;	Start address of component *a*
$code_{ld}[i + 1, j]$ 1 ρ;	As in Example 2.3

Procedures + Parameters

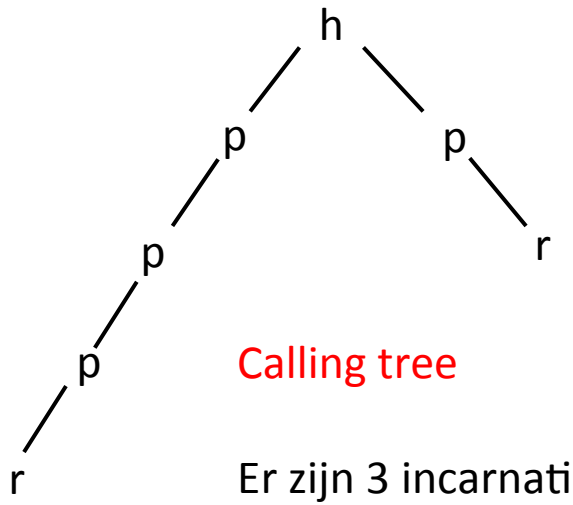
Procedure declaratie:

- naam
- specificatie van de formele parameters (value, variable)
- locale declaraties
- body
- (resultaat)

De oproepen in een uitvoering kunnen voorgesteld worden in een calling tree

De uitvoering komt dan overeen met een depth-first doortocht daarvan.

```
program h;
    var i: integer;
    proc p
        var x: real;
        proc r
        | …
        |
        if i > 0
           then  i := i - 1; p
           else r
        fi
     i := 2;
     p;
     p
end
```



Calling tree

Er zijn 3 incarnaties van p op het incarnatiepad van de eerste call naar r

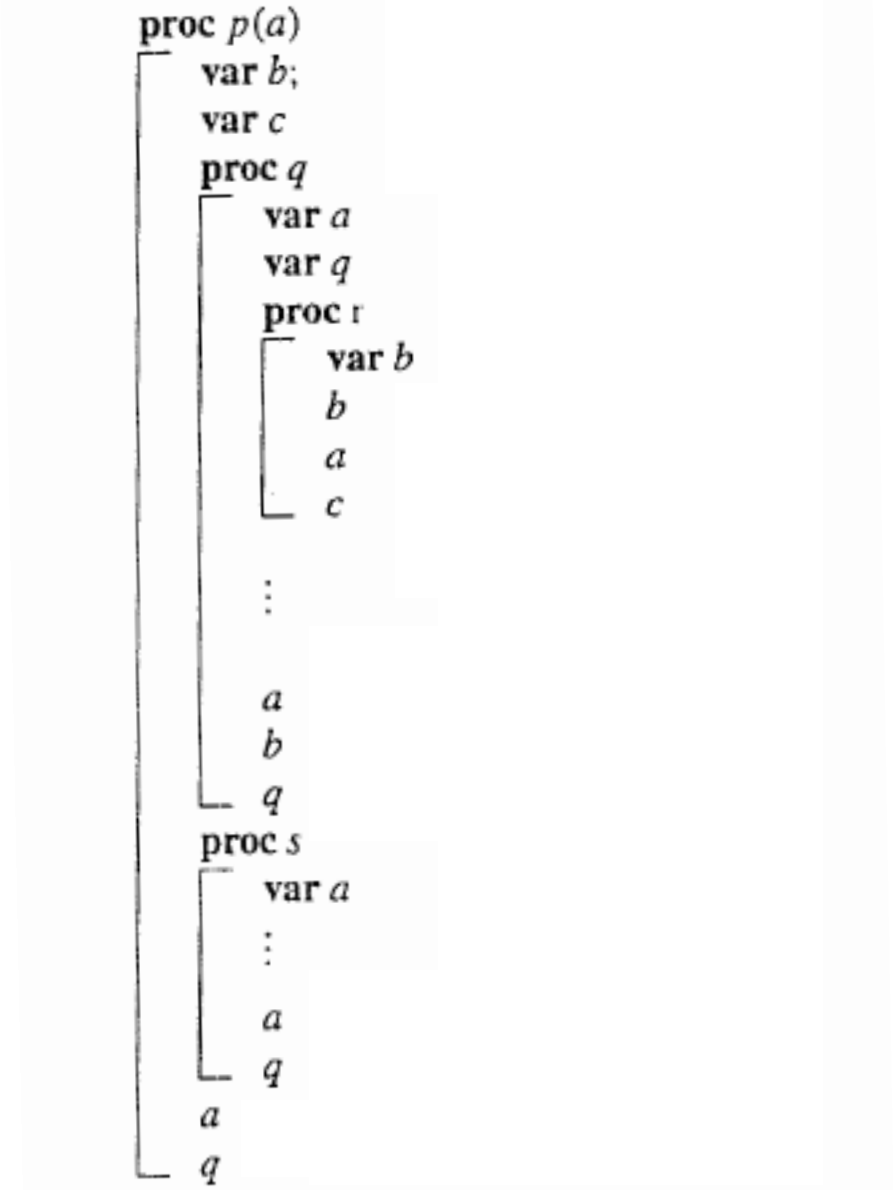
Elk daarvan heeft zijn eigen versie van de locale variabelen van p (zoals x)

p,r kunnen ook niet-locale variabelen gebruiken (zoals i)

Voor elke applied occurrence moet er een unieke defining occurrence te bepalen zijn

Er zijn scope rules nodig voor niet-locale namen

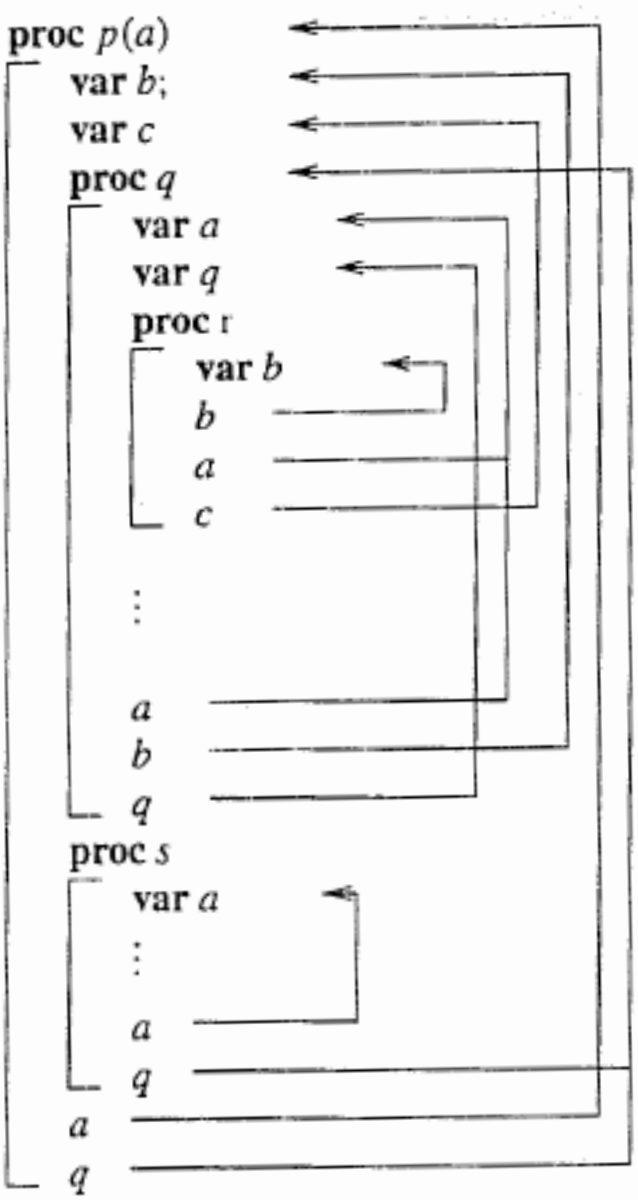
Meest gebruikte regel: voor een applied occurrence, zoek een defining occurrence in omsluitende programma-eenheden, tot er een gevonden is (= statische binding)



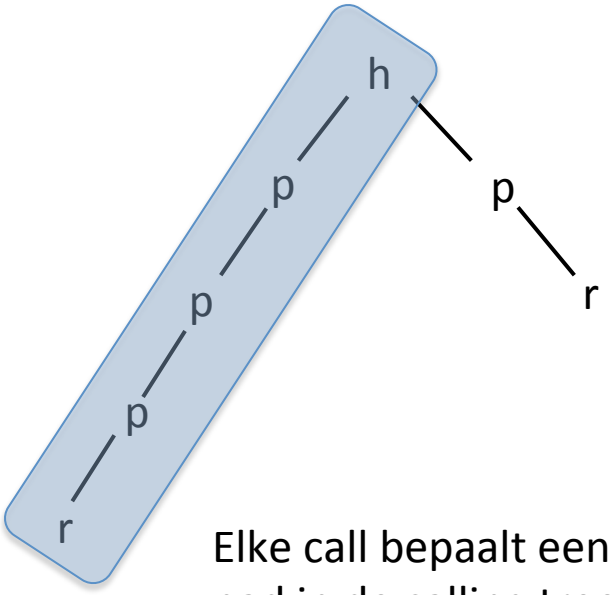
Voor elke applied occurrence moet er een unieke defining occurrence te bepalen zijn

Er zijn scope rules nodig voor niet-locale namen

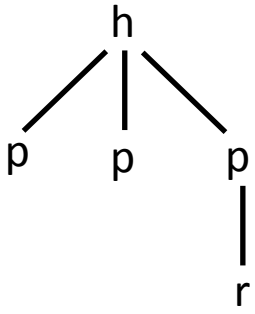
Meest gebruikte regel: voor een applied occurrence, zoek een defining occurrence in omsluitende programma-eenheden, tot er een gevonden is (= statische binding)



Om de juiste incarnatie van een niet-locale naam te vinden: zoek de laatste incarnatie van de omsluitende programma-eenheid (static predecessor) in de tree of static predecessors (evt meerdere stappen)
(zal voorgesteld worden door de statische links SL)



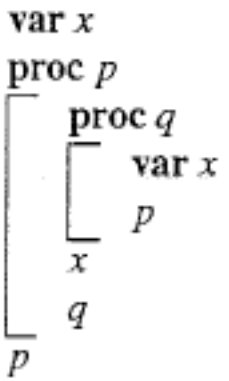
Elke call bepaalt een pad in de calling tree



Tree of static predecessors, voor het aangeduide pad

Alternatief: dynamische binding: defining occurrence zoeken via de laatst geactiveerde programma-eenheid

Example 2.7



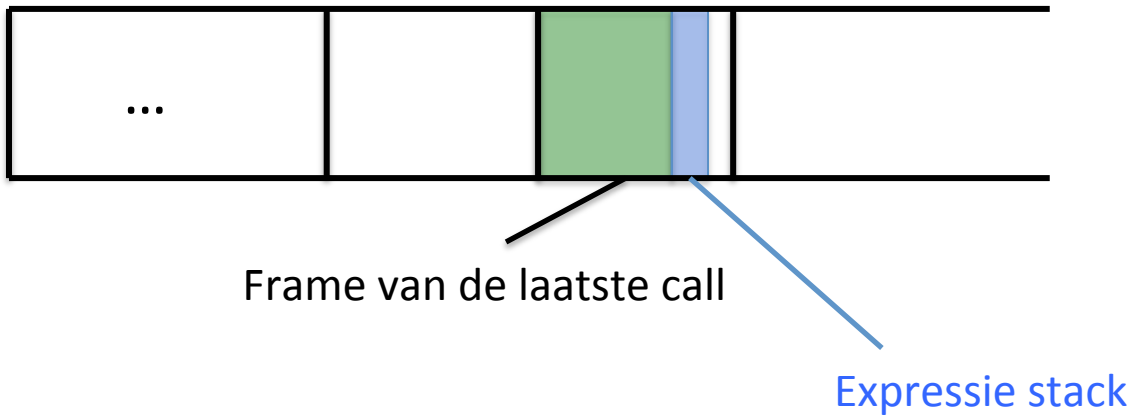
(1) Call to p (from outside p). For static and dynamic binding the applied occurrence of x refers to the external declaration.

(2) Call to p (in q). For static binding the applied occurrence of x refers to the external declaration of x; for dynamic binding, it refers to the declaration of x in q. It involves the most recently created incarnation of x. □

Geheugenorganizatie voor procedures

Elke call naar een procedure krijgt zijn eigen frame, een blok geheugen waarin de opgeroepen procedure haar locale data vindt, d.w.z. data die alleen relevant is voor de call.

Deze frames worden op een stack geplaatst; dus de stack heeft nu een geneste structuur: een "macroscopische" structuur (stack van frames) en een "microscopische" (stack voor expressie evaluatie)



Geheugenorganizatie voor procedures: “macro” stack met een stack frame voor elke incarnatie. Locale stack op de top van de frame.

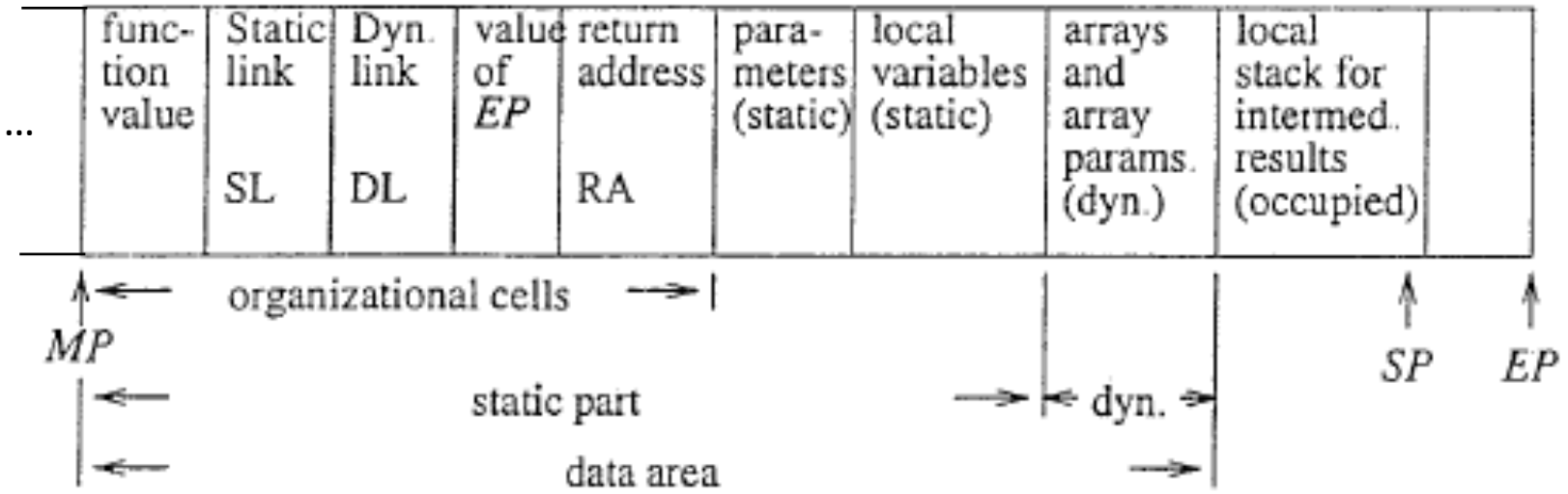
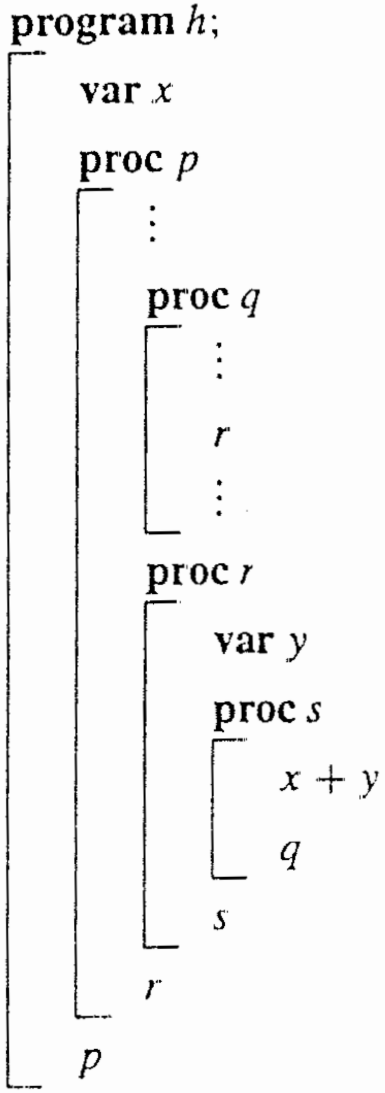


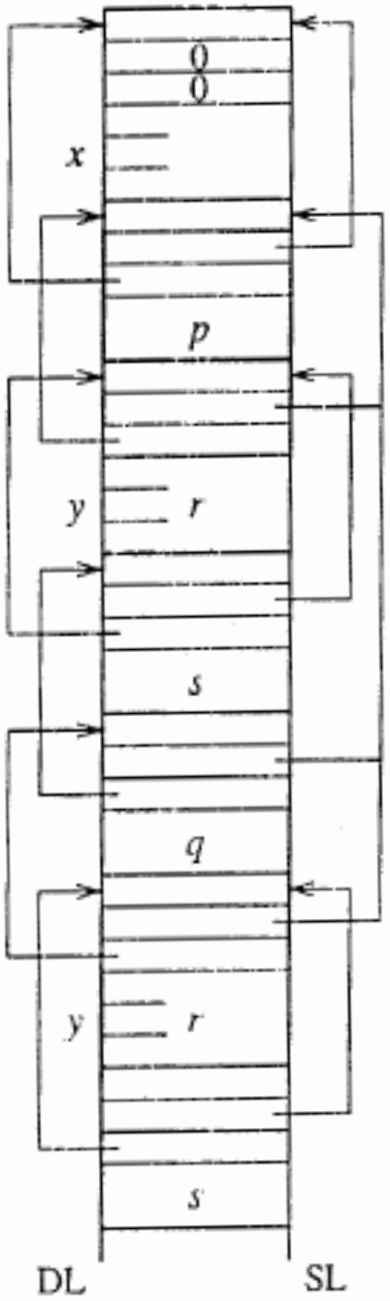
Figure 2.10 The highest stack frame.


program h;
var x
proc p
proc q
r
proc r
var y
proc s
x + y
q
s
r
p

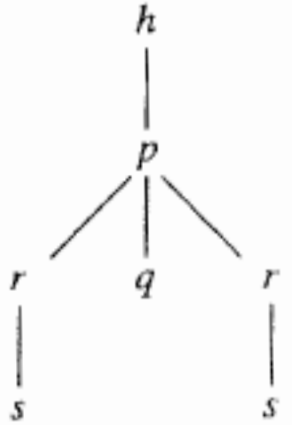
DL: incarnatiepad
SL : statische predecessors

```
program h;
    var x
    proc p
        ⋮
        proc q
            ⋮
            r
            ⋮
        proc r
            var y
            proc s
                x + y
                q
            s
        r
    p
```

(a)



(b)



Example 2.10 The nesting depths in Figure 2.11 are:

Defining occurrence		Applied occurrence	
p	1	p	1
q	2	q	4
r	2	r (in p)	2
s	3	r (in q)	3
		s	3

Gebruik van niet-locale variabelen

nesting diepte : aantal omsluitende programma-eenheden.

Voor een applied occurrence van een niet-locale variabele is de waarde te vinden door de statische link p keer te volgen, waar p het verschil in nesting diepte is tussen de applied occurrence en de overeenkomstige defining occurrence

Table 2.11 Loading and storing for a given difference in nesting depths p and a given relative address q. **lod** loads values, **lda** loads addresses.
$base(p,a) = \textbf{if } p = 0 \textbf{ then } a \textbf{ else } base(p-1, STORE[a+1])$.

Instr.	Meaning
lod $T\ p\ q$	$SP := SP + 1;$
	$STORE[SP] := STORE[base(p, MP) + q]$
lda $p\ q$	$SP := SP + 1;$
	$STORE[SP] := base(p, MP) + q$
str $T\ p\ q$	$STORE[base(p, MP) + q] := STORE[SP];$
	$SP := SP - 1$

Volgen van de SL is ”ingebouwd”

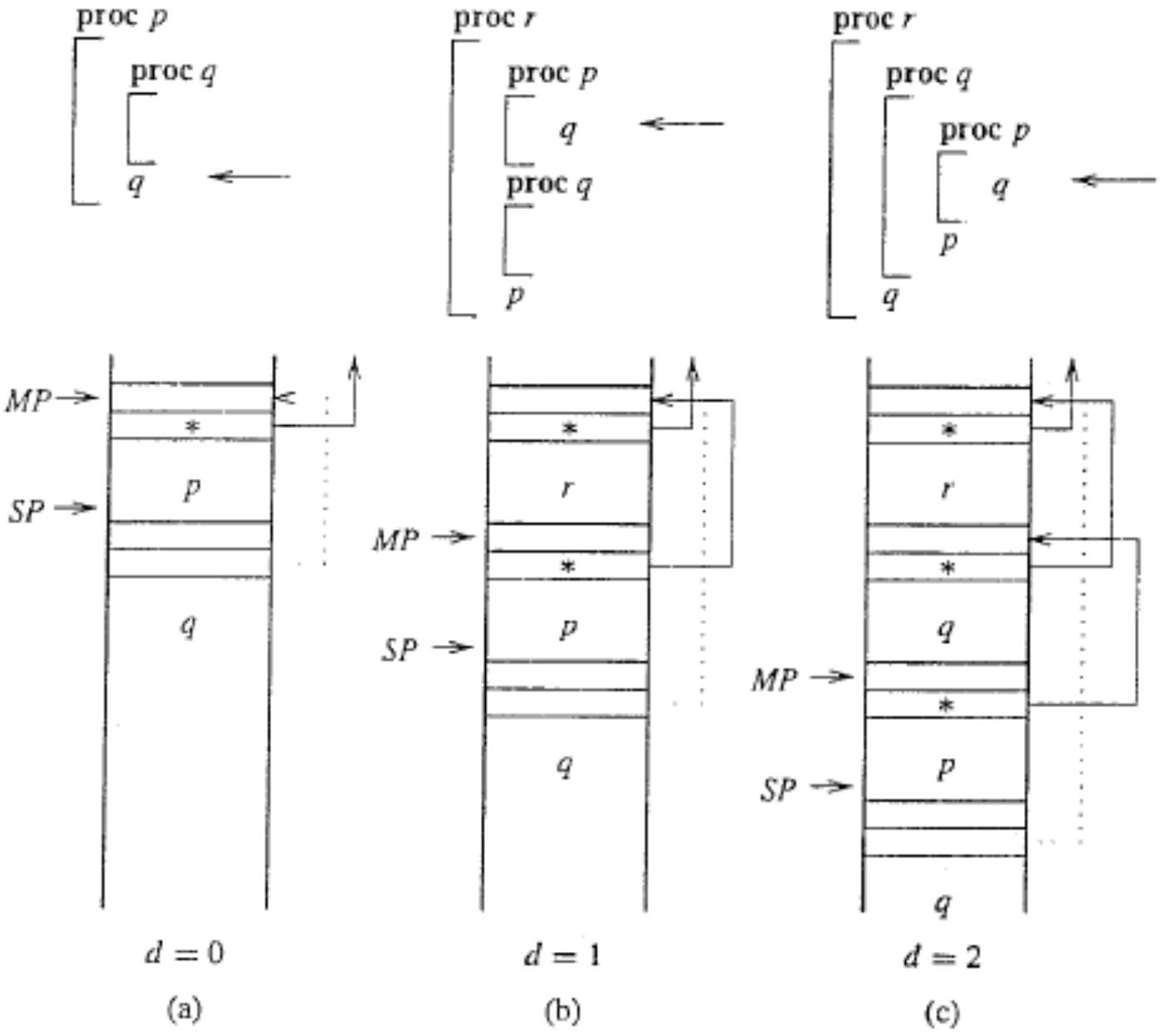
Zetten van de statische link (bij een call)

Laat procedure q gedeclareerd zijn op nesting diepte n. Dan kan q opgeroepen worden

- In de body van de programma-eenheid die q (direct) omsluit; dit is dus op nesting diepte n
- In procedures waarvan de declaraties genest zijn in q en waar q niet verborgen is door een nieuwe declaratie (van q). Deze calls hebben een nesting diepte groter dan n.

De correcte waarde van SL die ingevuld moet worden in het frame van de nieuwe call (van q) wordt verkregen door de statische links d keer te volgen, startend van de frame van de oproepende procedure, waar d het verschil is in nesting diepte tussen de call en de declaratie van q. Het is duidelijk dat, als $d \neq 0$, de oproepende procedure de call moet omsluiten.

We illustreren 3 gevallen, de stippellijn is de nieuwe statische link



d: difference in nesting depth