Vervolg: compilatie van een imperatieve taal

Nadruk op de HL features:

- Control flow
- Datatypes, inclusief arrays en structs
- Procedures + parameters
- Scoping

Centrale vraag:

Waar vinden de VM instructies hun data?

In het voorbeeld was dat triviaal omdat ook de doeltaal met symbolische variabelen werkte

Compilatie van een imperatieve taal (Pascal) naar een VM (P-machine)

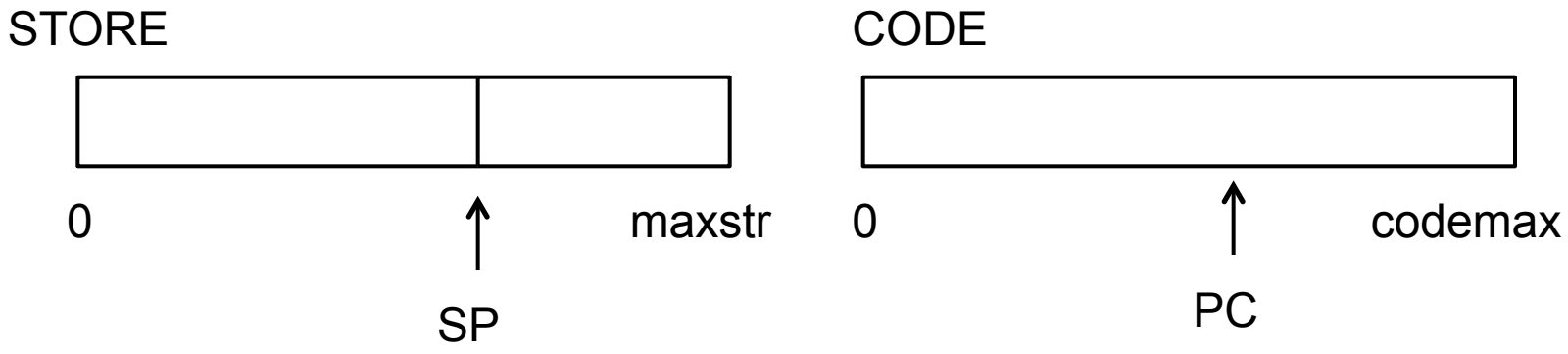
Features van een imperatieve taal:

- Variabelen
- State = mapping van variabelen op waarden
- State wordt gewijzigd d.m.v. assignments
- Expressions worden geëvalueerd
- Variabelen hebben een scope
- Complexe datastructuren (arrays,…) + types
- Expliciete specificatie van control flow (if-then-else, while, …)
- Procedures, met locale variabelen, parameters,…
- Variabelen kunnen meerdere incarnaties hebben (meerdere versies die tegelijk bestaan)

Architectuur van de P (Pascal) Machine – een VM op maat van de te vertalen taal. Vormt ook de definitie van de betekenis van die taal

Definitie: organizatie van het geheugen + instructies

Layout van het geheugen: een deel voor data, een deel voor code



Main loop

PC eerst aanpassen maakt de behandeling van jumps wat eenvoudiger

```
do
    PC := PC + 1
    execute instruction at location CODE[PC - 1]
od
```

De instructies van de P-Machine

- P-machine instructies zijn macro's bestaande uit meer elementaire instructies voor de P-machine. We introduceren ze stapsgewijs, voor de verschillende instructies van Pascal.

- We maken een onderscheid tussen L-values (locaties, waaraan waarden toegekend kunnen worden) en R-values (e.g. waarde van een expressie). Afhankelijk van de context zal een variabele dus omgezet moeten worden in ofwel een L-value , ofwel een R-value.

Formeel wordt de vertaling gedefiniëerd met behulp van de *code*-functies; die zetten hun argument (een element van de brontaal) om in een rij P-machine instructies.

Van de code-functies definiëren we meerdere varianten, te gebruiken naargelang de context: $code_L$,$code_R$ en *code*, voor het genereren van code die bij uitvoering ofwel een L-value of een R-value moet produceren, of voor het geval dat irrelevant is. Later hebben we nog een paar andere varianten nodig

We behandelen eerst expressies en assignments: we geven de eerst de relevante P-machine instructies en dan de code-functies.

De instructies:

1. Rekenkundige operatoren
2. Logische connectieven
3. Relationele operatoren
4. Transport van en naar het geheugen

Instructies voor evaluatie van expressies op de stack

Instr	Meaning	Cond	Result
add *N*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $+_N$ *STORE*[*SP*]; *SP* := *SP*−1	(*N*, *N*)	(*N*)
sub *N*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $-_N$ *STORE*[*SP*]; *SP* := *SP*−1	(*N*, *N*)	(*N*)
mul *N*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $*_N$ *STORE*[*SP*]; *SP* := *SP*−1	(*N*, *N*)	(*N*)
div *N*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $/_N$ *STORE*[*SP*]; *SP* := *SP*−1	(*N*, *N*)	(*N*)
neg *N*	*STORE*[*SP*] := $-_N$ *STORE*[*SP*]	(*N*)	(*N*)
and	*STORE*[*SP*−1] := *STORE*[*SP*−1] *and* *STORE*[*SP*]; *SP* := *SP*−1	(b, b)	(b)
or	*STORE*[*SP*−1] := *STORE*[*SP*−1] *or* *STORE*[*SP*]; *SP* := *SP*−1	(b, b)	(b)
not	*STORE*[*SP*] := *not* *STORE*[*SP*]	(b)	(b)
equ *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $=_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)
geq *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $\geq_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)
leq *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $\leq_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)
les *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $<_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)
grt *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $>_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)
neq *T*	*STORE*[*SP*−1] := *STORE*[*SP*−1] $\neq_T$ *STORE*[*SP*]; *SP* := *SP*−1	(*T*, *T*)	(b)

Types van de elementen boven op de stack

i: integer
r: real
b: boolean
a: adres
N: numeriek type
T: willekeurig type

(N,N): twee keer hetzelfde numerieke type

Instructies: **load** (van store naar stack)
store (van stack naar store)

Table 2.2 Store and load instructions. **ldo** loads from a location given by an absolute address, **ldc** loads a constant given in the instruction, **ind** loads indirectly using the highest stack location, **sro** stores in a location given by an absolute address, and **sto** stores in a location addressed by the second-highest location on the stack.

Instr.	Meaning	Cond.	Result
ldo *T q*	*SP* := *SP*+1; *STORE*[*SP*] := *STORE*[*q*]	*q* ∈ [0,*maxstr*]	(*T*)
ldc *T q*	*SP* := *SP*+1; *STORE*[*SP*] := *q*	Type(*q*) = *T*	(*T*)
ind *T*	*STORE*[*SP*] := *STORE*[*STORE*[*SP*]]	(a)	(*T*)
sro *T q*	*STORE*[*q*] := *STORE*[*SP*]; *SP* := *SP*−1	(*T*) *q* ∈ [0,*maxstr*]	
sto *T*	*STORE*[*STORE*[*SP*−1]] := *STORE*[*SP*]; *SP* := *SP*−2	(a, *T*)	

Definitie van de *code*-functie(s) voor de assignment instructie.

Omdat *code* code moet produceren waarin variabelen omgezet zijn in adressen, heeft die functie niet 1 maar 2 parameters:

- het te vertalen programmadeel
- een environment ρ; dit is een functie met voor elke variabele het corresponderende (relatieve) adres. ρ komt overeen met de symbooltabel, maar dan wel uitgebreid (type + adres, voor elke variabele)

Schema voor assignment x := y voor simpele variabelen x en y :

Bereken L-value van x
Bereken R-value van y
sto i

Code voor expressies en assignment

Uitvoering plaatst het resultaat op de stack, en laat de stack voor de rest ongewijzigd

Notatie: schrijf f x y ipv f(x,y)

code$_L$ voor x
code$_R$ voor e

Table 2.3 The compilation of assignments

Function		Condition
$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **equ** T	$\text{Type}(e_1) = \text{Type}(e_2) = T$
$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **neq** T	$\text{Type}(e_1) = \text{Type}(e_2) = T$
⋮		
$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **add** N	$\text{Type}(e_1) = \text{Type}(e_2) = N$
$code_R(e_1 - e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **sub** N	$\text{Type}(e_1) = \text{Type}(e_2) = N$
$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **mul** N	$\text{Type}(e_1) = \text{Type}(e_2) = N$
$code_R(e_1/e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **div** N	$\text{Type}(e_1) = \text{Type}(e_2) = N$
$code_R(-e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	$\text{Type}(e) = N$
$code_R\ x\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x variable identifier of type T
$code_R\ c\ \rho$	= **ldc** $T\ c$	c constant of type T
$code(x := e)\ \rho$	$= code_L\ x\ \rho;\ code_R\ e\ \rho;$ **sto** T	x variable identifier
$code_L\ x\ \rho$	= **ldc a** $\rho(x)$	x variable identifier

Stel dat we in een programma met integer variabelen a,b, c het assignment a := (b + (b × c)) willen vertalen, en dat voor ρ geldt: ρ(a) = 5, ρ(b) = 6, ρ(c) = 7. Dan is de vertaling dus het volgende:

Stel dat we in een programma met integer variabelen a,b, c het assignment a := (b + (b × c)) willen vertalen, en dat voor ρ geldt: ρ(a) = 5, ρ(b) = 6, ρ(c) = 7. Dan is de vertaling dus het volgende:

$code(a := (b + (b * c)))\ \rho$
$= code_L\ a\ \rho$; $code_R(b + (b * c))\ \rho$; **sto** i
$=$ **ldc** a 5; $code_R(b + (b * c))\ \rho$; **sto** i
$=$ **ldc** a 5; $code_R(b)\ \rho$; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b)\ \rho$; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; **ldc** a 7; **ind** i; **mul** i; **add** i; **sto** i

□

Control flow:
Conditionele en iteratieve statements, rijen van statements

Table 2.4 Conditional and unconditional branch.

Instr	Meaning	Comments	Cond	Result
ujp *q*	*PC* := *q*	Unconditional branch	*q* ∈ [0,*codemax*]	
fjp *q*	**if** *STORE*[*SP*] = *false*	Conditional branch	(b)	
	then *PC* := *q*		*q* ∈ [0,*codemax*]	
	fi			
	SP := *SP* − 1			

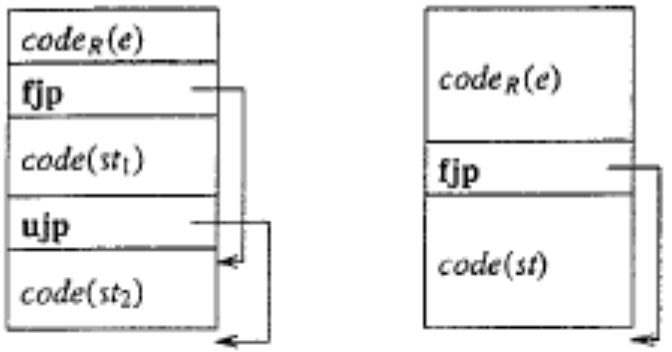
Control flow:
Conditionele en iteratieve statements, rijen van statements

Table 2.4 Conditional and unconditional branch

Instr	Meaning	Comments	Cond	Result
ujp *q*	*PC* := *q*	Unconditional branch	*q* ∈ [0,*codemax*]	
fjp *q*	**if** *STORE*[*SP*] = *false*	Conditional branch	(b)	
	then *PC* := *q*		*q* ∈ [0,*codemax*]	
	fi			
	SP := *SP* − 1			

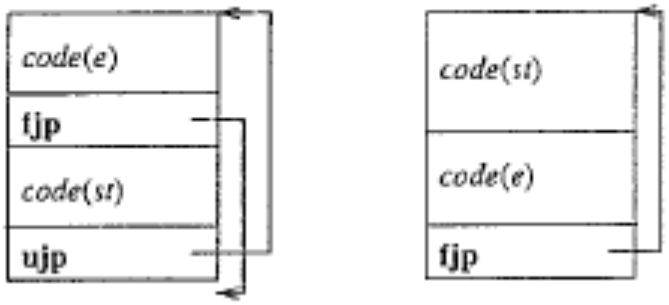
code (if e then st_1 else st_2 fi) ρ =
$code_R$ e ρ; fjp l_1; *code* st_1 ρ; ujp l_2; l_1: *code* st_2 ρ; l_2:

l_1, l_2 : labels



(a) (b)

Figure 2.2 Instruction sequences for conditional statements. (a) Bidirectional, (b) unidirectional.



(a) (b)

Figure 2.3 Code generation for loops, (a) **while** *e* **do** *st* **do** and (b) **repeat** *st* **until** *e*.

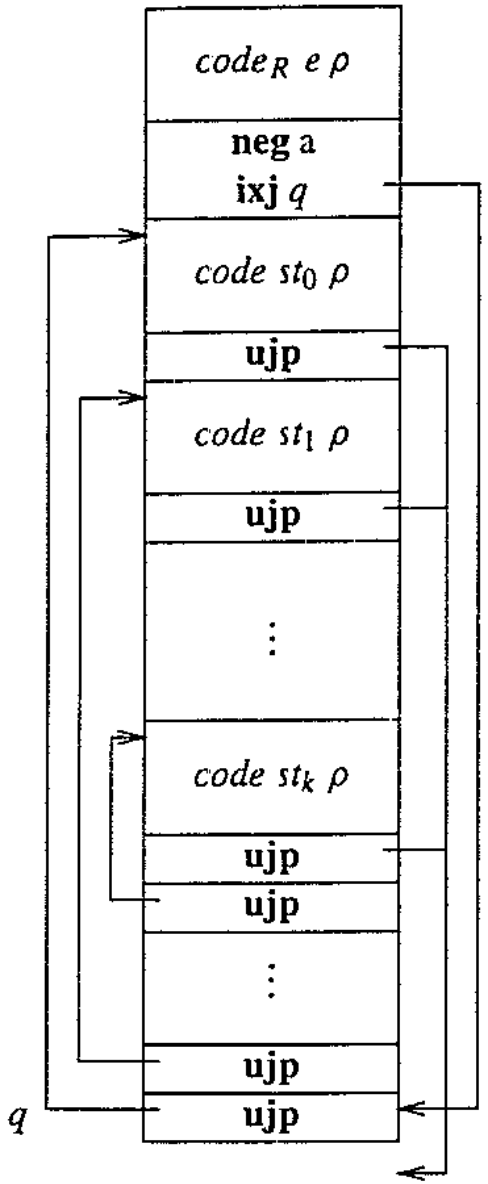
Switch / case statement

case e of
    0: st_0 ;
    1: st_1 ;
    …
    k: st_k ;
end

de waarde van e bepaalt welke van de st_i gekozen wordt

P-instructie: **ixj**

Instr	Meaning	Cond.	Result
ixj *q*	*PC* := *STORE*[*SP*] + *q*; *SP* := *SP* − 1	(*i*)	



leaves the value of the selector on the top of the stack

indexed branch to the end of the branching table

branch to the end of the statement

branching table

Om k opeenvolgende doel-adressen te hebben voor de jumps

Om de parameter q te kunnen invullen in dit schema hebben we de lengte van de stukken code st_i ρ nodig. Die is pas na hun generatie bekend, dus die q wordt pas achteraf ingevuld.

Er is ook nog een check nodig op de range waarin de waarde van e mag liggen.

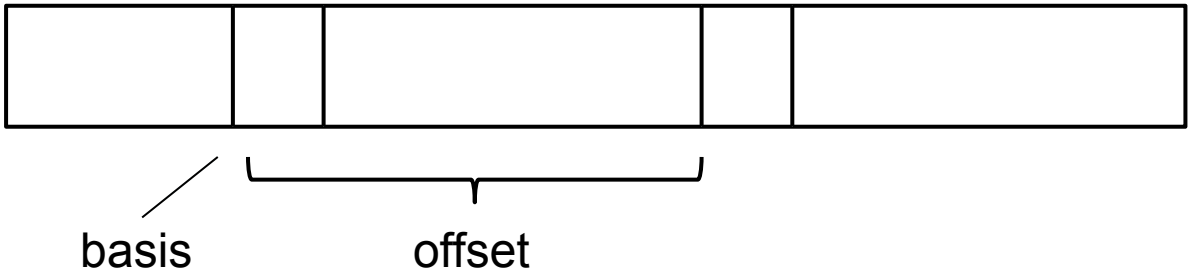
Vorm: **check** 0 k q

check 0 k q

```
        dup                 dupliceer de top van de stack
        ldc 0
        geq a
        fjp r
        dup
        ldc k
        leq a
        fjp r
        ixj q
r:      pop
        ldc k               k gebruikt als default
        ixj q
```

Geheugenallocatie: variabelen van een simpel type

ρ moet voor elke varabele een numeriek adres geven. Gewoonlijk is dat een relatief adres, dus een offset t.o.v. een basisadres. ρ moet uiteraard “at compile time” berekend worden, de adressen worden gebruikt bij uitvoering, dus “at runtime”.



We zullen aannemen dat locaties 0…4 gereserveerd zijn (zie verder: layout van de stack frames)

Geheugenallocatie: variabelen van een simpel type

Voor simpele types: assigneer aaneensluitende geheugenplaatsen

Voor een lijst van declaraties var n_1:t_1; … n_k:t_k;

Definiëer ρ door: $\rho(n_i) = i + 4$ of beter, rekening houdend met de grootte van de types:

$$\rho(n_i) = \begin{cases} 5 & \text{als } i = 1 \\ 5 + \sum_{j=1}^{i-1} \text{size}(t_j) & \text{anders} \end{cases}$$

waar size(t_j) het aantal plaatsen is nodig voor de voorstelling van een element van type t_j .

Geheugenallocatie: statische arrays

Probleem: bij gebruik van een component, bv a[e], is de waarde van e maar bekend bij uitvoering. Ze kan bovendien variëren, bv omdat die a[e] een lus voorkomt:

```
while … do
    ...
    x := a[e]
    ...
od
```

Het adres waarnaar a[e] verwijst is dynamische informatie, m.a.w. informatie die slechts bekend is at runtime (anders spreekt men van statische informatie, die is bekend at compile time)

Geheugenallocatie: statische arrays

Voorbeeld: var a : array[-5..5,1..9] of integer

Afspraak: we gebruiken row-major ordening: de elementen Van de array staan in de volgorde

a[-5,1], a[-5,2], … , a[-5,9],
a[-4,1], a[-4,2], … , a[-4,9],
…
a[5,1], a[5,2], … , a[5,9]

m.a.w. de rijen na elkaar. Anders gezegd, als we de elementen doorlopen varieert de meest rechtse index het snelst.

Geheugenallocatie: statische arrays

Algemeen geval, met k dimensies:

- var b : array $[u_1, \ldots, o_1, \ldots, u_k, \ldots, o_k]$ of integer
- dus op $b[i_1,\ldots,i_j,o_{j+1},\ldots,o_k]$ volgt $b[i_1,\ldots,i_j+1,u_{j+1},\ldots,u_k]$
- aantal elementen in dimensie i : $(o_i - u_i + 1)$
- size(t) for dit array type: $\Pi_1^k(o_i - u_i + 1)$ (als integer size 1 heeft)
- Voor een rij van declaraties: zoals voordien,

$$\rho(n_i) = \begin{cases} 5 & \text{als } i = 1 \\ 5 + \sum_{j=1}^{i-1} \text{size}(t_j) & \text{anders} \end{cases}$$

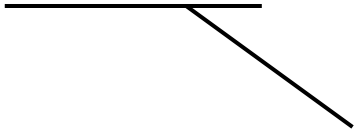
Geheugenallocatie: statische arrays

Voorbeeld: var a : array[-5..5,1..9] of integer

a[-5,1], a[-5,2], … , a[-5,9],
a[-4,1], a[-4,2], … , a[-4,9],
…
a[5,1], a[5,2], … , a[5,9]

Het element a[-2,6] staat op plaats (we tellen vanaf 0)

$(-2 - (-5)) \times (9 - 1 + 1) + 6 - 1 = (-2 + 5) \times 9 + 6 - 1 = 32$



aantal elementen in een rij (= aantal kolommen)

Geheugenallocatie: statische arrays

Toekenning van 0 aan de eerste component van een array a:

```
ldc a ρ(a);
ldc i 0;
sto i;
```

ρ(a) is het adres van het eerste element

waar ρ(a) bekend is bij compilatie.

Maar wat met a[i,j]:=0 ? De waarden î, ĵ van i en j zijn alleen bekend bij uitvoering. Die waarden zijn dynamische informatie. We moeten dus instructies genereren die at runtime de waarde

$$r = (\hat{i} - (-5)) \times (9 - 1 + 1) + \hat{j} - 1 = (\hat{i} + 5) \times 9 + \hat{j} - 1$$

berekenen. De component staat dan op adres ρ(a) + r

Geheugenallocatie: statische arrays

Het algemene geval:

$r = (\hat{i}_1 - u_1) \times \text{size}(\text{array}[u_2, \ldots, o_2, \ldots, u_k, \ldots, o_k] \text{ of integer}) +$
$(\hat{i}_2 - u_2) \times \text{size}(\text{array}[u_3, \ldots, o_3, \ldots, u_k, \ldots, o_k] \text{ of integer}) +$

…

$(\hat{i}_{k-1} - u_{k-1}) \times \text{size}(\text{array}[u_k, \ldots, o_k] \text{ of integer}) +$
$(\hat{i}_k - u_k)$

laat $d_i = o_i - u_i + 1$ (range in de i-de dimensie)

We kunnen r opsplitsen in een deel dat bekend is bij compilatie, en de rest.

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

Het deel dat bekend is bij compilatie (hier het tweede dus) kan uiteraard dan ook al berekend worden, en hoeft niet herberekend te worden elke keer een component gebruikt wordt (zie verder, dynamische arrays).

Aangepaste P-machine instructie

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

Berekening van geïndexeerd adres. STORE[SP-1] bevat een start adres, STORE[SP] bevat de index van de geselecteerde subarray en q bevat de grootte van de subarray. In het eerste voorbeeld dus 9.

Aangepaste P-machine instructies

Instr.	Meaning	Cond.	Result
inc $T\ q$	$STORE[SP] := STORE[SP] + q$	(T) and type $(q) = \mathrm{i}$	(T)
dec $T\ q$	$STORE[SP] := STORE[SP] - q$	(T) and type $(q) = \mathrm{i}$	(T)

Om de rij indices te compileren gebruiken we de functie $code_I$ met als tweede parameter de grootte van de componenten, g. Het startadres van array c is ρ(c) .

Verder is $d^{(j)} = d_{j+1} \times d_{j+2} \times \ldots \times d_k$ (met $d^{(k)} = 1$) en
$d = u_1 \times d^{(1)} + u_2 \times d^{(2)} + \ldots + u_k \times d^{(k)}$

$$
\begin{aligned}
code_L\ c[i_1, \ldots i_k]\ \rho &= \textbf{ldc a}\ \rho(c);\ code_I\ [i_1, \ldots, i_k]\ g\ \rho \\
code_I\ [i_1, \ldots, i_k]\ g\ \rho &= code_R\ i_1\ \rho;\ \textbf{ixa}\ g \cdot d^{(1)}; \\
&\quad\ code_R\ i_2\ \rho;\ \textbf{ixa}\ g \cdot d^{(2)}; \\
&\quad\ \vdots \\
&\quad\ code_R\ i_k\ \rho;\ \textbf{ixa}\ g \cdot d^{(k)}; \\
&\quad\ \textbf{dec a}\ g \cdot d;
\end{aligned}
$$

Met “rangecheck”: test of de waarden van de indices binnen de grenzen blijven.

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

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

g: componentgrootte u_i , o_i : onder- en bovengrens

Instr.	Meaning	Cond.	Result
chk *p q*	**if** (*STORE*[*SP*] < *p*) **or** (*STORE*[*SP*] > *q*) **then** *error*(‘value out of range’) **fi**	(i,i)	(i)

Voorbeeld: var i,j: integer;
a: array[-5..5,1..9] of integer

waar ρ(i)=5,ρ(j)=6 en ρ(a)=7

Code (a[i+1,j]:=0) ρ =

```
ldc a 7
ldc a 5
ind i 5
ldc i 1
add i
chk -5 5
ixa 9
ldc a 6
ind i
chk 1 9
ixa 1
```

```
dec a -44
ldc i 0
sto i
```