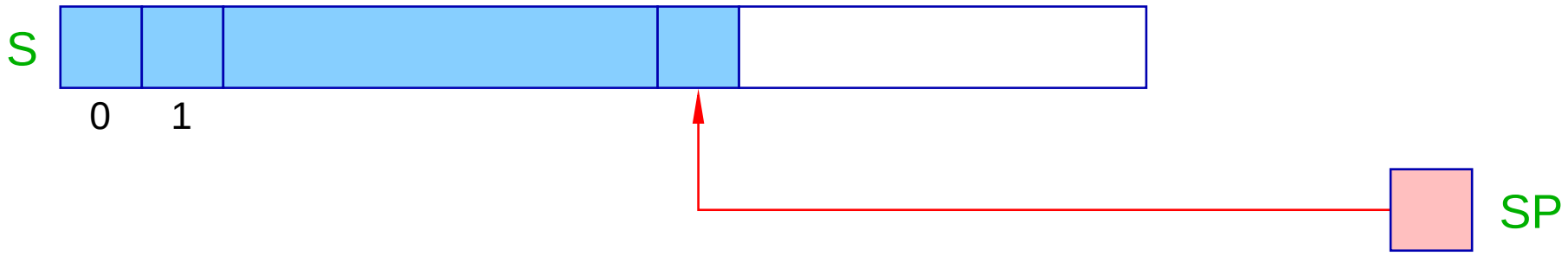
CMa — lihtsustatud C abstraktne masin

CMa arhitektuur

- Iga abstraktne masin määrab mingi hulga käske, mis on täitmiseks abstraktsel riistvaral.
- Seda abstraktset riistvara võib vaadelda kui teatavate andmestruktuuride kogumit, mida käsud kasutavad
- . . . ja mida juhib täitmisaegne süsteem (run-time system).

CMa arhitektuur

Magasin:



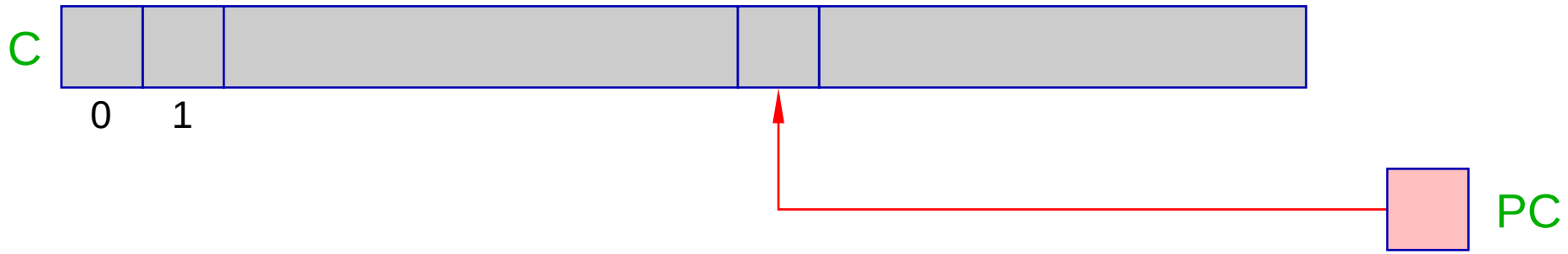
S = Stack — mälupiirkond andmete hoidmiseks, kus uute elementide lisamine/eemaldamine käib LIFO printsiibil.

SP = Stack-Pointer — register, mis sisaldab ülemise (so. viimasena lisatud) elemendi aadressit.

Lihtsustus: kõik mittestruktuurset tüüpi väärtused on sama suurusega ja mahuvad ühte magasini pessa.

CMa arhitektuur

Kood:



C = Code-store — mälupiirkond programmikoodi hoidmiseks; igas pesas on täpselt üks abstraktse masina käsk.

PC = Program Counter — register, mis sisaldab järgmisena täidetava käsu aadressit.

Algselt sisaldab PC aadressit 0; so. C[0] sisaldab kõige esimesena täidetavat käsku.

CMa arhitektuur

Programmi täitmine:

- Masin laadib käsu C[PC] registrisse IR (Instruction-Register), seejärel suurendab registrit PC ühe võrra ning lõpuks täidab selle käsu:

```
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```

- Käsu täitmine võib muuta registri PC sisu (hüpped).
- Masina peatsükli katkestab käsk `halt`, mis tagastab kontrolli väliskeskkonnale.
- Ülejäänud käsud toome sisse järk-järgult vastavalt vajadusele.

Lihtsad avaldised ja omistamine

Ülesanne: väärtustada avaldis $(1 + 7) * 3$

See tähendab: genereerida käskude jada, mis

- leiab avaldise väärtuse ja
- lisab selle magasini tippu.

Idee:

- kõigepealt arvutame alamavaldiste väärtused,
- salvestame need magasini tippu ning
- seejärel rakendame antud operaatorile vastavat käsku.

Lihtsad avaldised ja omistamine

Üldprintsiibid:

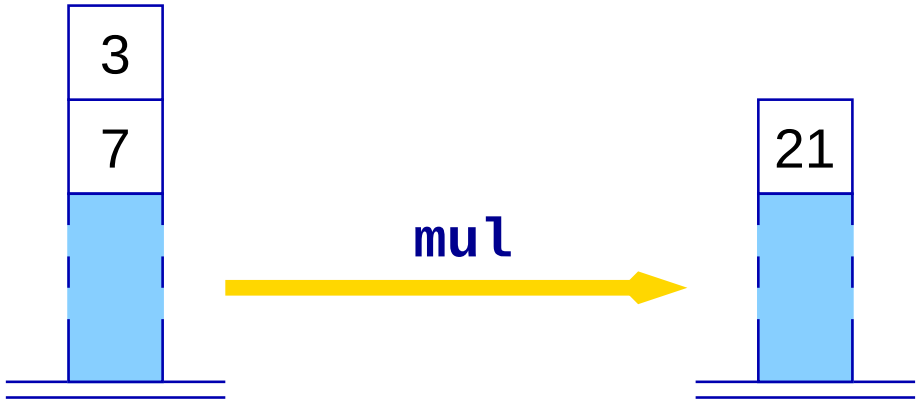
- käsud eeldavad, et nende argumendid on ülemistes pesades,
- käsu täitmine tarvitab oma argumendid ära,
- tulemus salvestatakse magasini tippu.


```
SP++;
S[SP] = q;
```

Käsk `loadc q` ei oma argumente ja salvestab konstandi `q` magasini tippu.

NB! Registri SP sisu on joonisel esitatud ainult kaudselt magasini kõrguse kaudu.

Lihtsad avaldised ja omistamine



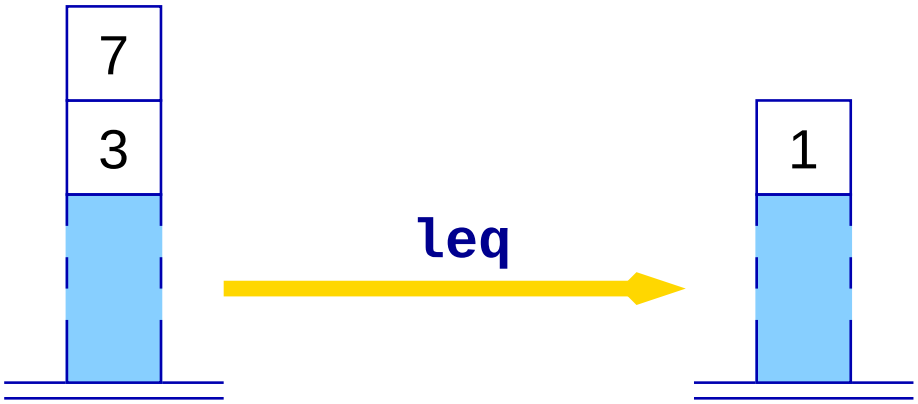

```
SP--;
S[SP] = S[SP] * S[SP+1];
```

Käsk `mul` eeldab magasinis kahte argumenti, tarvitab need ära ja salvestab nende korrutise magasini tippu.

Teised binaarsetele aritmeetilistele ja loogilistele operaatoritele vastavad käsud `add`, `sub`, `div`, `mod`, `and`, `or`, `xor`, `eq`, `neq`, `le`, `leq`, `ge` ja `geq` töötavad analoogselt.

Lihtsad avaldised ja omistamine

Näide: operaator `leq`

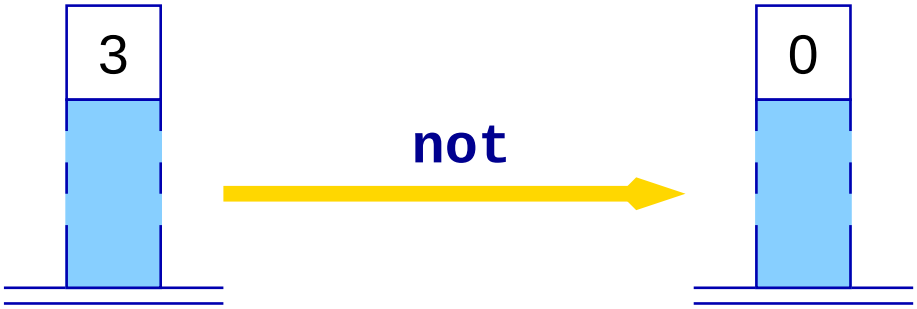


NB! Arv 0 esitab väära tõeväärtust, kõik teised täisarvud on tõesed väärtused.

Lihtsad avaldised ja omistamine

Unaarsed operaatorid `neg` ja `not` tarvitavad ühe argumendi ja produtseerivad ühe tulemuse:


```
S[SP] = -S[SP];
```



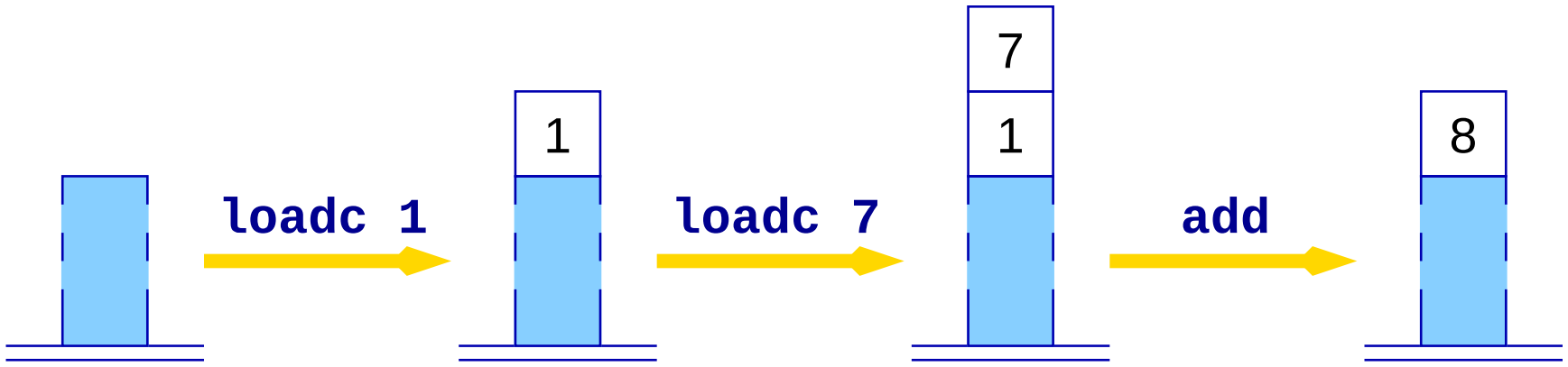

```
if (S[SP] ≠ 0)
    S[SP] = 0;
else
    S[SP] = 1;
```

Lihtsad avaldised ja omistamine

Näide: avaldisele $1 + 7$ vastab käskude jada:

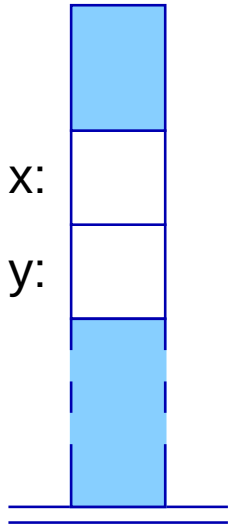
```
loadc 1
loadc 7
add
```

Selle koodi täitmine annab:



Lihtsad avaldised ja omistamine

- Muutujatele vastavad magasini S pesad:



- Koodi genereerimist kirjeldavad funktsioonid code, code_L ja code_R.
- Argumendid: transleeritav süntaktiline konstruktsioon ja aadresskeskond (so. funktsioon, mis igale muutujale seab vastavusse tema suhtaadressi magasinis).

Lihtsad avaldised ja omistamine

- Muutujaid kasutatakse kahel erineval moel.
- Näiteks omistuslauses $x = y + 1$ oleme huvitatud muutuja y väärtusest, kuid muutuja x aadressist.
- Muutuja süntaktiline asukoht määrab, kas me vajame tema L-väärtust või R-väärtust

  muutuja L-väärtus = tema aadress

  muutuja R-väärtus = tema väärtus

- Funktsioon $\text{code}_L\ e\ \rho$ emiteerib keskkonnas ρ avaldise e L-väärtust arvutava koodi.
- Funktsioon $\text{code}_R\ e\ \rho$ teeb sama sama R-väärtuse jaoks.
- NB! Mitte igal avaldisel pole L-väärtust (näit.: $x + 1$).

Lihtsad avaldised ja omistamine

- Binaarsete operaatorite transleerimine:

$$\begin{aligned} \text{code}_R\ (e_1 + e_2)\ \rho \quad = \quad & \text{code}_R\ e_1\ \rho \\ & \text{code}_R\ e_2\ \rho \\ & \texttt{add} \end{aligned}$$

  - Analoogselt teiste binaaroperaatorite korral.

- Unaarsete operaatorite transleerimine:

$$\begin{aligned} \text{code}_R\ (-e)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \texttt{neg} \end{aligned}$$

  - Analoogselt teiste unaaroperaatorite korral.

- Baasväärtuset transleerimine:

$$\text{code}_R\ q\ \rho \quad = \quad \texttt{loadc q}$$

Lihtsad avaldised ja omistamine

- Muutujate transleerimine:

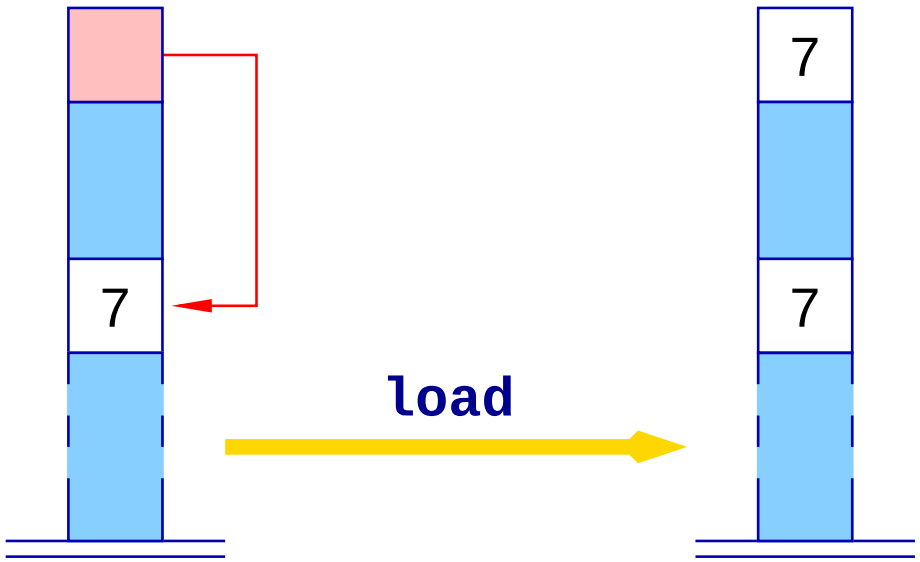
$$\begin{array}{lcl} \text{code}_L\ x\ \rho & = & \texttt{loadc}\ (\rho\ x) \\ \text{code}_R\ x\ \rho & = & \text{code}_L\ x\ \rho \\ & & \texttt{load} \end{array}$$

- Omistusavaldise transleerimine:

$$\begin{array}{lcl} \text{code}_R\ (x = e)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{code}_L\ x\ \rho \\ & & \texttt{store} \end{array}$$

Lihtsad avaldised ja omistamine

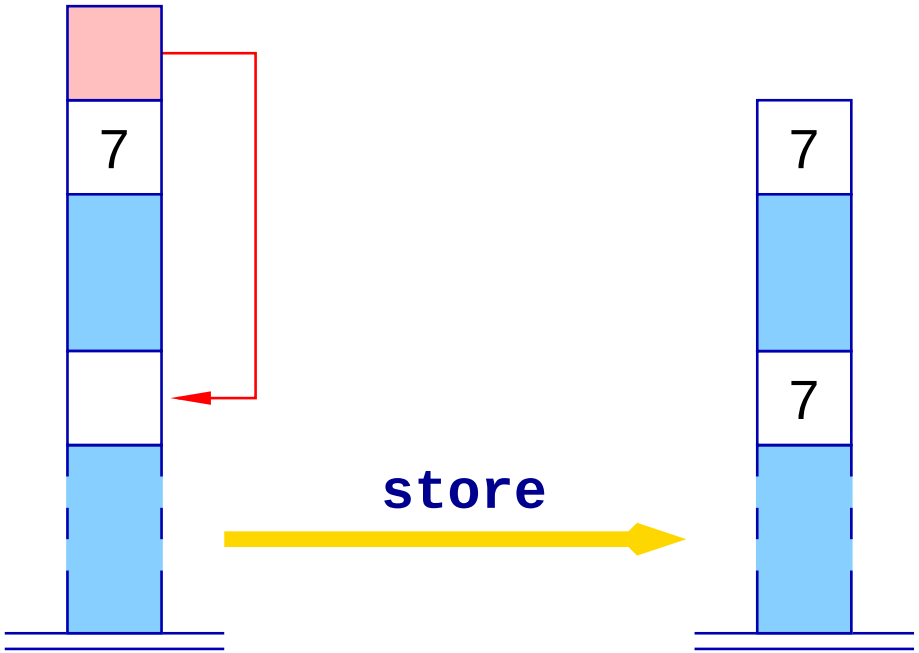
Käsk `load` salvestab magasini tippu selle magasini pesa sisu, mille aadress on ülemises pesas:




```
S[SP] = S[S[SP]];
```

Lihtsad avaldised ja omistamine

Käsk `store` kirjutab ülalt teise pesa sisu pessa, kuhu viitab ülemine pesa ja jätab kirjutatud väärtuse ka magasini tippu:




```
S[S[SP]] = S[SP-1];
SP--;
```

NB! Erineb Wilhelm/Maurer'i raamatus toodud vastavast P-machine'i käsust.

Ligipääs muutujatele

Näide: olgu $e \equiv (x = y - 1)$ ja $\rho = \{x \mapsto 4, y \mapsto 7\}$,
siis $\text{code}_R\ e\ \rho$ emiteerib koodi:

```
loadc 7          sub
load             loadc 4
loadc 1          store
```

Täiustusi: sagedasti esinevate käskukombinatsioonide jaoks võib sisse tuua uusi käske, näiteks:

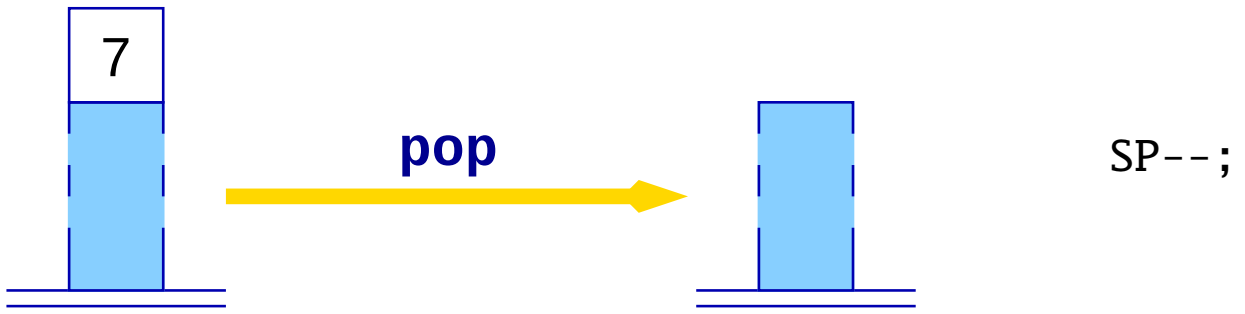
```
loada q    =   loadc q
               load

storea q   =   loadc q
               store
```

Laused ja lausete jadad

- Kui e on avaldis, siis $e;$ on lause.
- Lausel ei ole väärtust; seega registri SP sisu peab enne ja pärast lausele vastava koodi täitmist olema sama.

$$
\begin{array}{lcl}
\text{code } (e;)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \texttt{pop} \\
\text{code } (s\ ss)\ \rho & = & \text{code } s\ \rho \\
 & & \text{code } ss\ \rho \\
\text{code } \varepsilon\ \rho & = & \texttt{// tühi käskude jada}
\end{array}
$$

- Käsk pop eemaldab magasinist kõige ülemise peas:

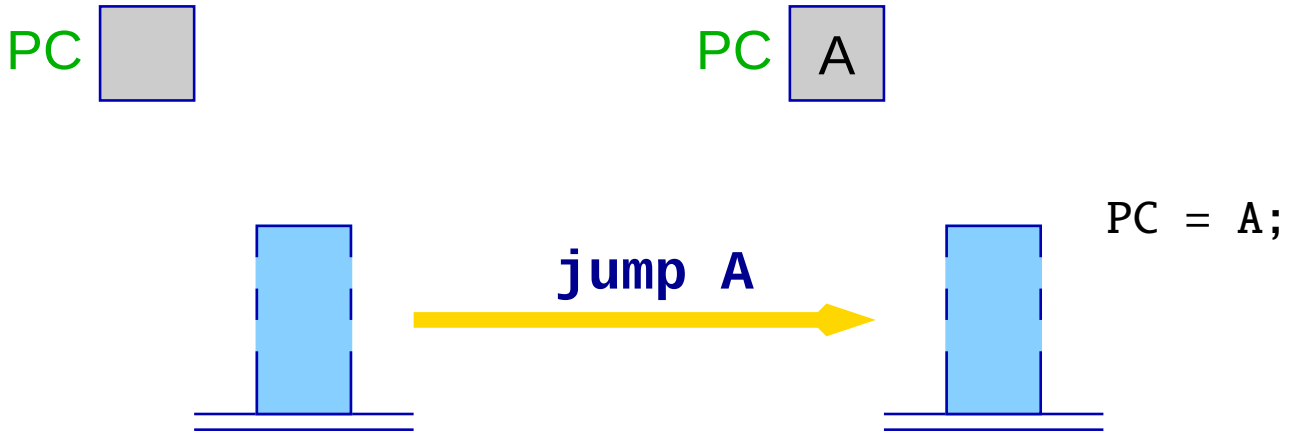


Tingimuslikud laused ja tsüklid

- Lihtsuse pärast kasutame hüpete sihtkohana sümbolmärgendeid, mis hiljem asendatakse absoluutsete aadressitega.
- Absoluutaadressite asemel võib kasutada ka suhtaadresseid; so. relatiivseid registri PC tegeliku väärtuse suhtes.
- Viimase lähenemise eeliseks on:
  - suhtaadressid on reeglina väiksemad;
  - kood on ringitõstetav (relocatable).

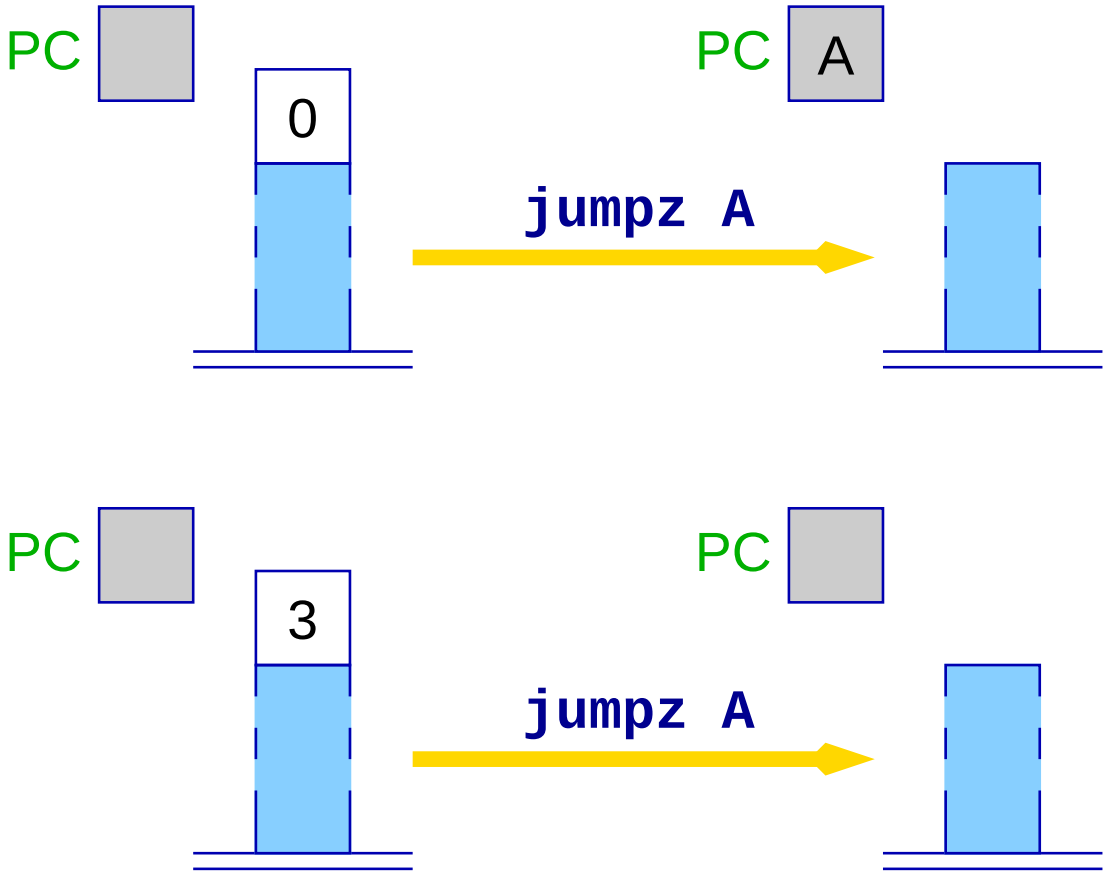
Tingimuslikud laused ja tsüklid

Käsk `jump A` teostab hüppe aadressile A; magasini ei muudeta:



Tingimuslikud laused ja tsüklid

Käsk `jumpz A` teostab tingimusliku hüppe; kui magasini tipus olev argument on 0, siis toimub hüpe aadressile A; vastasel korral hüpet ei toimu:

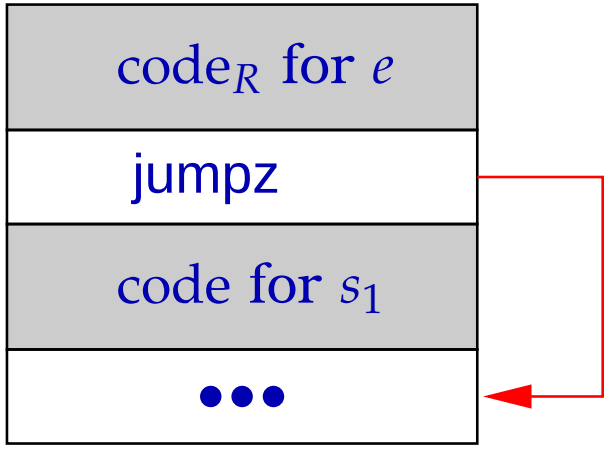



```
if (S[SP] = 0)
    PC = A;
SP--;
```

Tingimuslikud laused ja tsüklid

- Üheharulise tingimuslause $s \equiv \textbf{if}\ (e)\ s_1$ transleerimine:
  - genereerime koodi tingimuse e ja lause s_1 jaoks;
  - lisame tingimusliku hüppekäsu nende vahele.

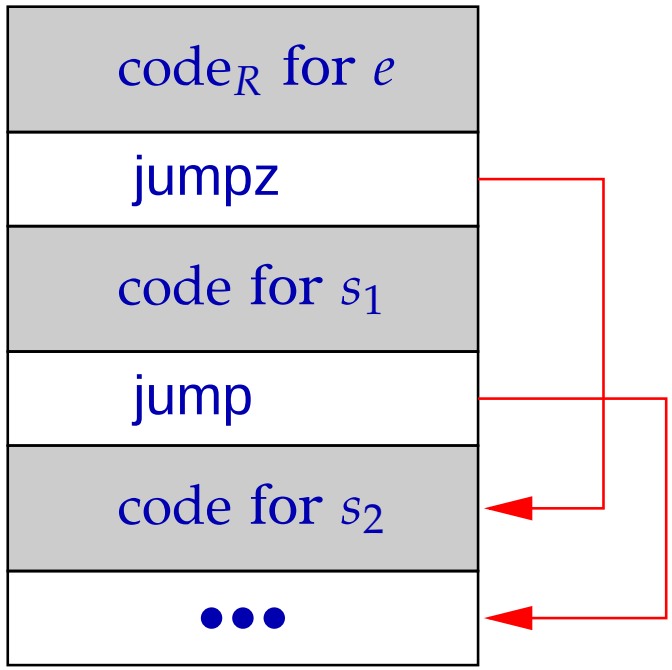
$$
\begin{array}{lll}
\text{code}\ (\textbf{if}\ (e)\ s_1)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \texttt{jumpz A} \\
 & & \text{code}\ s_1\ \rho \\
 & & \text{A:}\ \ldots
\end{array}
$$



Tingimuslikud laused ja tsüklid

- Kaheharulise tingimuslause $s \equiv$ **if** (e) s_1 **else** s_2 transleerimine:

$$
\begin{array}{lcl}
\text{code } (\textbf{if } (e)\ s_1 \textbf{ else } s_2)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \texttt{jumpz A} \\
 & & \text{code } s_1\ \rho \\
 & & \texttt{jump B} \\
 & \text{A:} & \text{code } s_2\ \rho \\
 & \text{B:} & \ldots
\end{array}
$$



Tingimuslikud laused ja tsüklid

Näide: olgu $\rho = \{x \mapsto 4, y \mapsto 7\}$ ja

$$
\begin{array}{lll}
s & \equiv & \textbf{if } (x > y) \qquad (i) \\
 & & \quad x = x - y; \qquad (ii) \\
 & & \textbf{else } y = y - x; \qquad (iii)
\end{array}
$$

siis code s ρ emiteerib koodi:

```
loada 4
loada 7
ge
jumpz A
```

(i)

```
loada 4
loada 7
sub
storea 4
pop
jump B
```

(ii)

```
A: loada 7
   loada 4
   sub
   storea 7
   pop
B: ...
```

(iii)

Tingimuslikud laused ja tsüklid

- while-tsükli $s \equiv$ **while** $(e)\ s_1$ transleerimine:

code (**while** $(e)\ s_1$) ρ =

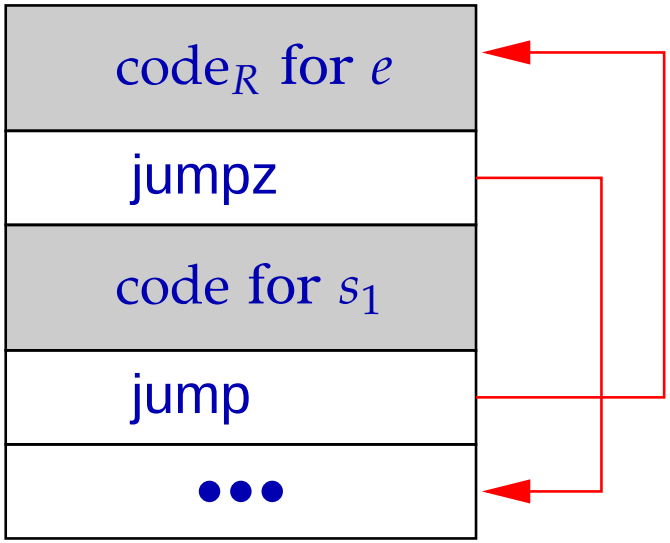
A: code$_R$ e ρ

`jumpz B`

code s_1 ρ

`jump A`

B: ...



Tingimuslikud laused ja tsüklid

Näide: olgu $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ ja

$$
\begin{array}{rcll}
s & \equiv & \textbf{while}\ (a > 0)\ \{ & (i) \\
 & & \quad c = c + 1; & (ii) \\
 & & \quad a = a - b; & (iii) \\
 & & \} &
\end{array}
$$

siis code $s\ \rho$ emiteerib koodi:

```
A: loada 7      loada 9      loada 7          jump A
   loadc 0      loadc 1      loada 8      B: ...
   ge           add          sub
   jumpz B      storea 9     storea 7
                pop          pop
```

(i) (ii) (iii)

Tingimuslikud laused ja tsüklid

- for-tsükkel $s \equiv$ **for** $(e_1;\ e_2;\ e_3)\ s_1$ on ekvivalentne while-tsükliga $e_1;$ **while** $(e_2)\ \{s_1\ e_3;\}$ (eeldusel, et s_1 ei sisalda continue-lauset)

$$
\begin{array}{lll}
\text{code}\ (\textbf{for}\ (e_1;\ e_2;\ e_3)\ s_1)\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \texttt{pop} \\
 & \text{A:} & \text{code}_R\ e_2\ \rho \\
 & & \texttt{jumpz B} \\
 & & \text{code}\ s_1\ \rho \\
 & & \text{code}_R\ e_3\ \rho \\
 & & \texttt{pop} \\
 & & \texttt{jump A} \\
 & \text{B:} & \ldots
\end{array}
$$

Tingimuslikud laused ja tsüklid

- Mitmeharulised tingimuslaused tuleb reeglina transleerida mitmeteks üksteisesse sisestatud if-lauseteks:

switch (e) {				x = e;
case c_0 :	ss_0 **break;**			**if** $(x = c_0)$ ss_0
case c_1 :	ss_1 **break;**			**else if** $(x = c_1)$ ss_1
...			$\Longrightarrow$	...
case c_{k-1} :	ss_{k-1} **break;**			**else if** $(x = c_{k-1})$ ss_{k-1}
default :	ss_k			**else** ss_k
}				

- Kui märgendid sorteerida ja kasutada kahendotsimist, siis saab võrdluste arvu vähendada logaritmini märgendite arvust.

Tingimuslikud laused ja tsüklid

- Erijuhtudel on võimalik konstantse ajaga hargnemine.
- Vaatame switch-lauset kujul:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \quad \textbf{case}\ 0\ :\ ss_0\ \textbf{break;} \\
 & & \quad \textbf{case}\ 1\ :\ ss_1\ \textbf{break;} \\
 & & \quad \ldots \\
 & & \quad \textbf{case}\ k-1\ :\ ss_{k-1}\ \textbf{break;} \\
 & & \quad \textbf{default}\ :\ ss_k \\
 & & \}
\end{array}
$$

Tingimuslikud laused ja tsüklid

$$
\begin{array}{lll}
\text{code } s\ \rho = & \text{code}_R\ e\ \rho & \mathrm{C}_0\text{: code } ss_0\ \rho \\
 & \text{check } 0\ k\ \mathrm{B} & \quad \texttt{jump D} \\
 & & \quad \ldots \\
 & & \mathrm{C}_k\text{: code } ss_k\ \rho \\
 & & \quad \texttt{jump D}
\end{array}
\qquad
\begin{array}{ll}
\mathrm{B}: & \texttt{jump } \mathrm{C}_0 \\
 & \ldots \\
 & \texttt{jump } \mathrm{C}_k \\
\mathrm{D}: & \ldots
\end{array}
$$

- Makro check $0\ k$ B kontrollib, kas tingimusavaldise R-väärtus on vahemikus $[0,k]$ ja seejärel teostab indekseeritud hüppe.
- "Hüppetabeli" B i-s element sisaldab otsehüppe käsu i-ndale harule vastava koodi algaadressile.
- Iga haru lõpus on otsehüpe switch-lausest välja.

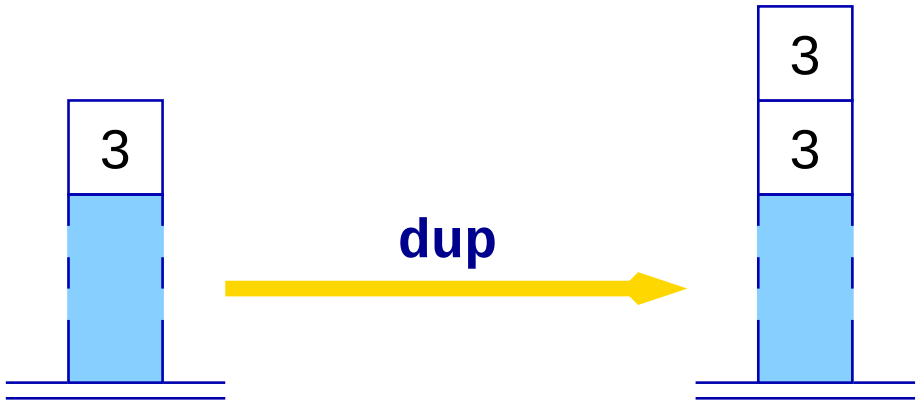
Tingimuslikud laused ja tsüklid

```
check 0 k B  =  dup          dup          jumpi B
                loadc 0      loadc k  A: pop
                geq          le           loadc k
                jumpz A      jumpz A      jumpi B
```

- Tingimusavaldise R-väärtust kasutatakse nii võrdlusteks, kui ka indekseerimiseks; seetõttu tuleb ta enne võrdlusi kopeerida.
- Kui R-väärtus ei ole vahemikus $[0,k]$, siis asendatakse ta enne hüpet konstandiga k.

Tingimuslikud laused ja tsüklid

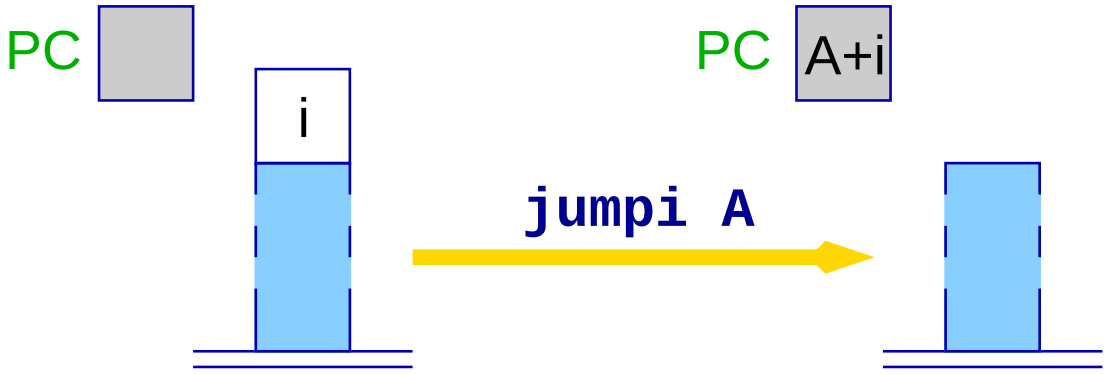
Käsk dup kopeerib magasini ülemise elemendi:




```
S[SP+1] = S[SP];
SP++;
```

Tingimuslikud laused ja tsüklid

Käsk `jumpi A` teostab indekseeritud hüppe:




```
PC = A + S[SP];
SP--;
```

Tingimuslikud laused ja tsüklid

- Hüppetabeli B võib paigutada kohe peale makrot check; see võimaldab hoida kokku mõned otsehüpped.
- Kui väärtuste vahemik algab u-st (ja mitte nullist), siis tuleb e R-väärtusest, enne tema kasutamist indeksina, lahutada u.
- Kui e kõik potentsiaalselt võimalikud väärtused on vahemikus $[0,k]$, siis pole makrot check vaja.

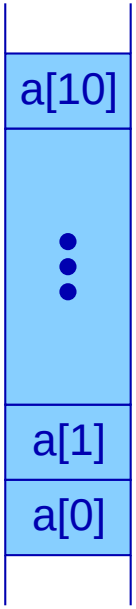
Massiivid, kirjed ning staatiline mäluhaldus

- Eesmärk: staatiliselt (so. kompileerimisajal) siduda iga muutujaga x fikseeritud (relatiivne) aadress $\rho\ x$.
- Eeldame, et baastüüpi muutujad (näit. **int**, ...) mahuvad ühte mälupessa.
- Seome muutujatega aadressid nende deklareerimise järjekorras alustades aadressist 1.
- Seega, deklaratsioonide $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$ (t_i on baastüüp) korral saame aadresskeskkonna ρ sellise, et

$$\rho\ x_i = i, \qquad i = 1, \ldots, k$$

Massiivid, kirjed ning staatiline mäluhaldus

- Massiiv on staatiliselt määratud suurusega järjestikuste mälupesade hulk.
- Ligipääs kasutades täisarvulisi indekseid, mis algavad nullist.
- Massiivi aadress on tema esimese elemendi $a[0]$ aadress.
- Näide: deklaratsioon **int**$[11]a$; defineerib 11 elemendilise massiivi.



Massiivid, kirjed ning staatiline mäluhaldus

- Defineerime funktsiooni sizeof (tähistus $|\cdot|$), mis leiab tüübi mäluvajadused:

$$|t| = \begin{cases} 1 & \text{kui } t \text{ on baastüüp} \\ k \cdot |t'| & \text{kui } t \equiv t'[k] \end{cases}$$

- Seega, deklaratsioonide $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$ korral

$$\begin{aligned} \rho\ x_1 &= 1 \\ \rho\ x_i &= \rho_{i-1} + |t_{i-1}| \quad i > 1 \end{aligned}$$

- Kuna $|\cdot|$ saab arvutada kompileerimise ajal, siis saab ka aadresskeskkonna ρ arvutada kompileerimise ajal.

Massiivid, kirjed ning staatiline mäluhaldus

- Olgu $t\ a[c]$; massiivi deklaratsioon.
- Siis i-nda komponendi algaadress on $\rho\ a + |t| \times (\text{rval of } i)$

$$
\begin{array}{rcl}
\text{code}_L\ (a[e])\ \rho & = & \texttt{loadc}\ (\rho\ a) \\
 & & \text{code}_R\ e\ \rho \\
 & & \texttt{loadc}\ |t| \\
 & & \texttt{mul} \\
 & & \texttt{add}
\end{array}
$$

- Üldjuhul võib massiiv olla esitatud avaldisena, mis tuleb enne indekseerimist väärtustada.
- C-s on deklareeritud massiiv viitkonstant, mille R-väärtus on massiivi algaadress.

Massiivid, kirjed ning staatiline mäluhaldus

$$
\begin{array}{lcll}
\text{code}_L\ (e_1[e_2])\ \rho & = & \text{code}_R\ e_1\ \rho & \\
 & & \text{code}_R\ e_2\ \rho & \\
 & & \texttt{loadc}\ |t| & \\
 & & \texttt{mul} & \\
 & & \texttt{add} & \\
\text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & e \text{ on massiiv}
\end{array}
$$

- NB! C-s on järgnevad (L-väärtustena) ekvivalentsed:

$$a[2] \qquad 2[a] \qquad a+2$$

- Normaliseerimine: massiivide nimed ja avaldised mis väärtustuvad massiiviks on enne indekseerimis-sulge; indeks-avaldis on sulgudes.

Massiivid, kirjed ning staatiline mäluhaldus

- Kirje on nime omavate, võimalik et erinevat tüüpi, komponentide hulk.
- Ligipääs komponentidele nimede (selektorite) abil.
- Lihtsustusena eeldame, et kirjekomponentide nimesid ei kasutata mujal.
  - Alternatiiv: hallata iga kirjetüübi st jaoks eraldi keskkonda ρ_{st}.
- Kuulugu **struct** $\{$ **int** a; **int** b; $\}$ x; deklaratsioonide hulka:
  - kirje x suhtaadress on tema esimese pesa aadress;
  - komponentide aadressid on relatiivsed kirje algaadressi suhtes; so. ülaltoodud näites $a \mapsto 0, b \mapsto 1$.

Massiivid, kirjed ning staatiline mäluhaldus

- Olgu $t \equiv$ **struct** $\{\ t_1\ c_1;\ \ldots\ t_k\ c_k;\ \ \}$, siis

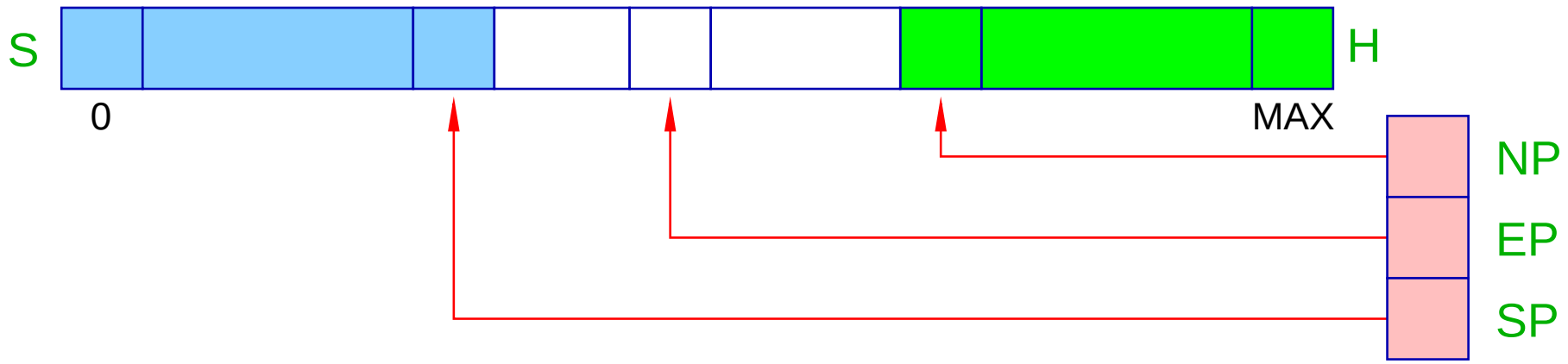
$$\begin{aligned} |t| &= \textstyle\sum_{i=1}^{k} |t_i| \\ \rho\ c_1 &= 0 \\ \rho\ c_i &= \rho\ c_{i-1} + |t_{i-1}| \qquad i > 1 \end{aligned}$$

- Seega komponendi $x.c_i$ aadress on $\rho\ x + \rho\ c_i$

$$\begin{aligned} \text{code}_L\ (e.c)\ \rho \quad = \quad & \text{code}_L\ e\ \rho \\ & \texttt{loadc}\ (\rho\ c) \\ & \texttt{add} \end{aligned}$$

Viidad ja dünaamiline mäluhaldus

Kuhi:



H = Heap — mälupiirkond dünaamilise elueaga andmetele.

NP = New-Pointer — register, mis sisaldab kuhja kõige alumise täidetud pesa aadressit.

EP = Extreme-Pointer — register, mis sisaldab kõige ülemise pesa aadressi, kuhu SP võib antud funktsiooni täitmisel viidata.

Viidad ja dünaamiline mäluhaldus

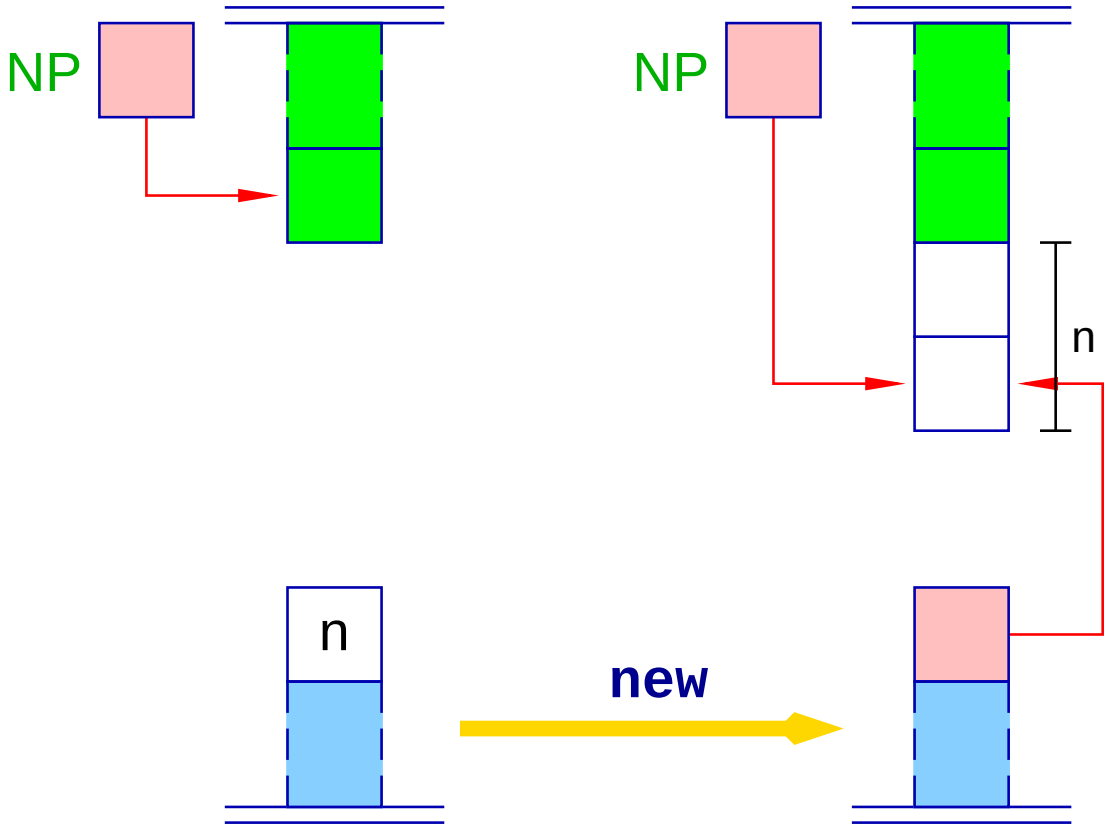
- Magasin ja kuhi kasvavad üksteise suunas, aga ei tohi kattuda (stack overflow).
- "Kokkupõrge" võib olla põhjustatud nii SP suurendamisest kui NP vähendamisest.
- EP aitab vältida ületäitumist magasini operatsioonide korral.
- EP väärtuse saab määrata staatiliselt.
- Kuhjast mälu võttes tuleb ületäitumist kontrollida.

Viidad ja dünaamiline mäluhaldus

- Viidad võimaldavad ligipääsu anonüümsetele, dünaamiliselt loodud, objektidele, mille eluiga ei allu LIFO printsiibile.
- Viitväärtuste saamiseks on kaks võimalust:
  - funktsiooni **malloc**(e) väljakutse reserveerib mälupiirkonna mille suurus on e väärtus ning väljastab selle piirkonna algusaadressi;
  - aadressoperaatori & rakendamine muutujale väljastab selle muutuja aadressi (so. L-väärtuse).

$$\begin{aligned} \text{code}_R\ (\textbf{malloc}(e))\ \rho &= \text{code}_R\ e\ \rho \\ &\quad \texttt{new} \\ \text{code}_R\ (\&e)\ \rho &= \text{code}_L\ e\ \rho \end{aligned}$$

Viidad ja dünaamiline mäluhaldus




```
if (NP - S[SP] ≤ EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

- `NULL` on spetsiaalne viitkonstant; samastatakse täisarvulise konstandiga 0.
- Ületäitumise korral väljastatakse `NULL`-viit.

Viidad ja dünaamiline mäluhaldus

- Viidatava väärtuste saamiseks on kaks võimalust:
  - operaatori $*$ rakendamine avaldisele e väljastab mälupesa sisu, mille aadress on avaldise e R-väärtus;
  - kirje komponendi selekteerimine läbi viida $e{\rightarrow}c$ on ekvivalentne avaldisega $(*e).c$.

$$
\begin{array}{lcl}
\text{code}_L\ (*e)\ \rho & = & \text{code}_R\ e\ \rho \\
\text{code}_L\ (e{\rightarrow}c)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \texttt{loadc}\ (\rho\ c) \\
 & & \texttt{add}
\end{array}
$$

Viidad ja dünaamiline mäluhaldus

Näide: olgu antud deklaratsioonid:

$$\textbf{struct } t \{ \textbf{ int } a[7]; \textbf{ struct } t * b; \};$$
$$\textbf{int } i, j;$$
$$\textbf{struct } t * pt;$$

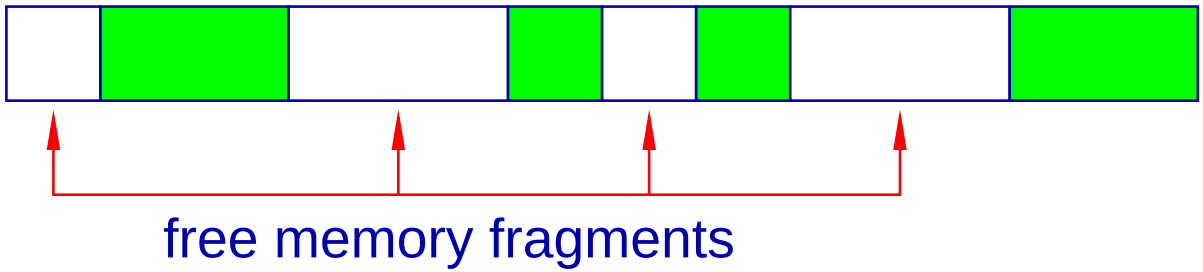
Siis $\rho = \{ a \mapsto 0,\ b \mapsto 7,\ i \mapsto 1,\ j \mapsto 2,\ pt \mapsto 3 \}$.

Avaldise $((pt \rightarrow b) \rightarrow a)[i+1]$ korral emiteeritakse kood:

```
loada 3        load           loada 1        loadc 1
loadc 7        loadc 0        loadc 1        mul
add            add            add            add
```

Viidad ja dünaamiline mäluhaldus

- Mälu vabastatakse C-s funktsiooni **free**(e) väljakutse abil.
- Antud mälupiirkond märgistatakse kui vaba ja pannakse spetsiaalsesse vabamälu listi (free list), kust **malloc** saab vajadusel selle uuesti kasutusele võtta.
- Probleemid:
  - peale mälupiirkonna vabastamist võivad mõned, ikka veel kättesaadavad, viidad sellele viidata (dangling references);
  - mälu võib ajapikku muutuda fragmenteerituks;



  - vabamälu listi haldamine võib olla kulukas.

Viidad ja dünaamiline mäluhaldus

- Alternatiiv: funktsiooni **free** väljakutse korral ei tehta midagi.

$$\text{code}\ (\textbf{free}(e);)\ \rho \quad = \quad \begin{array}{l}\text{code}_R\ e\ \rho \\ \text{pop}\end{array}$$

- Mälu täitumisel vabastatakse kindlasti mitte-kättesaadav mälu automaatselt kasutades prügikoristust (garbage collection).
  - \+ Mälu võtmine ja "vabastamine" on lihtne ja väga efektiivne.
  - \+ Pole "ripnevate viitade" probleemi.
  - \+ Mitmed prügikoristuse algoritmid defragmenteerivad kasutusel oleva mälu.
  - – Prügikoristus võib olla aeganõudev, mistõttu võivad tekkida programmi täitmises märgatavad pausid.

Funktsioonid

- Funktsiooni definitsioon koosneb neljast komponendist:
  - funktsiooni nimi, mille kaudu toimub tema väljakutsumine;
  - funktsiooni formaalsete parameetrite spetsifikatsioon;
  - funktsiooni resultaadi tüüp;
  - funktsiooni keha.
- C-s kehtib:

$$\text{code}_R\ f\ \rho \quad = \quad \_f \quad = \quad f\text{-i koodi algaadress}$$

- Seega peab aadresskeskkond haldama ka funktsioonide nimesid!

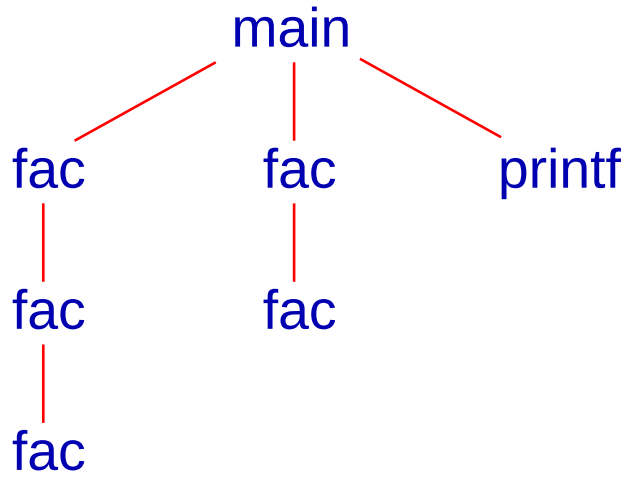
Funktsioonid

- Näide:

```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf("%d", n);
}
```

- Samast funktsioonist võib olla mitu aktiivset eksemplari.

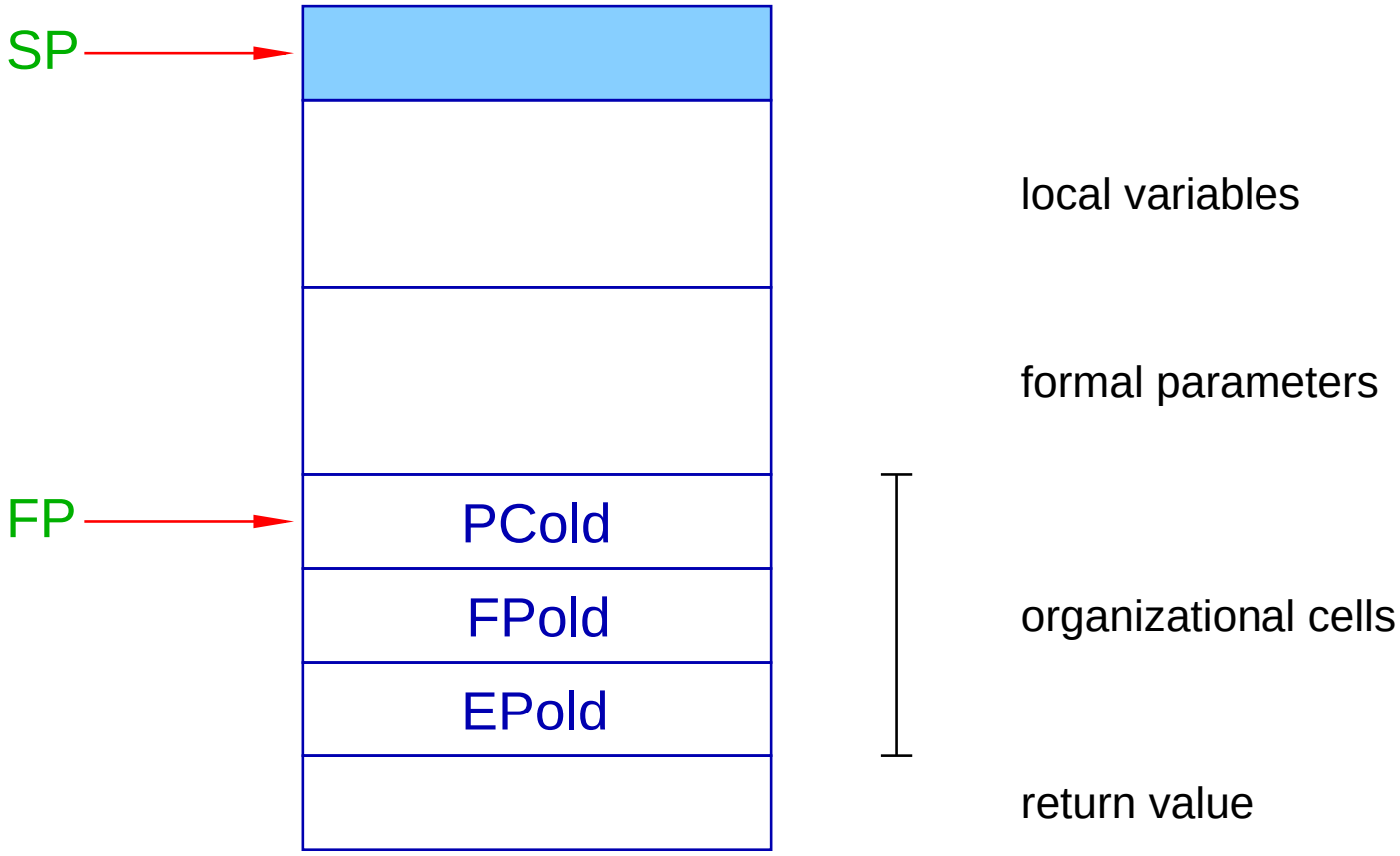


Funktsioonid

- Funktsiooni iga eksemplari fromaalsed parameetrid ja lokaalsed muutujad tuleb hoida eraldi.
- Selleks reserveerime funktsioonide iga väljakutse korral magasinis spetsiaalse mälupiirkonna, freimi (Stack Frame).
- FP (Frame Pointer) on register, mis viitab aktiivse freimi viimasele organisatoorsele pesale ja mida kasutatakse formaalsete parameetrite ja lokaalsete muutujate adresseerimiseks.

Funktsioonid

Freimi struktuur:



Funktsioonid

- Väljakutsuja peab peale väljakutsutud funktsiooni lõpetamist olema võimeline jätkama täitmist omas freimis.
- Seetõttu tuleb funktsiooni väljakutsel salvestada:
  - väljakutsuja freimi aadress FP;
  - koodi aadress, kust tuleb jätkata peale väljakutse lõpetamist (so. käsuregister PC);
  - väljakutsuja magasini maksimaalne võimalik aadress EP.
- Lihtsustus: eeldame, et funktsioonide resultaatväärtused mahuvad ühte mälupessa.

Funktsioonid

- Peame eristama kahte eri liiki muutujaid:
  - globaalsed muutujad, mis on defineeritud funktsioonidest väljaspool;
  - lokaalsed (ehk automaatsed) muuutujad (k.a. formaalsed parameetrid), mis on defineeritud funktsioonide siseselt.
- Aadresskeskkond ρ seob muutujanimedega paarid

$$(tag, a) \in \{G, L\} \times \mathbb{N}$$

- NB! Paljud keeled lubavad muutujate nähtavust kitsendada mingi ploki piiresse.
- Erinevate programmiosade transleerimine kasutab reeglina erinevaid aadresskeskkondi.

Funktsioonid

0
```
int i;
struct list {
    int info;
    struct list *next;
} *l;
```

1
```
int ith (struct list *x, int i) {
    if (i ≤ 1) return x→info;
    else return ith(x→next, i−1);
}
```

2
```
main () {
    int k;
    scanf("%d", &i);
    scanlist(&l);
    printf("%d", ith(l, i));
}
```

Funktsioonid

```
0   int i;
    struct list {
      int info;
      struct list *next;
    } *l;

1   int ith (struct list *x, int i) {
      if (i ≤ 1) return x→info;
      else return ith(x→next, i−1);
    }
```

```
2   main () {
      int k;
      scanf("%d", &i);
      scanlist(&l);
      printf("%d", ith(l, i));
    }
```

0 globaalne keskkond

ρ_0

$i \mapsto (G, 1)$

$l \mapsto (G, 2)$

$\text{ith} \mapsto (G, \_ith)$

$\text{main} \mapsto (G, \_main)$

Funktsioonid

0
```
int i;
struct list {
    int info;
    struct list *next;
} *l;
```

1
```
int ith (struct list *x, int i) {
    if (i ≤ 1) return x→info;
    else return ith(x→next, i−1);
}
```

2
```
main () {
    int k;
    scanf("%d", &i);
    scanlist(&l);
    printf("%d", ith(l, i));
}
```

---

1 keskkond fun-s ith

ρ_1

$x \mapsto (L, 1)$

$i \mapsto (L, 2)$

$l \mapsto (G, 2)$

$\text{ith} \mapsto (G, \_ith)$

$\text{main} \mapsto (G, \_main)$

Funktsioonid

0
```
int i;
struct list {
    int info;
    struct list *next;
} *l;
```

1
```
int ith (struct list *x, int i) {
    if (i ≤ 1) return x→info;
    else return ith(x→next, i−1);
}
```

2
```
main () {
    int k;
    scanf("%d", &i);
    scanlist(&l);
    printf("%d", ith(l, i));
}
```

---

2 keskkond fun-s main

ρ_2

$k \mapsto (L, 1)$

$i \mapsto (G, 1)$

$l \mapsto (G, 2)$

$\text{ith} \mapsto (G, \_ith)$

$\text{main} \mapsto (G, \_main)$

Funktsioonid

- Olgu f funktsioon, milles kutsutakse välja teine funktsioon g.
- Funktsiooni f nimetame kutsujaks (caller) ning funktsiooni g kutsutavaks (callee).
- Funktsiooni väljakutse puhul emiteeritav kood tuleb ära jagada kutsuja ja kutsutava vahel.
- Täpne jaotus sõltub sellest, et kes omab millist informatsiooni.

Funktsioonid

- Tegevused väljakutsel ja kutsutavasse sisenemisel:
  1. registrite FP ja EP salvestamine; } `mark`
  2. aktuaalsete argumentide arvutamine;
  3. kutsutava koodi $\_g$ algaadressi määramine;
  4. uue FP sättimine; } `call`
  5. PC salvestamine ja $\_g$ algusse hüppamine; } `call`
  6. uue EP sättimine; } `enter`
  7. lokaalsetele muutujatele mälu eraldamine. } `alloc`
- Tegevused kutsutavast lahkumisel:
  1. registrite FP, EP ja SP taastamine; } `return`
  2. tagasipöördumine f-i koodi; so. PC taastamine. } `return`

Funktsioonid

$$\begin{aligned} \text{code}_R\ (g(e_1,\ldots,e_n))\ \rho \quad = \quad & \texttt{mark} \\ & \text{code}_R\ e_1\ \rho \\ & \ldots \\ & \text{code}_R\ e_n\ \rho \\ & \text{code}_R\ g\ \rho \\ & \texttt{call n} \end{aligned}$$

- Aktuaalsete parameetritena esinevatel avaldistel leitakse nende R-väärtused
  - parameetrite edastamine väärtuse kaudu (call-by-value)
- Funktsioon g võib olla avaldis, mille R-väärtus on kutsutava funktsiooni algaadress

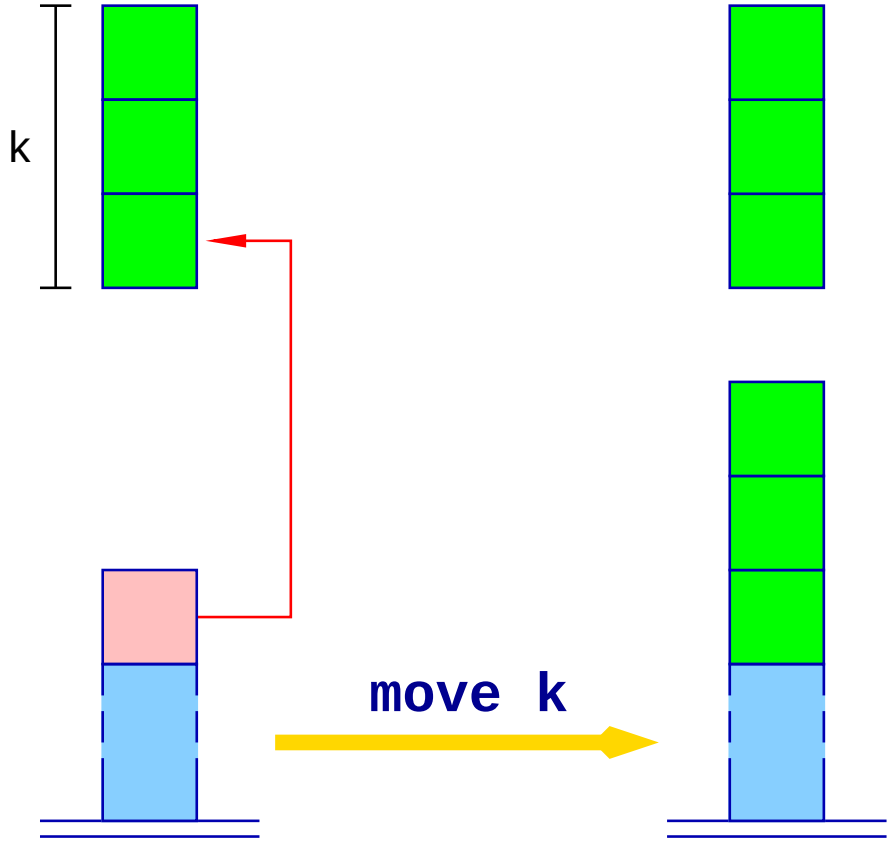
Funktsioonid

- Funktsiooni nimi on viitkonstant, mille R-väärtus on funktsiooni algaadress.
- Funktsiooni viida kaudu väärtuse võtmine väljastab selle sama viida.
  - Näide: deklaratsiooni **int** $(*)()g$; korral on väljakutsed $g()$ ja $(*g)()$ ekvivalentsed.
- Argumentidena antavad kirjed kopeeritakse.

$\text{code}_R\ f\ \rho$	=	`loadc` $(\rho\ f)$	f on funktsiooni nimi
$\text{code}_R\ (*e)\ \rho$	=	$\text{code}_R\ e\ \rho$	e on funktsiooni viit
$\text{code}_R\ e\ \rho$	=	$\text{code}_L\ e\ \rho$	e on kirje suurusega k
		`move k`	

Funktsioonid

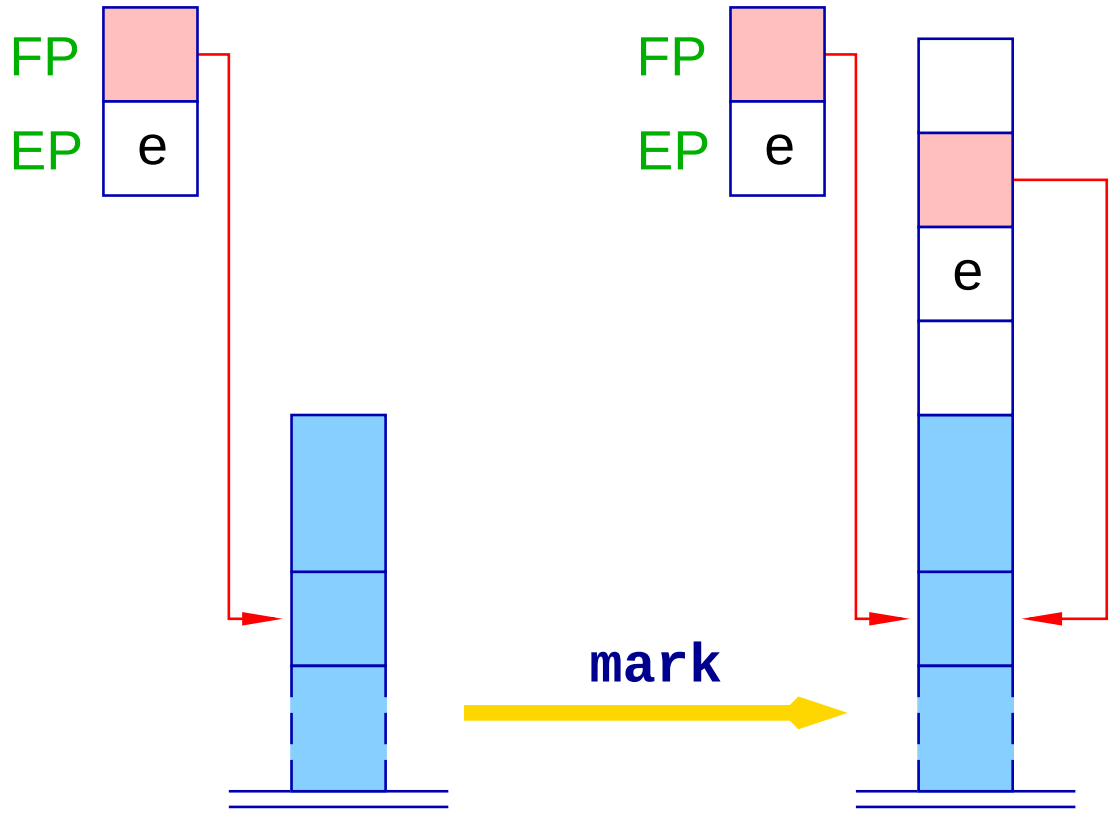
Käsk `move k` kopeerib k pesa magasini tippu:




```
for (i=k-1; i≥0; i--)
    S[SP+i] = S[S[SP]+i];
SP = SP + k - 1;
```

Funktsioonid

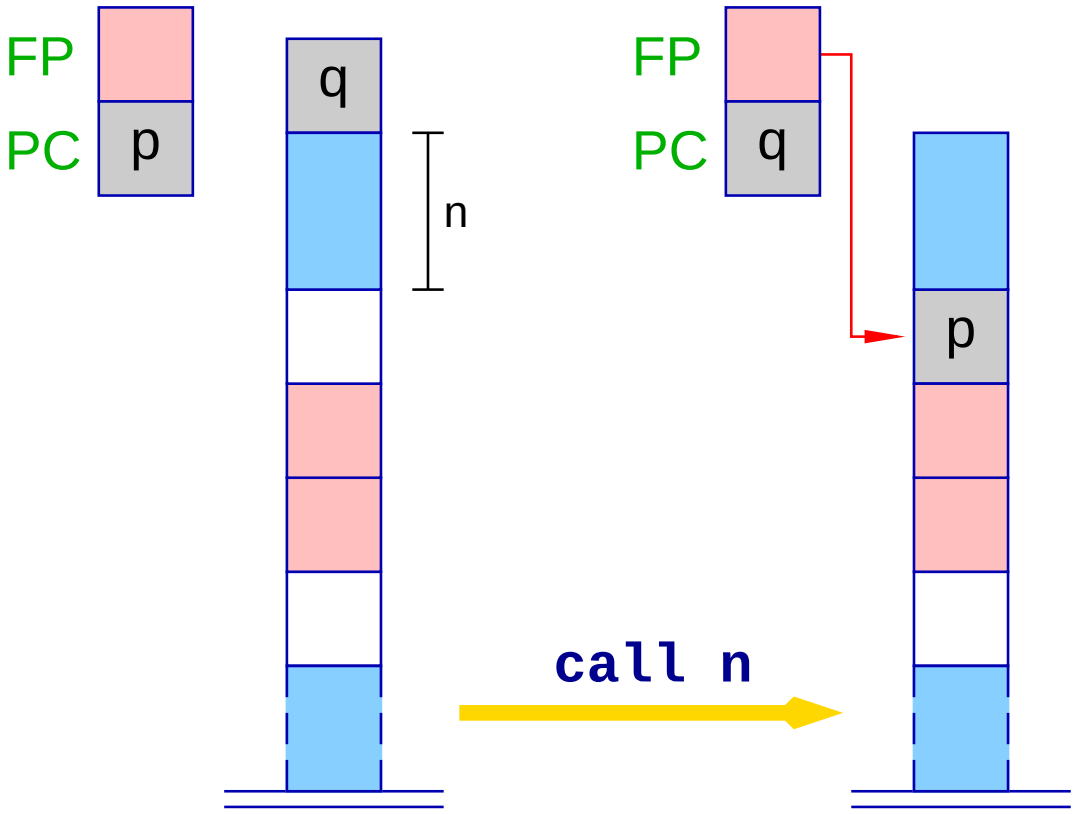
Käsk `mark` reserveerib mälu organisatoorsete pesade ja resultaatväärtuse jaoks ning salvestab registrid FP ja EP:




```
S[SP+2] = EP;
S[SP+3] = FP;
SP = SP + 4;
```

Funktsioonid

Käsk `call n` salvestab jätkuaadressi ning omistab registritele FP, SP ja PC uued väärtused:




```
FP = SP - n - 1;
S[FP] = PC;
PC = S[SP];
SP--;
```

Funktsioonid

$$\text{code}\ (t\ f\ (args)\{vars\ ss\})\ \rho \quad = \quad \_f\texttt{:}\ \texttt{enter q}$$
$$\texttt{alloc k}$$
$$\text{code}\ (ss)\ \rho_f$$
$$\texttt{return}$$

kus $q = maxS + k$

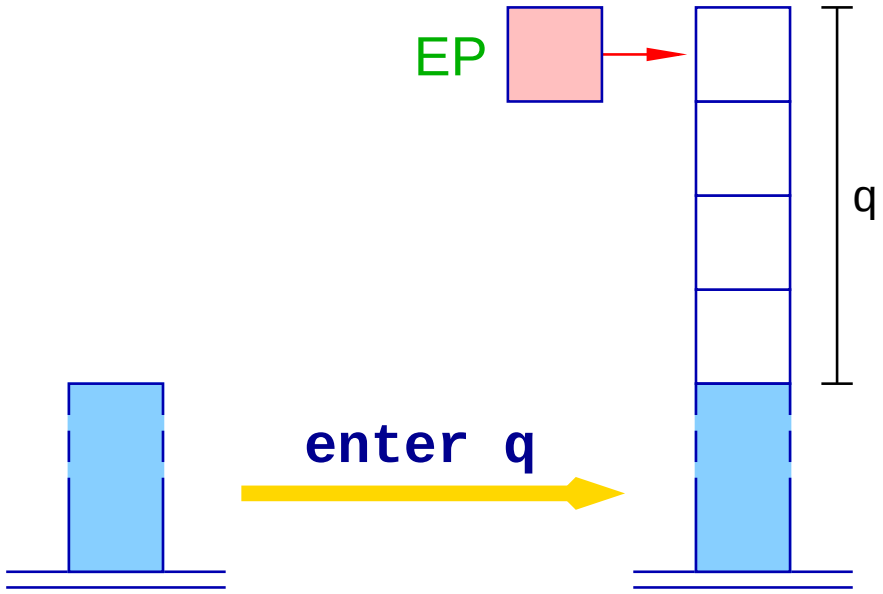
$maxS$ = lokaalse magasini maksimaalne sügavus

k = mälu lokaalsetele muutujatele

ρ_f = f-i aadresskeskkond

Funktsioonid

Käsk `enter q` seab registrile EP uue väärtuse:

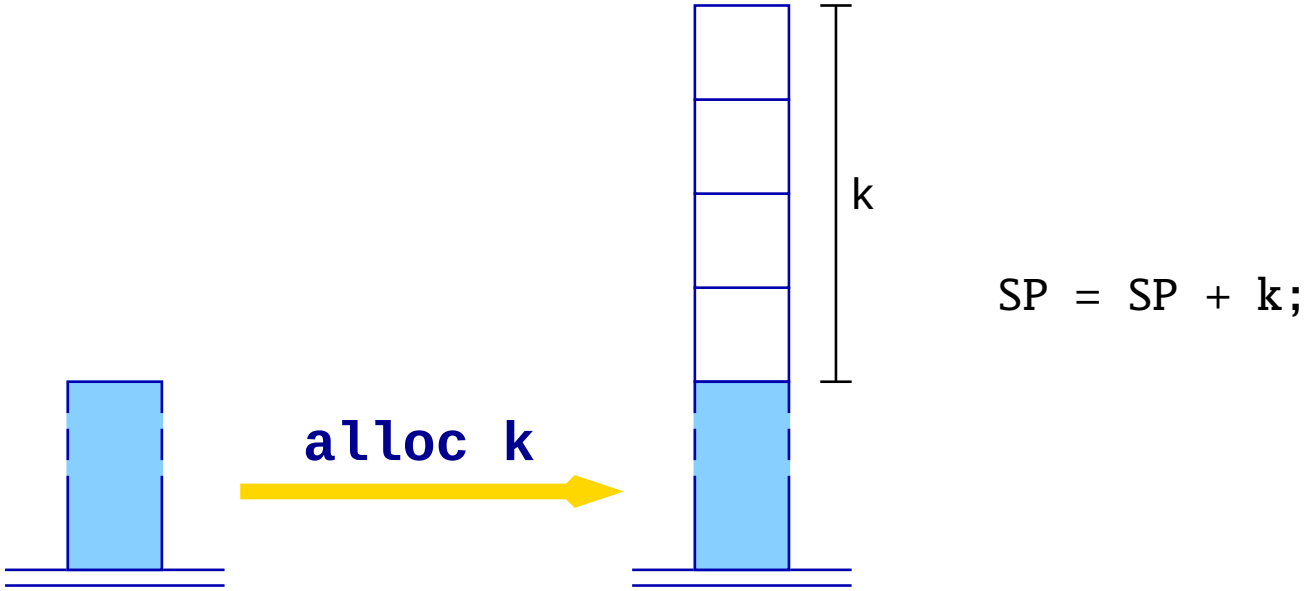



```
EP = SP + q;
if (EP≥NP)
    Error ("Stack Overflow");
```

NB! Kui mälu pole piisavalt, siis programmi töö katkestatakse.

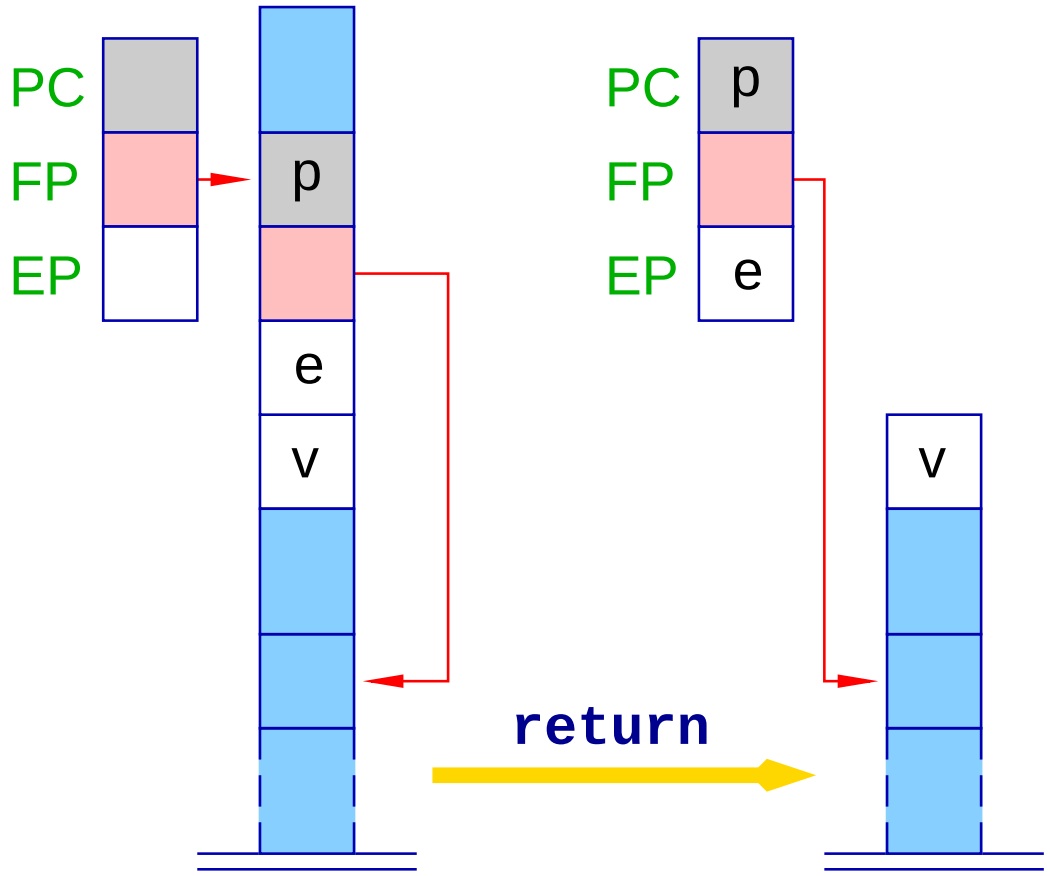
Funktsioonid

Käsk `alloc k` reserveerib magasinis mälu lokaalsete muutujate jaoks:



Funktsioonid

Käsk `return` taastab registrite PC, FP ja EP sisu ning jätab resultaatväärtuse magasini tippu:



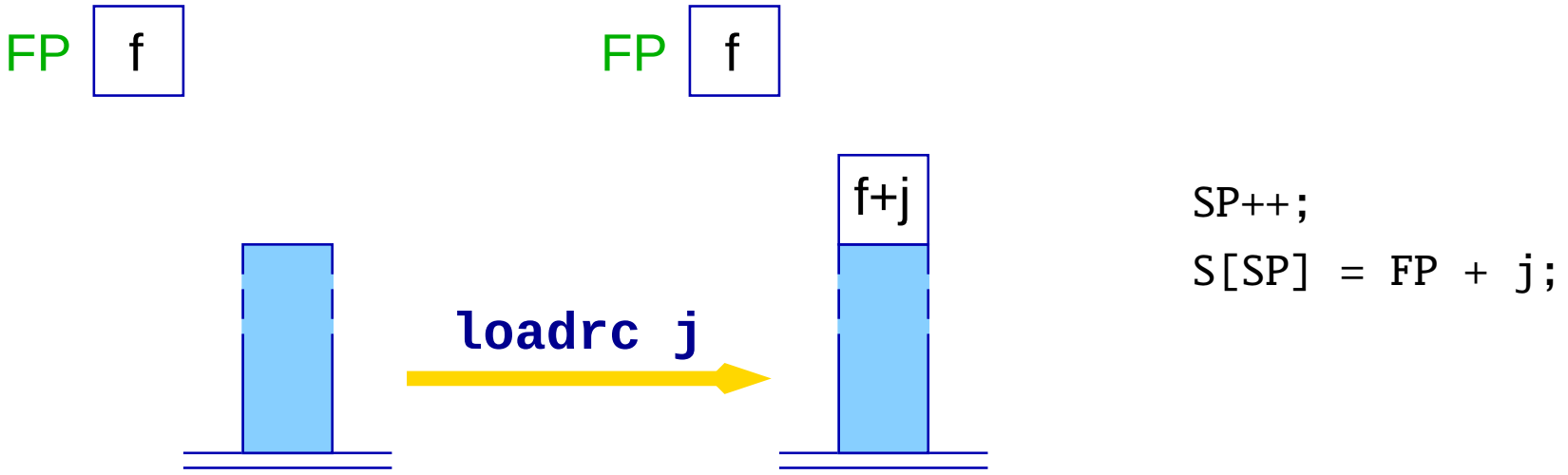

```
PC = S[FP];
EP = S[FP-2];
if (EP≥NP)
    Error ("Stack Overflow");
SP = FP - 3;
FP = S[SP+2];
```

Funktsioonid

Lokaalsetele muutujatele ja formaalsetele parameetritele ligipääs toimub relatiivselt registri FP suhtes:

$$\text{code}_L\ x\ \rho \quad = \quad \begin{cases} \texttt{loadc j} & \text{kui } \rho\ x = (G, j) \\ \texttt{loadrc j} & \text{kui } \rho\ x = (L, j) \end{cases}$$

Käsk `loadrc j` leiab FP ja `j` summa:




```
SP++;
S[SP] = FP + j;
```

Funktsioonid

Analoogselt käskudega `loada j` ja `storea j` toome sisse käsud `loadr j` ja `storer j`:

$$\begin{array}{lcl} \texttt{loadr j} & = & \texttt{loadrc j} \\ & & \texttt{load} \\ \texttt{storer j} & = & \texttt{loadrc j} \\ & & \texttt{store} \end{array}$$

return-lausele vastab väärtuse omistamine muutujale suhtaadressiga -3:

$$\begin{array}{lcl} \text{code}\ (\textbf{return}\ e;)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \texttt{storer -3} \\ & & \texttt{return} \end{array}$$

Funktsioonid

Näide:

int fac(**int** x) {
    if $(x \leq 0)$ **return** 1;
    else return $x * \text{fac}(x-1)$;
}

Siis $\rho_{fac} = \{x \mapsto (L, 1)\}$ ja emiteeritav kood on:

```
_fac: enter 7     loadc 1      A: loadr 1        mul
      alloc 0     storer -3       mark           storer -3
      loadr 1     return          loadr 1        return
      loadc 0     jump B          loadc 1     B: return
      leq                         sub
      jumpz A                     loadc _fac
                                  call 1
```

Kogu programmi transleerimine

Abstraktse masina olek enne programmi käivitamist:

SP = -1 FP = EP = 0 PC = 0 NP = MAX

Olgu $p \equiv vars\ fdef_1\ \ldots\ fdef_n$, kus $fdef_i$ on funktsiooni f_i definitsioon ja üks defineeritud funktsioonidest on nimega main.

Emiteeritav kood koosneb järmistest osadest:

- funktsioonide definitsioonidele $fdef_i$ vastav kood;
- globaalsetele muutujatele mälu reserveerimine;
- funktsiooni main(); väljakutse kood;
- käsk `halt`.

Kogu programmi transleerimine

$$
\begin{array}{lll}
\text{code } p\ \emptyset = & \texttt{enter}\ (k+6) & \texttt{pop} \\
& \texttt{alloc}\ (k+1) & \texttt{halt} \\
& \texttt{mark} & \_f_1 : \text{code } fdef_1\ \rho \\
& \texttt{loadc \_main} & \ldots \\
& \texttt{call 0} & \_f_n : \text{code } fdef_n\ \rho
\end{array}
$$

kus $\emptyset$ = tühi aadresskeskkond

ρ = globaalne aadresskeskkond

k = mälu globaalsetele muutujatele