CMa — lihtsustatud C abstraktne masin

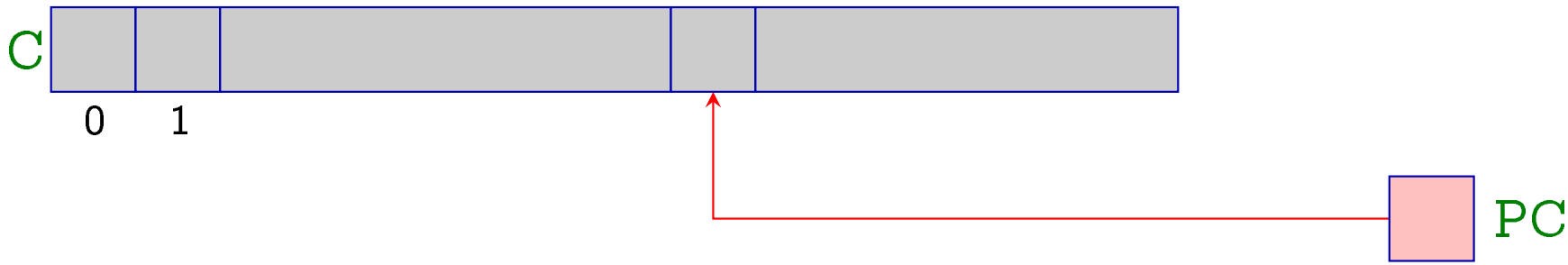
CMa arhitektuur
Avaldiste transleerimine
Lausete transleerimine
Funktsioonide transleerimine
Kogu programmi transleerimine

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

Kood:



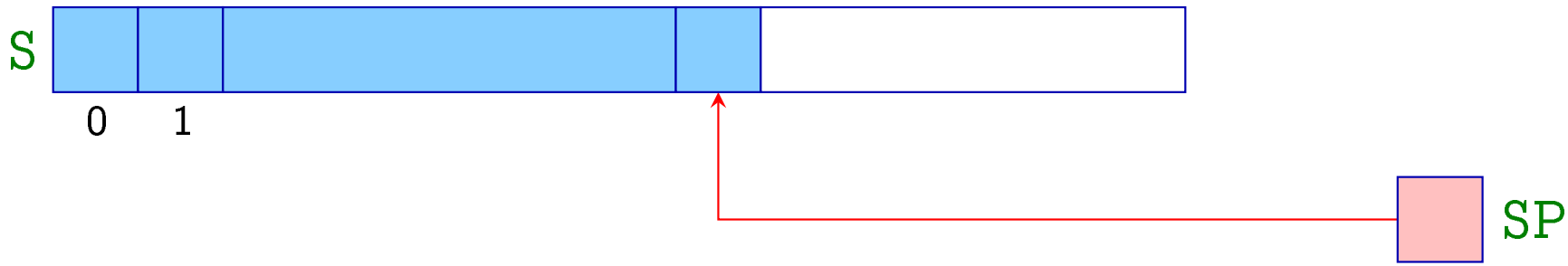
C = Code-store — mälupiirkond programmikoodi hoidmiseks; igas pesas on täpselt üks abstraktse masina käsk.

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

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

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 $(6 + 2) * 4 - 1$; s.t. genereerida käskude jada, mis

- leiab avaldise väärtuse ja
- lisab selle magasini tippu.

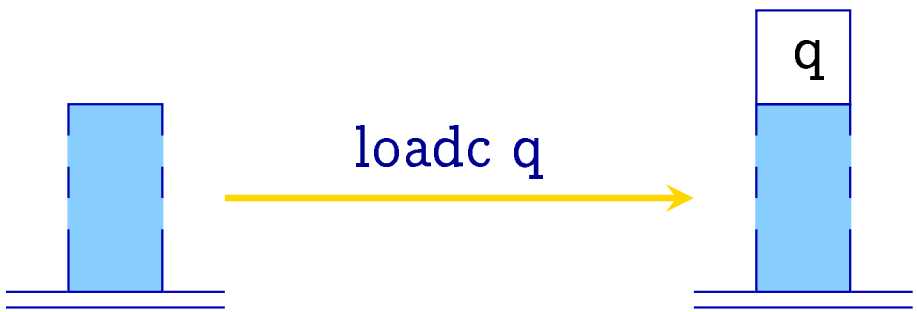
Idee:

- 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.

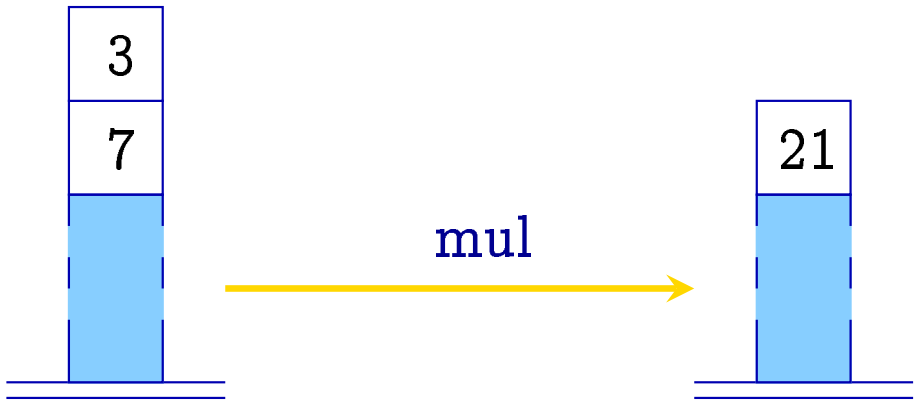



```
SP++;
S[SP] = q;
```

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



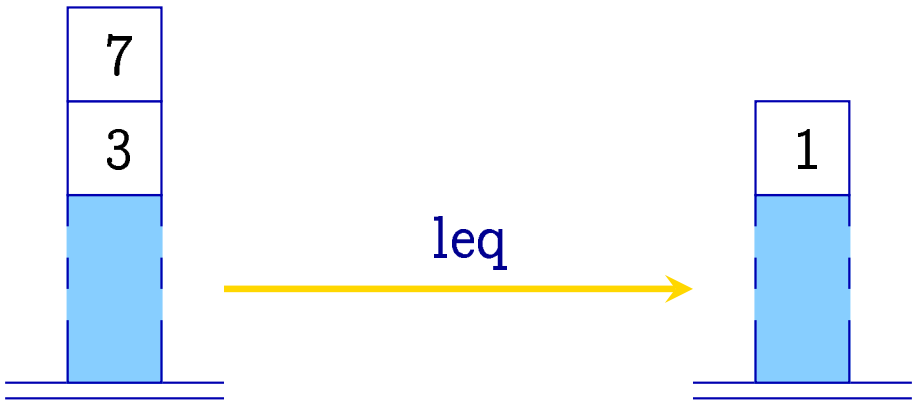

```
SP--;
S[SP] = S[SP] * S[SP+1];
```

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

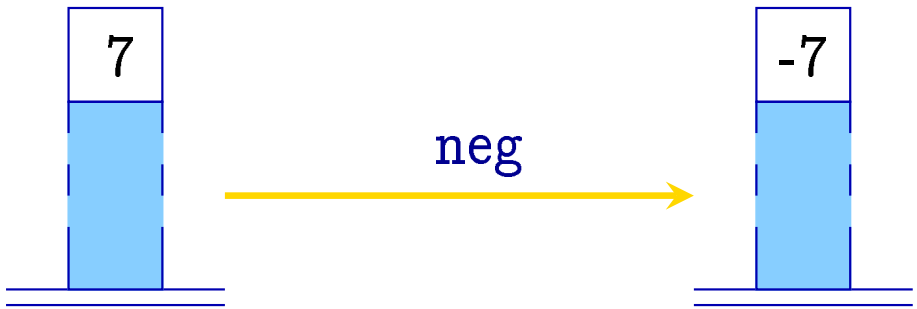
Näide: operaator `leq`



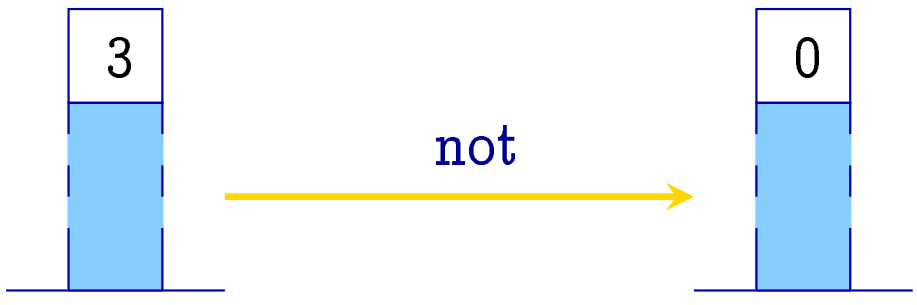
NB! Arv 0 esitab väära tõeväärtust, kõik teised täisarvud on tõesed väärtused.

Lihtsad avaldised ja omistamine

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




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



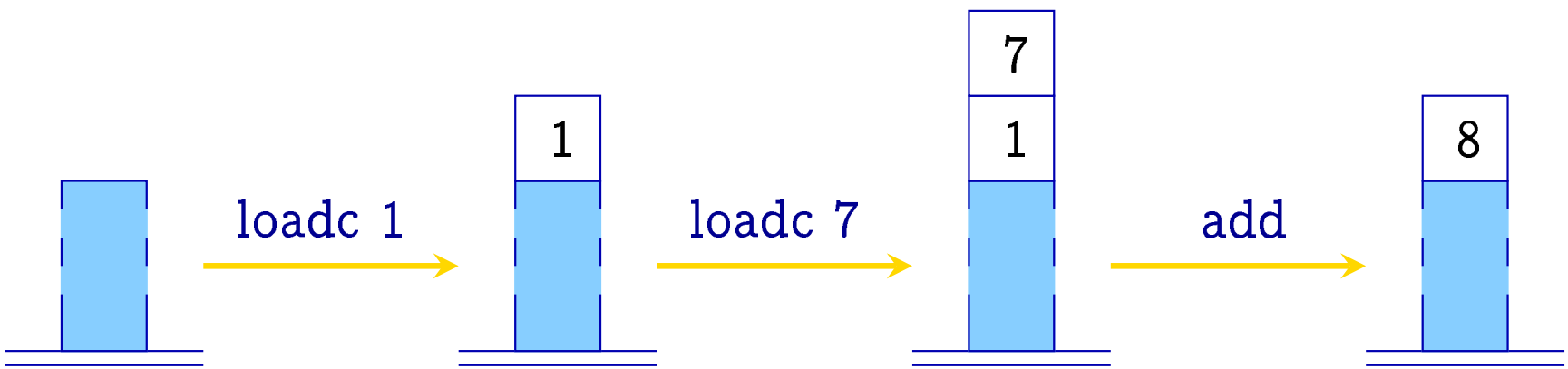

```
if (S[SP] ≠ 0)
    S[SP] = 0;
else
    S[SP] = 1;
```

Lihtsad avaldised ja omistamine

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

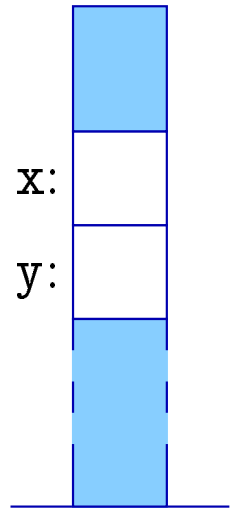
```
loadc 1
loadc 7
add
```

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 *aadresskeskkond* (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 R-väärtuse jaoks.
- NB! Mitte igal avaldisel pole L-väärtust (näit.: $x + 1$).

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

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

Lihtsad avaldised ja omistamine

- Binaarsete operaatorite transleerimine:

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

  - Analoogselt teiste binaaroperaatorite korral.
- Unaarsete operaatorite transleerimine:

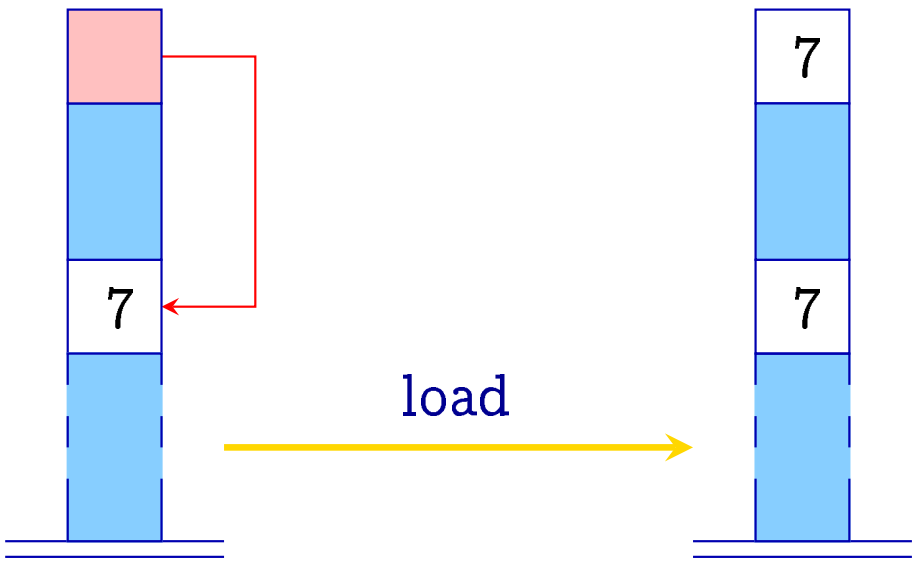
$$\begin{aligned} \mathtt{code}_R\ (-e)\ \rho \quad = \quad & \mathtt{code}_R\ e\ \rho \\ & \mathtt{neg} \end{aligned}$$

  - Analoogselt teiste unaaroperaatorite korral.
- Baasväärtuste transleerimine:

$$\mathtt{code}_R\ q\ \rho \quad = \quad \mathtt{loadc\ q}$$

Lihtsad avaldised ja omistamine

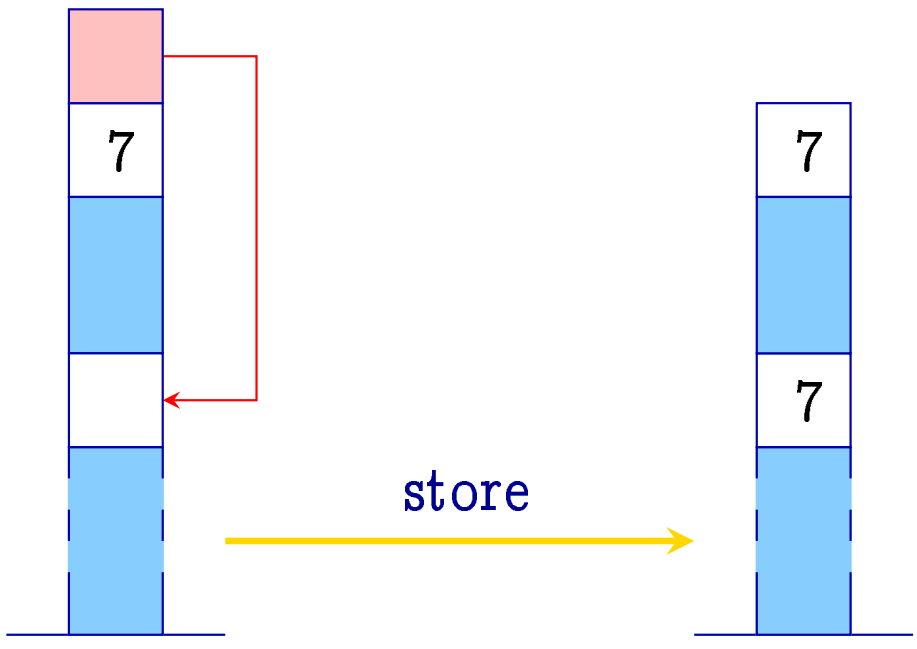
Käsk load salvestab magasini tippu selle magasini pesa sisu, mille aadress on ülemises pesas:




```
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--;
```

Lihtsad avaldised ja omistamine

- Muutujate transleerimine:

$$\begin{aligned} \mathtt{code}_L\ x\ \rho &= \mathtt{loadc}\ (\rho\ x) \\ \mathtt{code}_R\ x\ \rho &= \mathtt{code}_L\ x\ \rho \\ &\quad\ \mathtt{load} \end{aligned}$$

- Omistusavaldise transleerimine:

$$\begin{aligned} \mathtt{code}_R\ (x = e)\ \rho &= \mathtt{code}_R\ e\ \rho \\ &\quad\ \mathtt{code}_L\ x\ \rho \\ &\quad\ \mathtt{store} \end{aligned}$$

Lihtsad avaldised ja omistamine

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

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

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

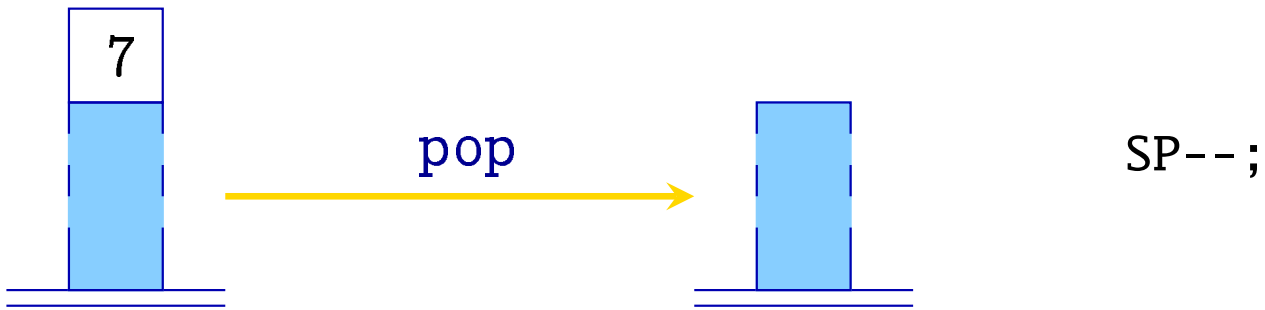
```
loada q    =  loadc q
              load
storea q   =  loadc q
              store
```

Laused ja lausete jadad

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

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

- Käsk pop eemaldab magasinist kõige ülemise peas:

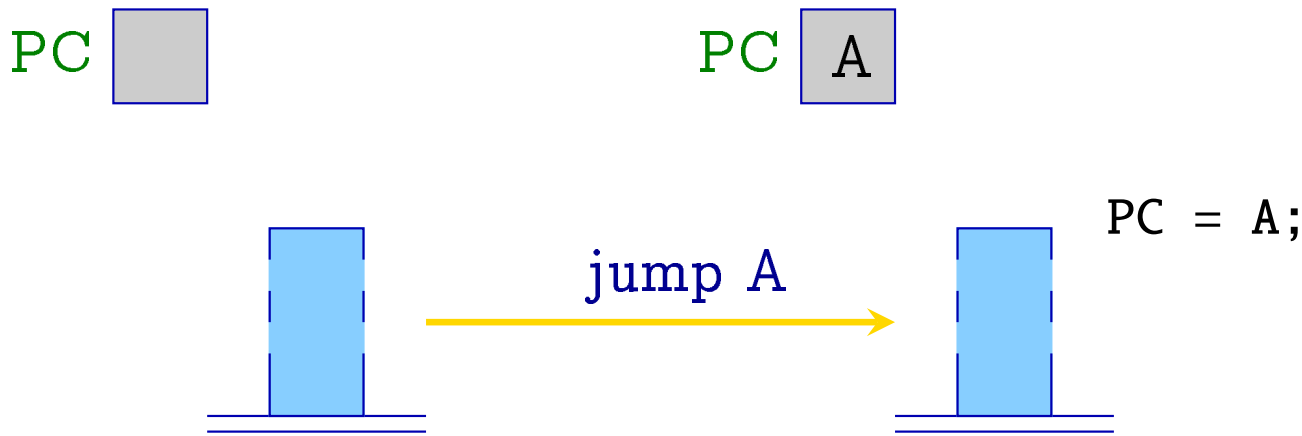


Tingimuslaused ja tsüklid

- Lihtsuse pärast kasutame hüpete sihtkohana sümbolmärgendeid, mis hiljem asendatakse absoluutaadressitega.
- 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).

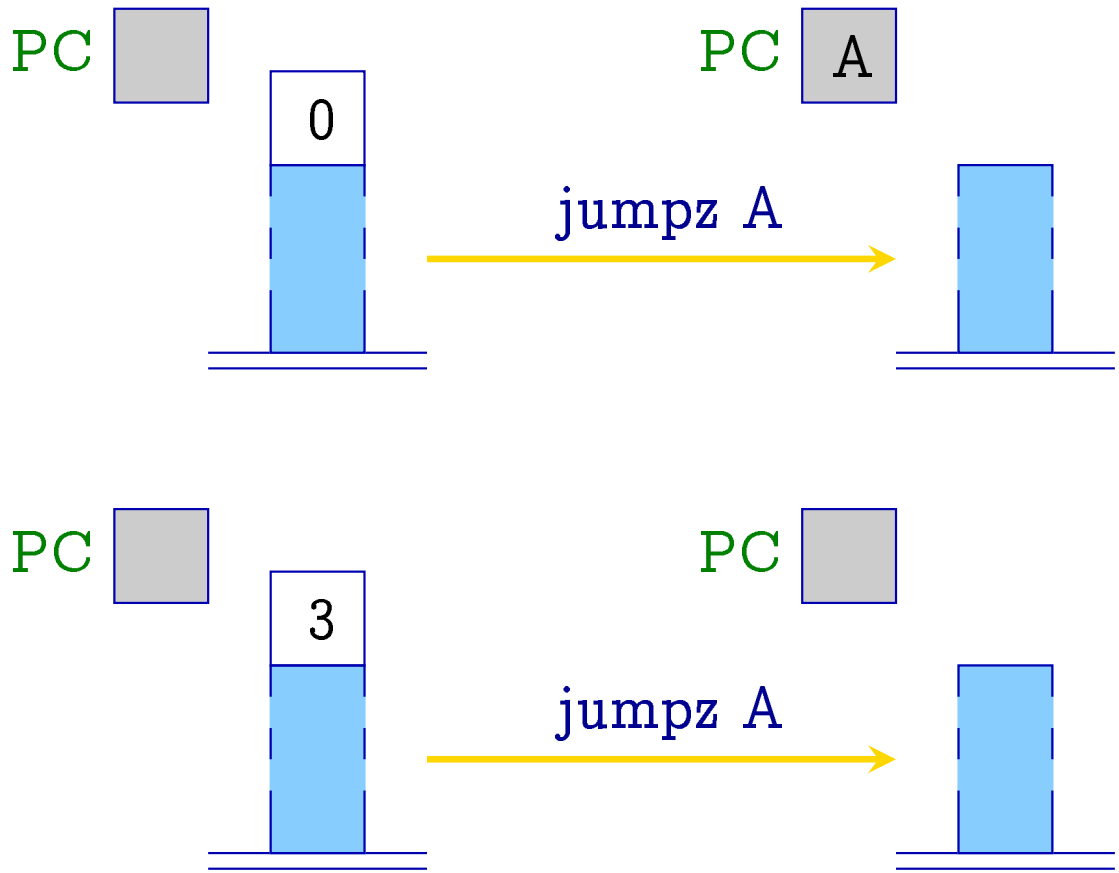
Tingimuslaused ja tsüklid

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



Tingimuslaused 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:



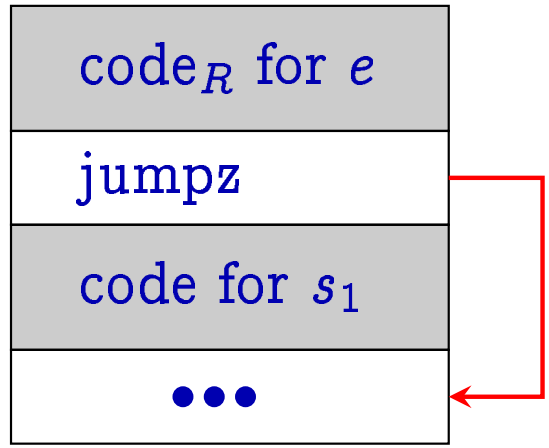

```
if (S[SP] = 0)
    PC = A;
SP--;
```

Tingimuslaused 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.

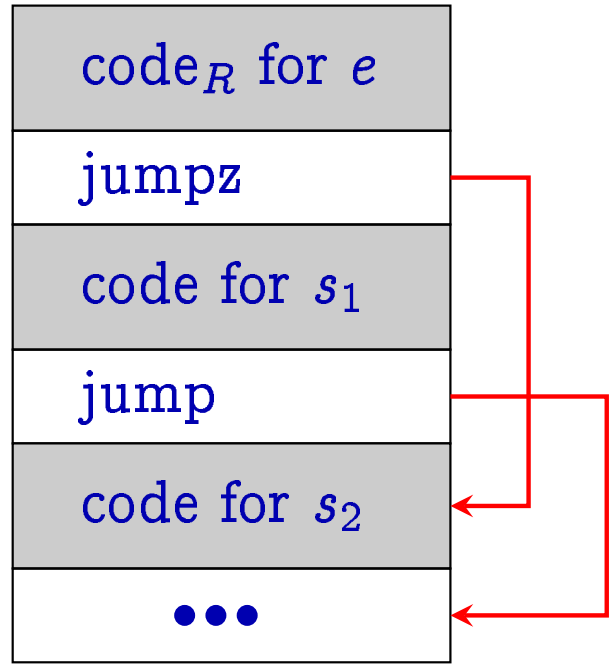
$$
\begin{aligned}
\texttt{code}\ (\textbf{if}\ (e)\ s_1)\ \rho\ \ = \\
&\texttt{code}_R\ e\ \rho \\
&\texttt{jumpz}\ \texttt{A} \\
&\texttt{code}\ s_1\ \rho \\
\texttt{A:}\ &\ldots
\end{aligned}
$$



Tingimuslaused ja tsüklid

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

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



Tingimuslaused ja tsüklid

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

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

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)

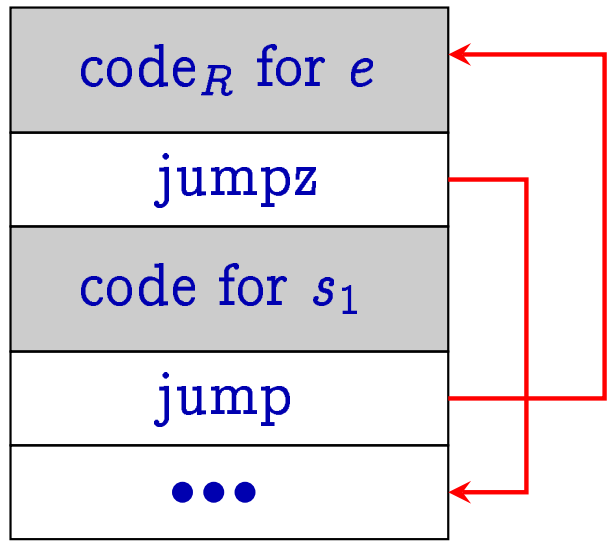
```
A: loada 7
   loada 4
   sub
   storea 7
   pop
B: ...
```

(iii)

Tingimuslaused ja tsüklid

- while-tsükli $s \equiv \textbf{while}\ (e)\ s_1$ transleerimine:

$$
\begin{aligned}
&\text{code}\ (\textbf{while}\ (e)\ s_1)\ \rho \quad = \\
&\qquad\qquad \text{A:}\ \text{code}_R\ e\ \rho \\
&\qquad\qquad\quad\ \ \texttt{jumpz B} \\
&\qquad\qquad\quad\ \ \text{code}\ s_1\ \rho \\
&\qquad\qquad\quad\ \ \texttt{jump A} \\
&\qquad\qquad \text{B:}\ \ldots
\end{aligned}
$$



Tingimuslaused 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 ρ 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)

Tingimuslaused ja tsüklid

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

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

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:

  $\mathtt{code}_R\ f\ \rho \quad = \quad \_f \quad = \quad$ starting address of f code

- Seega peab aadresskeskkond haldama ka funktsioonide nimesid!

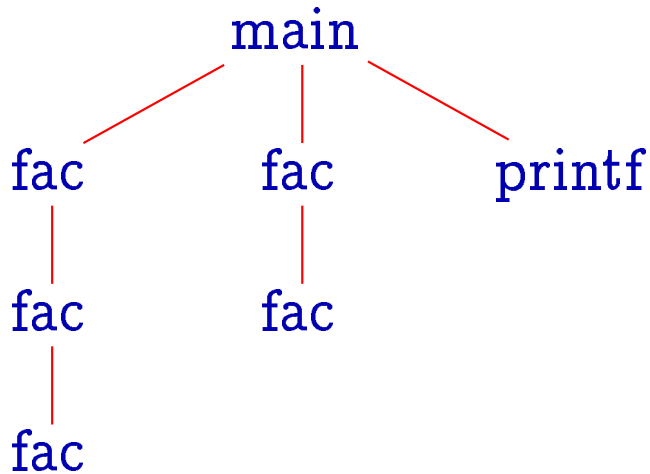
Funktsioonid

- 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.

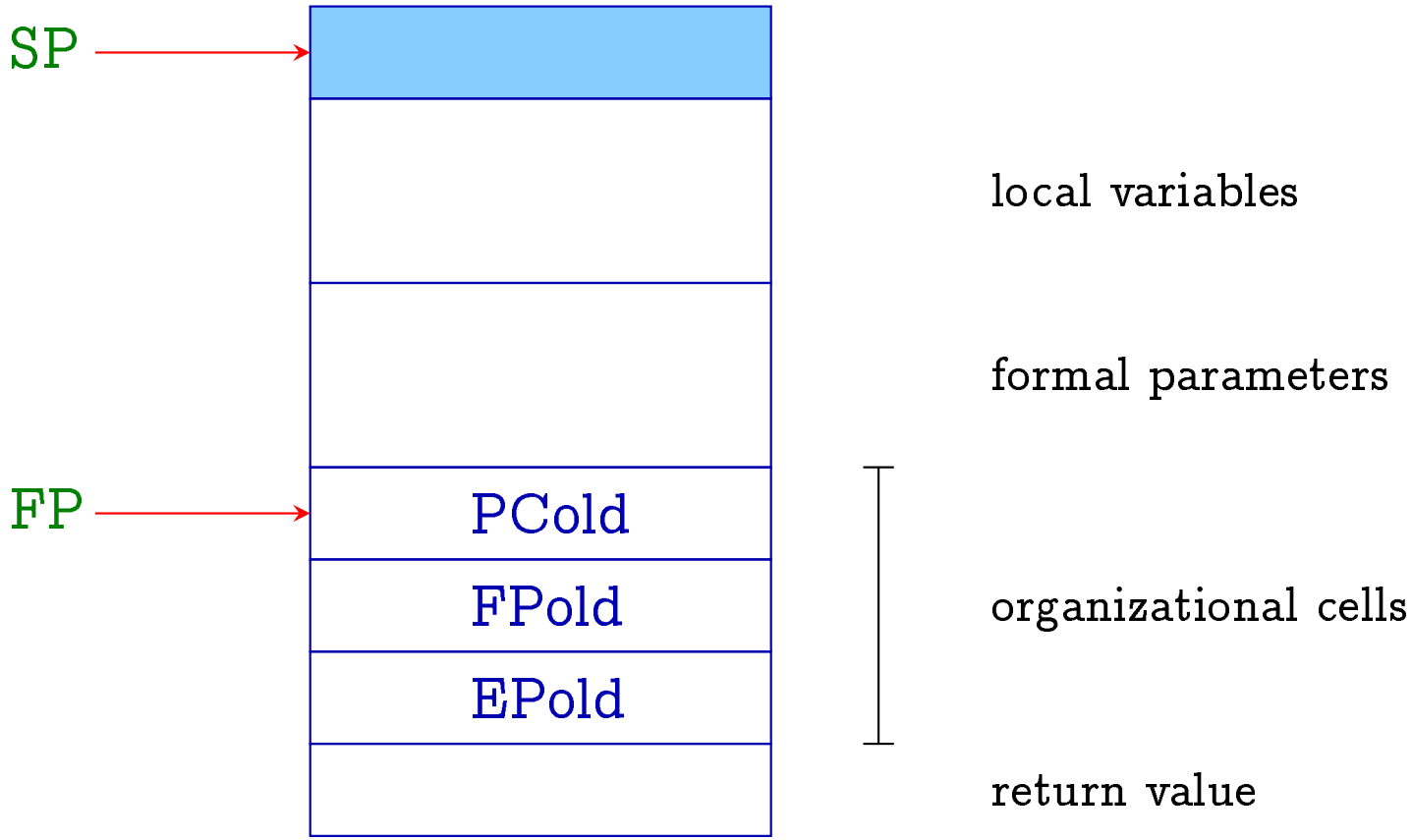


Funktsioonid

- Funktsiooni iga *eksemplari* formaalsed 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 global env.

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 env. for function ith

ρ_1

$$
\begin{aligned}
x &\mapsto (L, 1)\\
i &\mapsto (L, 2)\\
l &\mapsto (G, 2)\\
\text{ith} &\mapsto (G, \_ith)\\
\text{main} &\mapsto (G, \_main)
\end{aligned}
$$

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 env. for function main

ρ_2

$$
\begin{aligned}
k &\mapsto (L, 1)\\
i &\mapsto (G, 1)\\
l &\mapsto (G, 2)\\
\text{ith} &\mapsto (G, \_ith)\\
\text{main} &\mapsto (G, \_main)
\end{aligned}
$$

Funktsioonid

- Olgu f funktsioon, milles kutsutakse välja 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;
  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;
  2. tagasipöördumine f-i koodi; so. PC taastamine. } `return`

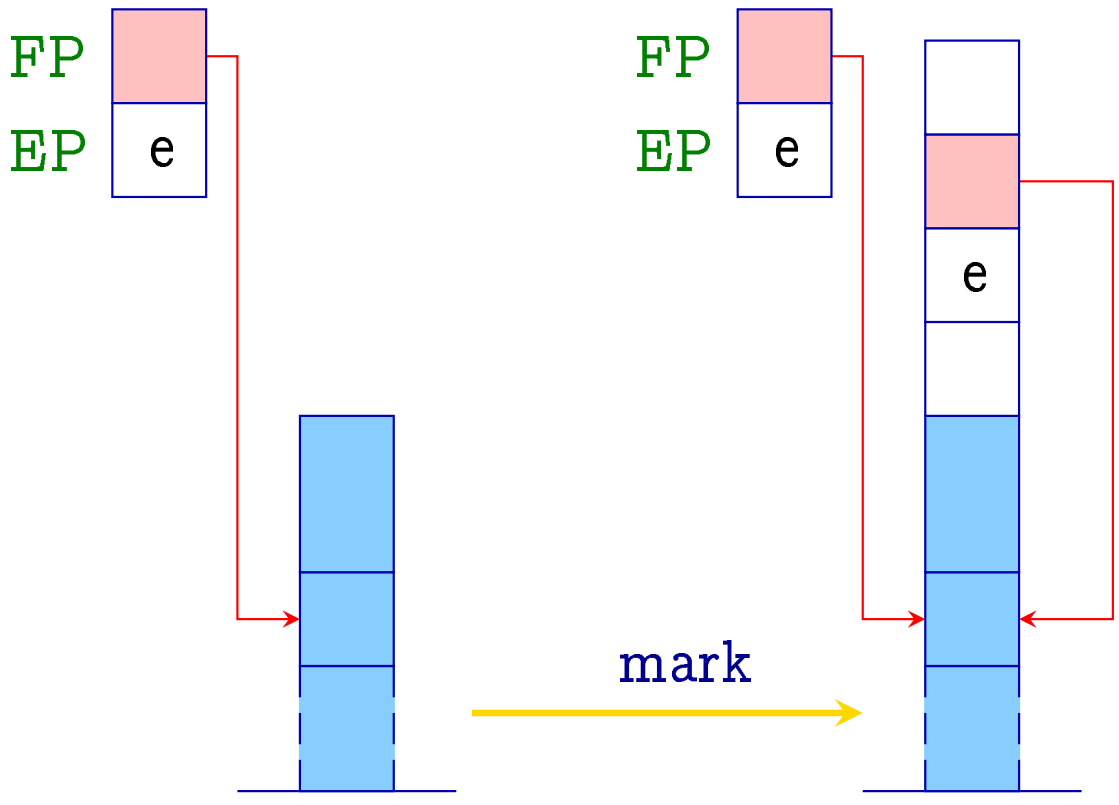
Funktsioonid

$$
\begin{array}{lcl}
\texttt{code}_R\ (g(e_1,\ldots,e_n))\ \rho & = & \texttt{mark} \\
 & & \texttt{code}_R\ e_1\ \rho \\
 & & \ldots \\
 & & \texttt{code}_R\ e_n\ \rho \\
 & & \texttt{code}_R\ g\ \rho \\
 & & \texttt{call n} \\
\texttt{code}_R\ f\ \rho & = & \texttt{loadc}\ (\rho\ f) \qquad\qquad f \text{ on funktsiooni nimi}
\end{array}
$$

- 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.
- Funktsiooni nimi on *viitkonstant*, mille R-väärtus on funktsiooni algaadress.

Funktsioonid

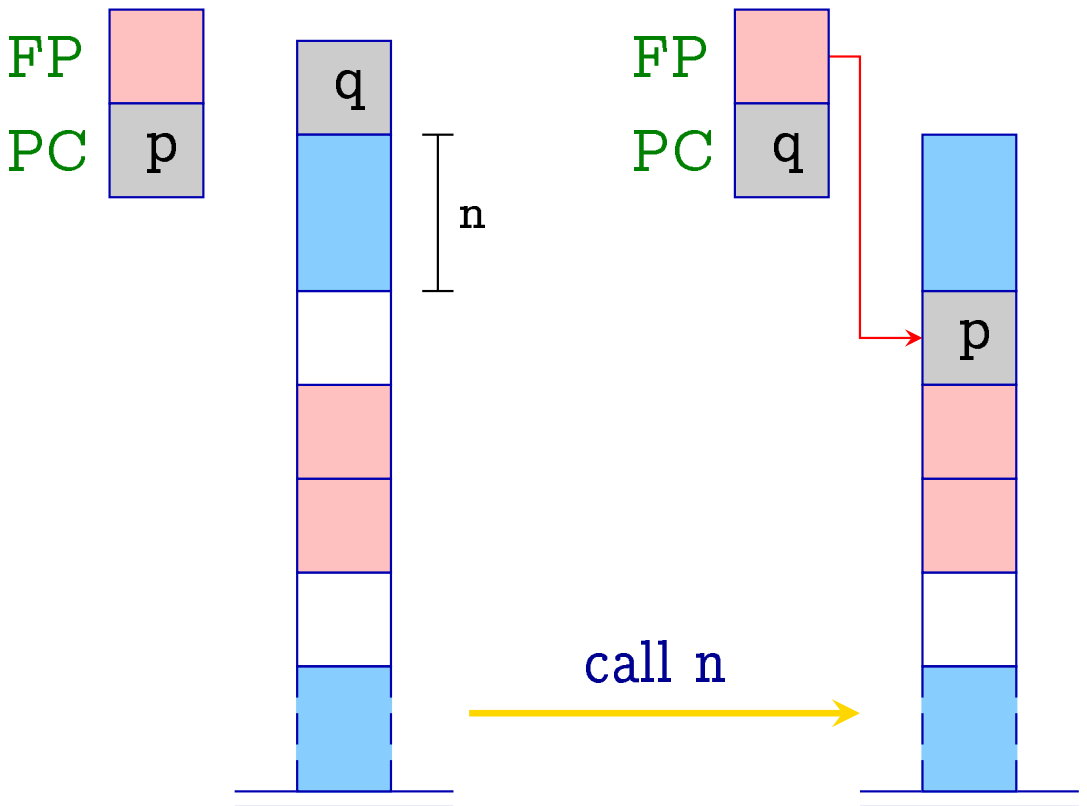
Käsk `mark` reserveerib mälu organisatoorsete pesade ja resultaatväärtuse jaoks ning salvestab registrid `FP` ja `EP`:




```
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

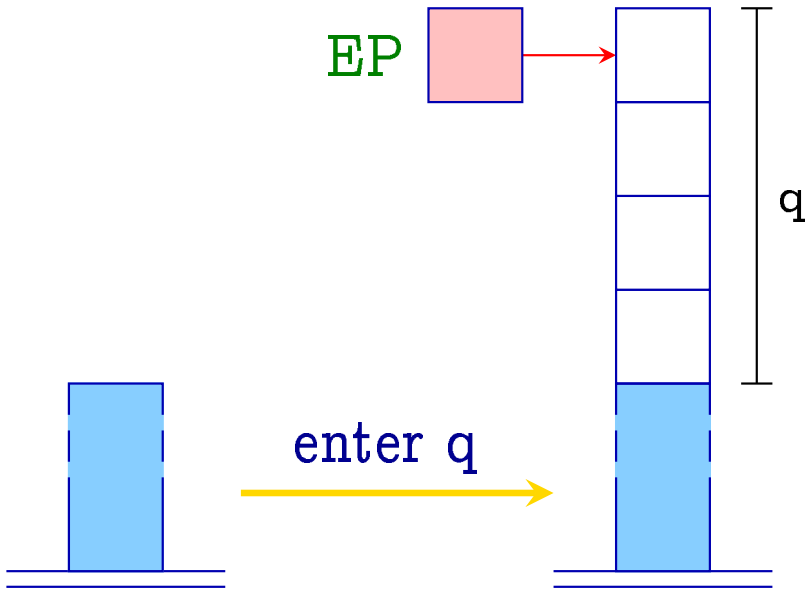
$$\texttt{code}\ (t\ f\ (args)\{vars\ ss\})\ \rho \quad = \quad \begin{array}{l} \_f\texttt{: enter q} \\ \qquad \texttt{alloc k} \\ \qquad \texttt{code}\ (ss)\ \rho_f \\ \qquad \texttt{return} \end{array}$$

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:

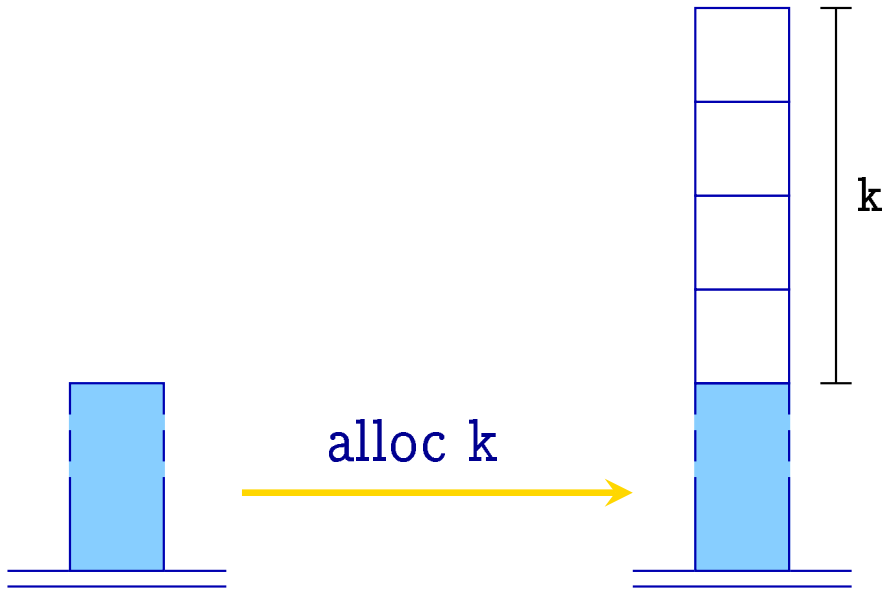



```
EP = SP + q;
if (EP≥NP)
    Error ("Stack Overflow");
```

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

Funktsioonid

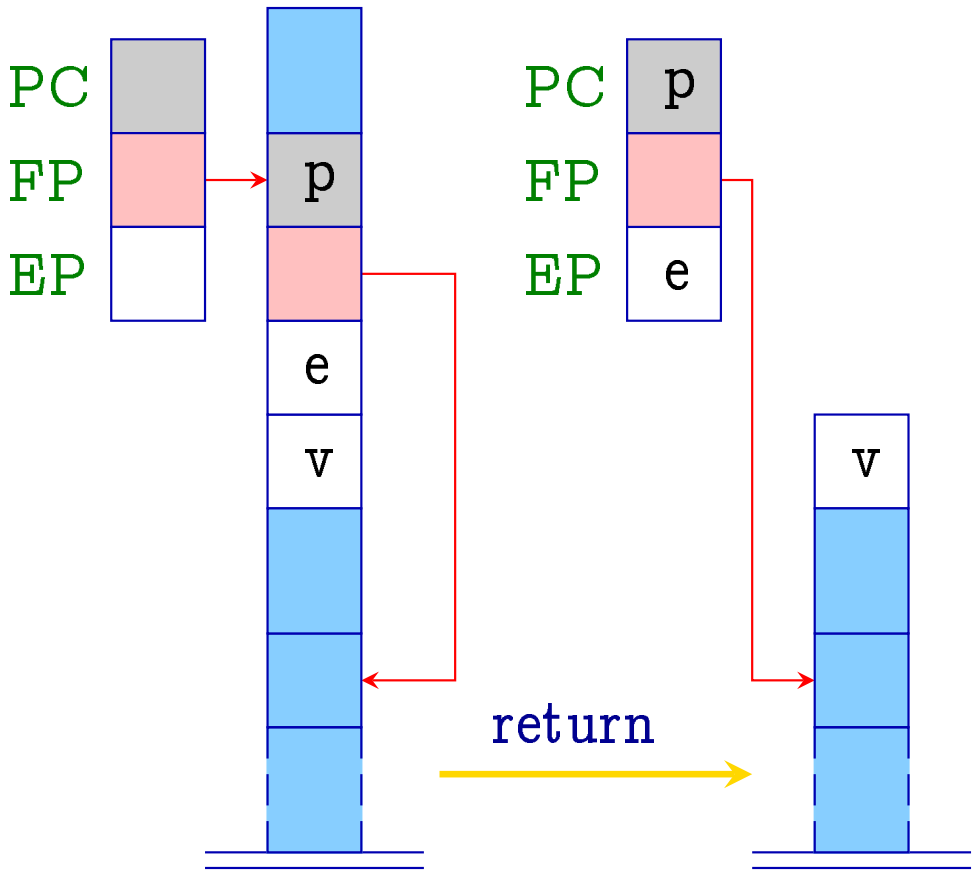
Käsk `alloc k` reserveerib magasinis mälu lokaalsete muutujate jaoks:




```
SP = SP + k;
```

Funktsioonid

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



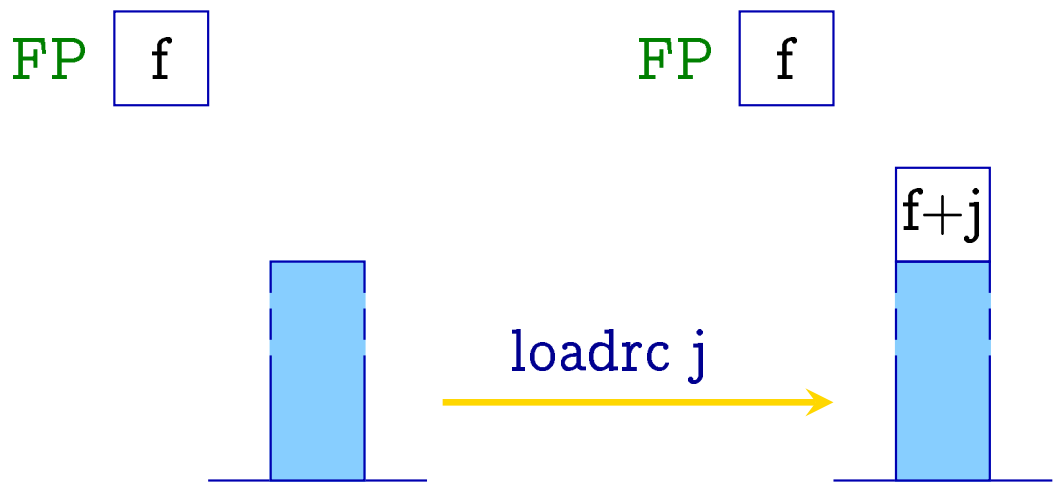

```
PC = S[FP];
EP = S[FP-2];
if (EP≥NP)
    Error ("Stack Overflow");
SP = FP - 3;
FP = S[SP+2];
```

Funktsioonid

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

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

Käsk `loadrc j` arvutab 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:

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

Funktsioonid

Näide:

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

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

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

Kogu programmi transleerimine

Abstraktse masina olek enne programmi käivitamist:

$$\texttt{SP} = \text{-1} \qquad \texttt{FP} = \texttt{EP} = 0 \qquad \texttt{PC} = 0 \qquad \texttt{NP} = \text{MAX}$$

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

$$
\begin{array}{lll}
\texttt{code}\ p\ \emptyset & = & \texttt{enter}\ (k+6) \\
 & & \texttt{alloc}\ (k+1) \\
 & & \texttt{mark} \\
 & & \texttt{loadc}\ \texttt{\_main} \\
 & & \texttt{call}\ \texttt{0} \\
 & & \\
 & & \texttt{pop} \\
 & & \texttt{halt} \\
 & & \texttt{\_}f_1 : \texttt{code}\ fdef_1\ \rho \\
 & & \ldots \\
 & & \texttt{\_}f_n : \texttt{code}\ fdef_n\ \rho
\end{array}
$$

kus $\emptyset$ = tühi aadresskeskkond

ρ = globaalne aadresskeskkond

k = mälu globaalsetele muutujatele