

# CMa — lihtsustatud C abstraktne masin

CMa arhitektuur

Avaldiste transleerimine

Lausete transleerimine

Staatiline ja dünaamiline mäluhaldus

Funktsioonide transleerimine

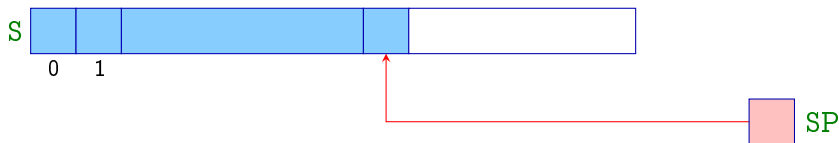
Kogu programmi transleerimine

## CMa arhitektuur

- Iga abstraktne masin määrab mingi hulga *käsk*e, 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:



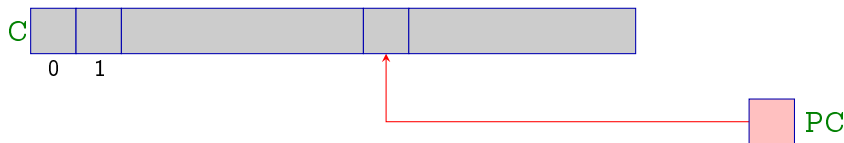
**S** = **Stack** — mälu piirkond andmete hoidmiseks, kus uute elementide lisamine/eemaldamine käib LIFO printsiibil.

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

**Lihtsustus:** kõik mittestruktuurset tüüpi väärtused on sama suurusega ja mahuvad ühte magasinisse.

# CMa arhitektuur

Kood:



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

**PC** = **P**rogram **C**ounter — register, mis sisaldab *järgmisena* täidetava käsu adressit.

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

- leiab avaldise väärtuse ja
- lisab selle magasinini tippu.

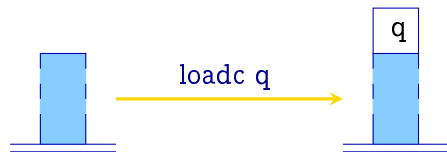
**Idee:**

- kõigepealt arvutame alamavaldiste väärtused,
- salvestame need magasinini 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 magasin tippu.



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

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

**NB!** Registri `SP` sisu on joonisel esitatud ainult kaudselt magasin kõrguse kaudu.

## Lihtsad avaldised ja omistamine



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

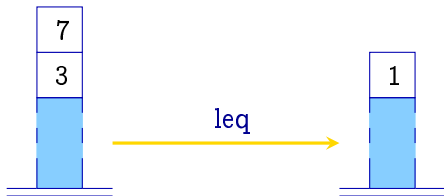
Käsk `mul` eeldab magasinis kahte argumenti, tarvitab need ära ja salvestab nende korrutise magasini tippu.

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

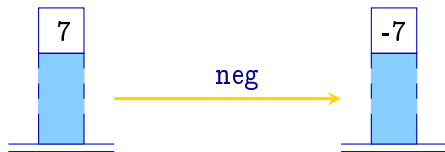
Näide: operaator `leq`



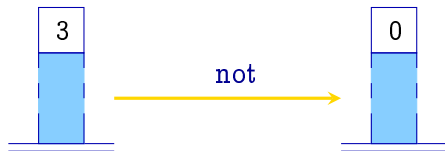
**NB!** Arv 0 esitab väärat tõeväärtust, kõik teised täisarvud on tõesed väärtused.

## Lihtsad avaldised ja omistamine

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



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



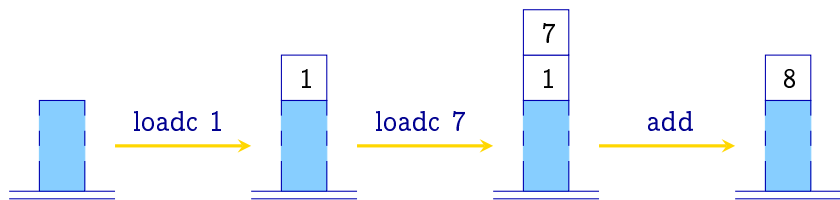
```
if (S[SP] ≠ 0)
    S[SP] = 0;
else
    S[SP] = 1;
```

## Lihtsad avaldised ja omistamine

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

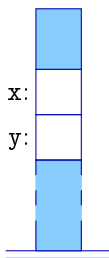
```
loadc 1  
loadc 7  
add
```

Selle koodi täitmine annab:



## Lihtsad avaldised ja omistamine

- Muutujatele vastavad magasinis  $S$  pesad:



- Koodi genereerimist kirjeldavad funktsioonid `code`, `codeL` ja `codeR`.
- 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  $code_L$   $e$   $\rho$  emiteerib keskkonnas  $\rho$  avaldise  $e$   $L$ -väärtust arvutava koodi.
- Funktsioon  $code_R$   $e$   $\rho$  teeb sama  $R$ -väärtuse jaoks.
- **NB!** Mitte igal avaldisel pole  $L$ -väärtust (näit.:  $x + 1$ ).

## Lihtsad avaldised ja omistamine

- Binaarsete operaatorite transleerimine:

$$\text{code}_R (e_1 + e_2) \rho = \begin{array}{l} \text{code}_R e_1 \rho \\ \text{code}_R e_2 \rho \\ \text{add} \end{array}$$

– Analoogselt teiste binaaroperaatorite korral.

- Unaarsete operaatorite transleerimine:

$$\text{code}_R (-e) \rho = \begin{array}{l} \text{code}_R e \rho \\ \text{neg} \end{array}$$

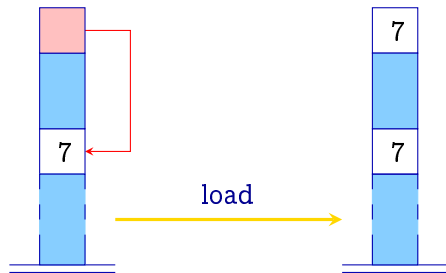
– Analoogselt teiste unaaroperaatorite korral.

- Baasväärtuste transleerimine:

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

## Lihtsad avaldised ja omistamine

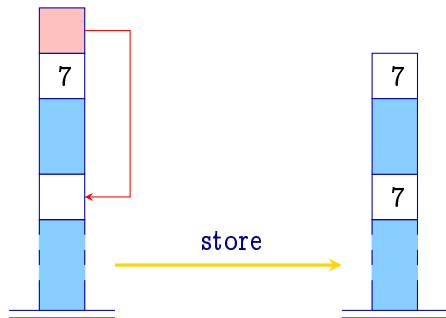
Käsk `load` salvestab magasinini tippu selle magasinini pesa sisu, mille aadress on ülemises pesas:



`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 magasinini tippu:



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



## Lihtsad avaldised ja omistamine

- Muutujate transleerimine:

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

- Omistusavaldise transleerimine:

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

## Lihtsad avaldised ja omistamine

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

$$\text{code}(e;)\rho = \text{code}_R e \rho$$

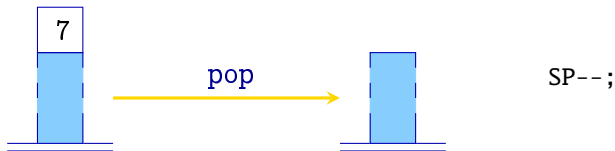
**pop**

$$\text{code}(s\ ss)\rho = \text{code}\ s \rho$$

$$\text{code}\ ss \rho$$

$$\text{code}\ \varepsilon \rho = \quad // \text{tühi käskude jada}$$

- Käsk **pop** eemaldab magasinist kõige ülemise peas:

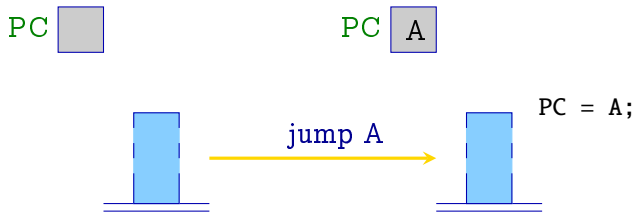


## Tingimuslaused ja tsüklid

- 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**).

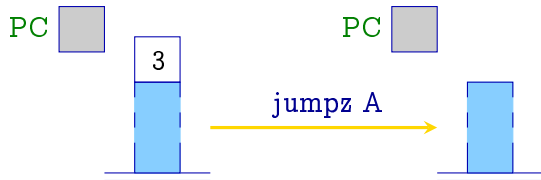
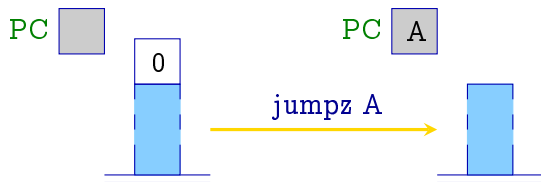
# Tingimuslaused ja tsüklid

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



## Tingimuslaused ja tsüklid

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



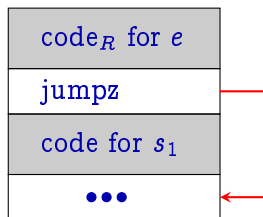
```
if (S[SP] == 0)
    PC = A;
    SP--;
```

## Tingimuslaused ja tsüklid

Üheharulise tingimuslause  $s \equiv \text{if } (e) s_1$  transleerimine:

- genereerime koodi tingimuse  $e$  ja lause  $s_1$  jaoks;
- lisame tingimusliku hüppekäsu nende vahele.

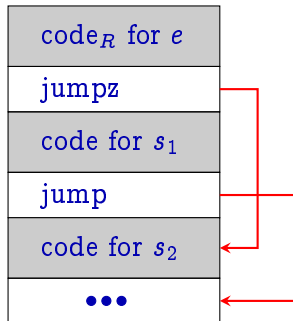
```
code (if (e) s1) ρ =  
    codeR e ρ  
    jumpz A  
    code s1 ρ  
A: ...
```



## Tingimuslaused ja tsüklid

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

```
code (if (e) s1 else s2) ρ =  
    codeR e ρ  
    jumpz A  
    code s1 ρ  
    jump B  
A: code s2 ρ  
B: ...
```





## Tingimuslaused ja tsüklid

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

```
s ≡ if (x > y)           (i)
      x = x - y;        (ii)
      else y = y - x;   (iii)
```

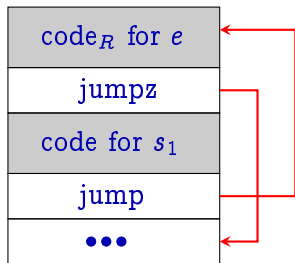
siis code  $s$   $\rho$  emiteerib koodi:

loada 4	loada 4	A: loada 7
loada 7	loada 7	loada 4
ge	sub	sub
jumpz A	storea 4	storea 7
	pop	pop
	jump B	B: ...
(i)	(ii)	(iii)

# Tingimuslaused ja tsükliid

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

`code` ( $\mathbf{while} (e) s_1$ )  $\rho =$   
A: `codeR`  $e$   $\rho$   
  `jumpz` B  
  `code`  $s_1$   $\rho$   
  `jump` A  
B: ...



## Tingimuslaused ja tsüklid

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

```
s ≡ while (a > 0) {           (i)
    c = c + 1;                 (ii)
    a = a - b;                 (iii)
}
```

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	
(i)	pop	pop	
	(ii)	(iii)	

## Tingimuslaused ja tsükliid

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

```
code (for (e1; e2; e3) s1) ρ = codeR e1 ρ
                                pop
                                A: codeR e2 ρ
                                jumpz B
                                code s1 ρ
                                codeR e3 ρ
                                pop
                                jump A
                                B: ...
```

## 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 mälupeassa.
- Seome muutujatega aadressid nende deklareerimise järjekorras alustades aadressist 1.
- Seega, deklaratsioonide  $d \equiv t_1 x_1; \dots t_k x_k$ ; ( $t_i$  on baastüüp) korral saame aadresskeskkonna  $\rho$  sellise, et

$$\rho x_i = i, \quad i = 1, \dots, k$$

## 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:

$$|t| = \begin{cases} 1 & \text{if } t \text{ is a primitive type} \\ k \cdot |t'| & \text{if } t \equiv t'[k] \end{cases}$$

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

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

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

## Massiivid, kirjed ning staatiline mäluhaldus

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

```
codeL (a[e]) ρ = loadc (ρ a)
                  codeR e ρ
                  loadc |t|
                  mul
                  add
```



## Massiivid, kirjed ning staatiline mäluhaldus

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

```
codeL (e1[e2]) ρ = codeR e1 ρ  
                    codeR e2 ρ  
                    loadc |t|  
                    mul  
                    add
```

```
codeR e ρ = codeL e ρ      e is an array
```

## 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  $\rho_{st}$ .
- 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  $t \equiv \text{struct } \{ t_1 c_1; \dots t_k c_k; \}$ , siis

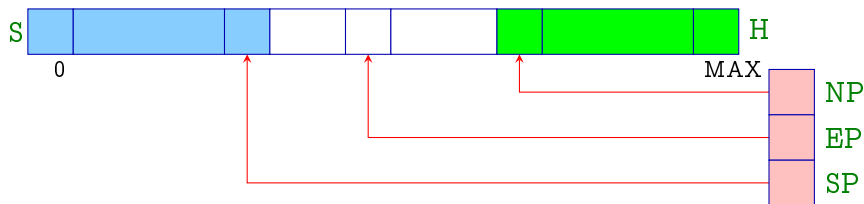
$$\begin{aligned} |t| &= \sum_{i=1}^k |t_i| \\ \rho c_1 &= 0 \\ \rho c_i &= \rho c_{i-1} + |t_{i-1}| \quad i > 1 \end{aligned}$$

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

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

# Viidad ja dünaamiline mäluhaldus

Kuhi:



- H** = **H**eap — mälupiirkond dünaamilise elueaga andmetele.
- NP** = **N**ew-**P**ointer — register, mis sisaldab kuhja kõige alumise täidetud pesa aadressit.
- EP** = **E**xtrême-**P**ointer — register, mis sisaldab kõige ülemise pesa aadressi, kuhu **SP** võib antud funktsiooni täitmisel viidata.

## Viidad ja dünaamiline mäluhaldus

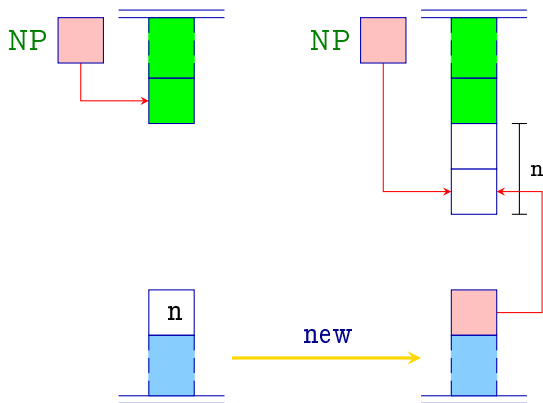
- Magasin ja kuhi kasvavad üksteise suunas, aga ei tohi kattuda ([stack overflow](#)).
- "Kokkupõrge" võib olla põhjustatud nii **SP** suurendamisest kui **NP** vähendamisest.
- **EP** aitab vältida ületäitumist magasinini operatsioonide korral.
- **EP** väärtuse saab määrata staatiliselt.
- Kuhjast 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).

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

# 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 mälupeesa sisu, mille aadress on avaldise  $e$  R-väärtus;
  - kirje komponendi selekteerimine läbi viida  $e \rightarrow c$  on ekvivalentne avaldisega  $(*e).c$ .

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



## Viidad ja dünaamiline mäluhaldus

Näide: olgu antud deklaratsioonid:

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

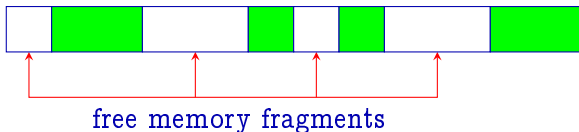
Siis  $\rho = \{ a \mapsto 0, b \mapsto 7, i \mapsto 1, j \mapsto 2, pt \mapsto 3 \}$ .

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.

$$\text{code}(\text{free}(e);) \rho = \text{code}_R e \rho \\ \text{pop}$$

- 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:

`codeR f ρ` = `_f` = starting address of *f* code

- Seega peab addresskeskkond haldama ka funktsioonide nimesid!

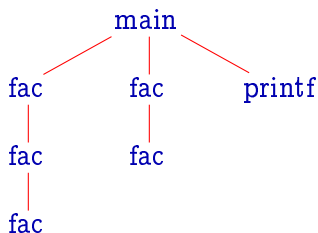
# Funktsioonid

- 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.

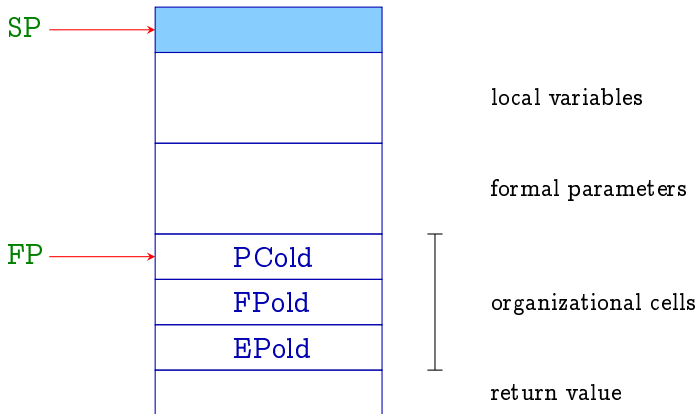


## Funktsioonid

- Funktsiooni iga *eksemplari* formaalsed parameetrid ja lokaalsed muutujad tuleb hoida eraldi.
- Selleks reserveerime funktsioonide iga väljakutse korral magasinis spetsiaalse mälupiirkonna, *freimi* (**Stack Frame**).
- **FP** (**F**rame **P**ointer) 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 väljakutsunud 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 magasinini 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) muutujad (k.a. formaalsed parameetrid), mis on defineeritud funktsioonide siseselt.
- Aadresskeskkond  $\rho$  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.

## Funktionsionid

```
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));
}
```

---

# Funktionsid

```
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.
ρ₀   i ↦ (G, 1)
      l ↦ (G, 2)
      ith ↦ (G, _ith)
      main ↦ (G, _main)
```

# Funktionsid

```
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
ρ1  x ↦ (L, 1)
     i ↦ (L, 2)
     l ↦ (G, 2)
     ith ↦ (G, _ith)
     main ↦ (G, _main)
```

# Funktionsid

```
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
ρ2  k ↦ (L, 1)
     i ↦ (G, 1)
     l ↦ (G, 2)
     ith ↦ (G, _ith)
     main ↦ (G, _main)
```

## 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; } return
  - 2 tagasipöördumine *f*-i koodi; so. **PC** taastamine.

## Funktsioonid

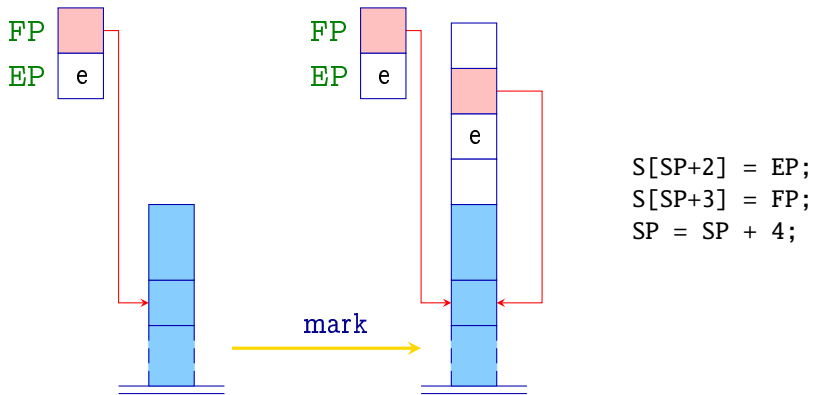
$$\text{code}_R (g(e_1, \dots, e_n)) \rho = \begin{array}{l} \text{mark} \\ \text{code}_R e_1 \rho \\ \dots \\ \text{code}_R e_n \rho \\ \text{code}_R g \rho \\ \text{call } n \end{array}$$

- 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.



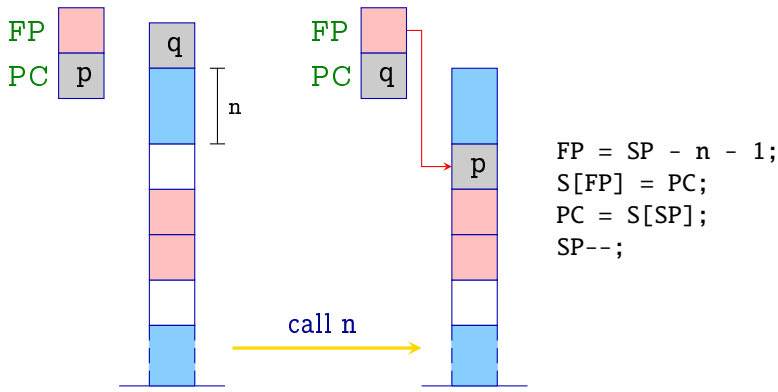
# Funktsioonid

Käsk **mark** reserveerib mälu organisatoorse pesade ja  
resultaatväärtuse jaoks ning salvestab registrid **FP** ja **EP**:



# Funktsioonid

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



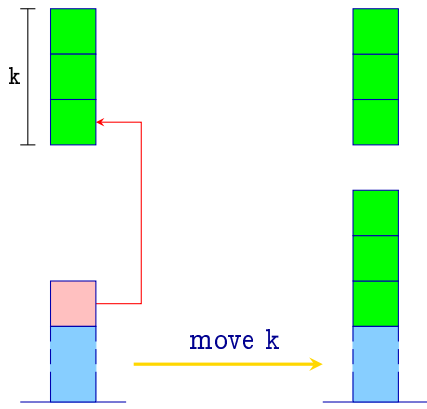
## Funktsioonid

- 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.

<code>code<sub>R</sub> f ρ</code>	=	<code>loadc (ρ f)</code>	$f$ on funktsiooni nimi
<code>code<sub>R</sub> (*e) ρ</code>	=	<code>code<sub>R</sub> e ρ</code>	$e$ on funktsiooni viit
<code>code<sub>R</sub> e ρ</code>	=	<code>code<sub>L</sub> e ρ</code> <code>move k</code>	$e$ on kirje suurusega $k$

# Funktsioonid

Käsk `move k` kopeerib  $k$  pesa magasinini tippu:



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

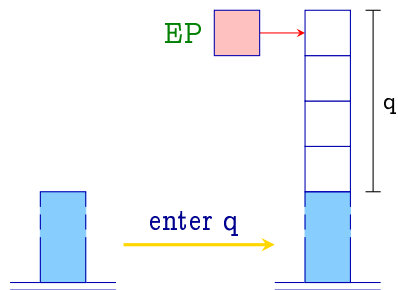
## Funktsioonid

```
code (t f (args){vars ss}) ρ =  _f: enter q  
                                alloc k  
                                code (ss) ρf  
                                return
```

kus  $q$  =  $maxS + k$   
 $maxS$  = lokaalse magasinini maksimaalne sügavus  
 $k$  = mälu lokaalsetele muutujatele  
 $\rho_f$  =  $f$ -i aadresskeskkond

# Funktsioonid

Käsk `enter q` seab registrile `EP` uue väärtuse:

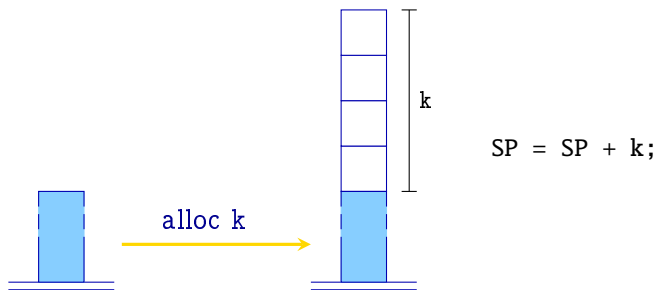


```
EP = SP + q;  
if (EP ≥ NP)  
    Error ("Stack Overflow");
```

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

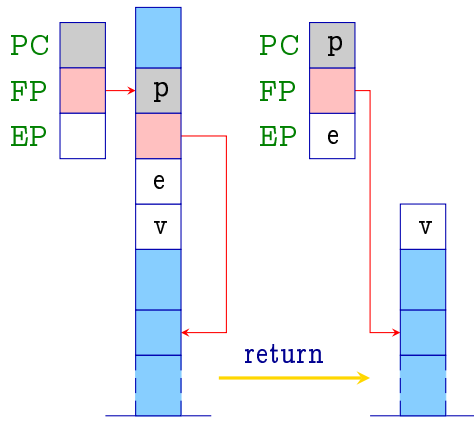
# Funktsioonid

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



# Funktsioonid

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



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



## Funksioonid

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

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

Käsk **loadrc j** arvutab **FP** ja **j** summa:

**FP** f

**FP** f



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

## Funktsioonid

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

$$\begin{aligned} \text{loadr } j &= \text{loadrc } j \\ &\quad \text{load} \\ \text{storer } j &= \text{loadrc } j \\ &\quad \text{store} \end{aligned}$$

return-lausele vastab väärtuse omistamine muutujale suhtaadressiga -3:

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

## 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      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
      jumpz A    sub
                loadc _fac
                call 1
```

## Kogu programmi transleerimine

Abstraktse masina olek enne programmi käivitamist:

$$SP = -1 \quad FP = EP = 0 \quad PC = 0 \quad NP = MAX$$

Olgu  $p \equiv vars \ fdef_1 \ \dots \ fdef_n$ , 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

```
code  $p \emptyset$  =   enter ( $k + 6$ )           pop
                  alloc ( $k + 1$ )       halt
                  mark                    $-f_1 : \text{code } fdef_1 \rho$ 
                  loadc _main           ...
                  call  $\emptyset$           $-f_n : \text{code } fdef_n \rho$ 
```

kus  $\emptyset$  = tühi aadresskeskkond  
 $\rho$  = globaalne aadresskeskkond  
 $k$  = mälu globaalsetele muutujatele