Izdelava programskega jezika

Nejc Mozetič

28. marec 2024

Povzetek

Ključne besede: programski jezik, prevajalnik, assembly

Programski jeziki podpirajo več različnih funkcionalnosti. V našem primeru so to izrazi, spremenljivke, izpisovanje, pogojni stavki, zanke ter funkcije. Programski jezik je preveden v assembly. Ta neposredno operira z strojno opremo - registri, skladi in operacije. Za prevajanje je potreben prevajalnik, ki lahko vse funkcionalnosti zapiše v assembly kodi.

Kazalo

Tabele

Slike

1 Uvod

Programski jeziki so ključna abstrakcija, ki nam pomaga pri komunikaciji z našimi računalniki. V seminarski nalogi bom ustvaril nov programski jezik ter predstavil, kako lahko ljudem berljivo kodo prevedemo v računalniku razumljivo obliko.

Glavni cilj naloge je izdelava prevajalnika, ki naš programski jezik spremeni v jezik assembly. Naš programski jezik bo omogočal osnovne algebrske operacije, shranjevanje podatkov v spremenljivke, operiranje s temi spremenljivkami, večkratno izvajanje kode s pomočjo zank ter razvejanje programa z uporabo pogojnih stavkov. Za prikazovanje rezultatov bomo implementirali tudi možnost izpisovanja.

2 Jezik

Za začetek moramo oblikovati naš programski jezik. Odločiti se moramo, katere funkcionalnosti bo imel in kako bodo le te izgledale. Za moj jezik sem se omejil na najpogostejše lastnosti, ki jih najdemo tudi v večini drugih jezikov. Omogočali bomo osnovne matematične operacije ter spremenljivke. Števila bodo omejena med $9.22 * 10^{18}$ in $9.22 * 10^{18}$. Več o tem v naslednjem poglavju ??kako deluje računalnik??.

Naša koda se ne bo izvajala le vrstica za vrstico, saj bomo podpirali tudi razvejevanje kode. Presledke in tabulatorje bo ignorirala, zato bomo veje končevali s ključno besedo "konec". Med razvejanje spadajo zanke, pogojni stavki in funkcije.

2.1 Izrazi

Naši računi bodo podpirali seštevanje, odštevanje, množenje, deljenje in oklepaje. Uporabljali bomo le cela števila, zato bo rezultat deljenja zaokrožen navzdol. Seveda moramo zagotoviti tudi pravilni vrstni red izvajanja ukazov.

V sledečem izrazu moramo najprej odšteti $4 - 5$ v oklepaju, nato zmnožiti rezultat $2 * 3 * 1$ ter končno temu prišteti $1 + 6$, da dobimo rezultat 7.

$$1 + 2 * 3 * (5 - 4)$$

Podpirali bomo tudi proste $-$, ki v matematiki niso dovoljeni, a nam to pogosto poenostavi pisanje kode:

$$1 + -2$$

$$3 * -4$$

$$5 * -(6 + 7)$$

2.2 Spremenljivke

Spremenljivke bodo delovale na podoben način kot pythonove spremenljivke. Ko želimo deklarirati novo spremenljivko, samo zapišemo njeno ime ter ji določimo vrednost. Ker bodo naše spremenljivke podpirale le cela števila, nam tipa ni potrebno deklarirati.

```
a = 5
```

Tu ima spremenljivka a vrednost 5. Vsakič, ko jo bomo v prihodnosti uporabili, bo enako, kot če bi napisali 5, dokler je ne spremenimo.

Spremenljivko lahko definiramo na enak način. Pri redeklaraciji lahko uporabimo tudi druge spremenljivke ter redefiniramo spremenljivko samo.

```
b = 5+a
b = b-3
```

V danem primeru bomo b najprej nastavili na vrednost $5 + a$, kar je $5 + 5 = 10$. V naslednji vrstici spremenimo b v $b - 3$. Ker b hrani vrednost 10, to pomeni, da je sedaj nastavljen na $10 - 3 = 7$.

Če bomo poskušali uporabiti vrednost spremenljivke, katere še nismo definirali, nam bo prevajalnik vrnil napako.

```
b = 2+c
```

2.3 Izpisovanje

Naš program bo sposoben izpisovati po eno število na vrstico. Podpirali bomo tako pozitivna kot negativna števila. Ko bomo želeli izpisati vrednost, bomo vrstico začeli s ključno besedo *pisi*, le tej pa bo sledila vrednost, katero želimo izpisati. Ta vrednost je seveda lahko tudi v obliki izraza.

```
pisi 5
pisi -4
pisi 2*6-7
```

Primer programa, ki ga lahko dosežemo z dosedanjimi funkcionalnostmi:

```
a = 3
pisi a+2 a = a+21
b = a -3
pisi a * b
```

Program nam mora izpisati vrednosti:

```
5
503
```

2.4 Pogojni stavki

Pogojni stavki omogočajo, da se nek del kode izvaja le, če je nekemu pogoju zadoščeno. Pogoje bomo omejili na primerjanje med dvema številoma.

```
1 a = 3
2 ce 5 < a
3     pisi a
4 konec
5
6 a = 30
7 ce 5 < a
8     pisi a
9 konec
```

V danem primeru se spremenljivka a prvič ne bo izpisala, saj trditev $5 < 3$ ni resnična. Zato bo izvajanje preskočilo 3. vrstico ter se nadaljevalo za ključno besedo *konec* v 4. vrstici. Naslednjič ko izvedemo primerjavo, smo spremenili vrednost a, tako da trditev $5 < 30$ postane resnična. Iz tega razloga se bo koda 7. vrstice tudi izvedla. Naš program bo tako izpisal le število 30.

Podpirali bomo tudi *sicer* stavke. To so ključne besede, ki sledijo pogojnim stavkom.

```
1 a = 3
```

```
2 ce 5 < a
3     pisi a
4 sicer
5     pisi 5
6 konec
```

Ker je trditev v pogojnem stavku neresnična, se vrstica 3 ne bo izvršila, se bo pa zato v vrstici 5 izpisala vrednost 5.

```
a = 3
ce 5 > a
    pisi a
sicer
    pisi 5
konec
```

V tem primeru je trditev resnična, saj je $5 > 3$. Tokrat se bo izvedla vrstica 3, ne pa vrstica 5. Naš nov izhod je 3.

2.5 Zanke

Podpirali bomo dve vrsti zank. Osnovne zanke so *dokler* zanke. To so deli kode, ki se izvajajo dokler je neka trditev resnična.

```
a = 0
dokler a <= 5
    pisi a
    a = a+1
konec
```

Ta zanka se izvaja, dokler je $a <= 5$. a začne z vrednostjo 0 ter se poveča vsako iteracijo. Ta koda bi nam izpisala izhod:

```
0
1
2
3
4
5
```

Po 6. iteraciji bo spremenljivka a držala vrednost 6, ki ne več ustreza pogoju zanke. Takrat se bo koda odvijala naprej od konca zanke.

Podpirali bomo še eno pogosto vrsto zanke *za*. To je bolj omejena zanka, saj se odvija le za vsak korak neke spremenljivke od začetne vrednosti do končne. Zanka bo sprejela vsaj dva argumenta. Prvi bo ime spremenljivke, ki jo bomo spreminjali do končne vrednosti. Če spremenljivka še ne obstaja, bo ustvarjena z začetno vrednostjo 0. Prejšnjo zanko bi lahko tako napisali še hitreje:

```
za a do 5
    pisi a
```

```
konec
```

Izhod bo enak kot prej, saj se bo a spreminjal po 1 od 0 do 5. Če želimo, lahko zanki dodamo še začetno vrednost spremenljivke. V primeru, da spremenljivka že obstaja, bo to preprosto spremenilo njeno vrednost pred izvajanjem zanke.

```
za a od 3 do 5
   pisi a
konec
```

Nova koda začne izvajati zanko z $a = 3$. Nov izhod:

```
3
4
5
```

2.6 Funkcije

Le nazadnje bomo podpirali tudi funkcije. Funkcije bodo definirane v telesu kode s ključno besedo *fun*. Preden je funkcija definirana, je ne moremo uporabljati. Funkcije tudi ne bomo mogli redefinirati. Funkcija ne bo mogla imeti istega imena kot že obstoječe spremenljikve.

```
1 fun tri
2    pisi 3
3 konec
4
5 tri
```

Vsakič, ko bomo poklicali funkcijo *tri*, kot smo jo tudi v 5. vrstici, se bo izvedla funkcija in nam izpisala 3.

Ob deklaraciji bomo lahko izbrali poljubno število argumentov, ki jih bo funkcija prejela. Vsakič, ko jo bomo klicali, bomo morali posredovati točno toliko vrednosti.

```
1 fun sesetej a b
2    pisi a + b
3 konec
4
5 sestej 3 5
6 sestej 6 4
```

Tukaj smo določili dva argumenta, ki ju funkcija uporabi. Peta vrstica izvede funkcijo z vrednostima 3 in 5, zato bo izpisana vrednost 8. Šesta vrstica bo izpisala 10.

Funkcije bodo lahko vrednost tudi vrnile, tako da jo bomo lahko uporabljali v prihodnosti. Rezultat se bo shranil v spremenljivki z imenom funkcije. Spremenil se bo šele, ko izvedemo funkcijo z drugačnimi argumenti.

```
1 fun sestej a b
2    vrni a + b
```

```
3  konec
4
5  sestej 3 5
6  pisi 2 * sestej
7
8  sestej 6 4
9  pisi sestej - 5
```

V peti vrstici spremenljivko *sestej* nastavimo na 8. Zato bo naslednja vrstica tudi izpisala 16. V 10. vrstici bo izpisna 5, saj bo vrednost spremenljivke seštej enaka 10.

Če bomo znotraj funkcije uporabljali spremenljivke, ki obstajajo zunaj nje, bomo spreminjali globalno spremenljivko. Če je spremenljivka prvič omenjena znotraj funkcije, od zunaj ne bo dostopna.

```
c = 11
fun sestej a b
    c = a + b
konec

sestej 1 2

pisi c
```

Ta koda bo izpisala 3, saj smo znotraj funkcije spremenili vrednost c.

```
fun fibonaci a b
    c = a
    a = a+b
    b = c
    pisi a
    pisi b
    pisi c
konec

fibonaci 1 2

pisi a
pisi c
```

Ta koda bi izpisala vrednosti 3, 2 in 1 znotraj funkcije, če ne bi imeli zadnjih dveh vrstic. Ker ti dve vrstici poskušata dostopati do spremenljivk definiranih v funkciji, naletimo na napako.

3 Kako deluje računalnik

Če želimo izdelati in prevajati programski jezik, moramo najprej vedeti, kako naj izgleda končni izdelek. Vsi končni ukazi se bodo izvajali na centralni računalniški enoti *CPU*, potrebovali pa bomo tudi bralno pisalni pomnilnik *RAM*. Poglejmo si, kako delujeta in zakaj sta pomembna.

3.1 Registri

Na cpu-ju imamo na voljo registre. To so delci spomina, v katere lahko zapišemo podatke, jih obdelamo ter nazaj preberemo. V tem projektu bomo uporabljali 64-bitno arhitekturo, kar pomeni, da je v vsakem registru prostora za do 2^{64} veliko vrednost.

Za računanje bomo uporabljali predvsem registra *rax* in *rbx*. Pri deljenju bomo potrebovali tudi register *rdx*. Kasneje bomo za pomnjenje potrebovali tudi registra *rbp* in *rsp*, ki kažeta na položaje v skladu (stack). O skladu bomo več povedali v kasnejšem podpoglavju Sklad.

3.2 Operacije

Cpu ima vgrajene različne operacije, ki jih lahko izvedemo na registrih. Primeri takih operacij so:

- *mov* - za tem ukazom navedemo prvi register, v katerega želimo prestaviti vrednost, nato pa še ali vrednost, ki jo želimo shraniti ali pa naslov, iz katerega želimo prenesti vrednost v prvi register. Primer: V register *rax* premaknemo vrednost 3. V register *rbx* premaknemo 3 iz registra *rax*.

  ```
  mov rax, 0
  mov rbx, rax
  ```

  Po drugi operaciji se 3 ne izbriše iz registra *rax*, pač pa ostane v njem. Tako se po teh dveh ukazih v obeh registrih nahaja število 3.

- *add* - temu ukazu sledita dva registra, katera želimo sešteti. Na mestu drugega registra lahko tudi tukaj napišemo kar neposredno vrednost, katero želimo prišteti, a naš prevajalnik tega ne bo delal. Rezultat se bo shranil v prvi navedeni register.

  Primer: V registra *rax* in *rbx* premaknemo vrednosti 3 in 2. Ti vrednosti nato seštejemo, rezultat 5 pa shranimo v registru *rax*.

  ```
  mov rax, 3
  mov rbx, 2
  add rax, rbx
  ```

  Tudi tukaj register *rbx* ostane po operaciji nespremenjen in tako ohrani vrednost 2.

- *imul* - to je ukaz, ki deluje na enak način kot seštevanje, le da izračuna produkt navedenih vrednosti.

- *sub* - tudi odštevanje deluje na podoben način, pri njem pa moramo biti pozorni, katera vrednost je navedena v prvem registru. Ta bo namreč zmanjševanec, druga vrednost pa odštevanec.

  Primer: V registra shranimo vrednosti, nato pa ju odštejemo. Ker je prvi naslov *rax*, bo število 3 zmanjševanec, naslov *rbx* pa odštevanec. Rezultat 1 bo shranjen v *rax*.

```
mov rax, 3
mov rbx, 2
sub rax, rbx
```

- *inc*, *dec*, *neg* - za temi ukazi sledi po en naslov, čigar vrednost s tem spremenimo. *inc* število poveča za 1, *dec* zmanjša za 1, *neg* pa negira. Zadnji ukaz nam bo prišel prav, saj bomo morali odštevati vrednosti v napačnem vrstnem redu.

  Primer: V *rax* premaknemo vrednost -3, *rax* zmanjšamo za 1 in dobimo -4, *rax* negiramo v 4, na koncu pa ga povečamo na 5.

```
mov rax, -3
dec rax
neg rax
inc rax
```

- *idiv* - deljenje je dokaj podobno odštevanju, le da smo bolj omejeni. Deljenec se bo vselej nahajal v registru *rax*, kamor bo rezultat tudi shranjen. Zato temu ukazu navedemo le en argument in sicer vrednost delitelja. Tukaj se tudi prvič srečamo z novim registrom *rdx*. Vanj se bo shranil ostanek pri deljenju. Tega sicer ne bomo potrebovali, a vseeno moramo poskrbeti, da je register pred operacijo prazen. Uporabili bomo binarno operacijo *xor*, rezultat pa je enak, kot če bi vanj zapisali 0 z ukazom *mov*.

  Primer: v register *rax* napišemo 6, v *rbx* 3, *rdx* pa izpraznimo.

```
mov rax, 6
mov rbx, 3
xor rdx, rdx
idiv rbx
```

  Po izvršitvi ukazov bo v *rax* rezultat $6 \div 3 = 0$, v *rbx* bo ostala 3, *rdx* pa bo ostal prazen, saj smo delili brez ostanka. Ko delimo z ostankom, dobimo rezultat zaokrožen navzdol.

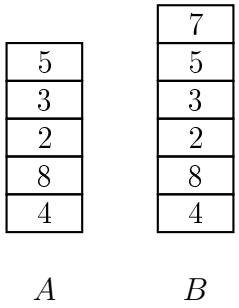
$$8 \div 3 = 2$$

3.3 Sklad

Sklad je podatkovna struktura, v katero lahko shranimo več vrednosti. Nahaja se v RAM-u, saj ima ta na voljo veliko več spomina. Medtem ko lahko na CPU shranimo le nekaj številk, lahko sklad znotraj rama drži več milijonov vrednosti. Če imamo v računalniku

16 gb rama, shranjujemo pa 64 bitna števila, to pomeni, da imamo pri polnem ramu shranjenih 250 milijonov števil.

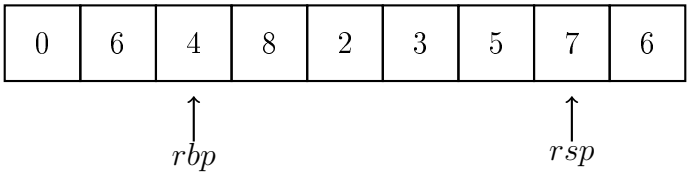
Sklad deluje na poseben način. Načeloma lahko dostopamo le do vrednosti na vrhu.



Slika 1: Dodajanje na sklad

Na sliki A imamo primer sklada. Če preberemo njegovo vrednost, bomo dobili 5, saj se ta nahaja na vrhu. Če na sklad potisnemo število 7, bomo dobili sliko B. Ko spet preberemo njegovo vrednost, vidimo 7, saj to postane vrhnje število. Če bi odstranili vrednost iz sklada, bi se vrnili nazaj na sliko A.

Rekel sem, da lahko do vrednosti načeloma dostopamo le na vrhu, saj je sklad v RAM-u implementiran znotraj tabele. Iz tega razloga lahko v resnici dostopamo do kateregakoli elementa, dokler si zapomnemo, kje se nahaja. Sklad je v tabeli definiran z dvema kazalcema: začetnim, ki nam pove, kje se nahaje prvi element ter končnim, ki kaže na zadnji element našega sklada. V assembly-u začetni kazalec označujemo z rbp, končnega pa z rsp. Ko želimo odstraniti element iz sklada, v resnici tabele ne spreminjamo, le končni kazalec pomaknemo nazaj. Spodaj je prikaz zgornjega sklada B znotraj tabele RAM-a.



Slika 2: Premikanje po skladu

Ko beremo vrednost sklada, pogledamo vrednost na položaju *rsp*. Če bi želeli odstraniti zgornjo vrednost sklada in se vrniti k sliki A, bi premaknili *rsp* za eno polje v levo. Če bi želeli dodati število 2 na sklad, bi kazalec premaknili na desno, na njegovem novem položaju pa zapisali 2. Ker uporabljamo 64-bitne vrednosti, vsako število v resnici zasede 8 bajtov, tako da kazalec pomikamo po 8 prostorov v levo in desno. Ko na sklad dodajamo elemente, se v resnici v RAM-u premikamo iz višjih naslovov proti nižjim.

Če v assembly-u želimo zapisati vrednost na sklad, uporabimo [in], znotraj njih pa napišemo naslov na skladu, s katerim želimo operirati.

Primer: Na sklad želimo dodati vrednost 5. Najprej moramo kazalec premakniti v desno. To dosežemo tako, da *rsp* zmanjšamo za 8. Na to mesto nato napišemo 5.

```
sub rsp, 8
mov [rsp], 5
```

Prepišimo zadnjo vrednost iz sklada v *rax* ter jo odstranimo.

```
mov rax, [rsp]
inc rsp, 8
```

Assembly nam to še bolj poenostavi ter nam da na voljo dve funkciji, ki naredita točno to, kar smo sedaj lastnoročno.

```
push 5
pop rax
```

Ukaz *push* doda vrednost na sklad, *pop* pa zadnjo vrednost napiše v želeni register ter odstrani iz sklada.

Če želimo dobiti zadnjo vrednost, ne da bi jo odstranili, uporabimo ukaz *mov* kot prej. Če želimo dolgoročno shraniti vrednosti, jih lahko dodamo na dno sklada, si zapomnemo, kje se katera nahaja, nato pa jih preko njihovega naslova beremo in spreminjamo.

Dodajmo recimo na začetku programa število 7. Vemo, da se nahaja en prostor ali 8 naslovov stran od začetnega kazalca *rbp*. Lahko jo beremo v drugi register ter jo spreminjamo:

```
push 7
mov rax, [rbp-8]
mov [rbp-8], 3
```

3.4 Pogojni stavki

Do sedaj se vsi ukazi, ki jih napišemo izvajajo zaporedno. Včasih pa se mora naša koda obnašati različno glede na pogoje. V našem programskem jeziku bomo to potrebovali pri pogojnih stavkih in zankah. V assembly-u to implementiramo z ukazi primerjanja ter s preskakovanjem.

Če želimo primerjati dve števili, ju moramo sprva posredovati ukazu *cmp*. Temu sledi ukaz primerjave ter oznaka kam želimo skočiti, če je primerjava resnična. Tu je tabela ukazov, ki jih imamo na voljo:

Ukaz	**Pomen**	**Enačba**
JE	Enako	$A = B$
JNE	Neenako	$A \neq B$
JG	Večje	$A > B$
JGE	Večje ali enako	$A \geq B$
JL	Manjše	$A < B$
JLE	Manjše ali enako	$A \leq B$

Tabela 1: Ukai primerjanja

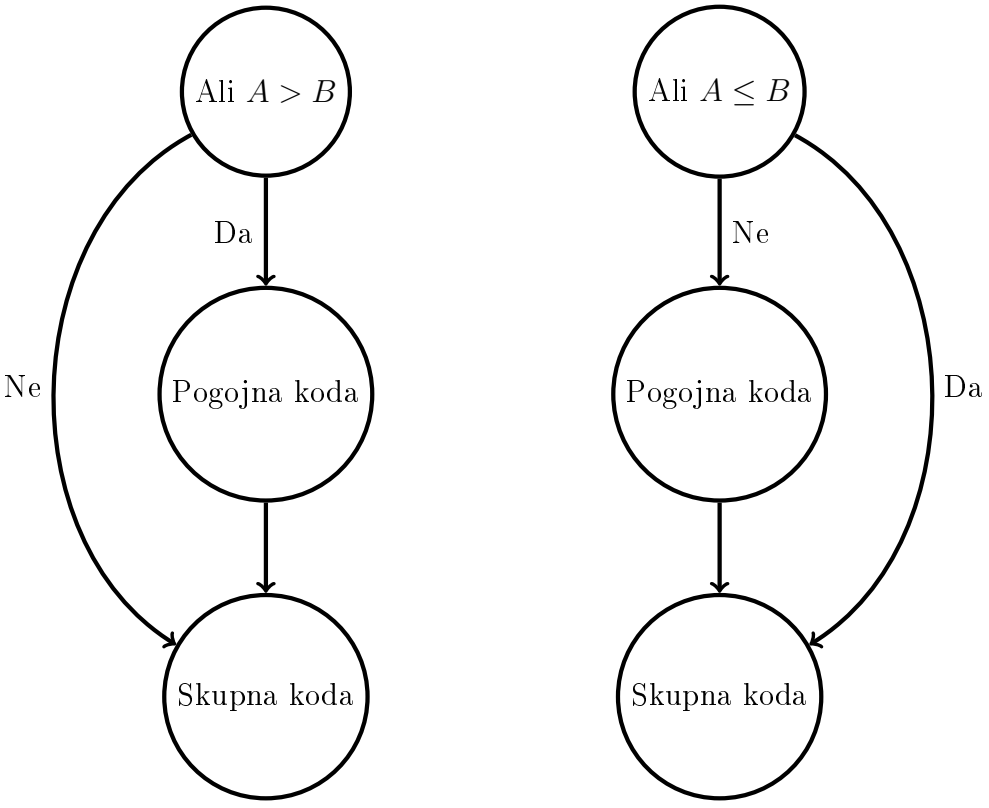
Primer kode s primerjanjem:

```
1 mov rax, 4
2 mov rbx, 3
3 cmp rax, rbx
```

```
4 jg skok
5
6 mov rax, 1
7
8 skok:
9 mov rbx , 9
```

Tretja in četrta vrstica določita, če je $rax > rbx$, se bo izvajanje nadaljevalo pri oznaki skok v vrstici 8 in tako bo ukaz pisanja 1 v rax preskočen. Ker je to v našem primeru bo končno stanje registrov $rax = 4$ in $rbx = 9$. Če bi v prvi vrstici rax nastavili na 2, se skok v šesti vrstici ne bi izvedel in bi tako končno stanje bilo $rax = 1$ in $rbx = 9$.

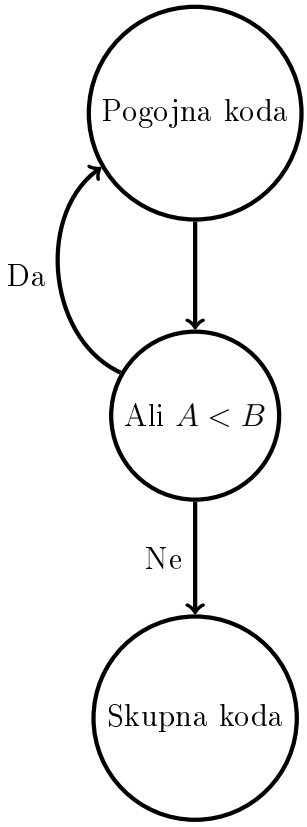
Vidimo, da se koda po oznaki izvede v vsakem primeru, koda med ukazom za skok in oznako pa le v primeru, da je pogoj neuresničen. Če bi na primer želeli, da se neka koda izvede le v primeru, da $rax > rbx$, bi morali za pogoj skoka uporabiti $rax \leq rbx$, oziroma *jle* ($\leq$). To je zato, ker sta si sledeča grafa enaka, desnega je le lažje implementirati.



Slika 3: Pogojna stavka

3.5 Zanke

Zanke lahko implementiramo na isti način kot pogojne stavke. Razmislimo, kako bi definirali osnovno zanko, ki se izvaja, dokler je nek pogoj resničen. To lahko dosežemo, če oznako, h kateri skačemo, postavimo pred primerjavo. Tako bomo skakali na začetek odstavka kode, dokler je pogoj resničen.



Slika 4: Preprosta zanka

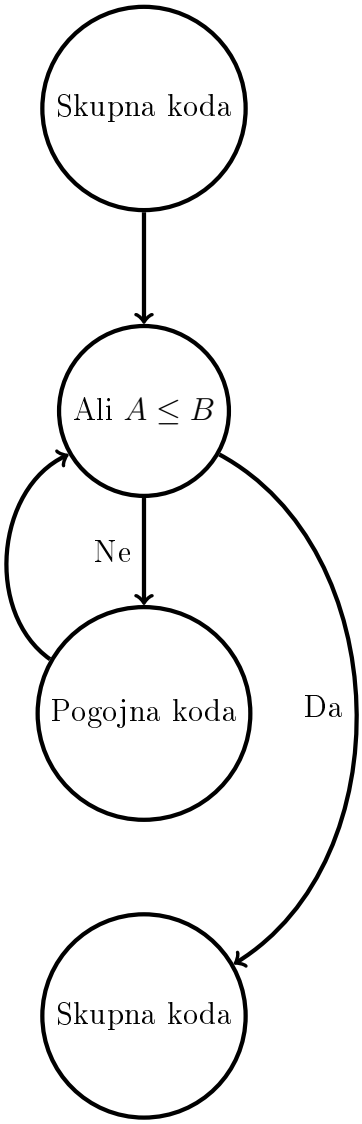
Primer implementacije zanke:

```
mov rax, 1
mov rbx, 7

zanka:
inc rax

cmp rax, rbx
jl zanka
```

Znotraj te zanke bomo povečevali *rax*, dokler je ta manjši od *rbx*. Ko se bo zanka prekinila, bo *rax* torej enak *rbx* - 7.

Težava s tem pristopom je, da se bo zanka vedno vsaj enkrat izvedla. Če bi bil *rax* na začetku 9, bi po koncu programa bil enak 10. Po navadi zanko želimo preskočiti, če pogoj za ponavljanje ni izpolnjen. To lahko dosežemo tako, da ustvarimo zanko, ki se ponavlja brez pogojev z ukazom *jmp*. Ta nadaljuje izvajanje pri definirani oznaki tako kot pogojni skoki. Na začetku vsakegega sprehoda skozi zanko bomo izvedli primerjavo, ki nas bo vrgla ven, če bo resnična. Tako kot pri pogojnih stavkih, moramo tudi tokrat

uporabiti nasprotni pogoj. Če želimo, da se zanka izvaja dokler je $rax > rbx$, bomo v primerjavi uporabili jle ($\leq$).



Slika 5: Izboljšana zanka

Primer:

```
mov rax, 9
mov rbx, 3

zanka:
cmp rax, rbx
jle konec
dec rax

jmp zanka

konec:
```

Ta koda se bo izvajala, dokler je $rax > rbx$. Po vsaki ponovitvi se rax zmanjša. Ko se pogoj $rax \leq rbx$ uresniči, skočimo na konec zanke in koda se nadaljuje.

3.6 Predložna koda

Če želimo zgraditi assembly program, moramo dodati nekaj predložne kode, ki jo bomo vključili v vsak program.

```
section .text
    global _start
_start:
    mov rbp , rsp

    ;Koda

    xor rdi, rdi
    mov rax , 60
    syscall
```

V prvih dveh vrsticah definiramo, da se naša koda začne izvajati v četrti vrstici. Peta vrstica definira naš stack, in sicer nastavi začetek na trenutni vrh. Za tem vstavimo kodo, ki jo želimo izvajati. Zadnje tri vrstice zaključijo program. Deveta vrstica izprazni register rdi, ki ob izhodu vrne izhodno kodo. Če smo prišli do konca programa, to pomeni, da se je pravilno izvedel, kar navadno nakažemo z izhodno kodo 0. V register rax moramo napisati število 60, saj je to koda za izhod. Zadnja vrstica posreduje ta registra operacijskemu sistemu, ki ju izvede. Za to kodo bomo dodali še funkcijo, ki izpiše trenutno vrednost v registru rax.

```
pisi:
    push rbp
    mov rbp, rsp

    test rax,rax
    jz pozitivno
    test rax, (1 << 63) - 1
    jg pozitivno

    neg rax
    push rax
    push '-'
    mov rax, 1
    mov rdi, 1
    mov rsi, rsp
    mov rdx, 1
    syscall
    add rsp, 8
    pop rax

pozitivno:

    mov rcx, 10
```

```
    mov rbx, 1
    sub rsp, 1
    mov byte [rsp], 10

stevka_loop:
    xor rdx, rdx
    div rcx
    add dl, '0'
    sub rsp, 1
    mov [rsp], dl
    inc rbx
    test rax, rax
    jnz stevka_loop

    mov rsi, rsp
    mov rax, 1
    mov rdi, 1
    mov rdx, rbx
    syscall

    mov rsp, rbp
    pop rbp
    ret
```

Ko želimo izvesti funkcijo v assembly-u uporabimo ukaz *call* in ime funkcije, v našem primeru *pisi*. Upoštevati moramo, da bodo pri klicanju funkcije vrednosti shranjene v registrih izgubljene. Preden jo izvedemo, moramo vsa pomembna števila shraniti v sklad.

Tu je primer kode, ki izpisuje število A. Število ima na začetku vrednost 8, vsako iteracijo se zmanjša za 1, dokler je $A > 5$.

```
section .text
    global _start

_start:
    mov rbp, rsp

    mov rax, 8
    mov rbx, 5
    push 0
    push 0

zanka:
    cmp rax, rbx
    jle konec

    mov [rbp-8], rax
    mov [rbp-16], rbx

    call pisi

    mov rax, [rbp-8]
```

```
    mov rbx, [rbp-16]

    dec rax
    jmp zanka

konec:

    xor rdi, rdi
    mov rax , 60
    syscall

pisi:
    push rbp
    mov rbp, rsp

    test rax,rax
    jz pozitivno
    test rax, (1 << 63) - 1
    jg pozitivno

    neg rax
    push rax
    push '-'
    mov rax, 1
    mov rdi, 1
    mov rsi, rsp
    mov rdx, 1
    syscall
    add rsp, 8
    pop rax

pozitivno:

    mov rcx, 10
    mov rbx, 1
    sub rsp, 1
    mov byte [rsp], 10

stevka_loop:
    xor rdx, rdx
    div rcx
    add dl, '0'
    sub rsp, 1
    mov [rsp], dl
    inc rbx
    test rax, rax
    jnz stevka_loop

    mov rsi, rsp
    mov rax, 1
    mov rdi, 1
```

```
    mov rdx, rbx
    syscall

    mov rsp, rbp
    pop rbp
    ret
```

Izhod:

```
8
7
6
4
```

4 Prevajanje

Končno moramo iz našega jezika, katerega smo oblikovali na začetku, dobiti delujočo assembly kodo.

4.1 Branje in pisanje

Našo kodo želimo pisati v datoteko, katero bo naš prevajalnik odprl in prebral.

```
ifstream vhod("koda.k");
if (!vhod.is_open()) {
    std::cerr << "Napaka pri odpiranju!" << std::endl;
    return 1; // Return an error code
}
```

Kodo bomo v obliki besedila pisali v *c++ stream*. Izhod želimo razdeliti na dva dela zaradi funkcij. Vedno, ko pišemo kodo, ki obstaja znotraj funkcije, želimo pisati v drugi *stream*, saj ga tako lahko preprosto dodamo na konec globalne kode.

```
ostringstream izhod[2];
```

Obstaja določena koda, ki jo bomo želeli vedno vključiti. Med to spada deklaracija assembly glavne funkcije, izhod iz globalne funkcije ter naša funkcija *pisi*, ki smo jo oblikovali v prejšnjem poglavju.

```
izhod[0] << "section .text\n\tglobal _start\n\n_start:\n\tmov rbp, rsp\n\tmov
    r15, rsp\n";

// prevajanje kode

izhod[0] << "\n\txor rdi, rdi\n\tmov rax, 60\n\tsyscall" << endl;
izhod[1] << "\npisi:\n\tpush rbp\n\tmov rbp, rsp\n\n\ttest rax,rax\n\tjz
    pozitivno\n\ttest rax, (1 << 63) - 1\n\tjg pozitivno\n\n\tneg rax\n\tpush
    rax\n\tpush '-'\n\tmov rax, 1\n\tmov rdi, 1\n\tmov rsi, rsp\n\tmov rdx, 1\n\
    tsyscall\n\tadd rsp, 8\n\tpop rax\n\npozitivno:\n\n\tmov rcx, 10\n\tmov rbx,
     1\n\tsub rsp, 1\n\tmov byte [rsp], 10\n\nstevka_loop:\n\txor rdx, rdx\n\
    tdiv rcx\n\tadd dl, '0'\n\tsub rsp, 1\n\tmov [rsp], dl\n\tinc rbx\n\ttest
    rax, rax\n\tjnz stevka_loop\n\n\tmov rsi, rsp\n\tmov rax, 1\n\tmov rdi, 1\n\
    tmov rdx, rbx\n\tsyscall\n\n\tmov rsp, rbp\n\tpop rbp\n\tret" << endl;
```

Ko končamo zapisovati assembly kodo v *stream*-a, ju združimo in zapišemo v izhodno datoteko.

```
ofstream izhodnaDatoteka;
izhodnaDatoteka.open("out.asm");
izhodnaDatoteka << izhod[0].str() << "\n\n" << izhod[1].str() << endl;
```

Izpisati bomo morali tudi napake v prevajanju, ko jih zaznamo. Ustvarimo lahko funkcijo, ki nam bo znala izpisati vrsto napake ter nam grafično prikazati, kje je bila zaznana.

```
int stVrstice = 0;
void napaka(string&s, int&ind, string sporocilo) {
    cout << "Vrstica " << stVrstice << ":" << endl;
    cout << s << endl;
    for (int i = 0; i < s.size(); i++) {
        if (i == ind) cout << "^";
        else cout << "~";
    }
    cout << endl << sporocilo << endl;
}
```

Ker je naš jezik neodvisen od presledkov, jih lahko tudi ignoriramo. To pomeni, da sta si sledeči kodi enakovredni:

Koda 1

```
za a do 4
    pisi a
        konec
```

Koda 2

```
za a do 4
pisi a
konec
```

To lahko dosežemo s funkcijo, ki jo kličemo vsakič, ko pričakujemo presledke:

```
void ign(string&s, int &ind) {
    while (isspace(s[ind])) ind++;
}
```

Preden procesiramo kodo, jo moramo tudi razdeliti na besede ali žetone. Žetoni so besede, ki določajo vrsto ukaza, lahko pa tudi imena spremenljivk in funkcij. Vedno se bodo začeli s črkami, kasneje pa lahko vsebujejo tudi števila.

```
string dobiZeton(string&s, int &ind) {
    ign(s,ind);
    string zeton = "";
    while (isalpha(s[ind]) || (zeton != "" && isdigit(s[ind]))) {
        zeton += s[ind++];
    }
    return zeton;
}
```

4.2 Struktura prevajanja

Prevajali bomo vsako vrstico posebej. Vrstico preberemo ter jo pošljemo naprej.

```
string s;
while (getline(vhod, s)) {
    vrstica(s);
    stVrstice++;
}
vhod.close();
```

Vsaki vrstici bomo nato po prvem žetonu določili vrsto.

```
void vrstica(string &s) {
    int ind = 0;

    ign(s, ind);
    if (ind == s.size()) return;
    izhod[tI] << "\n";

    string zeton = dobiZeton(s,ind);

    if (zeton == "pisi") {
        pisi(s, ind);
    } else if (zeton == "za") {
        zaZanka(s, ind);
    } else if (zeton == "dokler") {
        doklerZanka(s, ind);
    } else if (zeton == "ce") {
        pogojniStavek(s, ind);
    } else if (zeton == "sicer") {
        sicerStavek(s, ind);
    } else if (zeton == "konec") {
        konecVeje();
    } else if (zeton == "vrni") {
        izhodIzFunkcije(s, ind);
    } else if (zeton == "fun") {
        definirajFunkcijo(s, ind);
    } else if (funk.count(zeton)) {
        kliciFunkcijo(s, ind, zeton);
    } else if (zeton != "") {
        nastaviSpremenljivko(s, ind, zeton);
    } else {
        izraz(s, ind);
    }
}
```

4.3 Izrazi

Izraze bomo ovrednotili z rekurzijo.

$$2 + 3 * a - 4 * (5 + 2)$$

Vsak račun bomo razdelili na člene. To so skupki vrednosti in operacij, med seboj ločeni s presledki.

$$\underline{2} + \underline{3 * a} - \underline{4 * (5 + 2)}$$

To dosžemo s sledečo funkcijo:

```
void izraz(string &s, int &ind) {
    ign(s,ind);

    if (s[ind] == '+' || s[ind] == '-') izhod[tI] << "\txor rax, rax" << endl;
    else clen(s, ind);

    while (ind < s.length() - 1) {
        ign(s,ind);
        if (s[ind] == '+') sestej(s, ind);
        else if (s[ind] == '-') odstej(s, ind);
        else break;
    }
}
```

Te člene moramo med sabo tudi sešteti in odšteti:

```
void sestej(string &s, int &ind) {
    ind++;

    izhod[tI] << "\tpush rax" << endl;
    clen(s, ind);

    izhod[tI] << "\tpop rbx" << endl;
    izhod[tI] << "\tadd rax, rbx" << endl;
}

void odstej(string &s, int &ind) {
    ind++;

    izhod[tI] << "\tpush rax" << endl;
    clen(s, ind);

    izhod[tI] << "\tpop rbx" << endl;
    izhod[tI] << "\tsub rax, rbx" << endl;
    izhod[tI] << "\tneg rax" << endl;
}
```

Nekatere člene lahko še naprej razdelimo na več delov - poimenujmo jih vrednosti. Vrednosti bodo med seboj ločene z $*$ in $\div$:

$$3 * a$$

$$4 * (5 + 2)$$

```
void clen(string &s, int &ind) {
```

```
    vrednost(s, ind);
    while (ind < s.length() - 1) {
        ign(s,ind);
        if (s[ind] == '*') zmnozi(s, ind);
        else if (s[ind] == '/') deli(s, ind);
        else break;
    }
}

void zmnozi(string &s, int &ind) {
    ind++;

    izhod[tI] << "\tpush rax" << endl;
    vrednost(s, ind);

    izhod[tI] << "\tpop rbx" << endl;
    izhod[tI] << "\timul rax, rbx" << endl;
}

void deli(string &s, int &ind) {
    ind++;

    izhod[tI] << "\tpush rax" << endl;
    vrednost(s, ind);

    izhod[tI] << "\tmov rbx, rax" << endl;
    izhod[tI] << "\tpop rax" << endl;
    izhod[tI] << "\txor rdx, rdx" << endl;
    izhod[tI] << "\tidiv rbx" << endl;
}
```

Vidimo, da so vrednosti lahko en od treh tipov: števila, spremenljivke ali še en izraz znotraj oklepaja.

```
void vrednost(string &s, int &ind) {
    ign(s,ind);
    if (s[ind] == '-') {
        ind++;
        vrednost(s, ind);
        izhod[tI] << "\tneg rax" << endl;
    } else if (s[ind] == '+') {
        ind++;
        vrednost(s, ind);
    } else if (s[ind] == '(') {
        ind++;
        izraz(s, ind);
        ind++;
    } else if (isdigit(s[ind])) {
        stevilo(s, ind);
    } else {
        spremenljivka(s, ind);
```

```
    }

    return ;
}
```

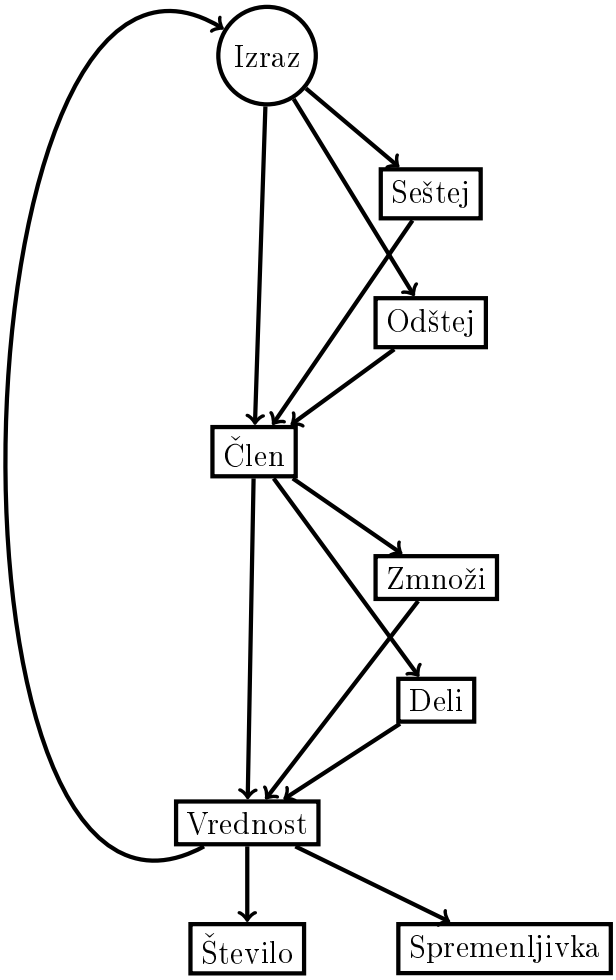
V primeru, da je vrednost število, ga lahko samo preberemo in zapišemo neposredno v assembly.

```
void stevilo(string &s, int &ind) {
    ign(s,ind);
    string st = "";

    while ('0' <= s[ind] && s[ind] <= '9') {
        st += s[ind];
        ind++;
    }
    izhod[tI] << "\tmov rax, " << st << endl;
}
```

Če je vrednost spremenljivka, moramo pogledati, če in kje se nahaja. Assembly-ju nato povemo, na katerem naslovu naj jo išče.

```
void spremenljivka(string &s, int &ind) {
    string zeton = dobiZeton(s,ind);
    if (!spre.count(zeton)) {
        napaka(s, ind, "Nedefinirana spremenljivka " + zeton + ".");
        return;
    }

    izhod[tI] << "\tmov rax, [" << glob[spre[zeton].glob] << " - " << spre[zeton
        ].pol*8 << "]" << endl;
}
```

Če je vrednost nov izraz, le tega nadalje vrednotimo z rekurzijo. Tukaj je graf klicev funkcij med vrednotenjem izraza:



Slika 6: Zaporedje računanja

4.4 Spremenljivke

Če vrstico začnemo z imenom spremenljikve, to pomeni, da jo želimo nastaviti.

```
void nastaviSpremenljivko(string &s, int &ind, string ime) {
    if (!spre.count(ime)) {
        definirajSpremenljivko(ime);
    }
    ign(s,ind);
    if (s[ind] != '=') {
        napaka(s, ind, "Neveljavno nastavljanje spremenljivke " + ime + ".");
        return;
    }
    ind++;
    ign(s, ind);

    izraz(s, ind);

    izhod[tI] << "\tmov [" << glob[spre[ime].glob] << " - " << spre[ime].pol * 8
         << "], rax" << endl;
}
```

Če tokrat prvič omenjamo spremenljivko, jo moramo še definirati. To pomeni, da ji določimo mesto shranjevanja v RAM-u in njen index shranimo v prevajalnik, tako da bomo vedeli, kje se nahaja, ko naslednjič naletimo nanjo.

```
struct sprem{
    int pol;
    int glob;
};
map<string, sprem> spre;
vector<int> stSpre = {1, 1};

void definirajSpremenljivko(string ime) {
    izhod[tI] << "\tpush 0" << endl;
    shraniSpremenljivko(ime);
}

void shraniSpremenljivko(string ime) {
    spre[ime] = (sprem){stSpre[tI]++, tI};
    if (tI) {
        lokSpre.push_back(ime);
    }
}
```

Potem ko ugotovimo, kje se spremenljivka nahaja, moramo izračunati njeno novo vrednost in jo posodobiti. To dosežemo s funkcijo izraz, ki bo ovrednotila račun, ki sledi imenu spremenljivke in enečaju.

4.5 Izpisovanje

Če se vrstica začne s ključno besedo *pisi*, bomo na podoben način s funkcijo izraz ovrednotili, kar ji sledi. Za razliko od prej, te vrednosti ne bomo pisali v RAM, pač pa jo bomo pustili v registru rax, katerega bo tudi assembly funkcija *pisi* uporabila.

```
void pisi(string &s, int &ind) {
    ign(s,ind);

    izraz(s, ind);

    izhod[tI] << "\tcall pisi" << endl;
}
```

4.6 Pogojni stavki

Pogojni stavki so naš prvi primer prevajanja razvejane kode. V vsakem trenutku moramo vedeti, na koliko vejah se nahajamo ter tipe teh vej (pogoji, zanke, funkcije). Pogojne stavke bomo označili s številom 0.

```
int stVej = 0;
stack<pair<int, int>> veje; // st veje, tip veje

void pogojniStavek(string &s, int &ind) {
    veje.push({++stVej, 0});
    pogoj(s, ind);
}
```

Pogoj je primerjava dveh izrazov, ki je lahko resnična ali pa ne. Če je pogoj resničen, se bo koda, ki leži na njegovi veji, izvedla, sicer pa ne.

```
void pogoj(string &s, int &ind) {
    izraz(s, ind);
    izhod[tI] << "\tpush rax" << endl;

    ign(s, ind);
    string relacija = "";
    while (relacijskiZnak(s[ind])) relacija += s[ind++];

    ign(s, ind);

    izraz(s, ind);
    izhod[tI] << "\tpop rbx\n\tcmp rax, rbx\n\t";

    if (relacija == ">") izhod[tI] << "jge";
    else if (relacija == ">=") izhod[tI] << "jg";
    else if (relacija == "<") izhod[tI] << "jle";
    else if (relacija == "<=") izhod[tI] << "jl";
    else if (relacija == "=") izhod[tI] << "jne";
    else napaka(s, ind, "Neznana relacija.");
```

```
    izhod[tI] << " konec" << stVej << endl;
}

bool relacijskiZnak(char c) {
    string znaki = "<>=";
    for (char&z:znaki) if (z==c) return true;
    return false;
}
```

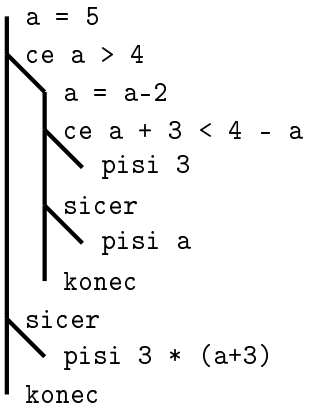
Veji *ce* lahko sledi še veja *sicer*. Ukazi na tej veji se bodo izvedli le, če prejšnjemu pogoju ni bilo zadoščeno.

```
void sicerStavek(string &s, int &ind) {
    if (veje.top().second) {
        napaka(s, ind, "Napana uporaba \"sicer\" stavka.");
        return;
    }

    int naslednjaVeja = veje.top().first;
    veje.pop();

    veje.push({++stVej, 0});

    izhod[tI] << "\tjmp konec" << stVej << endl;
    izhod[tI] << "konec" << naslednjaVeja << ":" << endl;
}
```

sicer bo preden ustvari svojo vejo tudi prekinil prejšnjo. Drugi način prekinjanja veje je z besedo *konec*.

```
void konecVeje() {
    if (veje.top().first == -1) {
        izhod[tI] << "\tmov rsp, rbp\n\tpop rbp\n\tret" << endl;
        for (string&s:lokSpre) {
            spre.erase(s);
        }
        tI = 0;
        veje.pop();
        return;
    }
    if (veje.top().second) {
        izhod[tI] << "\tjmp zanka" << veje.top().first << endl;
    }

    izhod[tI] << "konec" << veje.top().first << ":" << endl;
    veje.pop();
}
```

Na spodnji sliki je grafični prikaz vej, na katere je razdeljena koda.



Slika 7: Razvejanje kode

4.7 Zanke

Prevajali bomo dve vrsti zank. Če je prva beseda *dokler*, bo naše prevajanje podobno prevajanju pogojnih stavkov. Tokrat bomo vrsto veje označevali z 1.

```
void doklerZanka(string &s, int &ind) {
    veje.push({++stVej, 1});
    izhod[tI] << "zanka" << stVej << ":" << endl;
    pogoj(s, ind);
}
```

Druga vrsta zanke je *za* zanka, ki je precej zapletenejša, saj že v prevajalniku definiramo celotno logiko delovanja. V eni vrstici moramo ugotoviti, katero spremenljivko bomo uporabljali za števec, ali ta spremenljivka že obstaja, na kakšno vrednost jo moramo na začetku nastaviti in do katere vrednosti jo bomo povečevali.

```
void zaZanka(string &s, int &ind) {
    stVej++;
    veje.push({stVej, 1});

    ign(s, ind);
    string ime = dobiZeton(s, ind);

    if (!spre.count(ime)) {
        definirajSpremenljivko(ime);
    }

    ign(s,ind);

    string zeton = dobiZeton(s,ind);

    if (zeton == "od") {
        ign(s,ind);
        izraz(s, ind);

        izhod[tI] << "\tmov [" << glob[spre[ime].glob] << " - " << spre[ime].pol
```

```
            * 8 << "], rax\n" << endl;

        ign(s, ind);
        zeton = dobiZeton(s, ind);
    }


    if (zeton != "do") {
        napaka(s, ind, "Neveljavna zanka" + zeton);
        return;
    }
    ind++;
    ign(s,ind);


    izhod[tI] << "\tmov rax, [" << glob[spre[ime].glob] << " - " << spre[ime].
        pol*8 << "]\n\tdec rax\n\tmov [" << glob[spre[ime].glob] << " - " <<
        spre[ime].pol*8 << "], rax\nzanka" << stVej << ":\n\tmov rax, [" << glob
        [spre[ime].glob] << " - " << spre[ime].pol*8 << "]\n\tinc rax\n\tmov ["
        << glob[spre[ime].glob] << " - " << spre[ime].pol*8 << "], rax" << endl;

    izraz(s, ind);
    izhod[tI] << "\tmov rbx, [" << glob[spre[ime].glob] << " - " << spre[ime].
        pol * 8 << "]\n\tcmp rax, rbx\n\tjl konec" << stVej << endl;
}
```

4.8 Funkcije

Naša zadnja vrsta razvejanja bodo funkcije. Funkcije bodo delovale malo drugače kot pogoji in zanke. Veja funkcije bo vedno ena in vedno bo najzunanjejša veja. To pomeni, da veja funkcije ne bo nikoli otrok veje zanke, pogoja ali sebe. Funkcijo bomo definirali z ključno besedo fun. To bo delovalo le, če se trenutno ne nahajamo v nobeni veji. Funkcijo bomo označili kot vejo tipa -1. Besedi *fun* sledi ime funkcije in poljubno število argumentov. To število prevajalnik shrani. Ko definiramo funkcijo, zamenjamo izhod, v katerega zapisujemo. Zapisali si bomo tudi seznam vseh spremenljivk, ki jih na novo definiramo znotraj funkcije, zato da lahko ta imena kasneje sprostimo.

```
struct funkcija{
    int pol;
    int stArgumentov;
};

int tI = 0;
map<string, funkcija> funk;

string trenFunk = "";
void definirajFunkcijo(string &s, int &ind) {
    if (tI) {
        napaka(s, ind, "Poskus definiranja funkcije znotraj funkcije");
        return;
```

```
    }
    if (veje.size()) {
        napaka(s, ind, "Poskus definiranja funkcije znotraj veje.");
        return;
    }
    stVej++;

    veje.push({-1, -1});

    ign(s,ind);

    string ime = dobiZeton(s, ind);

    if (spre.count(ime)) {
        napaka(s, ind, "Napaka pri definiranju funkcije.\nObstaja ze
            spremenljivka z imenom " + ime + ".");
    }
    definirajSpremenljivko(ime);
    trenFunk = ime;

    tI = 1;
    izhod[tI] << "funkcija" << stVej << ":" << endl;
    ign(s, ind);

    int stArgumentov = 0;
    while (ind < s.length()) {
        string zeton = dobiZeton(s, ind);

        if (spre.count(zeton)) {
            napaka(s, ind, zeton + " ze obstaja kot globalna spremenljivka.");
            return;
        }
        shraniSpremenljivko(zeton);
        stArgumentov++;
        ign(s, ind);
    }

    funk[ime] = (funkcija){stVej, stArgumentov};
    izhod[tI] << "\tpush rbp\n\tmov rbp,rsp\n\tsub rsp, " << stArgumentov*8 <<
        endl;
}
```

Funkcije imajo možnost vrniti vrednost. V tem primeru se bo rezultat funkcije shranil v spremenljivki z enakim imenom kot funkcija.

```
void izhodIzFunkcije(string &s, int &ind) {
    if (!izhod) {
        napaka(s, ind, "Poskus izhoda iz funkcije v globalni kodi.");
        return;
    }
```

```
    izraz(s, ind);
    izhod[tI] << "\tmov [" << glob[spre[trenFunk].glob] << " - " << spre[
        trenFunk].pol * 8 << "], rax" << endl;

    izhod[tI] << "\tmov rsp, rbp\n\tpop rbp\n\tret" << endl;
}
```

Ko funkcijo pokličemo, ji moramo pripisati enako število argumentov, kot ko smo jo definirali.

```
void kliciFunkcijo(string &s, int &ind, string ime) {
    izhod[tI] << "\tpush 0\n\tpush 0" << endl;
    while (ind < s.length()) {
        izraz(s, ind);
        izhod[tI] << "\tpush rax" << endl;
    }
    izhod[tI] << "\tadd rsp, " << (funk[ime].stArgumentov+2)*8 << "\n\tcall
        funkcija" << funk[ime].pol << endl;
}
```

5 Viri in literatura

1. *LET'S BUILD A COMPILER!*
   Jack W. Crenshaw, Ph.D.
   24 July 1988

2. https://www.cs.virginia.edu/ evans/cs216/guides/x86.html
   x86 Assembly Guide
   Datum dostopa: 14. november 2023

3. https://en.cppreference.com/w/
   C++ reference
   Datum dostopa: 15. november 2023

4. https://eucbeniki.sio.si/
   i-učbenik za matematiko
   Datum dostopa: 20. december 2023

5. https://lusy.fri.uni-lj.si/ucbenik/book/index.html
   Informatika 1
   Datum dostopa: 15. november 2023

6. https://www.cs.man.ac.uk/ pjj/farrell/comp3.html
   Anatomy of a compiler
   Datum dostopa: 15. november 2023

6 Program in uporaba

Kot prilogo dodajam kodo prevajalnika. Za uporabo jo morate prevesti s pomočjo C++ prevajalnika. Na sistemih linux, za katerega je ta program namenjen, je to lahko g++. Poleg prevedenega prevajanika dodamo datoteko z izvorno kodo programa v našem novem jeziku. To datoteko dodamo kot argument ko zaženemo prevajalnik.

```
g++ -o prevajalnik prevajalnik.cpp
./prevajalnik koda.k
```

Ta ukaza bosta prevedla datoteko koda.k v assembly. V mapi poleg prevajalnika se bo pojavila nova datoteka out.asm. To je prevedena koda. Če želimo assembly spremeniti v program, ki ga lahko zaženemo moramo uporabiti še nekaj ukazov.

```
nasm -f elf64 out.asm -o out.o
ld -o out out.o
./out
```