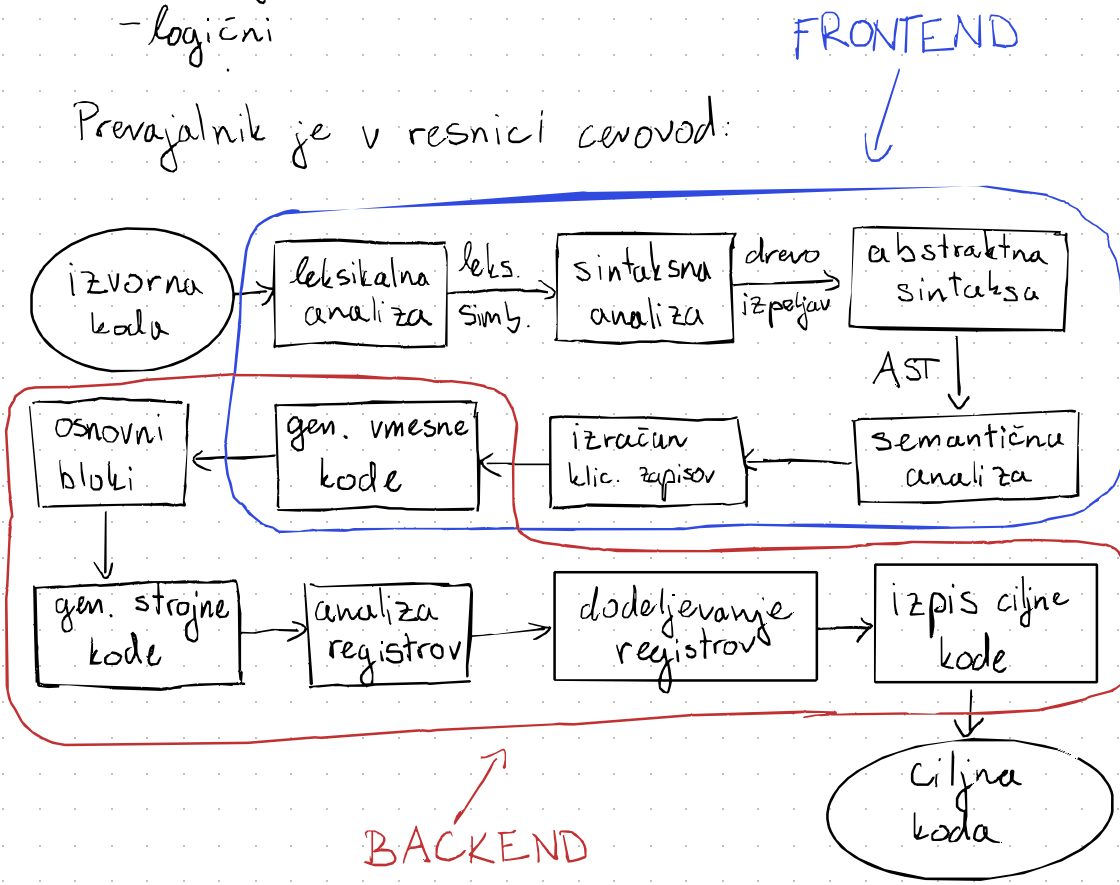
PREVAJALNIKI

Jeziki ki so:
- prevedeni

in niso
- objektno orientirani
- funkcijski
- logični

Compiler Red Dragon Book

Prevajalnik je v resnici cevovod:



LEKSIKALNA ANALIZA

vhod: program v izvornem jeziku
izhod: seznam leksikalnih simbolov

Kaj je leksikalni simbol

1. vrsta leksikalnega simbola
   - ↳ ime tipa, funkcije
   - ↳ ključna beseda
   - ↳ konstante
   - ↳ prirejanje ...
2. znakovna predstavitev
3. položaj v izvorni datoteki

(1.) Naredimo DKA

(2.a) Tabela

	ASCII znaki
stanja	v katero stanje gremo

(2.b) Naredimo s switchi (2 nested)

main → vmesni simbol, ampak ga rabimo obdržati? zelo dolgo → do linkerja

1. leks. analiza je požresna
2. numerične konstante so vedno $\underline{\text{ne}}$predznačene

Orodje ANTLR

Gramatika brez Omejitev	TS
Kontekstno odrisna gramatika	LBA
KNG	SA
Linearna gramatika	KA

SINTAKSNA ANALIZA

vhod: zaporedje leksikalnih simbolov
izhod: drevo izpeljav (sled izpeljave)

skrajno leva / desna izpeljava
(lm) (rm)

sintakso jezika opišemo s KNG

java language specification version 1

algoritmi:
- LL leva sled izpeljave, od zgoraj navzdol
- LR leva sled izpeljave, od spodaj navzgor
- PEG
- sintaksni kombinatorji (parser combinators)

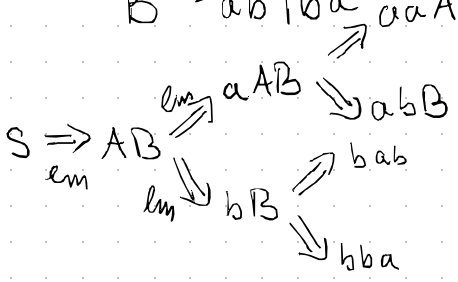
↳ kanonični LL, LALL, SLL, ... ALL(*), ... GLL
↳ kanonični LR, LALR, SLR, ... GLR

Algoritem LL

primer: $S \to AB$

$A \to aA \mid b$

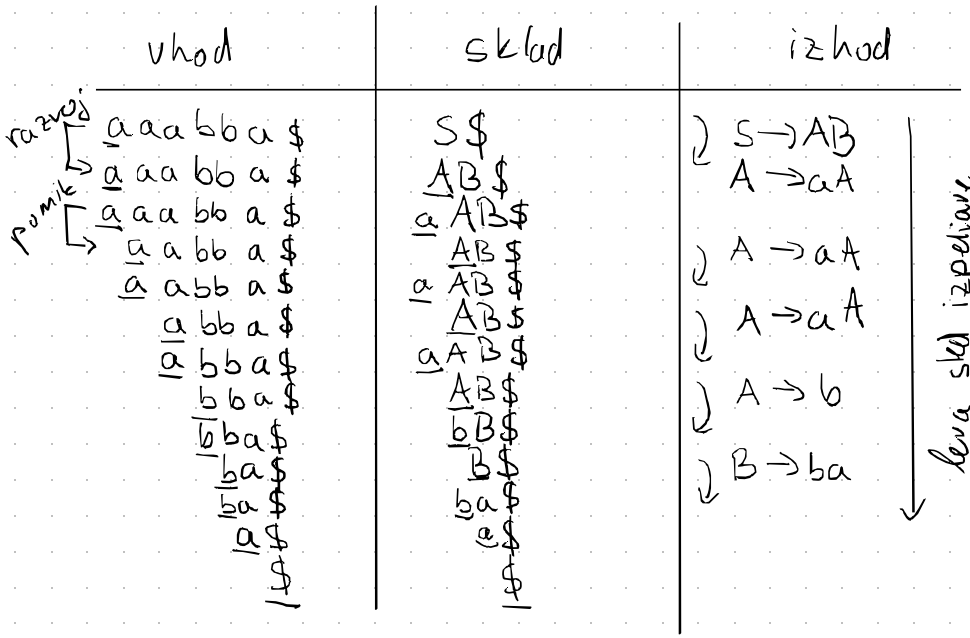
$B \to ab \mid ba \mid aaAB$




```
void parseS(){
  switch(peek()){
    case 'a':
    case 'b':
      parseA()
      parseB()
      break
    default: err()
  }
}
```

```
void parseA(){
  switch(peek()){
    case 'a':
      next('a')
      parseA()
      break
    case 'b'
      next('b')
      break
    default err()
  }
}
```

```
void parseB()
  switch(peek()){
    case 'a':
      next('a')
      next('b')
      break
    case 'b'
      next('b')
      next('a')
      break
    defaut err()
  }
}
```



	a	b	$
S	S → AB	S → AB	
A	A → aA	A → b	
B	B → ab	B → ba	

LL1 gramatika

(v vsakem kvadratku ≤1 vpis)

LL1 podmnožica deterministični, nedvoumnih KNG jev

na foro RT: prevajalniki → seminar o zagonju

$E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to \underline{id}$

$\notin LL(1)$
$\in LR(1)$

slabo berljivi LL(1)

$E \to TE'$
$E' \to \varepsilon \mid +TE' \mid -TE'$
$T \to FT'$
$T' \to \varepsilon \mid *FT' \mid /FT'$
$F \to \underline{id}$

$\in LL(1)$

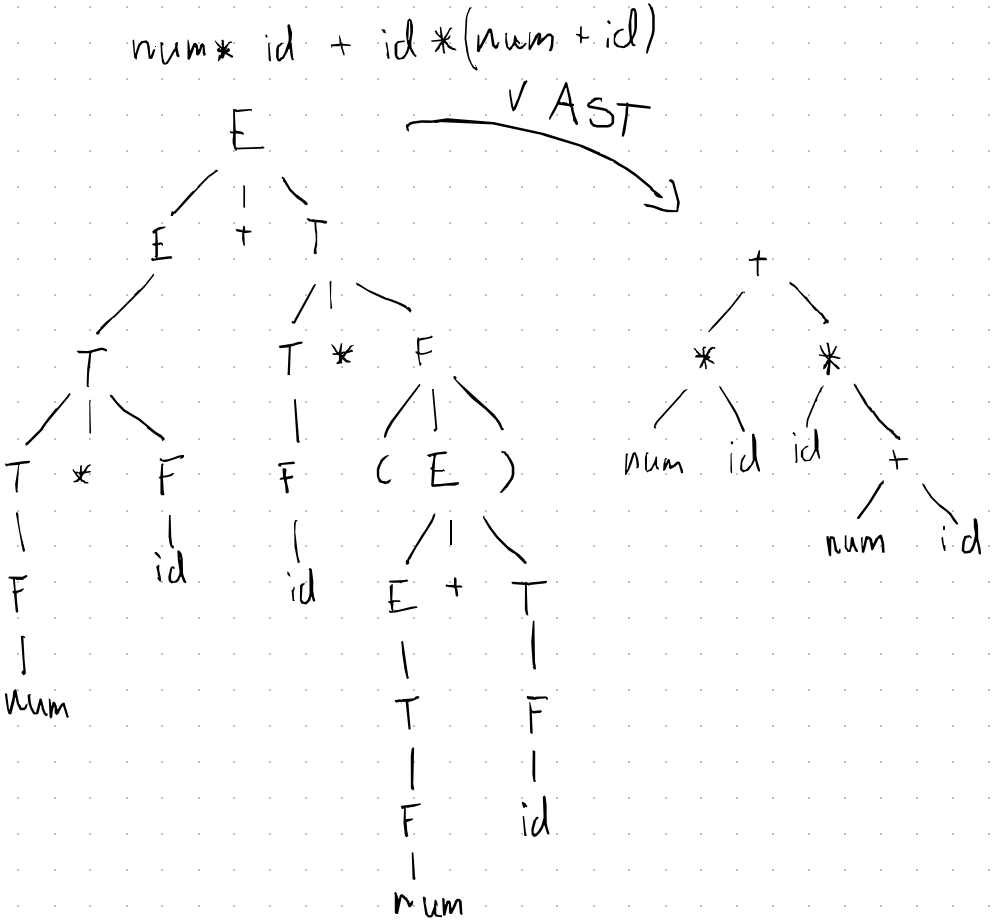
RAKU – v resnici verzija PERLA

$E' \to E==E \mid E$
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id \mid num$

ABSTRAKTNA SINTAKSA

vhod: drevo izpeljav

izhod: abstraktno sintaksno drevo



$E \rightarrow E_1 + T$	$E.AST = +(E_1.AST, T.AST)$
$E \rightarrow T$	$E.AST = T.AST$
$T \rightarrow T * F$	$T.AST = *(T_1.AST, F.AST)$
$T \rightarrow F$	$T.AST = F.AST$
$F \rightarrow \underline{id}$	$F.AST = \underline{id}$
$F \rightarrow \underline{num}$	$F.AST$
$F \rightarrow (\ E\ )$	

```
ast ParseE(){
    (
    (
    (
    (
    (

}
```

Sintaksno usmerjeno prevajanje

od izvorne kode do abstraktnega sintaksnega drevesa in nato prevajanje poddreves

↳ orodje: atributne gramatike

= KNG + atributi + semantična pravila za izračun atributov

SEMANTIČNA ANALIZA

vhod: abstraktno sintaksno drevo
izhod: abstraktno sintaksno drevo + atributi

- razreševanje imen } delamo mi
- preverjanje tipov }
- odkrivanje nedosegljive kode
- analiza strogosti v kodi

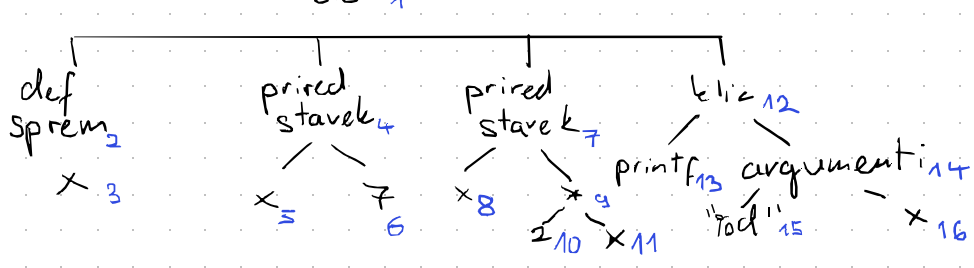
Razreševanje imen

↳ali so vsa imena "znana, definirana..."
↳povezava med mestom definicije in mestom uporabe

```
C {
    int x; x = 7; x = 2 * x;
    printf("%d", x);
}
```

mesto definicije

mesta uporabe



simbolna tabela - preslikava imena v definicij

med obhodom AST ja:

↳ na mestu definicije imena vstavi (ime, def) v S.T.

↳ na mestu uporabe imena
- če def obstaja: usmerimo kazalec na def
- če def NE obstaja: javimo napako

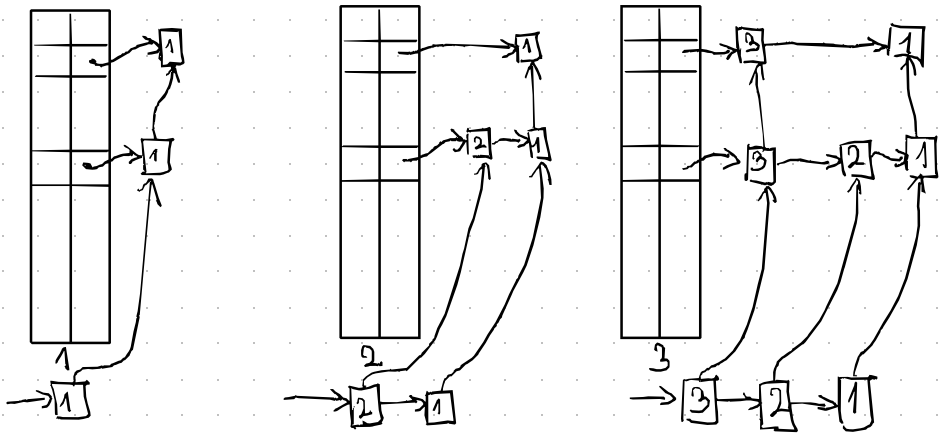
↳ če je ime vidno: • javimo napako
• nova def. zasenči staro

insert: dodaj ime, def v s.t. ALI javi napako

find : vrne def danega imena ALI javi napako

new_scope: začne novo področje dosega

old_scope: konča trenutno področje dosega



Razreševanje tipov

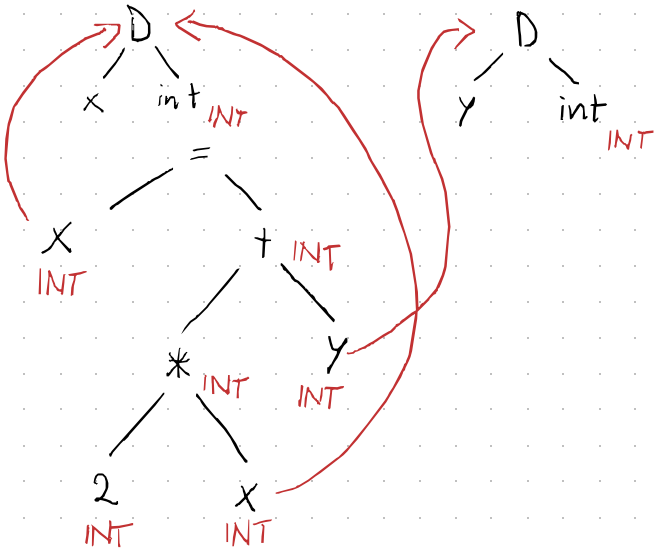
tip = množica vrednosti in operacije nad njimi

tipizacija → statična (prevajanje)
tipizacija → dinamična (izvajanje)

statično preverjanje tipov

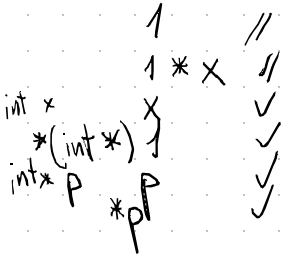
1. za vsak tip v programu določimo opis in predstavitev
   - vgrajeni tipi (int, char ...)
   - definirani tipi (struct, union, class ...)
2. preverjanje tipov

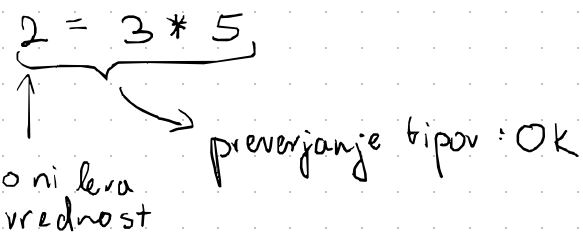
primer: `x = 2 * x + y`



za znavanje neskončnih tipov
↳ razreševanje rekurzivnih tipov

Preverjanje levih vrednosti (l-values)





Izračun konstantnih podizrazov

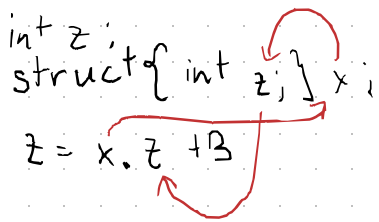
x = 1+3; ⟹ x = 4 ← ok

sizeof (int) ← NUJNO !!!

char str[128+1];

preobtežene metode:

int max(int i, int j){...}
int max(int i, int j, int k){...}
} to dela name mangling



KLICNI ZAPISI IN SPREMENLJIVKE

vhod: AST z atributi

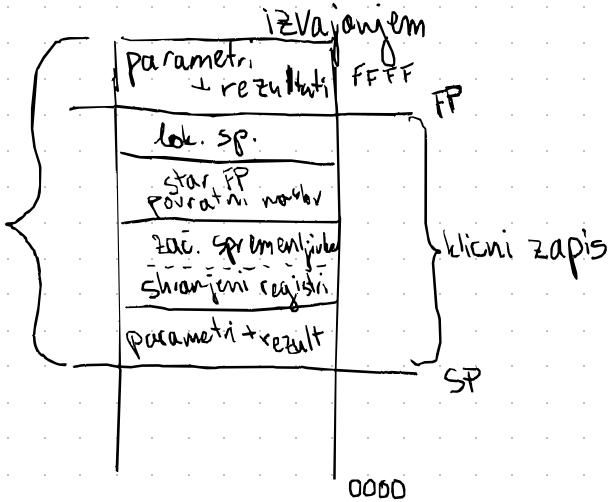
izhod: AST z več atributi

spremenljivke:

- ↳ statične spremenljivke ← fiksni naslov, čas izvajanja programa
- ↳ avtomatične spremenljivke ← spremenljiv naslov, čas izvajanja funkcije
- ↳ registrske spremenljivke
- ↳ zunanje spremenljivke

komponente zapisov (odmik, velikost)

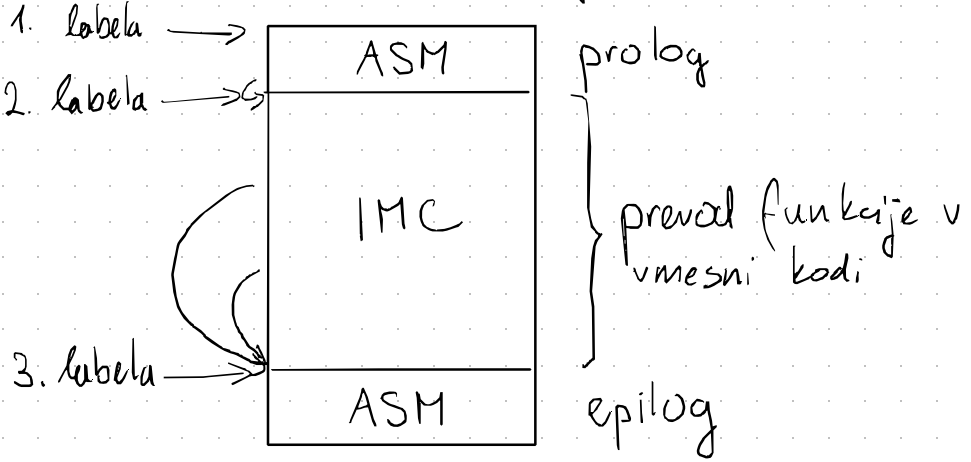
klicni zapis = del sklada, ki pripada funkciji med izvajanjem



GENERIRANJE VMESNE KODE

vhod: AST + atributi

izhod: vmesna koda za posamezne funkcije



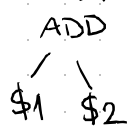
3-adresna vmesna koda:

ADD $1, $2 ; $1 = $1 + $2

4-adresna vmesna koda:

ADD $1, $2, $3 ; $1 = $2 + $3

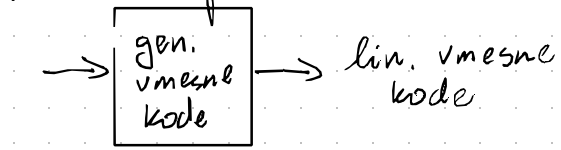
drevesna vmesna koda:



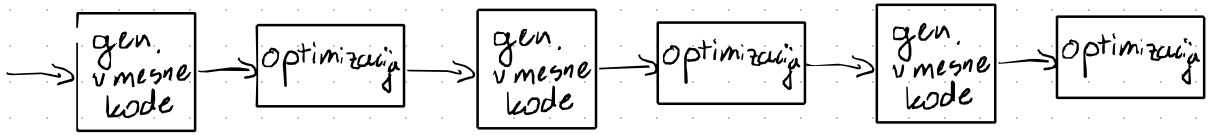
skladovna vmesna koda:

java bytecode

enostavni primeri:



"pravi" prevajalniki



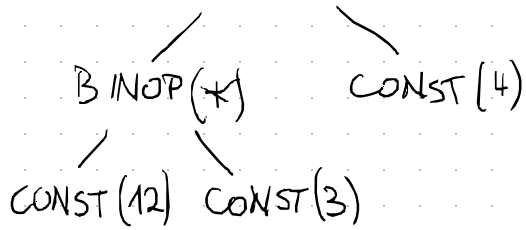
bližu jeziku — vmes — bližu asmju

drevesna koda: ukazi za stavke in izraze

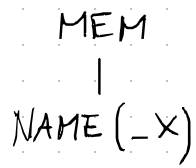
izrazi:
- CONST
- UNOP
- BINOP
- CALL
- NAME
- MEM
- TEMP

12 => CONST(12)

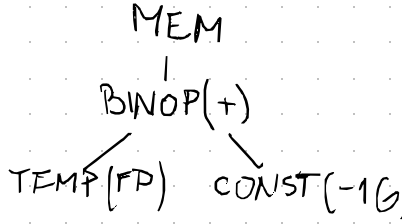
12*3+4 => BINOP(+)



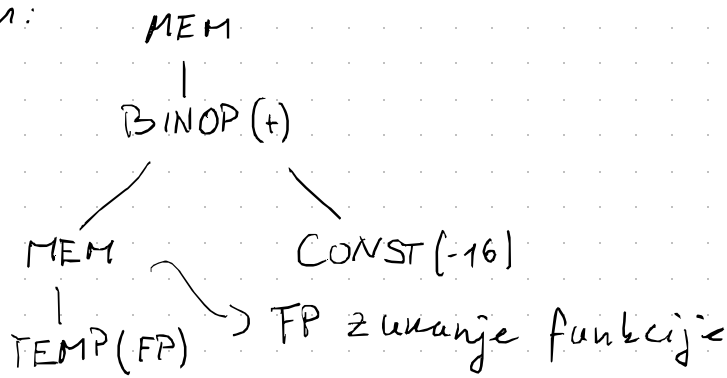
globalna spremenljivka X:



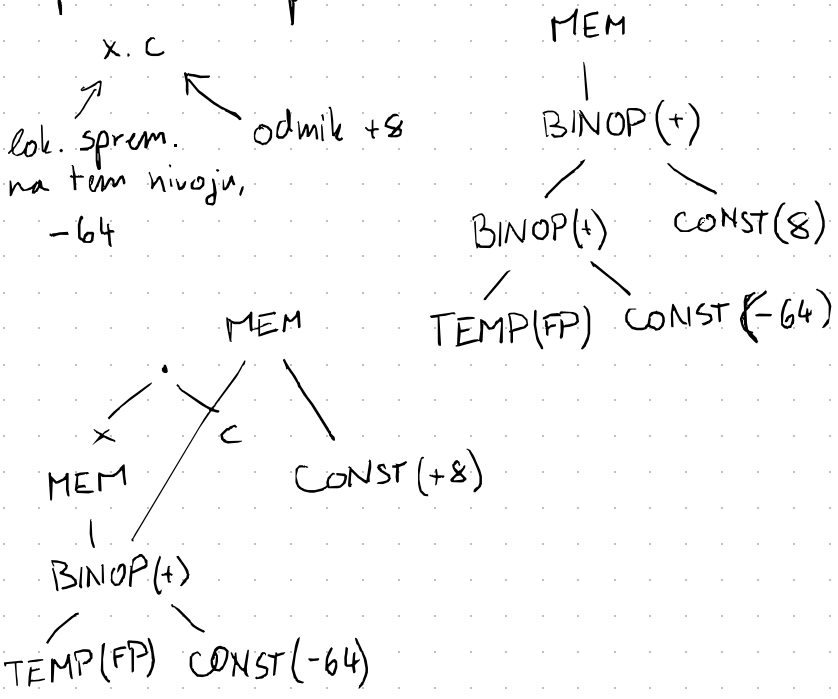
lok. sprem. X:
na trenutnem nivoju
(negativen odmik)



en nivo ven:



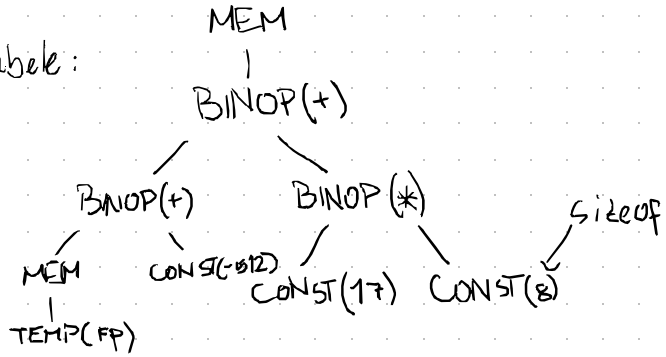
komponenta zapisa:



element tabele:

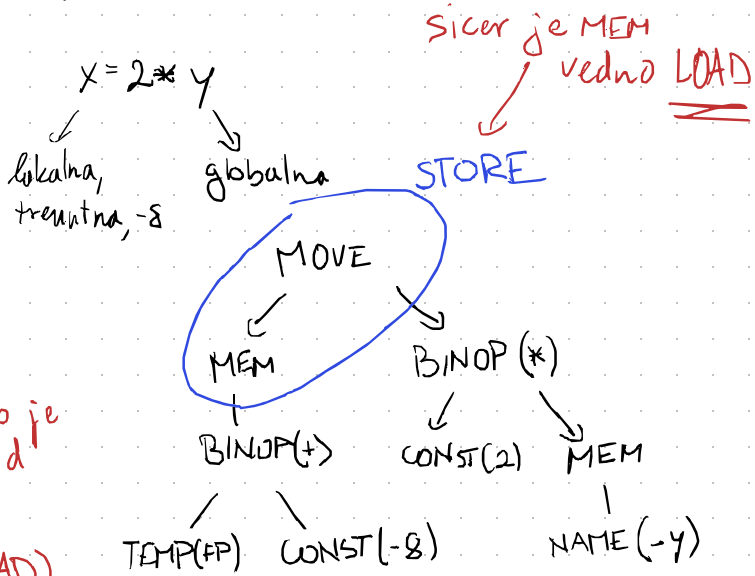
X[17]

en nivo ven

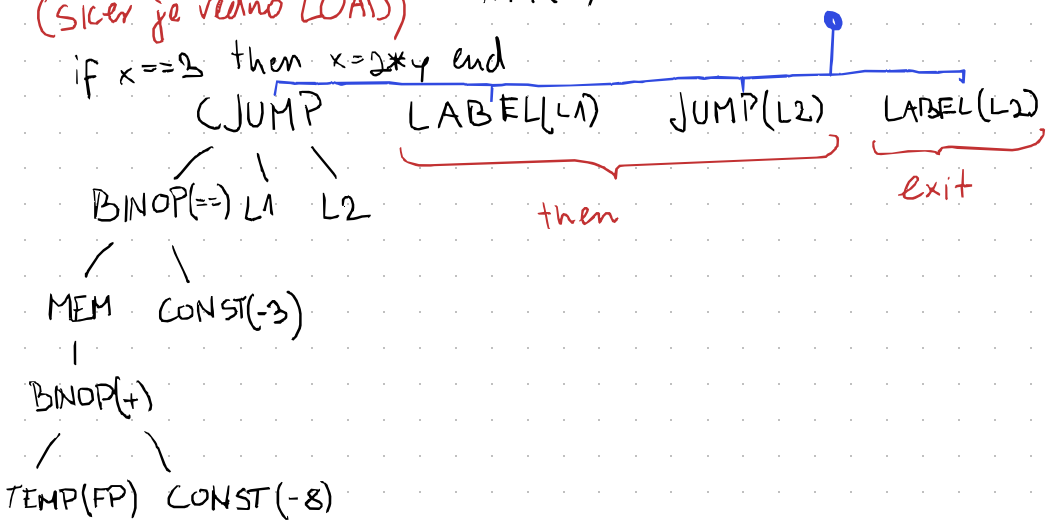


rabimo en argument več, kot jih ima funkcija:
STATIC LINK!!!

starki:
MOVE
JUMP
CJUMP
LABEL



(MEM je STORE natanko tedaj, ko je neposredno levo od MOVE)
(sicer je vedno LOAD)



LINEARIZACIJA VMESNE KODE

vhod: vmesna koda posamezne funkcije

izhod: linearizirana vmesna koda

1. gnezdeni klici funkcij
   $f1(f2(f3(5)))$
2. ukaz SEXPR
3. ukaz CJUMP

CILJ:

MOVE — TEMP, CALL

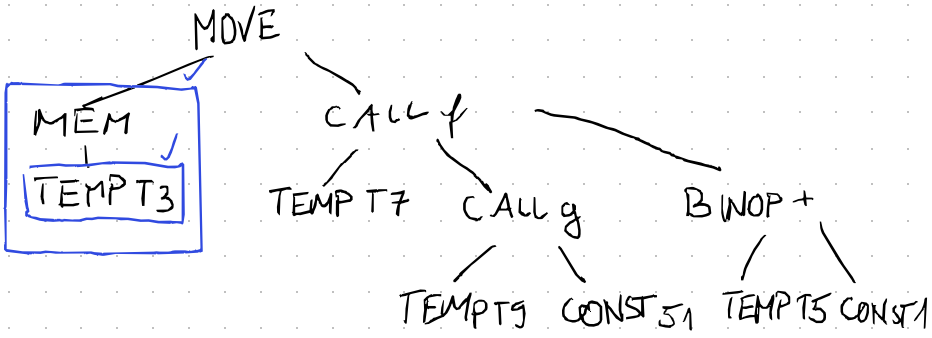
ali

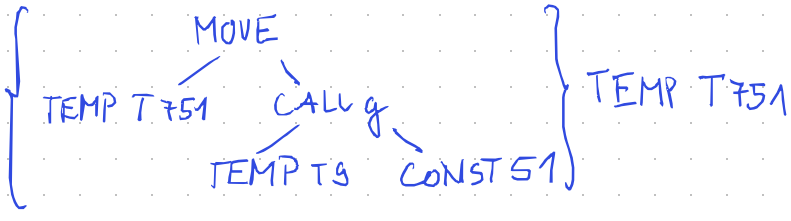
ESTMT — CALL

dve vrsti vmesne kode:

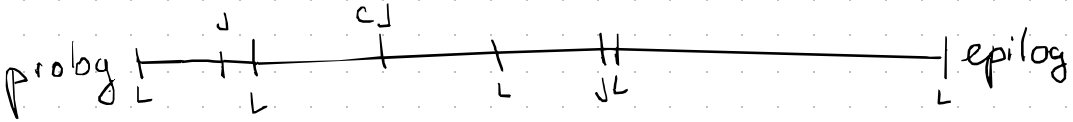
za stavke stavek → {stavkov}

za izraze izraz → {stavkov} izraz



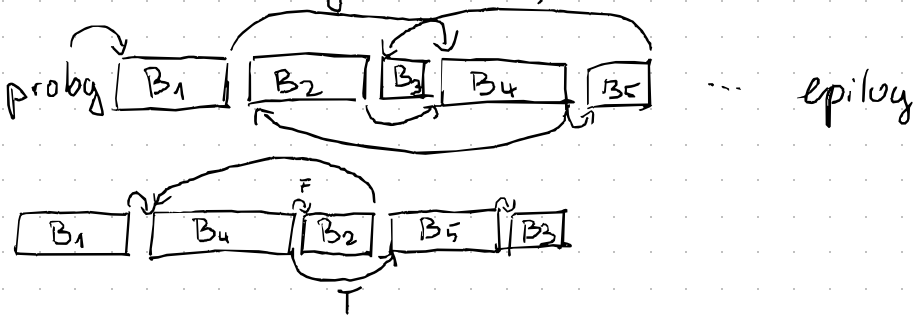


(Tukaj manjkajo primeri)



osnovni bloki ≡ zaporedje stavkov, ki:

1. se začne z LABEL
2. se konča z JUMP ali CJUMP
3. vmes ne vsebuje LABEL, JUMP in CJUMP



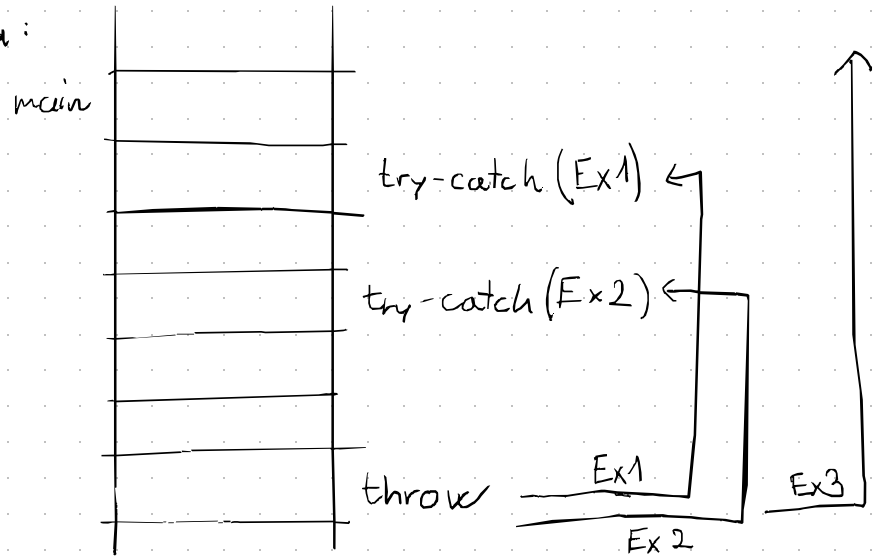
ALGORITHM ENGINEERING

IZJEMF

C: setjump ~try-catch

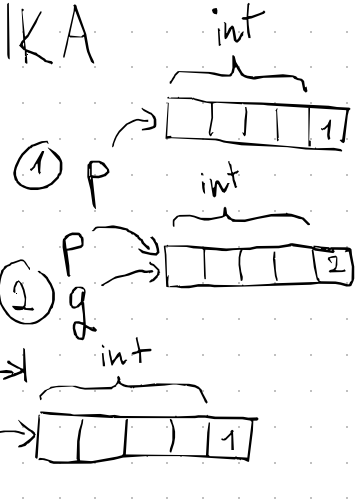
longjump ~throw

Java:



AVTOMATSKO ČIŠČENJE POMNILNIKA

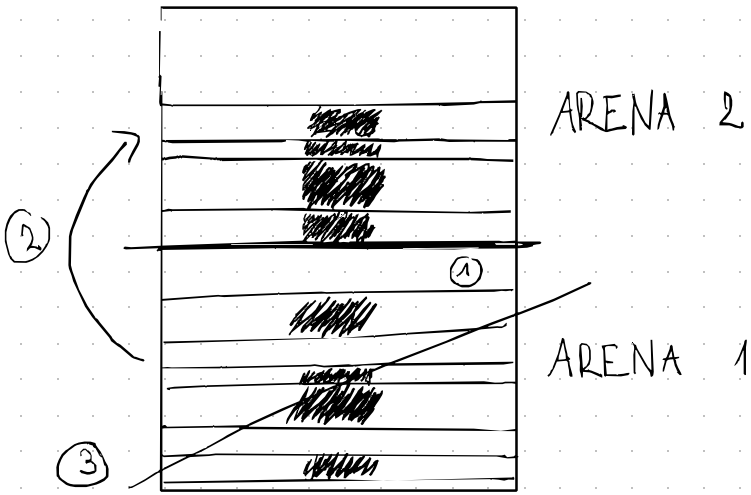
```
int *p = NULL;
(1) p = malloc (sizeof(int));
(2) int *q = p;
(3) p = NULL
```



problem:

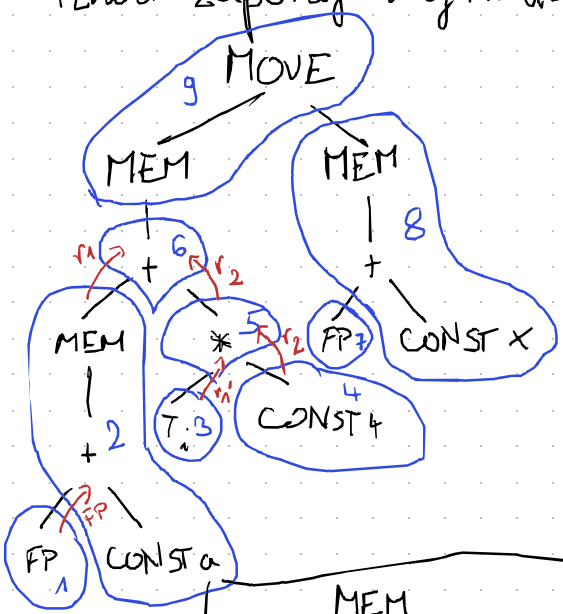
ref counting

mark and sweep

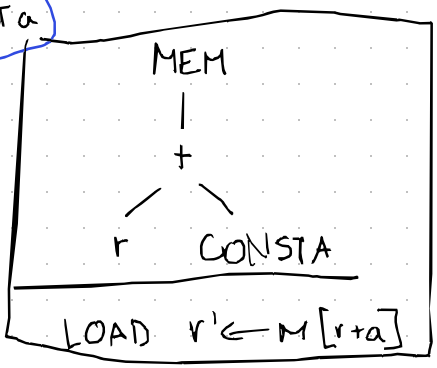


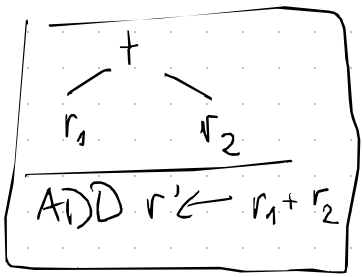
GENERIRANJE STROJNIH UKAZOV

vhod: zaporedje kanonicnih dreves (vsaka funkcija posebej)
izhod: zaporedje strojnih ukazov z zac. sprem.



2: LOAD $r_1 \leftarrow M[FP + a]$
4: ADDI $r_2 \leftarrow r_0 + 4$
5: MUL $r_2 \leftarrow r_1 * r_2$
6: ADD $r_1 \leftarrow r_2 + r_1$
8: LOAD $r_2 \leftarrow M[FP + x]$
9: STORE $M[r_1 + 0] \leftarrow r_2$





MAXIMAL MUNCH ALGORITEM

RAZREDI:

↳ LABEL

↳ OPER → vsi ukazi razen

↳ MOVE

↳ ukazi za prepis vrednosti iz enega v drug register

ADD T1 T2 *

BRZ T3, L17

LOAD T3, T2, \

MOVE T1, T2

ADD T1, T2, 0

MUL T1, T2, 1

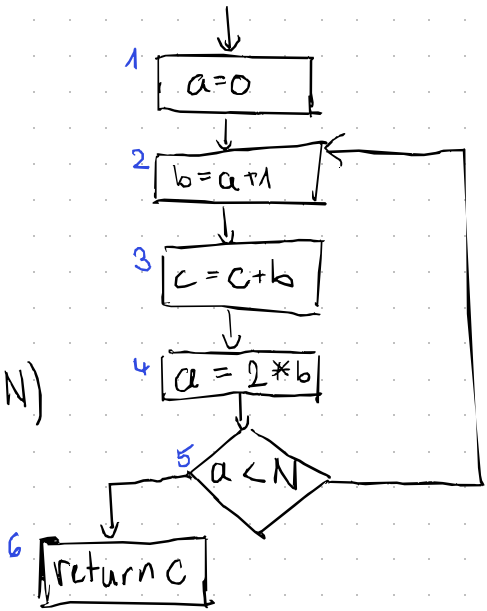
} T1 ⇐ T2

AKTIVNOST SPREMENLJIVK

vhod: zaporedje strojnih ukazov z zač. sprem.
izhod: +1- + interferenčni graf spremenljivk

primer:

```
a := 0
do {
    b := a+1
    c := c+b
    a := 2*b
} while (a < N)
return c
```



use(n) množica začasnih spremenljivk, iz katerih ukaz bere
def(n) mn. zač. spr. , v katere ukaz piše
pred(n) mn. vozl. iz katerih je dosegljivo vozlišče n
succ(n) mn. vozl. ki so dosegljive iz vozl. n.

za cjump tudi 2 kabeli

$$in(n) = use(n) \cup [out(n) \setminus def(n)]$$

$$out(n) = \bigcup_{s \in succ(n)} in(s)$$

znak za unijo

$a = a + b$
DEF USE USE

$c = a + c$
DEF USE USE

$in(n) = USE(n) \cup [out(n) \setminus def(n)]$

$a \quad [a, c \setminus c] = a$

ponovimo ker imamo loop (5 ima 2 succ)

	use	def	out	in	out	in
6	c			c		c
5	a		c	c, a	a, c	a, c
4	b	a	c, a	b, c	a, c	b, c
3	c, b	c	b, c	b, c	b, c	b, c
2	a	b	b, c	a, c	b, c	a, c
1		a	a, c	c	a, c	c

2. iteracija

$\forall n: in(n) = \emptyset, \; out(n) = \emptyset$

$i = 1$

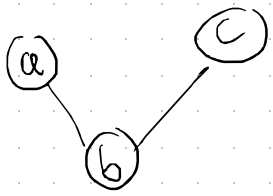
do {

$\forall n: out^{(i)}(n) = \bigcup_{s \in succ(n)} in^{(i)}(s)$

$in^{(i)}(n) = use(n) \cup [out^{(i-1)}(n) \setminus def(n)]$

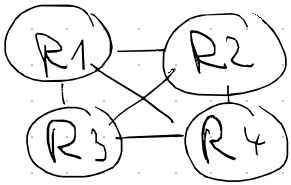
} while $\forall n \; out^{(i-1)}(n) \neq out^{(i)}(n) \; || \; in^{(i-1)}(n) \neq in^{(i)}(n)$

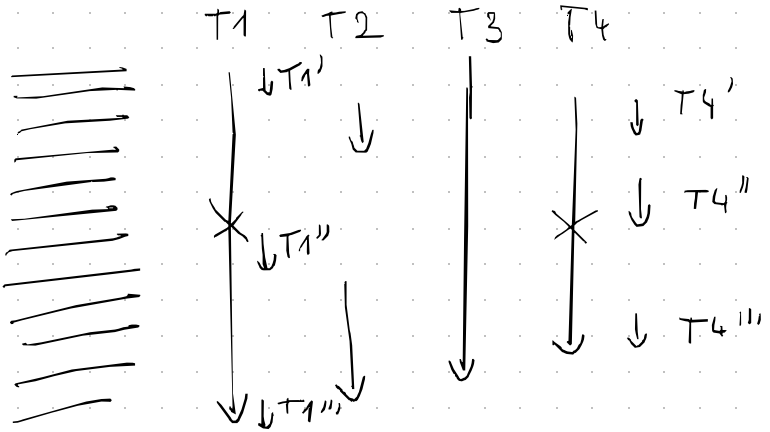
$\forall n: out(n) = out^{(i)}(n) \quad in(n) = in^{(i)}(n)$



} interferenční graf

precoloring





v skrcajnosti:

ADD T1, T2, T3

load T2'

load T3'

ADD T1' T2' T3'

store T1'

DODELJEVANJE REGISTROV

vhod: zaporedje strojnih ukazov + interferenčni graf

izhod: zaporedje strojnih ukazov z registri

1. če obstaja vozlišče z manj kot K sosedi ga umaknemo iz grafa na sklad

2. izberemo vozlišče in ga umaknemo na sklad in ga razglasimo za morebitni preliv

3. s sklada prestavim vozlišče v graf
   če vozlišče lahko pobarvam → ga pobarvam
   če vozlišča ne morem pobarvati → dejanski

4. če obstaja dejanski preliv: popravim kodo + int. graf (preliv!!)
   sicer uspeh