**U IZRADI**

**TOČNOST ZADATAKA NIJE NIMALO SIGURNA ☺ SVE TREBA PROVJERITI**

**1.6.**

**1.**

neka fizička mašina - kalkulator, server poslužitelj web aplikacija, Java Virtual Machine

**2. **

First, it fetches the next instruction to execute from memory. The instruction is then decoded to determine the operation to be performed as well as its operands. As many

operands as required by the instruction are fetched from memory. After this, the instruction, which must be one of the machine’s primitives, is executed. Once execution of the operation has completed, any results are stored. Then, unless the instruction just executed is a halt instruction, execution passes to the next instruction in sequence and the cycle repeats.

**3.**

**čisto interpretirana implementacija**:

Programi napisani u L se ne prevode eksplicitno u L0, već samo dolazi do “dekodiranja”. To znači da se izvršava kod koji pripada naredbi iz L (taj kôd je u jeziku L 0). Dakle, interpreter kao rezultat rada **NE** daje kôd u L 0.

**čisto prevedena implementcija :**

Postupak translacije iz L u L0 naziva se prevođenjem (eng. compilation) te kao rezultat daje program koji je moguće izvršiti u bilo kojem trenu kada za to postoji potreba

**Usporedba čisto interpretirane i čisto prevedene implementacije**

Čisto interpretirana implementacija

*Niska efikasnost* zbog utroška vremena na dekodiranje elemenata iz L .

Također, za višestruka pojavljivanja iste naredbe iz L , interpreter obavlja

višestruka dekodiranja

*Visoka razina fleksibilnosti* zbog mogućnosti interakcije s interpreterom

tijekom izvođenja programa (pogodno za korištenje u alatima za

*debuggiranje*). Također, izrada interpretera je jednostavnija od izrade

prevoditelja te koristi manje memorije u radu.

Čisto prevedena implementacija

Izvršavanje prevedenog programa brže nego kod interpretera zbog

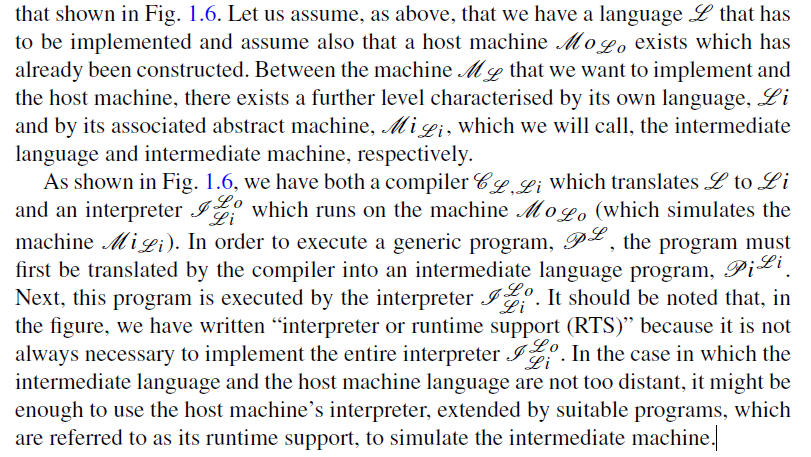
nepostojanja potrebe za dekodiranjem. Također, višestruka pojavljivanja

iste naredbe iz L se prevode jednom i izvršavaju više puta.

Gubitak većine informacija o izvornom programu otežava korištenje u

alatima za *debuggiranje*.

**4.**

****

**5.**

Kada se jezik želi implementirati na mnogo različitih platformi (svojstvo prenosivosti (eng. portability)), često se koristi prevođenje u među-jezik te (po potrebi) interpretiranje međujezika na različitim platformama

**6.**

**-**

**7.**

Evaluacijom izraza „(, P)“ dobivamo

odnosno dobivamo interpreter koji uvijek interpretira program P za bilo koji ulaz Y.

Objašnjenje Futamurine 1. projekcije:

Parcijalnom evaluacijom se rješavamo ulaznih argumenata koji su konstanti tako da ih ubacimo u sam program odnosno specijaliziramo program za taj konstanti ulazni argument. Kada imamo interpreter (kojem je prvi ulazni parametar program X koji se interpretira, a drugi ulazni parametar Y su ulazni podaci za taj program) pomoću kojeg želimo interpretirati samo jedan određeni program P (za ulazni argument X će uvijek vrijediti X = P gdje je P taj određeni program) onda ćemo parcijalnom evaluacijom dobiti odnosno dobiveni interpreter je specijalizirani interpreter koji će uvijek interpretirati samo program P za ulazne podatke Y i taj postupak je Futamurina 1. projekcija (kada je program koji specijaliziramo interpreter, a specijaliziramo za određeni izvorni kod).

**3.7**

**1.**

Da bi prevodilac H ispravno funkcionirao, mora za svaki ulazni program P (x) pisan u jeziku L završiti s radom

Pretpostavimo da postoji program P koji kao ulaz prima parametar x (piše se P (x)).

Postoji li program H koji kao parametar prima program P (x) i koji:

– završava i ispisuje *“da”* ako i program P (x) završava, te

– završava i ispisuje *“ne”* ako program P nikada ne završava?

Dokaz nepostojanja *problem završetka programa:*

Da bi dokazali (preciznije, opovrgli) poslužit ćemo se dokazom po kontradikciji:

Pretpostavimo da postoji program H kako je navedeno u prethodnoj definiciji

Korištenjem H možemo napisati novi program K koji kao ulaz prima drugi program P koji završava i ispisuje “da” ako H (P, P ) ispisuje “ne”; te koji ne završava ako H (P, P ) ispisuje “da”

Dakle, ako P (P ) ne završava, K (P ) ispisuje “da”, a ako P (P ) završava, K (P ) ne završava

Primjenom K na vlastiti tekst K, možemo temeljem gornjeg pisati: ako K (K ) ne završava, K (K ) ispisuje “da”, a ako K (K ) završava, K (K ) ne završava

Vidljivo je da prethodna tvrdnja nema smisla! Time je dokazana kontradikcija

**2.**



a)

b) Qp stalno završava sa izlazom 1, ako P(P) završi

c) Qp nikad ne završava, ako P(P) ne završi

d) iz b) i c) se vidi da Qp izvršava konstantnu funkciju 1 samo akko P(P) završi

e) da sada možemo odlučiti dali program izračunava konstantu funkciju, onda bi mogli odlučiti i o zabranjenom problemu završetka programa, s obzirom da transformacija, obzirom na P, konstruira Qp koji je skroz općenit.

**4.6**

**1.**

Ako je unesen Y > 0 :

ispis : 2

Ako je unesey Y < 0 :

ispis: 2

**2.**

Ako je unesen Y > 0 :

ispis : 1

Ako je unesen Y < 0 :

ispis: 0

**3.**

a)

\* =

int foo(){

return 0;

}

\*\* =

**4.**

p = malloc();

\*P=123;

q=p;

free(p);

**5. [ BY: *Di je vnago?*** <http://www.fer2.net/member.php?u=7527>**]**

int\* p = (int\*) malloc(sizeof(int)); // p pokazuje na objekt veličine 4B u memoriji

\*p = 5; // taj objekt je broj 5

free(p) // uništimo objekt, ali p još uvijek pokazuje tamo (dangling pointer)

**6.**

ispis: 5 7

**7.,8.,10. : [ BY: *letlampa*** <http://www.fer2.net/member.php?u=7494>**]**

**7.**

ispis: 10 2

**8.**

ispis: 6 7

**9.**

ispis: 6 7

**10.**

ispis: 6 8

**11.**

ispis: 7 1

**12. [ BY: *Di je vnago?*** <http://www.fer2.net/member.php?u=7527>**]**

ispis: 15

**13.**

ispis:

**5.8**

**1.**

int x;

int foo(int n)

{

if( n <= 1 ) return 1;

return foo(n-1);

}

read(x);

foo(x);

**2.**

int foo(int n)

{

return n+1;

}

foo(1);

foo(2);

foo(3);

1. i 2. zadatak **[ BY: *Di je vnago?*** <http://www.fer2.net/member.php?u=7527>**]**  
  
Kôd u Scali.   
Gornja funkcija ***factorial*** je rješenje 1. zadatka. Na stogu će biti **n** aktivacijskih zapisa, a broj n ne mora biti poznat za vrijeme prevođenja pa tako ni broj zapisa nije moguće odrediti.  
Donja funkcija ***factorial\_tr*** je također rekurzivna funkcija, ali se ovdje radi o rekurziji repa. Neovisno o broju n, moguće je za vrijeme prevođenja odrediti broj aktivacijskih zapisa - jedan, jedini zapis dijele svi pozivi funkcije.

html kod:

object Factorial {

def factorial(n: Int): Int = {

if (n < 2) 1

else n \* factorial(n - 1)

} //> factorial: (n: Int)Int

def factorial\_tr(n: Int): Int = {

def fact(n: Int, acc: Int): Int = {

if (n < 2) acc

else fact(n - 1, n \* acc)

}

fact(n, 1)

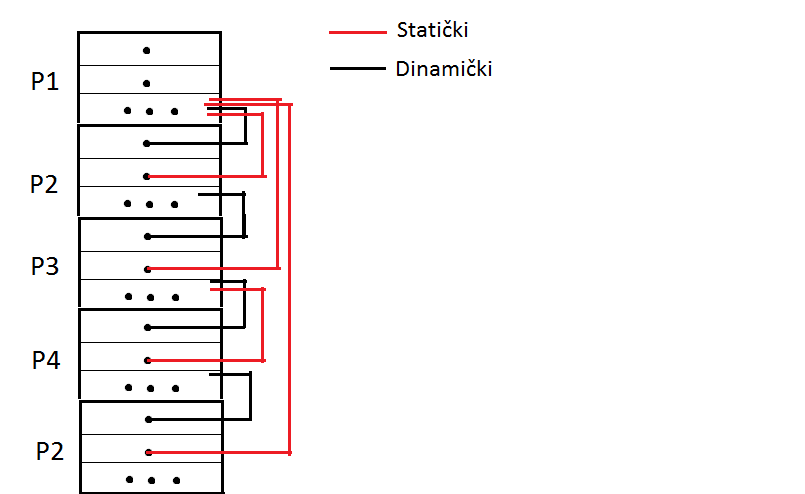
} //> factorial\_tr: (n: Int)Int

factorial\_tr(2) //> res0: Int = 2

}

**3.**

**4.**

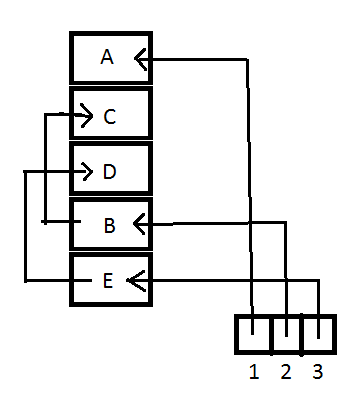
****

**5.**

****

POZIV: A,C,D,B,E

DISPLAY:



**6.**

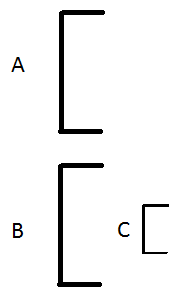
Implementacija dinamičkog dosega je konceptualno jednostavnija od implementacije statičkog s obzirom da to da se svodi na pretraživanje aktivacijskih zapisa od vrha prema dnu stoga (u cilju pronalaska ne-lokalnog imena)

**7.**

POZIV:

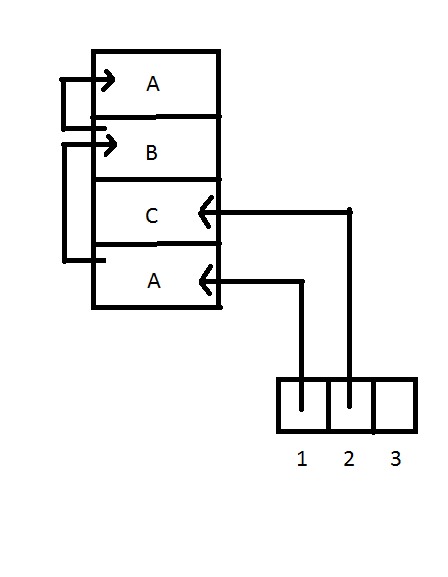
A,B,C,A,B

Gnijezdo:

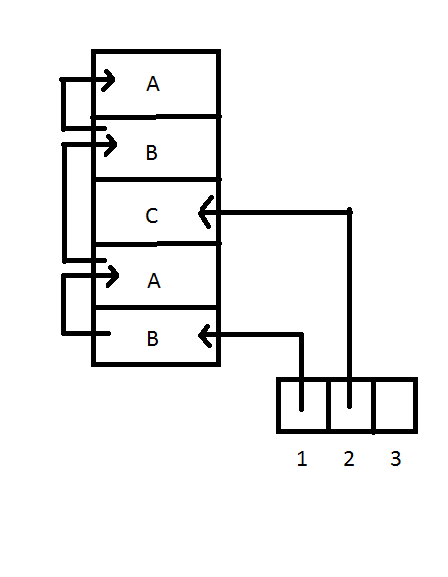


DISPLAY:

NAKON DRUGOG POZIVA A:



NA KRAJU IZVRŠAVANJA:



**6.8**

**1.**

**2.**

**3.**

**4.**

**5.**

**6.**

**7.**

**7.6 [ BY: *Di je vnago?*** <http://www.fer2.net/member.php?u=7527>**]**

**1.**  
"A block can be assigned a name, in which case the name is part of the local  
environment of the block which immediately includes the block to which  
the name has been assigned. This is the case also for blocks associated with  
procedures."  
  
Kada deklariramo funkciju, to ime je dio lokalne okoline bloka u kojem smo deklarirali tu funkciju. S obzirom na gniježđenje, to ime je (nelokalno) dostupno unutar bloka funkcije. To nam omogućava da unutar bloka funkcije pozovemo tu istu funkciju (rekurzija).  
  
"...therefore, because of  
nesting, the association is also visible in the block which constitutes the body of  
the procedure, a fact which permits the definition of recursive procedures)."  
  
**2.** ISPIS: 2 2 2  
  
**3.** ISPIS 2 4  
  
**4.** Call by constant se može implementirati kao call by value (za podatke male veličine) ili kao call by reference (ako se radi o velikim podatkovnim strukturama). Ako ćemo to gledati kao poziv po vrijednosti onda će se ispisati: 2 2. Ako ćemo gledati kao poziv po referenci onda će se ispisati 2 i 3. Ili će biti dojavljena greška??  
  
**5.** ISPIS: 4 4

**6.**

**7.** Value-result: ISPIS: 0  
Reference: ISPIS: 1  
  
**8.** ISPIS: 3 0  
  
**9.** Return 4

**10.**

**8.14. [ BY: *Di je vnago?*** <http://www.fer2.net/member.php?u=7527>**]**

**1.**

Adresa 904. (Da je column-order onda bi bilo 2092)

**2.**

Ne kužim šta hoće ovdje. Pristupam skroz jednako samo što za svaki redak alociram memoriju posebno.

html kod:

#include <stdio.h>

#include <stdlib.h>

#define ROWS 10

#define COLS 10

int main() {

int i, j;

int\* rowPointers[ROWS];

for (i = 0; i < ROWS; i++) {

rowPointers[i] = malloc(sizeof(int) \* COLS);

}

for (i = 0; i < ROWS; i++)

for (j = 0; j < COLS; j++)

rowPointers[i][j] = i + j;

for (i = 0; i < ROWS; i++) {

for (j = 0; j < COLS; j++)

printf("%4d", rowPointers[i][j]);

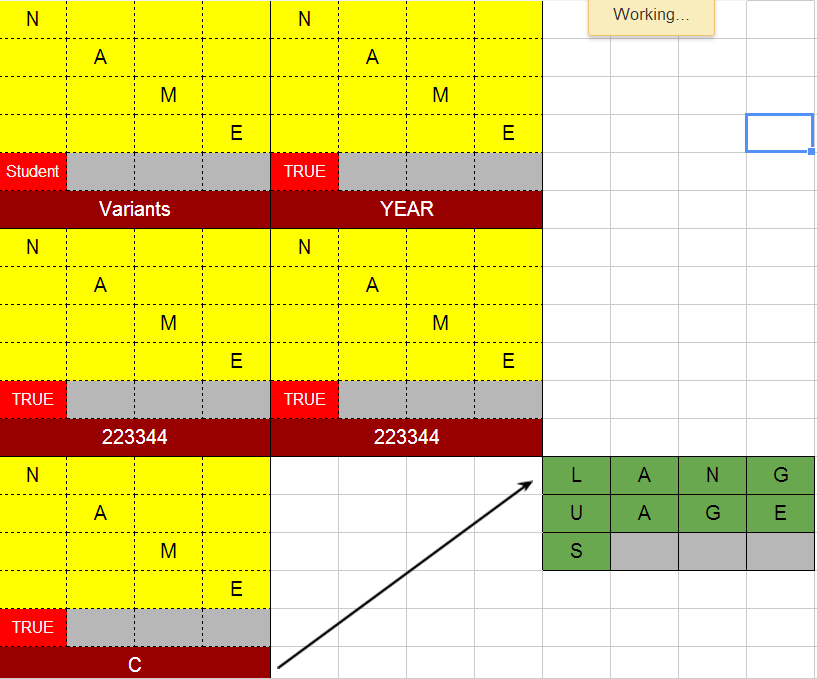
printf("\n");

}

return 0;

}

**3.**



**4.**

Integer rekurzivno. Nisam siguran što se ovdje htjelo. Ja sam to napravio kao niz dekadskih znamenki. Npr. ovdje se radi o brojevima 102. Valjda je to bio cilj. Moglo bi se još i kao npr. binarni zapis, itd.

html kod:

object IntegerRecursive {

def checkEqual(i: Integer, j: Integer): Boolean = {

if (i == null && j == null)

true

else if (i == null || j == null)

false

else if (i.digit != j.digit)

false

else

checkEqual(i.next, j.next)

}

val i = new Integer(1, new Integer(0, new Integer(2, null)))

val j = new Integer(1, new Integer(0, new Integer(2, null)))

checkEqual(i, j)

}

class Integer(x: Int, y: Integer) {

def digit = x

def next = y

}

**5.**

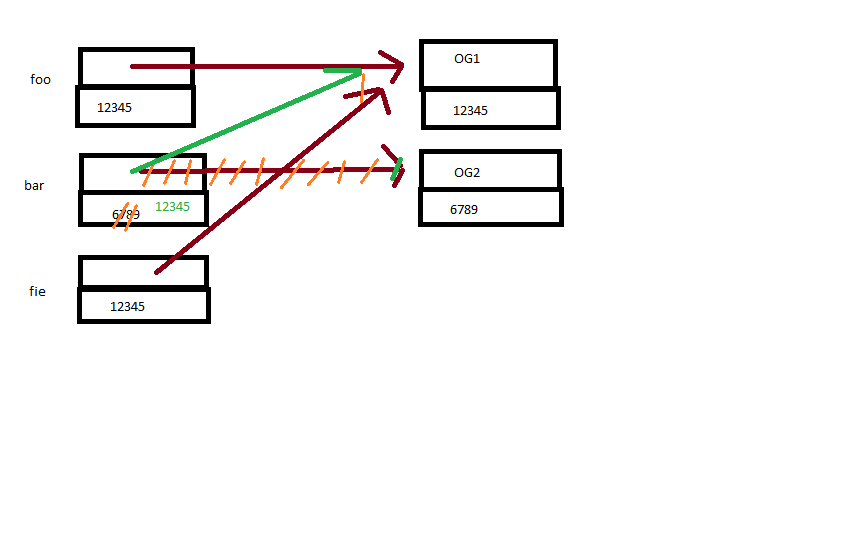
Objašnjenje: T1 i T2 su strukturno ekvivalentni. T3 i T4 se sastoje od struktura T1 i T2. Međutim, nisu navedeni u istom redoslijedu, no kako su T1 i T2 ekvivalentne strukture onda nam je to jedno te isto te su i T3 i T4 ekvivalentne.

**6.**

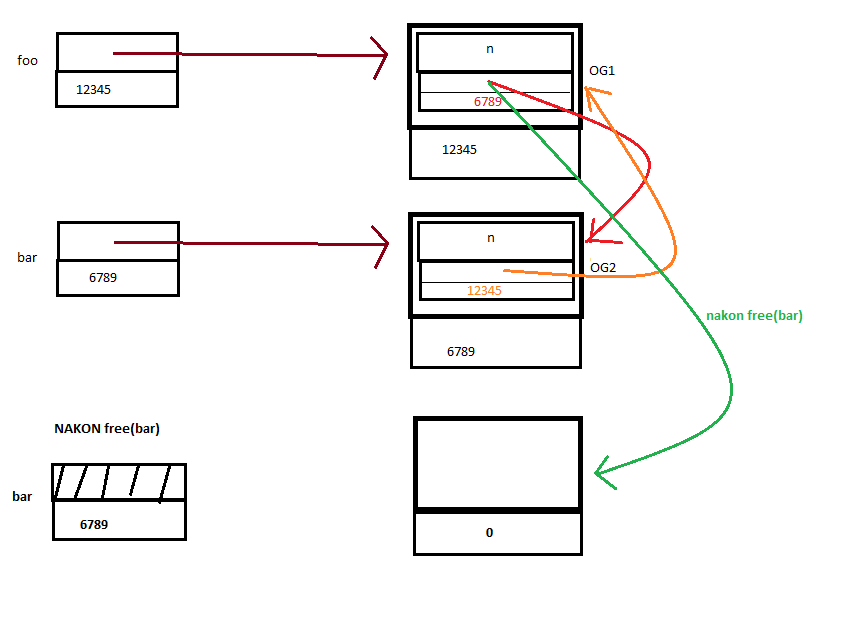
1) 'a \* 'a -> 'a  
2) ('a -> Boolean) \* 'a \* 'a -> 'a  
3) 'a \* 'b -> 'a

**7.**

**8.**



**9.**



**10.**

Nakon linije 6.:  
OGG1 - 1  
OGG2 - 2  
OGG3 - 1  
  
Nakon linije 9.:  
OGG1 - 0  
OGG2 - 1  
OGG3 – 0

**11.**

OG2 može na free list.

**12.**

OG1 i OG2 na free list.