.NET Framework

.NET nije operativni sistem.
.NET je softver koji povezuje informacije, ljude, sisteme i uređaje.

Delovi .NET-a

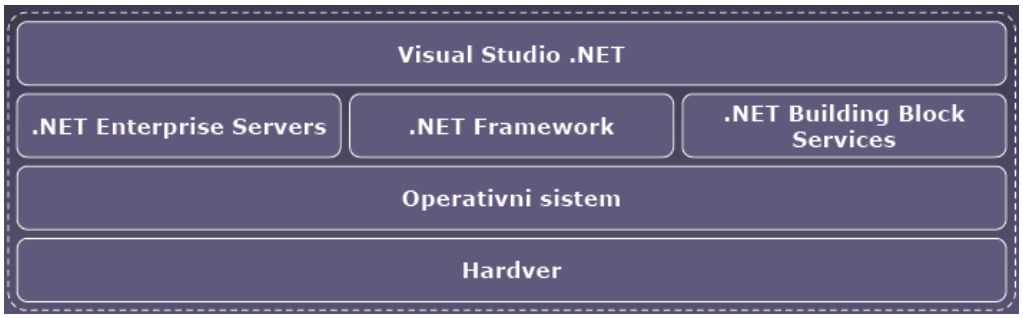
- svi urešaji su međusobno povezani na globalnoj mreži
- softveri kao usluge dostupni na globalnoj mreži

.NET Framework

- osnova za ispunjenje vizije
- servisi i nove tehnologije koje omogućavaju razvoj distribuiranih aplikacija

.NET Enterprise Servers
SQL Server 2000, BizTalk 2000, Commerce Server 2000, ... koriste ih .NET Framework aplikacije

.NET platform



Hardver
Serveri, radne stanice, personalni računari i neračunarski uređaji kao što su mobilni telefoni, pejdžeri, ...

Operativni sistem
Windows Server 2003,Windows XP, Windows 2000, Windows CE, BSD Unix, Linux, ...

.NET Enterprise Servers
Application Center,BizTalk Server 2000, Commerce Server 2000, Exchange Server 2000, Host Integration Server 2000, Internet Securitz and Acceleration Server 2000, SQL Server 2000

.NET Framework - nova razvojno-izvršna infrastruktura za kreiranje distribuiranih aplikacija.

.NET Building Block Services
paradigma “softver kao servis”
XML Web Services (Internet servisi) - komponente koje su dostupne na Internetu i koje se mogu koristiti prilikom razvijanja sopstvenih aplikacija
.NET My Services (HailStorm) - skup Microsoft-ovih korisnički orijentisanih Internet servisa kao što su pasoš, podsetnik, skladište, eMail, ...

Visual Studio .NET
alat za razvoj softvera
integrisano okruženje za kreiranje distribuiranih aplikacija (IDE for RAD)

Najvažniji deo .NET platforme je .NET Framework

.NET Framework

Nova platforma za razvoj softvera.
Sistemska aplikacija koja omogućava

- **razvoj** (projektovanje, kodiranje, uklanjanje grešaka, instalacija, održavanje) i
- **izvršenje** distribuiranih aplikacija

Obezbeđuje

- komponentnu infrastrukturu (Component infrastructure)
- integraciju programskih jezika (Language integration)
- internet interoperabilnost (Internet interoperability)
- jednostavan razvoj (Simple Development)
- jednostavnu instalaciju (Simple Deployment)
- pouzdanost (Reliability)
- bezbednost (Security)

Arhitektura .NET Framework-a

Common Language Runtime - CLR
Najvažniji deo .NET Framework-a. **Nadležan** za aktiviranje objekata, izvršavanje bezbednosnih provera nad njima, njihovo smeštanje u memoriju, izvršavanje i uklanjanje iz memorije.

Base Class Libraries - BCL
Biblioteke tipova koje nudi .NET Framework. Sastoji se iz klasa, interfejsa i vrednosnih tipova koji omogućavaju korišćenje funkcija sistema.

Class Libraries – Data and XML classes
Specijalizovane biblioteke tipova. Zasniva se na tipovima datim u BCL-u.

Tehnologije za razvoj aplikacija
Web Service, Web Forms, Windows Forms, Console Applications. Interfejsne i neinterfejsne aplikacije.

Common Type System - CTS
Skup pravila koje prevodioci .NET jezika moraju da poštuju. Njime su dati predefinisani tipovi podataka.

Common Language Specification - CLS
Specifikacija minimalnih zahteva koje svaki .NET jezik mora da podrži u cilju postizanja integracije jezika.

.NET jezici

C# (C Sharp) – nov jezik razvijen posebno za .NET Framework. Postojeći jezici su ili redizajnirani ili prošireni (VB.NET, managed C++, ...)

Common Language Runtime (CLR)
Common Language Runtime = .NET
N**ajvažniji** deo .NET Framework-a, predstavlja **izvršno okruženje** .NET Framework-a, **nadležan** je za aktiviranje objekata, izvršavanje bezbednosnih provera nad njima. Njihovo smeštanje u memoriju, izvršavanje i uklanjanje iz memorije ono što je Java Virtual Machine (JVM) za Java platformu, to CLR za .NET Framework. I JVM i CLR omogućavaju izvršavanje aplikacija na različitim platformama (nezavisno od hardvera i operativnog sistema) dok JVM podržava jedino programski jezik Java, CLR podržava MS programski jezici: C#, VB.NET, C++.NET, JScript.NET programski jezici drugih proizvođača: COBOL, Eiffel, Perl, Pzthon, SmallTalk, ...
CLR podržava sve programske jezike koji se mogu prevesti u Microsoft Intermediate Language (MSIL)

Proces prevođenja izvornog u mašinski kod odvija se u dva koraka

- izvorni kod se prevodi u Microsoft Intermediate Language **MSIL**
- prevođenje MSIL koda u konkretan platformski kod koji izvršava CLR

Microsoft Intermediate Language (MSIL)
Jezik nižeg nivoa sa jednostavnom sintaksom, koji se vrlo brzo prevodi u mašinski kod podržava sve osobine objektno orijentisanih jezika uključujući apstrakciju podataka, nasleđivanje, polimorfizam i korisne koncepte kao što su izuzeci i događaji uvođenjem ovog međujezika omogućena je:
platformska nezavisnost

- kod napisan na bilo kom .NET jeziku može se izvršiti na bilo kojoj platformi
- Java - bytecode ; .NET - MSIL

poboljšanje performansi

- dok se Java kod interpretira, MSIL se uvek prevodi (ne postoji gubitak performansi koji je neminovan prilikom interpretacije)
- ne prevodi se cela aplikacija odjednom, već samo deo koji se pozove
- prevedeni kod se čuva sve dok se aplikacija ne završi

jezička interoperabilnost

- kod dobijen prevođenjem iz jednog programskog jezika u MSIL, je interoperabilan sa kodom koji je na isti način dobijen iz nekog sasvim drugog programskog jezika

Microsoft Intermediate Language (MSIL)

- osnovne karakteristike
- objektna orijentisanost i korišćenje interfejsa
- razlikovanje vrednosnih i referentnih tipova
- stroga tipiziranost podataka
- upravljanje greškama putem izuzetaka

- korišćenje atributa

Karakteristike MSIL-a

Objektna orijentisanost i korišćenje interfejsa

- podržava sve osobine objektno orijentisanih jezika uključujući apstrakciju podataka, nasleđivanje, polimorfizam i korisne koncepte kao što su izuzeci i događaji
- podržava jednostruko nasleđivanje klasa
- klase koje implementiraju dati interfejs moraju da obezbede implementaciju metoda i svojstava naznačenih konkretnim interfejsom

Razlikovanje vrednosnih i referentnih tipova

vrednosni tipovi

- promenljiva direktno čuva svoje podatke
- čuvaju se na steku

referentni tipovi

- promenljiva sadrži adresu na kojoj se nalaze odgovarajući podaci
- čuvaju se u delu memorije koji se naziva kontrolisani hip

stroga tipiziranost

- svaka promenljiva pripada određenom, konkretnom tipu podataka
- nisu dozvoljene operacije koje ostavljaju mogućnost dvosmislenog tumačenja na koju vrstu podataka se njihov rezultat odnosi (Variant u VB-u)

šta je omogućeno ?

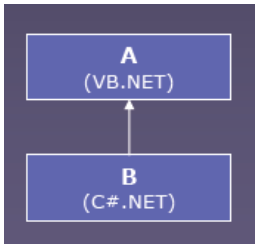
- jezička interoperabilnost
- automatsko upravljanje memorijom
- bezbednost
- aplikacioni domeni

Jezička interoperabilnost

klase napisane u jednom jeziku mogu direktno da komuniciraju sa klasama koje su napisane u drugom jeziku, odnosno:

- klasa napisana u jednom jeziku može da nasledi klasu napisanu u drugom jeziku
- klasa može da sadrži primerke drugih klasa koje su realizovane korišćenjem različitih jezika
- objekat može direktno da pozove metodu drugog objekta koja je napisana u drugom jeziku
- objekti se mogu slobodno prenositi između metoda

- klasa “B” mora da razume sve tipove podataka koje klasa “A” koristi
- različiti jezici – različite ključne reči za iste tipove
- 32-bitni označeni celobrojni tip
- u jeziku VB.NET definisan kao **integer**
- u jeziku C# definisan kao **int**

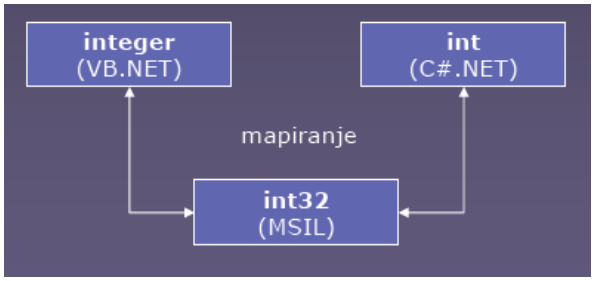


Common Type System

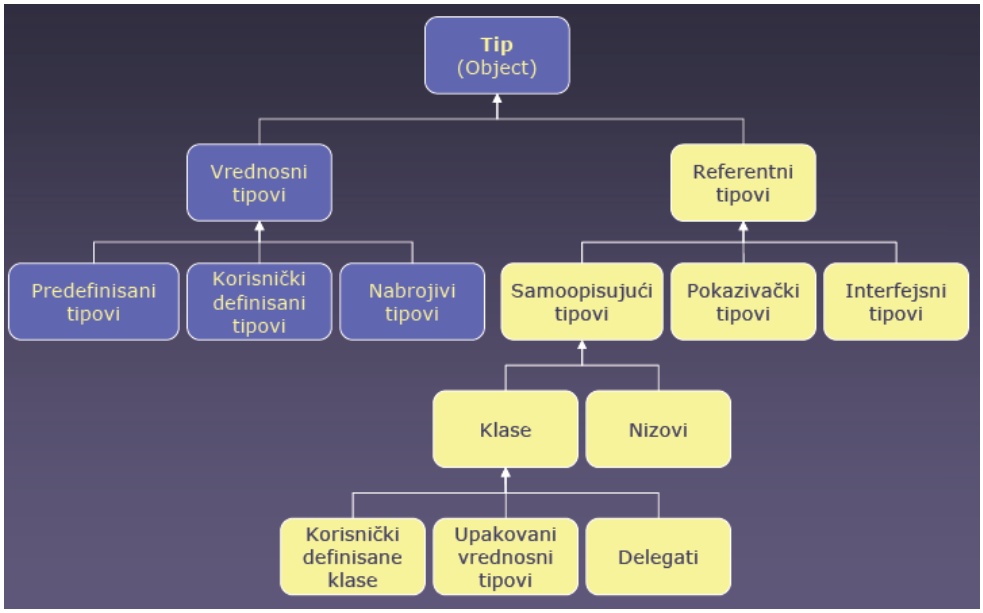
Zajednički sistem tipova definiše skup pravila, koje prevodioci .NET jezika moraju da poštuju da bi definisali, referencirali i smestili i referentne i vrednosne tipove.

CTS definiše predefinisane tipove podataka koji su dostupni u MSIL-u, tako da svi jezici .NET Framework-a proizvode MSIL kod koji se zasniva na ovim tipovima. (VB.NET – integer, C# – int, MSIL – System.int32)

CTS tipovi imaju istu semantiku, bez obzira u kojem su jeziku definisani



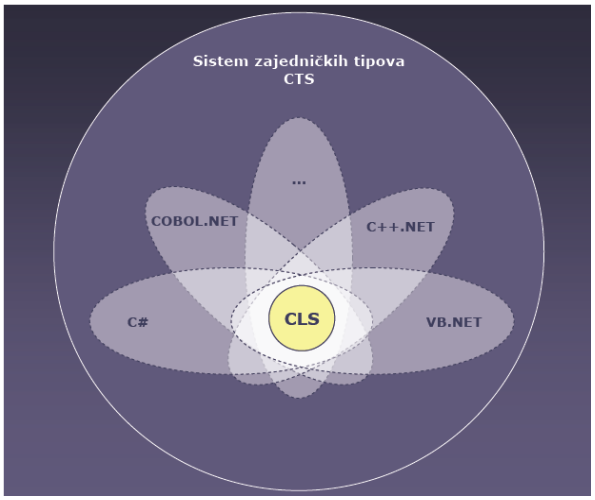
Hijerarhijska struktura CTS-a



Common Language Specification – CLS

- specifikacija zajedničkog jezika predstavlja skup minimalnih zahteva koje svaki .NET jezik mora da podrži, u cilju postizanja interoperabilnosti.
- jedan od zahteva koji nameće CLS je da nije dozvoljeno korišćenje identifikatora koji se razlikuju samo po veličini slova za razliku od VB.NET-a, C# pravi razliku između velikih i malih slova
- nisu sve klase .NET Frameworka u saglasnosti sa CLS-om na primer, uint je definisan u C#-u, a nije u VB.NET-u
- ova specifikacija se odnosi samo na javne i zaštićene članove klasa i javne klase (privatni delovi nisu dostupni)

CTS & CLS



Automatsko upravljanje memorijom

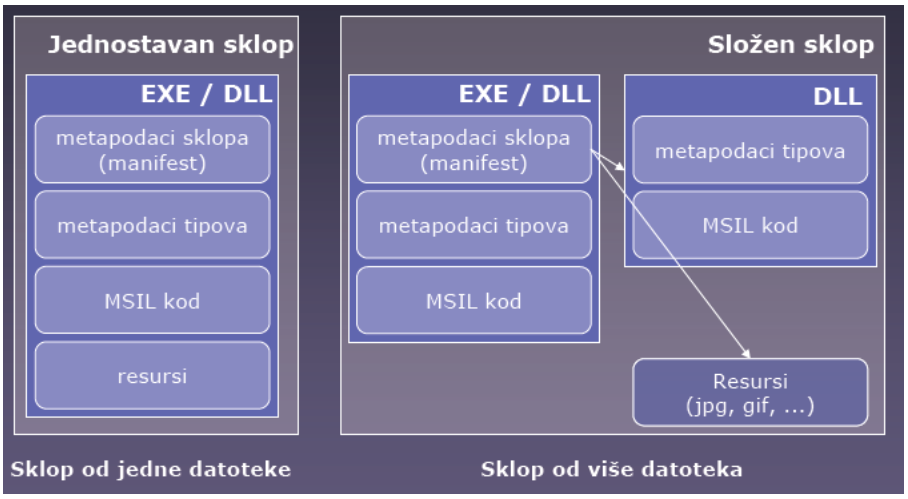
- **komponenta za sakupljanje otpadaka** (garbage collector)
- objekti na hipu do kojih ne vodi ni jedna referenca se automatski oslobađaju
- sprečava se curenje memorije (memory leak), a isto tako sprečavaju situacije gde memorija ostaje zauzeta čak i posle završetka procesa
- **napomena**: vreme kada će upotrebljena memorija biti oslobođena nije tačno određena
- posledica: oslobađanje resursa ne treba da se radi u destruktoru clase

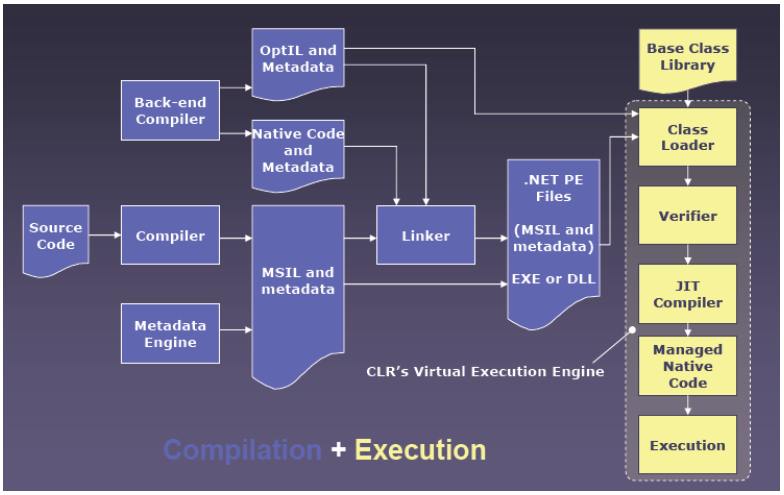
Sklop (assembly)

- Sklop je kolekcija koja se sastoji od jedne ili više datoteka, pri čemu jedna od njih sadrži metapodatke poznate pod nazivom **manifest sklopa**
- manifest sklopa definiše šta sve ulazi u sastav sklopa, verzije, autora sklopa, kakve su bezbednosne dozvole potrebne da bi sklop radio
- IL kod + metapodaci kojima se opisuju tipovi i definisane metode u kodu
- sklop je u potpunosti samoopisujuća logička jedinica

dve vrste

- **privatni sklopovi**
  - zero impact installation
  - može ih koristiti samo njihov softverski paket
- **deljeni sklopovi**
  - instaliraju se u poseban direktorijum – globalni keš sklopa
  - deljena biblioteka – može ih koristiti bilo koja druga aplikacija

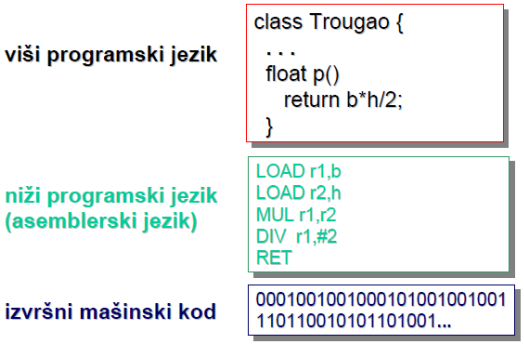




Uvod u programske jezike

Veliki broj različitih viših programskih jezika je razvijen i implentiran (preko **8000** programskih jezika - podatak iz 2006): Fortran, Cobol, Simula, Pascal, Algol, Modula-2, PHP, Visual Basic, Lisp, Prolog, Smalltalk, Java, JavaScript, C, C++, C# , ...
• Učenje i sticanje znanja o fundamentalnim konceptima i karakteristikama programskih jezika

- Niži programski jezici (mašinski zavisni jezici)
  - Mašinski kod je izuzetno težak za citanje, pisanje i razumevanje
  - Nije portabilan
- Viši programski jezici (mašinski nezavisni jezici)
  - Kod viših programskih jezika je razumljiv i lak za citanje i pisanje
  - Skrivanje detalja stvarne mašine



2. Značajni uticaji na razvoj programskih jezika
I. Aplikacioni domeni
II. Metode programiranja
III. Komjuterske arhitekture

Aplikacioni domeni PJ
• Razvijen veliki broj vrlo različitih jezika za različite aplikacione domene.
Svaki aplikacioni domen ima specifične potrebe.

- ***Inžinjerski domen***
- ***Poslovni domen***
- ***Veštačka inteligencija***
- ***Sistemsko programiranje***
- ***Internet i Web***

Poslovni domen
prvi viši programski jezik projektovan za poslovnu obradu podataka bio je COBOL kasnije su razvijeni sistemi za pravljanje bazama podataka
Inžinjerski domen
proste strukture podataka (nizovi i matrice), ali se zato zahteva veliki broj preciznih numeričkih računanja FORTRAN - projektovan za precizna numerička računanja
Veštačka inteligencija
Korišćenje simbola (umesto numeričkih računanja) logički i funkcionalni jezici

Sistemsko programiranje

- koristi se za sistemski softver
- zahteva se efikasnost u izvršavanju, zbog neprekidnog korišćenja ove vrste softvera

- operativni sistem UNIX napisan u programskom jeziku C

Internet i Web

veoma aktuelan domen; ovo distribuirano i heterogeno okruženje uticalo je na razvoj novih jezika koji su posebno pogodni za njega

- Web script jezici (JavaScript, PHP)
- Markup jezici (HTML, XML)

Metode programiranja

Od 1950 do sredine 60-tih god. prošlog veka nepostojanje metoda programiranja

- proste aplikacije
- glavna briga efikasnost koda
- stariji programski jezici: FORTRAN, COBOL

Kraj 60-tih i početak 70-tih god. prošlog veka:

- reducirana cena hardvera, cena razvoja softvera u porastu, efikasnost programera postaje veoma važna
- glavna briga produktivnost u programiranju (bzo i lako pisanje razumljivih programa)
- *Strukturno programiranje* - prva metoda programiranja
- sistematičan pristup razvoju razumljivih i korektnih programa: povećana razumljivost i čitljivost programa, adekvatnije kontrolne strukture,
- razvijen pj **Pascal** za podršku koncepata strukturnog programiranja
- Proceduralno-orjentisano projektovanje programa

Kraj 70-tih: Pomeranje od procedurlno-orjentisanog projektovanja ka projektovanju orjentisanom ka podacima

- Apstrakcija podatka - inkapsulira obradu sa podacima (Apstraktni tip podatka)
- Programski jezik **Simula** za podršku koncepta apstrakcije podatka

Sredina 80-tih: Objektno-orjentisano programiranje

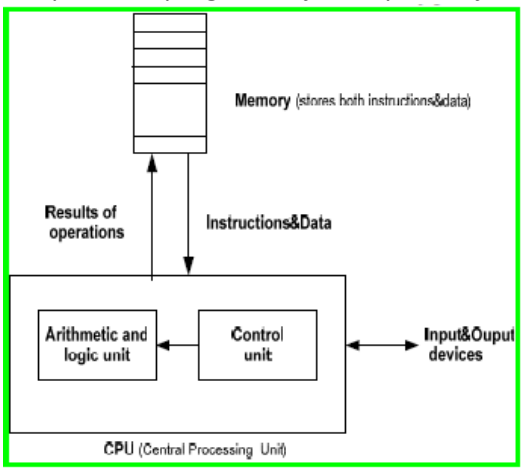
- apstraktni tip podatka + naslenivanje + polimorfizam
- **Smalltalk, C++, Java, C#**

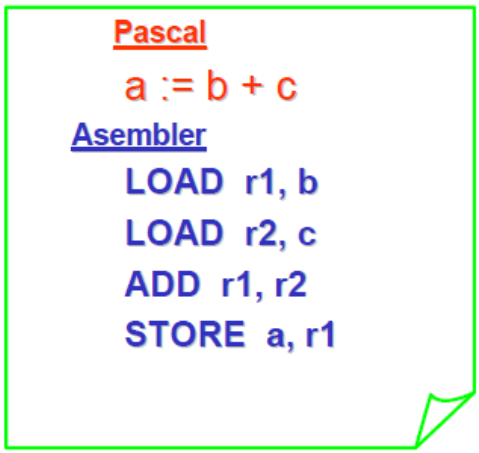
1995 do danas: Web, Distribuirane aplikacije

- Web script jezici (JavaScript, PHP)
- Markup jezici (HTML, XML)

Kompjuterske arhitekture

• Imperativni programski jezici apstrahuju Von Neumann-ovu kompjutersku arhitekturu





Kompjuterske arhitekture

- *Promenljive reprezentuju memoriju* (model memorijskih lokacija)
- *Instrukcija dodeljivanja za promenumemorijskih vrednosti*
- *sekvencijalno izvršavanje instrukcija*

3. Fundamentalni koncepti programskih jezika
I. Apstrakcije u višim PJ
II. Jezičke paradigme
III. Specifikacija i implementacija PJ

Apstrakcije u višim programskim jezicima
i. Bazne apstrakcije
ii. Strukturne apstrakcije
iii. Proceduralne apstrakcije
iv. Apstrakcije podataka

Apstrakcije u programskim jezicima
• ***Bazne apstrakcije***
Apstrahovanje

- memorijskih ćelija,
- interne reperezentacije primirtivnih tipova podataka
- i implementacije operacija nad vrednostima takvih tipova

Apstrakcije u programskim jezicima
• ***Strukturne apstrakcije***

- *Struktuirani tipovi podataka* (apstrahovanje kolekcija menusobno povezanih podataka – niz, zapis, lista)
- *Kontrolne structure* (sekvenca, selekcija, iteracija)

Proceduralne apstrakcije

- Sakrivanje složenog koda sa prostim interfejsom
- proceduralne apstrakcije u višim PJ podržane su potprogramima i proceudrama

Apstrakcije podataka

- sakrivanje interne reprezentacije korisničkih tipova podataka pomoću skupa operacija (javni interfejs) preko kojih se stanja objekata tipa jedino mogu menjati
- koncept se u savremenim višim PJ implementira preko klase

Vrste jezika (Jezičke paradigme)
Razlike izmenu pojedinih vrsta programskih jezika baziraju se na ***modelu***, tj. ***paradigmi*** koju podržavaju

Četiri glavne vrste jezika:
i. ***Imperativni*** (FORTRAN, COBOL, Pascal, C)
ii. ***Objektno-orjentisani*** (Smalltolk, C++, Java, C#)
iii. Funkcionalni (LISP, Scheme, Haskell)
iv. *Logički* (Prolog)
v. **Ostale vrste jezika** (“script” jezici, “markup” jezici, upitni jezici, ...)

Specifikacija i implementacija programskih jezika
i. Specifikacija
_ **sintaksa** forma (struktura) programa
_ **semantika** značenje programa

ii. Implementacija
implementacioni modeli:

- **Kompajleri**
- **Interpreteri**
- **Hibridni implementacioni sistemi**

5. Značaj izučavanja programskih jezika i kriterijumi za njihovu evaluaciju
II. Kriterijumi za evaluaciju programskih jezika

Značaj izučavanja programskih jezika
Značajni razlozi za izučavanje koncepata programskih jezika:

- *Lakše i brže učenje novih jezika*
- *Bolji izbor odgovarajućeg jezika za odreneni problem*
- *Novi načini razmišljanja u rešavanju odrenenog problema*
- *Bolje razumevanje značaja implementacije programskog jezika*

Kriterijumi za ocenu PJ
i. **Čitljivost**
ii. Lakoća učenja i pisanja
iii. Pouzdanost
iv. Cena

Čitljivost programa
Čitljivost programa je mera koliko je lako čitanje i razumevanje programa napisanog u nekom programskom jeziku.
Značajni faktori koji utiču na lako čitanje i razumevanje programa:

- ***Jednostavnost pj***
- ***Ortogonalnost pj***
- ***Kontrolne strukture***
- ***Tipovi i strukture podataka***
- ***Sintaksa jezika***

Jednostavnost programskog jezika veoma utiče na njegovu čitljivost.
Sledeći faktori smanjuju čitljivost:

- veliki broj baznih konponenti jezika (otežano učenje i razumevanje i za autora i za čitaoca programa)
- jezik dozvoljava alternativne načine za realizaciju neke konstrukcije na primer u C++ postoje 4 načina za inkrementiranje celobrojne promenljive: y++; ++y; y+=1; y = y + 1;

Ortogonalnost u programskom jeziku znači da se relativno mali broj baznih koncepata mogu kombinovati pomoću konzistentnog skupa pravila

- Nedostatak ortogonalnosti u pj manifestuje se kao veliki broj specijalnih slučajeva (pravila izuzetaka). Na primer u programskom jeziku C postoji veći broj pravila izuzetaka, na primer funkcija može da vraća zapise(structs) ali ne i nizove (arrays), element zapisa može da bude bilo kog tipa osim void ili zapisa istog tipa, itd.
- Ortogonalnost je povezana sa jednostavnošću jezika. Veća ortogonalnost podrazumeva i mali broj pravila izuzetaka što čini jezik čitljivijim i razumljivijim

Kontrolne strukture

- metoda strukturnog programiranja koja se pojavila 1970. bila je reakcija na lošu čitljivost uzrokovanu neadekvatnim kontrolnim strukturama (na primer **goto** instrukcija) u tadašnjim programskim jezicima.
- danas su kontrolne strukture manje važan faktor za čitljivost jer većina programskih jezika podržava adekvatne kontrolne structure

Tipovi podataka i strukture Postojanje adekvatnih mogućnosti za definisanje tipova i struktura podataka u jeziku povećavaju čitljivost programa
_ na primer neki jezici koji ne podržavaju logički tip (boolean), umesto njega koriste celobrojni tip, tj (0,1) vrednosti umesto logičkih vrednosti (false, true).
krajListe = 1 zančenje nejasno
krajListe = true značenje je veoma jasno u jezicima koji podržavaju logički tip

Sintaksa ili struktura elemenata jezika ima značajan uticaj na čitljivost programa. Prosta i nedvosmislena sintaksa: forma koda jasna.

Lako učenje i pisanje programa u PJ. Pisanje programa je mera koliko se lako programski jezik može koristiti za kreiranje programa u odrenenom aplikacionom domenu.
Faktori koji utiču na lako učenje i pisanje programa:

- ***Jednostavnost i ortogonalnost jezika***
- ***Podrška apstrakcijama***

Lako učenje i pisanje programa u PJ
Podrška apstrakcija
apstrakcija označava mogućnost definisanja i korišćenja komplikovanih struktura ili operacija na način koji dozvoljava skrivanje mnogih detalja
_ primer realizacija binarnog stabla u pj FORTRAN 77 (koji ne podržava dinamičke strukture podataka) i u pj Java:
Java: FORTRAN 77:
public class Node { Integer : Left (100)
Node left; Integer : Value(100)
Integer value; Integer : Value(100)
Node right;
}

Pouzdanost programa
Program je ***pouzdan*** ako se izvršava u skladu sa njegovom specifikacijama pod svim uslovima.
Sledeće karakteristike imaju značajan uticaj na pouzdanost programa u datom programskom jeziku:

- ***Provera tipa (Type Checking)***
- ***Obrada grešaka (Exception handling)***
- ***Ograničavanje alijasa (Aliasing)***
- ***Čitljivost; lakoća učenja i pisanja***

Provera tipa (Type Checking)
_ provera tipa je testiranje grešaka tipa u datom programu, u vreme kompilacije ili za vreme izvršenja
_ prednost se daje otkrivanju grešaka tipa za vreme kompajliranja programa

Obrada grešaka (Exception handling)
_ "Hvatanje" grešaka u vreme izvršavanja programa, preduzimanje korektivnih mera i nastavljanje izvršavanja.
_ jezici C++, Java, C# podržavaju exception handling.

Ograničavanje alijasa (Aliasing)
_ postojanje dve ili više referenci na istu memorijsku lokaciju.
_ Zbog poznate činjenice da je "aliasing" opasna pojava u programskim jezicima i da loše utiče na pouzdanost, ograničava se mogućnost korišćenja alijasa u njima.

Čitljivost i lakoća učenja i pisanja
programi napisani u jeziku koji ne zadovoljava kriterijume čitljivosti i pisanja prgrama, reduciraju pouzdanost.

Cena
Ukupna cena projektovanja i relizacije programa u nekom programskom jeziku je funkcija više njegovih karakteristika:

- ***obuka programera za korišćenje jezika***
- ***vreme pisanja programa u datom jeziku***
- ***kompajliranje programa***
- ***izvršavanja programa***
- ***Implementacioni sistem jezika***
- ***Pouzdanost***
- ***Održavanje***

Obuka programera
Ovo je funkcija jednostavnosti i ortogonalnosti jezika i iskustva programera***.***
Vreme pisanja programa u datom jeziku
Ovo je funkcija lakoće pisanja programa u datom jeziku. Cena obuke programera i pisanja programa može se značajno reducirati pomoću dobrog programerskog okruženja.
Implementacioni sistem jezika
Jezik čiji je implementacioni sistem skup, ima umanjene šanse za šire prihvatanje i korišćenje.

Pouzdanost - Loša pouzdanost softvera izuzetno utiče na povećanje cene.
Održavanje - Održavanja obuhvata korekcije i modifikacije softvera koji se već koristi. Primarni uticaj na cenu održavanja softvera ima čitljivost programa.

5. Programerska okruženja
Programersko okruženje je kolekcija alata koja se koristi u razvoju softvera
_ kolekcija može da sadrži samo tekst-editor, linker i kompajler
_ ili to može da bude kolekcija integrisanih alata, tako da se svakom pristupa preko uniformnog korisničkog interfejsa

Programerska okruženja
• **UNIX** starije prog. okruženje; grafički korisnički interfejs bio je glavni nedostatak.
• **JBuilder** prog. okruženje koje uključuje kolekciju integrisnih alata za razvoj **Java** aplikacija; pristup alatima preko grafičkog korisničkog interfejsa
• **Microsoft Visual Studio .NET** novije softversko razvojno okruženje. Koristi se za razvoj softvera u jednom od sledećih .NET jezika: **C#, Visual**
BASIC.NET, Jscript, J#, managed C++

2. Specifikacija I implementacija programskih jezika

1. Sintaksa i semantika
Specifikacija programskih jezika obuhvata
• **Sintaksu** i
• **Semantiku**

Sintaksa i semantika
• ***Sintaksa -*** forma (struktura) programa.
• ***Semantika -*** značenje sintaksnih konstrukcija jezika

Primer:
Sintaksna forma Java while instrukcije je
while (<boolean_expr>) <statement>

Semantika ove instrukcije je da kad god je tekuća vrednost logičkog izraza (<boolean_expr>) **true,** izvršava se telo **while** instrukcije (<statement>). Proces evaluacije logičkog izraza i izvršavanja tela while petlje se ponavlja sve dok je računata vrednost logičkog izraza true.

Semantika
Semantiku ili značenje jezika je teško precizno i formalno opisati, jer se može opisati na više različitih načina
• Neformalni pristup
• Pristupi za fomalni opis semantike
- Operaciona semantika
- Translaciona semantika
- Aksiomatska definicija jezika
- Denotaciona semantika

Sintaksa
Specifikacija:
• Leksička specifikacija
• Sintaksna specifikacija

2. **Leksička specifikacija**
Opis sintaksnih atomskih jedinica u jeziku daje se preko *leksičke specifikacije* i ova specifikacija razdvaja se od sintaksne.
Token je najmanja sintaksna jedinica u programu koja ima značenje U prirodnom pisanom jeziku to je reč ili interpunkcijski znak.

Leksička specifikacija
Tokeni u PJ su:
- Identifikatori
- Ključne reči
- Operatori
- Numeričke konstante
- Specijalni znaci (zagrade, separatori, ..)

Primer prevonenja teksta programa u niz tokena

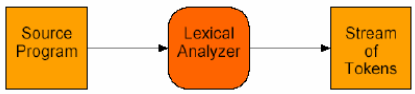
```
if (x  > 20)
   break;
```

if	Ključna reč
(	Otvorena zagrada
x	Identifikator
>	Rel. operator
20	Num. konstanta
)	Zatvorena zagrada
break	Ključna reč
;	Separator

Leksička specifikacija
Leksička specifikacija zadaje se u formalnoj notaciji preko **regularnih izraza**
- Regularni izraz za celobrojnu konstantu:
[0-9]+
Celi brojevi koji ce biti prepoznati: 123 7 1072
Greška -100 A1
- Regularni izraz za ključne reči
int|break|if|then|...

Leksička analiza
• ***Leksička analiza*** je faza u procesu translacije programa

• ***Leksički analizator (skener)*** *prevodi tekst programa u niz tokena (tj.* prepoznaje tokene u ulaznom tekstu, na osnovu zadatih leksičkih specifikacija – **regularnih izraza**)



lex, flex, JLex – softverski alati za generisanje leksičkog analizatora (skenera)

3. **Sintaksa specifikacija PJ**
• sintaksa jezika se *opisuje* pomoću **gramatike** koja definiše pravila za pisanje korektnih programa u nekom programskom jeziku
• **konteksno-slobodne gramatike** (CFG-Contex Free Grammars)
- Koriste se za formalni opis sintakse PJ
• CFG obično se zapisuju u **BNF***(Backus-Naur form)* notaciji

BNF notacija
BNF(Backus-Naur form) notacija
- Formalna notacija za opis sintakse jezika
- BNF je meta jezik za programske jezike
- BNF prvi put korišćenja za opis sintakse programskogh jezika ALGOL 60

BNF notacija
BNF koristi apstrakcije za opis sintaksnih struktura
_ apstraktna definicija instrukcija dodeljivanja:

```
< assign> → <id> = <expression>
<id> → a |b | c
<expression> → <id>
                    | <id> + <id>
                    | <id> - <id>
```

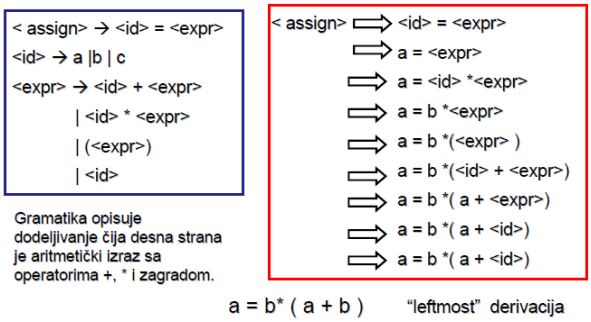
< assign> Apstrakcija koja se definiše (neterminal)
_ **<id> = <expression>** definicija pravila, sadrži reference na druge apstrakcije (neterminale)
_ instrukcija dodeljivanja opisana pomocu datih pravila:
a = a + b

Sintaksa u BNF notaciji se specificira korišćenjem:
- Skupa ***terminala*** (tokena)
- Skupa ***neterminala***
- Skupa ***produkcionih pravila***
- ***Startni symbol***

• Produkciono pravilo u BNF notaciji:
N ::= a
N - neterminal
a - sekvenca terminala i neterminala

Derivacije
sintaksne konstrukcije se *izvode* preko primene niza pravila derivacija počinje od specijalnog neterminala tzv. *startnog simbol.*

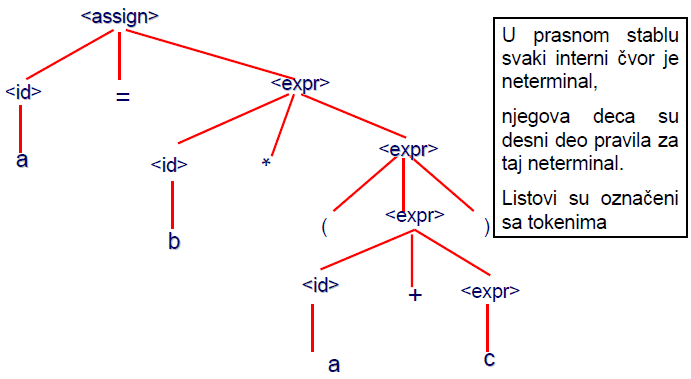
Ilustracija derivacije



Parsna stabla

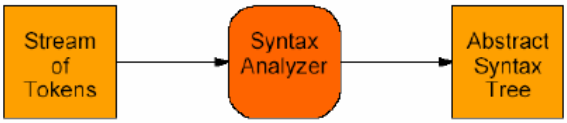
• Atraktivna karakteristika gramatika je da opisuju *hijerarhijsku sintaksnu strukturu* jezika.
• Takve hijerarhijske strukture nazivaju se ***parsnimstablima***
• Na sledećoj strani derivacija instrukcije dodeljivanja a = b * (a + b) reprezentovana je preko parsnog stabla.

Primer parsnog stabla za a = b * (a + b)



Sintaksna analiza

• **Sintaksna analiza** je faza u procesu translacije u kojoj se provera da li je program napisan skaldu sa gramatikom jezika
• **Parser (sintaksni analizator)** je program koji, na osnovu date gramatike, proverava da li je data sekvenca tokena i neterminala ispravna ili ne, i generiše parsno stablo koje reprezentuje sintaksnu strukturu programa.

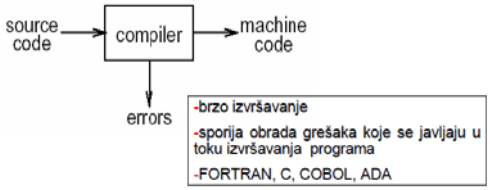


yacc, bison- softverski alat za generisanje parsera

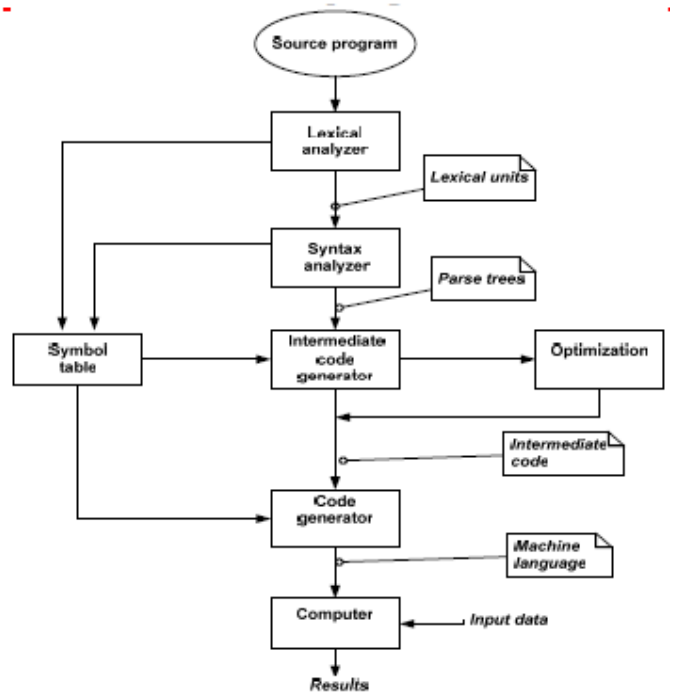
4. Implementacione metode

• **Kompajleri**
• **Interpreteri**
• **Hibridni implementacioni sistemi**

Implementacione metode: Kompajleri - programi koji prevode programe napisane u izvornom kodu u izvršni kod.



Implementacione metode: Struktura kompajlera



Faze procesa kompilacije

• **Leksicka analiza** – u ovoj fazi koristi se *leksički analizator (skener)* koji prevodi tekst programa u niz tokena *(tj.* prepoznaje tokene u ulaznom tekstu, na osnovu zadatih leksičkih specifikacija)

• **Sintaksna analiza** – u ovoj fazi *parser* kao ulaz uzima niz tokena i na osnovu date gramatike, proverava da li je data sekvenca tokena i neterminala ispravna ili ne, i generiše parsno stablo.

U *tabelu simbola* se beleže korisne informacije o promenljivima, tipovima podataka i upotrebi simbola u programu.

Generisanje menukoda – u ovoj fazi vrši se generisanje menukoda (generše se program u jeziku koje je sličan asembleru)

Optimizacija menukoda – u ovoj fazi, (koja je opciona) vrši se optimizacija menukoda u smislu brzine izvršavanja i/ili smanjenja veličine programa

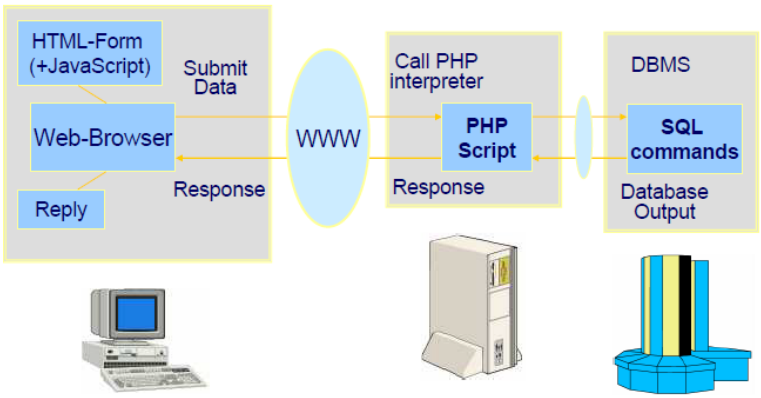
Generisanje koda – generator koda prevodi menukod u izvršni kod (mašinski jezik)

Implementacione metode: Interpreteri

Interpreter: program napisan u izvornom kodu interpretira se pomoću drugog programa – interpretera (softverska simulacija mašine)

- brz razvoj programa i laka obrada grešaka koje se javljaju u toku izvršavanja programa
- sporo izvršavanje
- LISP, APL, SNOBOL,

-Web script jezici: JavaScript, PHP

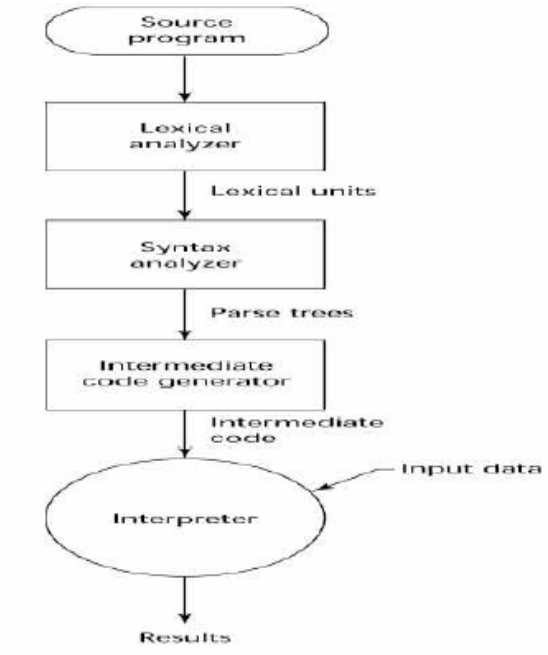


Implementacione metode: hibridni implementacioni sistemi

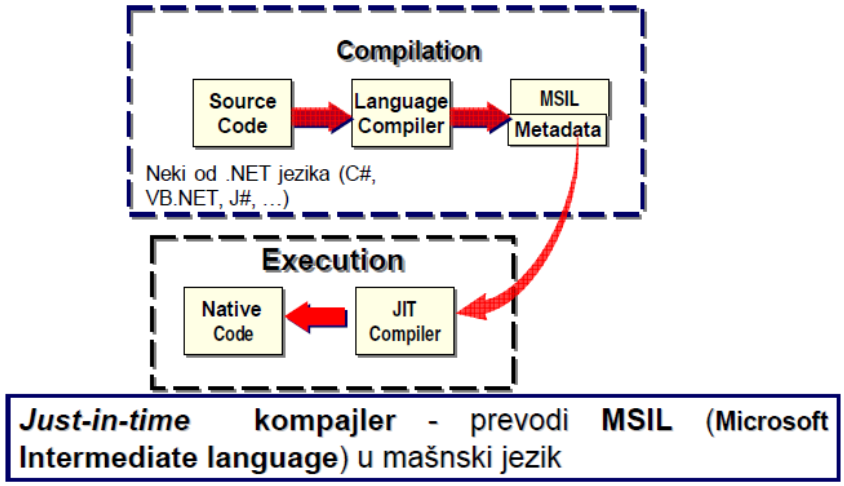
• **Hibridni:** prevode programe napisane u izvornom kodu u menukod (intermediate code) koji omogućava laku interpretaciju

• **Java byte code**

menukod; interpretacija na JVM (Java Virtual Machine)

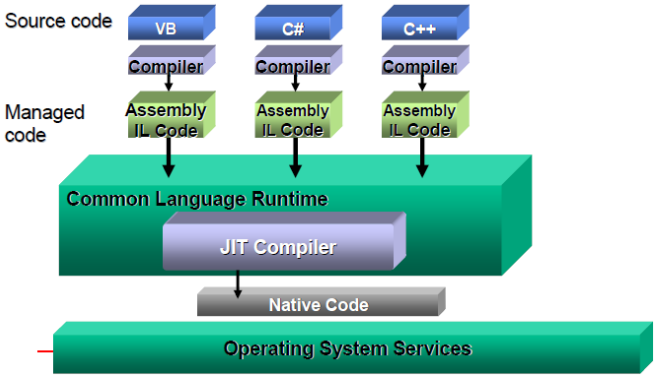


Implementacione metode: Just-in-time sistemi



Just-in-time **kompajler** - prevodi **MSIL** (**Microsoft Intermediate language**) u mašnski jezik

Implementacione metode: .NET izvršni model



3. Jezičke paradige

Razlike izmenu pojedinih vrsta programskih jezika baziraju se na ***modelu***, tj. ***paradigmi*** koju podržavaju

Imperativni jezici
• Imperativni (proceduralni) jezici pogodni za predstavljanje algoritama kao serijske kompozicije koraka
• Imperativni programski jezici apstrahuju Von Neumann-ovu kompjutersku arhitekturu

Glavne karakteristike imperativnih jezika:
• *Promenljive:* model memorijskih lokacija
• *Ključna operacija: dodeljivanje* – promena memorijskih vrednosti
• *Program je niz instukcija:* sekvencijalno izvršavanje instrukcija

• Podržavaju:
- bazne apstrakcije,
- strukturne apstrakcije,
- proceduralne apstrakcije
• FORTRAN, Algol, COBOL, Pascal, C

• Bočni efekti (side-effects) funkcija

```
int i = 1
main () {
int y = 5;
printf("%d/n", f(y)+g(y));
printf("%d/n", g(y)+f(y));
}
int f(int x) {
i= i*2; return i*2 }
int g(int x) {
return i*x; }
```

f(y)+g(y) razlicito od *g(y)+f(y)*
bočni efekat funkcije *f* na globalnoj promenljivoj *i*

Funkcionalni jezici
Baziraju se na matematičkim funkcijama, ali su implementirani u von Neumann-ovoj arhitekturi

Karakteristike:
• Osnovni koncept je *funkcija*
• Nema eksplicitne naredbe dodeljivanja posledica: nema bočnog efekta (side effects), na primer uvek važi $f(a)+f(b) = f(b)+f(a)$
• Rad sa memorijom implicitan
– nema eksplicitne deklaracije promenljivih kojima se dodeljuje memorijski proctor

Karakteristike:
• Redosled evaluacije funkcija kontrolisan je rekurzijom
• Jezik je netipiziran
– nema eksplicitnih tipova, provera slaganja tipova vrši se tek u vreme izvršavanja programa

Funkcionalna kompozicija
fukcionalna forma koja kao parametre uzima funkcije;
funkcionalna kompozicija se zapisuje kao izraz koji koristi operator o: h = f o g

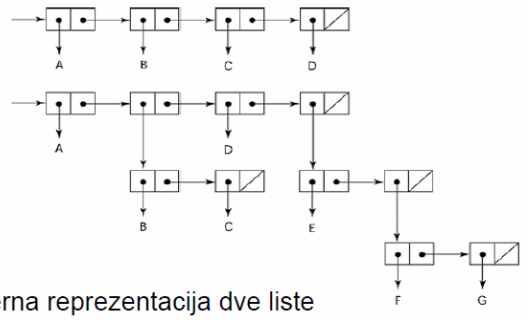
primer:
Ako je
f(x) = x + 2 g(x) = 3*x
onda se h definiše kao
h(x) = f(g(x)), ili h(x) = (3*x) + 2

Program: se predstavlja jednom funkcijom
Ključna operacija: primena funkcije

Obezbenuju:
- skup primitivnih funkcija
- funkcionalne forme za konstruisanje složenih funkcija
- strukture za reprezentovanje podataka

Prvi predstavnik funkcionalnog programiranja je programski jezik **LISP** (LISt Processing – obrada lista)
Do danas razvijen je veći broj dijalekata LISP-a i drugih funkc. jezika: COMMON LISP, Scheme, ML, Haskell

Funkcionalni jezici - LISP
Sadrži samo dva tipa: *atome* i *liste*
- Prosta lista: (A B C D)
- Ugnježdena lista koja sadrži 4 elementa:
(A (B C) D (E (F G)))



Interna reprezentacija dve liste

Primer LISP programa
defun factorial (n)
(cond ((equal n 1) 1)
(* n (factorial (- n 1))))

Logički jezici
• Pripadaju klasi deklarativnih (neproceduralnih) programskih jezika
• Logički programski jezici zasnovani su na *predikatima* (logičkim izrazima)

• Dve osnovne karakteristike logičkog programiranja su
- upotreba činjenica i pravila za predstavljanje informacija i
- upotreba dedukcije (izvodjenja) za odgovaranje na pitanja
• Program: skup *činjenica* i *pravila*

Činjenica - tvrnenje o nekoj osobini objekta ili vezi izmenu dva ili više objekta
Činjenice se u programu predstavljaju predikatima oblika: P(X1, . . . , X*n*)

• *Primer:*
Činjenica otac(Petar, Milan) ima značenje da su Petar i Milan u relaciji "otac", tj.Petar je Milanov otac"

Pravilo omogućava izvonenje o postajanju osobine ili veze, koje se zasniva na preduslovima
pravilo se zapisuje u obliku
P(X1, . . . , X*n*) :– F(X1, . . . , X*n*, Y1, . . . , Y*m*)
Značenje pravila je da je predikat P(X1, . . . , X*n*) tačan ako je logička formula F(X1, . . . , X*n*, Y1, . . . , Y*m*) tačna.

Primer:
Pravilo roditelj(X, Y) :- otac(X,Y) ima značenje da su X i Y u relaciji "roditelj" ako su X i Y u relaciji "otac".
Dakle, ovo pravilo kaže "Osoba X je roditelj osobe Y AKO je X otac osobe Y"

Primer Prolog programa:
otac (petar,milan).
majka(vesna,milan).
roditelj(M,D):-majka(M,D).
roditelj(O,D):-otac(O,D).
Izvršavanje programa zahteva se zadavanjem pitanja u formi predikata:
?- otac(petar,milan) ?- otac(petar,X)
yes milan
• Odgovor na postavljeno pitanje dobija se mehanizmom

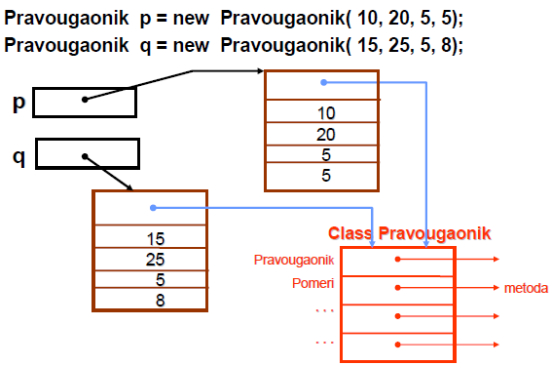
Unifikacije
Ključna operacija: *unifikacija* (izjednačavanje)
Unifikacijom se izjednačavaju vrednosti promenljivih odnosno konstanti na istoj poziciji istog predikata.
?- otac(petar,X)
milan
Daje odgovor "milan" zato što postoji činjenica (predikat)
otac(petar, milan), pa se X "unifikuje" sa milan

Logički jezici:
Prolog (PROgramming in LOGic), Datalog

Objektno-orjentisani jezici
javili su se kao odgovor na softversku krizu, kao način da se, grupisanjem objekata sa sličnim osobinama u klase, (a ne uz aplikacije), dobije programski kod koji se može ponovo upotrebiti
• ekstenzija imperativne paradigm

Objektno-orjentisanu paradigmu karakterišu sledeće osobine:
• osnovni koncept: *objekat*
• Objekti se grupišu u *klase*
• Objekat je izvršna *instanca* klase

```
class Pravougaonik {
private int lokacijaX, lokacijaY, sirina, visina; // atritbuti
public Pravougaonik (int x, int y, int s, int v) // konstruktor
{ lokacijaX = x; lokacijaY = y; sirina = s; visina = h; }
public void Pomeri( int deltaX, int deltaY) // metoda
{ lokacijaX = lokacijaX + deltaX;
lokacijaY = lokacijaY + deltaY;
}
}
```



• Program*:* objekti izmenu sebe komuniciraju *porukama*
porukom se od objekta koji ih prima zahteva da izvrši neku od operacija definisanih u klasi kojoj pripada.
• Ključna operacija: *prenošenje poruka*

• Skrivanje informacija – objektu se može pristupiti samo preko poruka tj. metoda koje su definisane za klasu kojoj objekat pripada.
• *Učaurenje podataka i operacija* u celinu(objekat)
• Naslenivanje
Nova klasa može se definisati kao *izvedena* klasa (*podklasa)* već definisane klase (*osnovne* klase ili *nadklase*).
• Polimorfizam

Zaključak: O-O paradigma podržava:
- apstraktne tipove podataka,
- naslenivanje,
- polimorfizam

O-O jezici: Simula, Smalltolk, C++, Java, C#

Ostale vrste jezika
• "Script" jezici
  - Najraniji: komandni jezici (shell)
  - Web script jezici: JavaScript, PHP
• "Markup" jezici
  - HTML (*HyperText Markup Language*)
  - XML (eX*tensible Markup Language*)
• Upitni jezici
• Jezici za specifikaciju i modelovanje
  - UML *(Unified Modeling Language*)

Bazne apstrakcije

- ***Apstrakcija*** je reprezentacija entiteta koja uključuje samo atribute značajne u određenom kontekstu
  - Koncept apstrakcije je fundamentalan u programiranju i razvoju softvera
  - Intelektualni alat koji koristimo u savlađivanju složenosti (realni sistema i softverski sistema)

Apstrakcije u programskim jezicima
Apstakcije koje podržavaju programski jezici mogu se klasifikovati u sledeće apstraktne nivoe:

I. Bazne apstrakcije
II. Strukturne apstrakcije
III. Proceduralne apstrakcije
IV. Apstrakcije podataka

- ***Bazne apstrakcije*** apstrahovanje
  - memorijskih ćelija (*promenljive*)
  - interne reperezentacije primitivnih tipova podataka i implementacije operacija nad vrednostima takvih tipova (*deklaracija promenljive*)
  - proces računanja i memorisanje sračunate vrednosti (*instrukcija dodeljivanja*)

- ***Strukturne apstrakcije***
  - *Strukture podataka* metod za apstrahovanje kolekcije međusobno povezanih podataka.
    - *pr. strukture u pj C:* struct osoba {

```
        char* ime;
        char*adresa;
        int starost;
    } os;
```

- *strukture podataka* u većini programskih jezika su: niz, zapis, lista
- *Struktuirani tip podatka:*

```
typedef struct {
        char* ime;
        char*adresa;
    int starost;
    }TOsoba;
```

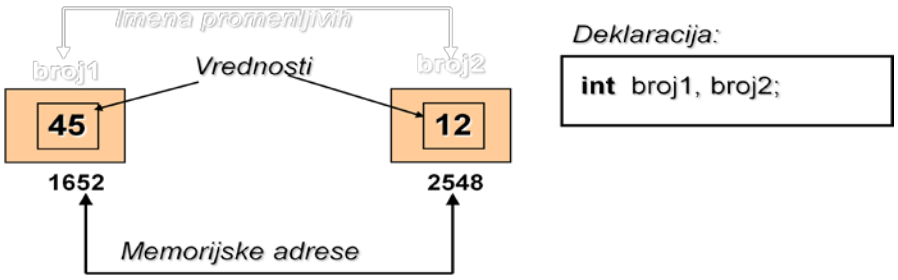
  - *Kontrolne strukture:* sekvenca, selekcija, iteracija

- ***Proceduralne apstrakcije***
  - Sakrivanje složenog koda sa prostim interfejsom
  - proceduralne apstrakcije u višim PJ podržane su potprogramima i proceudrama

- ***Apstrakcije podataka***
- sakrivanje interne reprezentacije korisničkih tipova podataka pomoću skupa operacija (javni interfejs) preko kojih se stanja objekata tipa jedino mogu menjati
- koncept se u savremenim višim PJ implementira preko klase

Uvod u bazne apstrakcije

- Imperativni jezici apstrahuju von Neumann arhitekturu.
- Jezičke apstrakcije za memorijske ćelije su promenljive



Memorija i promenljive

Promenljive

- ***Promenljiva*** je apstrakcija memorijske ćelije (memorijskih ćelija)
- Fizička & apstraktna memomirjska ćelija
  - vrednost floating-point tipa zauzima 4by
  - na jezičkom nivou, to je jedna apstraktna memorijska ćelija

Atributi promenljive

i. Identifikator
ii. Adresa
iii. Vrednost
iv. Tip
v. Životni vek (lifetime)
vi. Doseg (scope)

Identifikatori

- korisnički definisana imena

- Identifikator se koristi za identifikovanje
  nekog programskog eniteta (promenljive, tipovi, formalni parametari, potprogrami, klase, metode, itd.)
- Identifikatori u većini programskih jezika imaju isti oblik: *string koji sadrži slova, cifre i konektore*
- Neke razlike u formi identifikatora
  - Konektori
  - Dužina
  - Case Sensitivity
  - Rezervisane i ključne reči

- **Konektori**
  - "underscore" (_) karakter se kao konektor u imenu promenljive široko koristio tokom 1970 i 1980; (pr. **max_ceo_broj**), danas je manje popularan
  - Danas je zamenjen sa tzv. "Camel" notacija (pr. **maxCeoBro**j)
- **Dužina (length)**
  - FORTRAN I: maximum 6
  - COBOL: max 30 karaktera
  - FORTRAN 90 i ANSI C: max 31
  - Ada, Java, C#: nema ograničenja, ali svi karakteri su značajni

- **Case Sensitivity**
  - U nekim programskim jezicima u identifikatoru se razlikuju mala i velika slova, tako da su identifikatori u tim jezicima *case sensitive*
  - Imena promenljivih **MaxBroj**, **maxbroj** i **MAXBROJ** u jezicima koji su "case sensitivi" su različita
  - Identifikatori u C, C++, Java i C# su case sensitivity
  - Nedostak: loša čitljivost, imena koja izgledaju vrlo slično zapravo su različita, jer identifikuju različite programske enititete

- **Rezervisana reč (reserved words)**
  specijalna reč koja se u programu *ne može* koristiti kao identifikator
  - **float** u C-u koristi se za deklaraciju promenljive, ali ne i kao ime promenljive
- **Ključna reč (keyword)**
  reč koja je specijalna samo u određenom kontekstu
  - *Primer:* FORTRAN

  REAL X deklaracija
  REAL = 7 dodeljivanje
  - Nedostatak: loša čitljivost

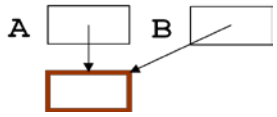
Sve promenljive *nemaju ime* (*anonymous*)
Primer:
dinamičke promenljive se referenciraju preko pointera ili referentnih promenljivih

Adresa promenljive
- memorijska **adresa** sa kojom se promenljiva povezuje
- Isto ime promenljiva može da bude povezano sa *različitim adresama*, ako se nalazi na različitim mestima u programu
  - pr. lokalna promenljiva X deklarisana u dva različita potprograma

– Isto ime promenljive može da ima *različite adrese* u *različitim vremenima izvršenja* programa
  – pr. ako je promenljiva Y deklarisana kao lokalna u nekom potprogramu, onda u svakom novom pozivu potprograma promenljiva može da se alocira na različitim memorijskim adresama)

– ***Alijasi*** *(aliases):*
dve ili više promenljivih koje pristupaju istoj memorijskoj lokaciji

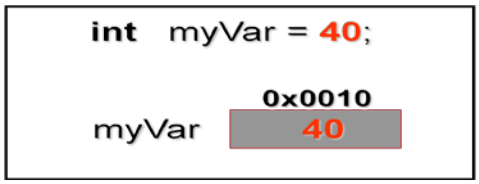
– *Kako se kreiraju alijasi?* Jedan od načina je preko pointera i referentnih promenljivih.



- loša čitljivost
- Problem *Dangling reference*

Vrednost promenljive

- **sadržaj** memorijske lokacije sa kojom je promenljiva povezana



l-value i r-value **promenljive**

Promenljiva ima *dve vrednosti*:

- ***l-value*** adresa promenljive (l-left)
- ***r-value*** vrednost promenljive (r-right)

Primer:

X = Y;

adresa (X) vrednost (Y)
l-value (X) *r-value* (Y)

l-value je adresa od X, a *r-value* je vrednost od y

Poniter je promenljiva čija vrednost (*r-value*) je adresa (*l-value)* druge promenljive

```
int  x = 5;
int*p;
p = &x;
```

Tip promenljive

- definiše *skup vrednosti* i *skup operacija* za manipulisanje vrednostima datog tipa.
- Tip određuje veličinu memorije koja će biti alocirana

C# tip	veličina mem.	dozvoljene vrednosti
int	4 By	[-2147483648, 2147483647]
Aritmetičke operacije		

- Jezici obično uključuju *predefinisane tipove* (boolean, integer, float, character *)*
- Jezici omogućavaju programeru da definiše nove tipove - *korisnički definisani tipovi*

Koncept povezivanja (Binding)

- ***Povezivanje*** *(binding)*

povezivanje atributa sa programskim entitetom (na primer, atributi identifikator, tip, vrednost povezuju se sa promenljivom kao entitetom)

- Postoje raličite vrste povezivanja u programskim jezicima i kao i različita vremena povezivanja
- ***Vreme povezivanja*** *(Binding Time)*
  vreme uspostavljanja (kreiranja) povezivanja

Moguća vremena povezivanja:

- Vreme definisanja jezika
- Vreme implementacije jezika
- Vreme pisanja programa
- Vreme kompajliranja
- Link time
- Load time
- Vreme izvršenja

- **Vreme definisanja jezika**
  - povezivanje operatora-simbola sa odgovarajućom operacijom (def. značenje operatora)
- **Vreme implementacije jezika**
  - primitivnih tipova se povezuju sa njihovom internom reprezentacijom
- **Vreme kompajliranja**
  - povezivanje promenljive sa tipom (u imperativnim i objektno-orjentisanim jezicima)
  - određivanje mehanizama alociranja memorije za promenljivu
- **Link time**
  povezivanje poziva bibliotečkih funkcija sa njihovim izvršnim kodom
- **Vreme punjenja programa** (Load time)
  povezivanje promenljive (globalne promenljive, C **static** promenljive) sa memoriskom lokacijom
- **Vreme izvršenja** (Run time)
  povezivanje dinamičkih promenljivih sa memorijskimm lokacijama

Vreme i način na koji se promenljiva povezuje sa svojim atributima
je ključ za ponašanje/implementaciju jezika.

- **Statičko** (Static binding, Early binding times)

povezivanje dogodilo pre izvršenja i ostaje nepromenjeno do kraja izvršavanja programa obično veća efikasnost

- **Dinamičko** *(Dynamic binding, Late binding times)*

povezivanje se dogodilo u vreme izvršenja i može se menjati za vreme izvršenja programa
obično veća fleksibilnost

Povezivanje tipa sa promenljivom

- **Statičko** i **dinamičko** povezivanje
- Vrsta povezivanja tipa utiče na implementciju jezika:
  statičko → kompilacija
  dinamičko → interpretacija

Statičko povezivanje

- Dešava se u vreme kompajliranja
- Tip promenljive se specificira pomoću **eksplicitne/implicitne** deklaracije
- ***Eksplicitna deklaracija*** je programski izraz koji se koristi za deklaraciju tipa promenljive u većini programskih jezika

```
x : integer;        Pascal
int  x;             C++/Java/C#
```

- ***Implicitna deklaracija*** je "default" mehanizam za specifikaciju tipa promenljive

- Inicijalno projektovana u FORTRAN-u. Imena promenljivih koja počinju slovima I-N su celobrojnog tipa, inače su realnog tipa
- implicitna deklaracija karakteristika starijih programskih jezika FORTRAN, PL/I, BASIC

Dinamičko povezivanje

- Dešava se u vreme izvršenja programa
- promenljiva se povezuje sa tipom dodeljivanjem vrednosti tipa u trenutku izvršavanja instrukcije dodeljivanja. Tip promenljive se može menjati u toku izvršenja programa, dodeljivanjem promenljivoj vrednosti drugog tipa.
- *Primer:* JavaScript

- **list** = [2 4 6 8]; lista dužine 4
- **list** = 17.3; lista je skalrna promenljiva

JavaScript, PHP

Dinamičko povezivanje tipa sa promenljivom

- Prednost: fleksibilnost
- Nedostaci:
  - Visoka cena
    - dinamička provera tipa
    - za promenljivu koja memoriše vrednosti različitih tipova u različitim vremenima izvršavanja programa zahtevaju se različite veličine memorije
    - Dodatne informacije o tekućem tipu promenljive
    - vreme interpretacije
  - Loša detekcija grešaka tipa

  Nekorektni tipovi na desnoj strani instrukcije dodeljivanja se ne detektuju kao greška, već se tip promenljive na levoj strani dodeljivanja prosto menja u nekorektni tip

Provera tipa (Type Checking)

- **Provera tipa** je aktivnost koja obezbeđuje primenu operatora na operande kompatibilnih tipova
  - Operandi: promenljive, izrazi, parametri
  - Operatori: dodeljivanje, relacioni operatori, funkcije
- ***Kompatibilan tip*** je onaj koji je ili
  - legalan za operator ili
  - se konvertuje (*implicitno* ili *eksplicitno*) u legalan tip

- *Eksplicitna konverzija* između različitih tipova mora biti specificirana
  - *Primer u C#:*

  Eksplicitna konverzija **long** u **int**:

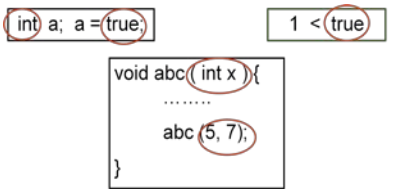
  int intValue = **(int)** longValue;
- *Implicitna konverzija* između različitih tipova određena je definicijom jezika
  - *Primer u C#:*

  Implicitna (automatska) konverzija **int** u **long**:

```
int     intValue = 123;
long    longValue = intValue;
```

- ***Greška tipa*** (*type error*): primena operatora na operand nekompatibilnog tipa

Primeri grešaka tipa:



- Statička
- Dinamička
- Sinteza prethodne dve tehnike

Statička provera tipa

- Tipovi svih promenljivih poznati su u vreme kompajliranja (uslov: statička povezanost tipa sa promenljivom)
  - sve greške tipa detektuju se u vreme kompajliranja
  - efikasnost
  - u većini modernih programskih jezika akcenat je na statičkoj proveri tipa
- Statička provera tipa je otežana u slučaju kada jezik dozvoljava da se u jednoj promenljivoj memorišu vrednosti različitih tipova u različitim vremenima izvršavanja

Primeri:

union type u C i C++, **variant records** u Pascal-u

Dinamička provera tipa

Tipovi promenljivih poznati su samo u vreme izvršavanja programa

- provera tipa se vrši u vreme izvršenja (run-time) pre svake primene operacije na vrednost promenljive
- dodatna memorija za informacije o tipu
- Neefikasnost
- fleksibilnost

script jezici: JavaScript, Perl, PHP

funkcionalni i logički jezici

Definicija Lisp funkcije za sabiranje dva broja: (defun f (a,b) (+ a b))

- Nelegalan poziv

  (f nil nil)

  ne otkriva se u vreme kompajliranja
- Greska se otkriva tek u vreme izvršavanja – dinamička provera tipa

Sinteza statičke i dinamičke provere tipa

- Objektno-orjentisani programski jezici pored statičke provere tipa, (ova tehnika preovladava), zahtevaju i dinamičku proveru tipa (polimorfizam)

C++, Java, C#

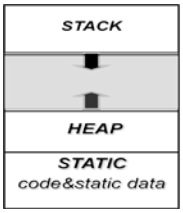
Životni vek promenljive i memorijsko povezivanje

- ***Životni vek promenljive***
  vreme vezivanja promenljive za određenu memorijsku lokaciju
- ***Vreme vezivanja*** je vreme između alokacije i dealokacije memorijske lokacije

Kategorije promenljivih (prema životnom veku)

- **Statička**
- **Stack-dinamička**
- **Eksplicitna heap-dinamička**
- **Implicitna heap-dinamička**

Mehanizmi alokacije memorije

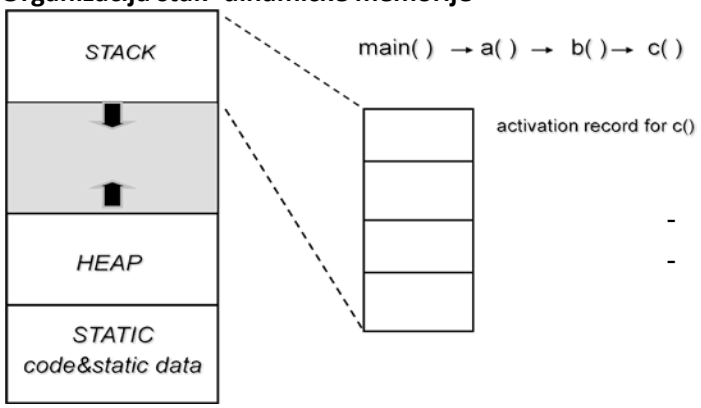


Stack-zasnovana alokacija

Heap-zasnovana alokacija

Statička alokacija

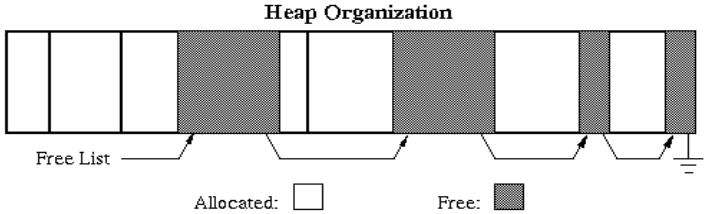
Organizacija stak- dinamičke memorije



- LIFO (last-in, first-out) alokacija/dealokacija
- alokacija se događa uvek kada se pozove procedura

Organizacija dinamičke heap memorije

Heap je memorijski segment u kojem se memorijski blokovi mogu alocirati i dealocirati u proizvoljnim vremenima



Statička promenljiva

- Povezivanje promenljive sa *memorijskom lokacijom* pre početka izvršenja i promenljiva ostaje vezana za istu memorijsku lokaciju do kraja izvršenja programa.
- Statičko povezivanje promenljive i sa *tipom*

- Statička alokacija memorije (promenljiva se jednom alocira)
- Životni vek prom. = vreme izvršenja programa
- Vrednost promenljive je perzistentna od poziva do poziva funkcije
- Statičke promenljive u PJ:
  - U starijim programskim jezicima (npr. FORTRAN) sve promenljive se statički alociraju
  - **static** lokalne promenljive u funkcijama **C** prog. jezika

Primer: **static** promenljiva deklarisana u C funkciji

```
int brojac ( )  {
    static int  k = 0;
    return  ++ k;
}
int main ( )  {
    printf (" brojac = %d  /n", brojac ( ) );
    printf (" brojac = %d /n", brojac ( ) );
    return 0;
}
```

Izlaz: brojac = 1
brojac = 2

- Prednost:
  - efikasnost (direktno adresiranje)
- Nedostaci:
  - nefleksibilnost (nije podžana rekurzija)
  - memorija nije deljiva između promenljivih

Stack-dinamička promenljiva

- Po pozivu funkcije (metode) i prepoznavanju deklaracija promenljivih, kreira se memorijsko povezivanje (u vreme izvršenja)
    - Povezivanje promenljive sa *tipom* je statičko
- Meorija za promenljivu alocira se na *run-time stack-u*
    - LIFO (last-in, first-out) alokacija/dealokacija
    - alokacija se događa uvek kada se pozove funkcija (metoda)
- životni vek stack-dinamičke promenljive = vreme izvršenja funkcije (metode)
    - vrednost promenljive nije perzistentna; reinicalizuje se pri svakom pozivu funkcije (metode)

```
int brojac ( )  {
    int  k = 0;
    return  ++ k;
}
int main ( )  {
    printf (" brojac = %d  /n", brojac ( ) );
    printf (" brojac = %d /n", brojac ( ) );
    return 0;
}
```

Izlaz: brojac = 1
brojac = 1

- Prednosti:
  Podržana rekurzija
  Ponovno korišćenje memorije
- Stack-dinamičke promenljive u PJ:
    - lokalne promenljive deklarisane u metodama u C++, Java i C#
    - lokalne promenljive deklarisane u C i C++ funkcijama

Eksplicitna heap-dinamička promenljiva

- Alokacija i dealokacija memorije vrši se za vreme izvrašavanja programa na osnovu eksplicitnih direktiva, koje specificira programer
    - Statičko povezivanje sa tipom, dinamičko sa mem. lokacijom
- Dinamičke promenljive alociraju se i dealociraju na *heap-u* i referenciraju se preko *pointera* ili *referentnih promenljivih*

Koriste se za implementaciju dinamičkih struktura podataka

- liste
- stabla
- grafovi

Primer heap-dinamičke promenljive u C++:

```
int *intcvor;
intcvor = new int;
delete intcvor;
```

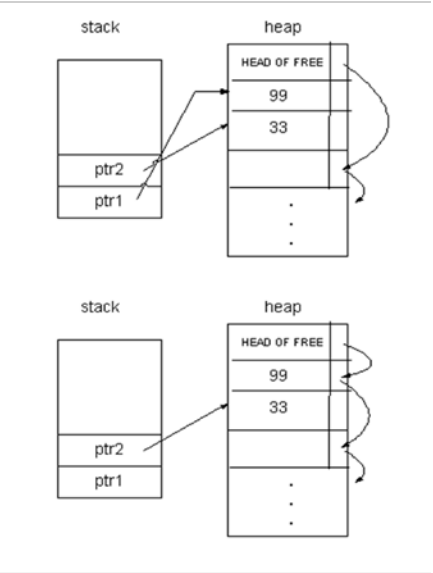
- Eksplicitna dealokacija heap-dinamičke promenljive (operator **delete**)

/* kreiranje pointera */

/*Kreira se heap-dinamička promenljiva tipa **int**; na promenljivu se referencira preko pointera intcvor */
/* Dealokacija promenljive na koju pokazuje intcvor */

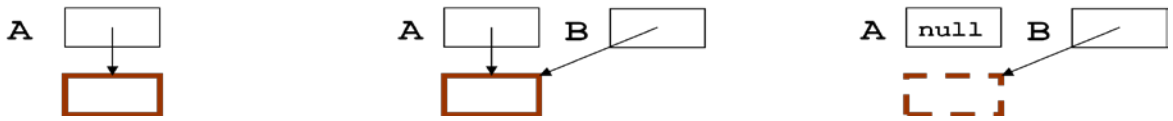
```
int * ptr1 = new int;
*ptr1 = 99;

int * ptr2 = new int;
*ptr2 = 33;
```




```
delete ptr1;
```

- Problem ***Dangling reference*** (dangling pointer)
  pointer koji sadrži adresu heap-dinamičke promenljive koja je dealocirana



- Problem **"Lost"** heap-dinamička promenljiva - alocirana promenljiva kojoj se ne može pristupiti (tzv. **garbage** promenljiva)
  - Ovaj problem dele jezici u kojima se zahteva eksplicitna dealokacija heap-dinamičke promenljive

- Java
  - Objekti su heap-dinamički i pristupa im se preko referentnih promenljivih
  - Implicitna dealokacija (garbage collection)
  - C#
  - Referentne promenljive, implicitna dealokacija
  - Pointeri (interoperabilnost sa C i C++ kodom)

Primer 1 (C#)

```
public class Count {
        private int k;
        public Count() {
                k = 0; }
        public int counter() {
                return ++k; }
    static void Main() {
        Count c = new Count();
        System.Console.WriteLine("Counter is " + c.counter());
        System.Console.WriteLine("Counter is " + c.counter());
      }
```

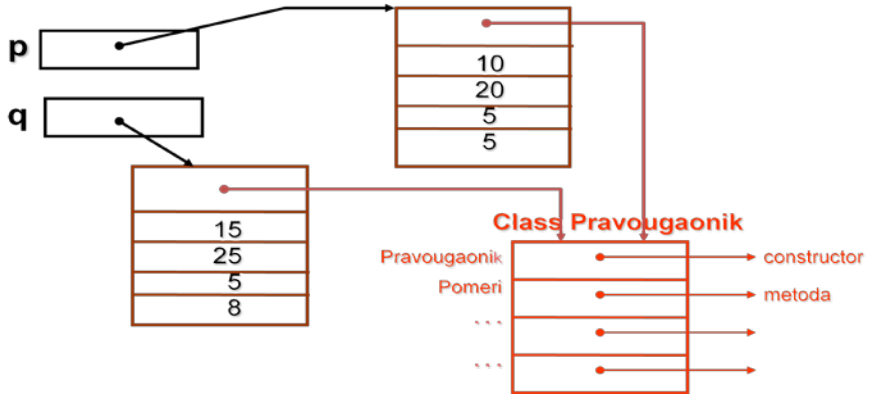
```
    }
```

Primer 2 (C#)

```
class  Pravougaonik {
 private int lokacijaX, lokacijaY, sirina, visina;  // atritbuti
 public Pravougaonik (int x, int y, int s, int v)   // konstruktor
  { lokacijaX = x; lokacijaY = y;  sirina = s; visina = h; }
 public void Pomeri( int deltaX, int deltaY)  // metoda
  {    lokacijaX = lokacijaX  + deltaX;
       lokacijaY = lokacijaY + deltaY;
   }
}
```

Alociranje heap-dinamičkih objekata

```
Pravougaonik  p = new  Pravougaonik( 10, 20, 5, 5);
Pravougaonik  q = new  Pravougaonik( 15, 25, 5, 8);
```



Eksplicitna heap-dinamička promenljiva

- Alocira se na heap-u
- Vrednost promenljive je perzistentna od poziva do poziva metode
- Životni vek promenljive vreme između alokacije i dealokacije heap-dinamičke promenljive

- Prednosti
  - dinamičko upravljanje memorijom
- Nedostaci
  - povećavaju složenost programiranja
    - dangling reference problem
    - Garbage promenljiva – u slučaju eksplicitne dealokacije

Implicitna heap-dinamička promenljiva

- Implicitna alokacija i dealokacija uzrokovana operacijom dodeljivanja (u vreme izvršenja programa)
- Dinamičko povezivanje promenljive sa tipom i memorijskom lokacijom

PERL, JavaScript

- Prednosti
  - Velika fleksibilnost
  - transparentna za korisnika (reduciranje složenost programiranja)
- Nedostaci
  - Neefikasnost, (svi atributi se dinamički povezuju)
  - Loša detekcija grešaka tipa

Doseg promenljive (Scope)

- **Doseg promenljive** je segment koda programa u kome je promenljiva poznata i može se koristiti
- *Pravila dosega* jezika određuju kako se pojavljivanja imena promenljivih u programu povezuju sa deklaracijama promenljivih
    - *Lokalna* promenljiva, vidljiva u dosegu ako je u njemu deklarisana
    - *Nelokalna* promenljiva, vidljiva u dosegu u kome nije deklarisana
    - Doseg je *prostorna* osobina promenljive

Promenljiva se povezuje sa dosegom:

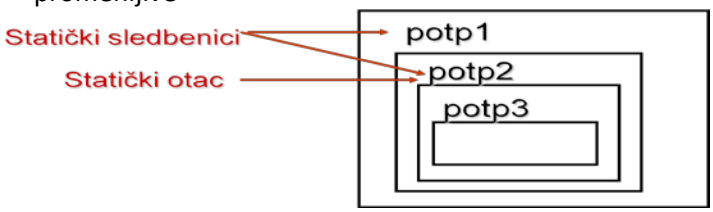
- ***Statički***
- ***Dinamički***

Statički doseg (leksički doseg)

- **Statički doseg** bazira se na tekstu programa i definiše se u terminima *leksičke* strukture programa
- Doseg promenljive se statički određuje, pre izvršenja programa (vreme kompajliranja)
- Dve kategorije programskih jezika, koje određuje metoda za kreiranje statičkog dosega:
    - PJ u kojima je dozvoljeno ugnježđavanje *potprograma* (Pascal, Ada, JavaScript)
    - PJ u kojima se ugnježđeni dosezi kreiraju samo pomoću ugnježđavanja *definicija klasa i blokova* (C++, Java, C#)

Statički doseg(ugnježđavanje potprograma)

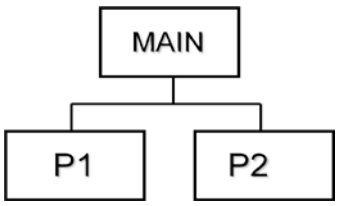
- **Ugnježđavanje potprograma** - povezivanje promenljive sa deklaracijom: kompajler pronalazi deklaraciju promenljive



Deklaracija se prvo pretražuje lokalno, zatim u najbližem dosegu sledbeniku (ocu) itd.

Statički doseg

```
program MAIN;
    var a : integer;
    procedure P1;
    begin
            writeln ( a );
    end; { P1}
    procedure P2;
            var a : integer;
    begin
             a := 0;  P1;
    nd; {P2}
    begin
            a := 7;  P2;
    end. { MAIN}
```

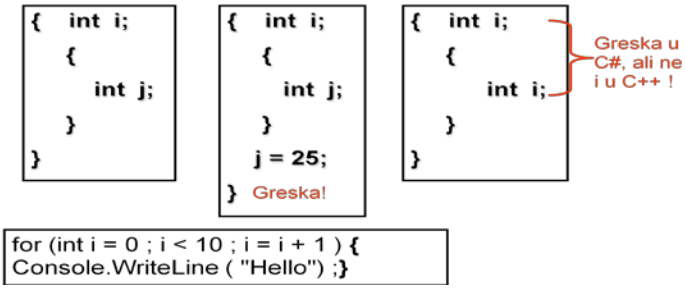


Povezivanje nelokalnih promenljivih sa deklaracijama zasniva se na leksičkoj strukturi programa.

Rezultat writeln **7**

- **Blok:** metoda za kreiranje statičkog dosega u programskom unit-u
- Blok je sekcija koda u kojoj se lokalne promenljive alociraju/dealociraju na početku/kraju bloka
    - ALGOL 60 inicijalno uveden blok

Primeri ugnježđavanja blokova u C#:



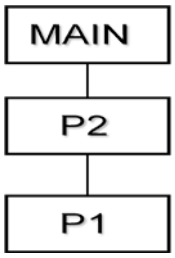
- Napredna rešenja za problem ogranizovanja složenih programa su inkapsulacione konstrukcije:
  - Klasa
  - Packages (Java)
  - Namespaces (C#, XML)

  } Imenovane inkapsulacije
- *Imenovana inkapsulacija* koristi se za kreiranje novog dosega za promenljivu

Dinamički doseg

- **Dinamički doseg** definiše doseg promenljive u terminima izvršavanja programa
- Lisp, APL, Snobol, Perl
- Dinamički doseg se zasniva se na *sekvenci poziva potprograma*
  - za povezivanje promenljivih sa njihovim deklaracijama pretražuje se sekvenca poziva potrograma;
  - pretraživanje počinje od poslednjeg pozvanog potprograma (princip stack-a)

```
program MAIN;
    var a : integer;
    procedure P1;
    begin
            writeln ( a );
    end; { P1}
    procedure P2;
            var a : integer;
    begin
             a := 0;   P1;
    nd; {P2}
    begin
            a := 7;  P2;
    end. { MAIN}
```



Povezivanje promenljivih sa deklaracijama zasniva se na sekvenci poziva potprograma

Rezultat writeln **0**

Životni vek i doseg promenljive

- Često povezani, ali različiti koncepti

Primer: **static** promenljiva u C funkciji

```
void fun() {
    static int x = 0;
    int y = 0;
    …
}
```

- Doseg prom. **x** i **y** od deklaracije do kraja bloka funkcije
- Životni vek prom. **x** = vreme izvršenja programa
- Životni vek prom. **y** = vreme izvršenja funkcije

Elementarni tipovi podataka

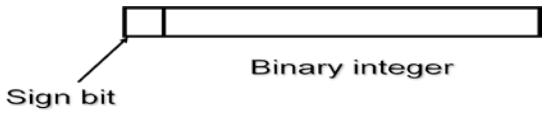
- Primitivni tipovi
- Prosti korisnički definisani tipovi

Primitivni tipovi

- Nisu definisani u terminima drugih tipova
- Predifinisani naziv tipa
- Predefinisani skup vrednosti
- Predefinisani skup operacija
  - Celobrojni tip
  - Realni tip
  - Logički tip
  - Znakovni tip

Celobrojni tip (Integer type)

- Podskup skupa celih brojeva
- Interna reprezentacija: binarna, hardverski podržana



- Programski jezici podražavaju različite dužine celih brojeva i u nekim PJ se mogu označiti kao **unsigned** ili **signed**

Java podržava četiri različite dužine **signed** celih brojeva:

- **byte** (8 bitova),
- **short** (16 bitova),
- **int** (32 bitova),
- **long** (64 bitova)

Celobrojni tipovi u C#:

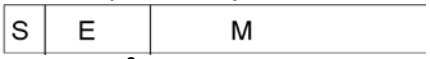
- **sbyte** - 8-bit signed integer
- **byte** - 8-bit unsigned integer.
- **short** - 16-bit signed integer.
- **ushort** - 16-bit unsigned integer.
- **int** - 32-bit signed integer.
- **uint** - 32-bit unsigned integer.
- **long** - 64-bit signed integer.
- **ulong** - 64-bit unsigned integer.

Realni tip (Floating point data type)

- Model realnih brojeva, ali reprezentacije su samo aproksimacije za realne vrednosti
- Većina programski jezici obično podržavaju dva tipa realnih brojeva
  - Float (realni broj sa jednostrukom tačnošću, 4 by)
  - Double (realni broj sa dvostrukom tačnošću, 8by)

Realni tip (Floating point data type)

Interna reprezentacija realnih vrednosti: preko **mantise** i **eksponenta**



3E2 = 3x10^2

Većina novih računara koristi *IEEE Floating-Point Standard format* za reprezentovanje realnih brojeva
IEEE Floating-point formati: jednostruka i dvostruka preciznost

Znakovni tip (Character type)

- Konačan i uređen skup znakova
- Interna reprezentacija:

Karakteri se reprezentuju kombinacijom bitova u binarni kod (Binary code)

- BCD
- ASCII
- Unicode

- **BCD code** (Binary coded decimal) 4 - bitni kod

Interna reprezentacija:

BCD code

128 = 0001 0010 1000

- **ASCII code (**American Standard Code for Information Interchange) 8-bitni kod (256 različitih znakova) koristi se u većini pj
- **Unicode** 16- bitni kod

Java, JavaScript, C#

Decimalni tip (Decimal type)

- Decimalni tip se koristi se u poslovnim (finansijskim) aplikacijama (COBOL, Decimal Type u C#)
- Memoriše fiksni broj decimalnih cifara sa fiksnom decimalnom tačkom na fiksnoj poziciji
- Interna reprezentacija u BCD code

128 = 0001 0010 1000

Logički tip (Boolean type)

- Samo dve vrednosti tipa: **true** i **false**
- Svi jezici ne podržavaju logički tip (na primer Perl i C)
- Logički tip je prvi put uveden u programskom jeziku ALGOL 60
- Interna implementacija

Tipična implementacija logičke vrednosti je jedan byte.

- Čitljivost

Prosti korisnički definisani tipovi

- Nabrajući tip (Enumeration Type)
- Podintervalni tip (Subrange Type)

Nabrajajući tip (Enumeration Type)

- **Nabrajajući tip** definiše uređeni skup vrednosti, koje su imenovane konstante
- Nabrajajući tip obezbeđuje način za organizovanje i grupisanje imenovanih konstanti
- Imenovanim konstantama implicitno se dodeljuju celobrojne vrednosti (0, 1, …), ali moguća su i ekplicitna dodeljivanja

- **Pascal**

```
type colortype  = (red, blue, green, yellow, black);
var color : colortype;
color := blue;
```

- **C++**

```
enum colors  (red, blue, green, yellow, black);
colors myColor = blue;
```

- **C#**

```
enum colors {red, blue, green, yellow, black};
colors a = colors.blue
```

Podintervalni tip (Subrange Type)

- **Podintervalni tip** definiše podintervalne vrednosti drugog primitivnog ili nabarajajućeg tipa
  - Definiše se pomoću gornje i donje granice
  - Prvi put uveden u programskom jeziku Pascal, korišćen u pj Ada

    type pozitivan = 0 .. MAXINT;

Unified Type System – C#