Programske paradigme

— Osnovna svojstva programskih jezika —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Svojstva programskih jezika

Postoji veliki broj pitanja koja se mogu postaviti na temu odnosa programskih jezika. Na primer,

- Šta je ono što čini dva programska jezika sličnim? (Na osnovu čega neki jezik pripada nekoj paradigmi?)
- Šta je ono što čini dva programska jezika različitim? (Na osnovu čega neki jezik ne pripada nekoj paradigmi?)
- Šta je zajedničko za sve programske jezike?
- Šta je ono što je specifično za svaki programski jezik?
- Šta je ono što je specifično za neki programski jezik?

Svojstva programskih jezika

- Postoje različita svojstva programskih jezika koja je potrebno razmatrati:
  - Leksika, sintaksa, semantika, pragmatika
  - Pravila vezana za imena, doseg, povezanost (posebno za imperativne jezike)
  - Kontrola toka, potprogrami
  - Tipovi
  - Prevođenje (kompajlirani/interpretirani jezici) i izvršavanje

Sadržaj

1 Leksika, sintaksa, semantika, pragmatika

1.1 Leksika

Leksika

- Programski jezici moraju da budu precizni
- Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika.
- U okviru leksike, definišu se reči i njihove kategorije
- U programskom jeziku, reči se nazivaju lekseme, a kategorije tokeni.
- U prirodnom jeziku, kategorije su imenice, glagoli, pridevi
- U programskom jeziku, tokeni mogu da budu identifikatori, ključne reči, operatori...

Leksika

- `a = 2 * b + 1` — lekseme su a, =, 2, *, +, 1 i b, a njima odgovarajući tokeni su identifikator (a i b), operator =, operator *, operator +, celobrojni literali 2 i 1.
- Neki tokeni sadrže samo jednu reč, a neki mogu sadržati puno različitih reči
- Programski jezik C sadrži više od 100 različitih tokena: 44 različite ključne reči, identifikatore, celobrojne vrednosti, realne vrednsoti, karakterske konstante, stringovske literale, dve vrste komentara, 54 operatora...
- Drugi moderni programski jezici (npr. C++, C#, Java, Ada) imaj sličan nivo kompleksnosti tokena

Leksika

- Razlikujemo ključne reči i identifikatore: identifikator ne može biti neka od ključnih reči, npr ne možemo da napravimo promenljivu koja bi se zvala `if` ili `while`
- Postoje ključne reči koje zavise od konteksta, engl. *contextual keywords* koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.
- Na primer, u C# -u reč `yield` može da se pojavi ispred `break` ili `return`, na mestima na kojima identifikator ne može da se pojavi. Na tim mestima, ona se interpretira kao ključna reč, ali može da se koristi na drugim mestima i kao identifikator.

Leksika

- Na primer, C# 4.0 ima 26 takvih konteksno zavisnih ključnih reči, C++11 ih takođe ima dosta.
- Većina je uvedena revizijom postojećeg jezika sa ciljem da se definiše novi standard: sa velikim brojem korisnika i napisanog koda, vrlo je verovatno da se neka reč već koristi kao identifikator u nekom postojećem programu. Uvođenjem konteksno zavisne ključne reči, umesto obične ključne reči, smanjuje se rizik da se postojeći program neće kompilirati sa novom verzijom standarda.

Leksika

- Reči su obično definišu regularnim izrazima
- Na primer, `[a-zA-Z_][a-zA-Z_0-9]*`
- Regularnim izrazima se definišu reči, dok se konačnim automatima prepoznaju ispravne reči (generisanje je bitno programerima, a prepoznavanje kompajlerima)
- Leksikom programa obično se bavi deo prevodioca koji se naziva leksički analizator: on dodeljuje ulaznim rečima odgovarajuće kategorije, što je bitno za dalji proces prevođenja.

1.2 Sintaksa

Sintaksa

- Sintaksa programskog jezika definiše strukturu izraza, odnosno načine kombinovanja osnovnih elemenata jezika u ispravne jezičke konstrukcije.
- Formalno, sintakse se opisuju kontekstno slobodnim gramatikama, a prepoznaju parserima (potisni automat)

Sintaksa

- Na primer, sledeća gramatika (zadata koristeći Bakus-Naurovu formu) definiše jednostavne aritmetičke izraze:

```
<exp> ::= <exp> "+" <exp>
<exp> ::= <exp> "*" <exp>
<exp> ::= "(" <exp> ")"
<exp> ::= "a"
<exp> ::= "b"
<exp> ::= "c"
```

- Isprvan izraz u ovom jeziku je proizvod ili suma od 'a', 'b' i 'c', na primer $a * (b + c)$

Sintaksa

- Sintaksa modernih programskih jezika je često veoma slična
- Veliki broj jezik je nastao pod uticajem C-a
- Ključne reči su uglavnom preuzete iz engleskog jezika
- Često programski jezici koji pripadaju istoj paradigmi imaju sličnu sintaksu (često ali ne uvek, npr Lisp i Haskell imaju potpuno različite sintakse)
- Sintaksa aritmetičkih izraza je najčešće veoma slična

Sintaksa — Ezoterični programski jezici

- Ezoterični programski jezici (eng. *esolang*) su programski jezici dizajnirani da testiraju granice dizajna programskih jezika
- Njihova sintaksa je obično veoma kompleksna i potpuno drugačija
- Ovi jezici imaju čudne ciljeve: da budu maksimalno nerazumljivi, da budu što manji, da budu što teži za kompilaciju i debagovanje, da budu zabavni, da budu šaljivi...
- Njihov cilj nije da budu upotrebljivi, korisni niti da rešavaju neki konkretan problem, već da zabave i da ispitaju granice i mogućnosti programskih jezika
- Pored drugačije sintakse, često imaju i potpuno neočekivanu semantiku

Sintaksa — Ezoterični programski jezici

- Primer programa u programskom jeziku AsciiDots

```
/#$<.
*-[+]
\#1/
```

- Primer programa u programskom jeziku Befunge

```
"dlroW olleH">:v
              ^,_@
```

- Primer programa u programskom jeziku Brainfuck:

```
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++
..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.
```

Sintaksa — Ezoterični programski jezici

- Primer programa u programskom jeziku LOLCODE:

```
HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

- Primer programa u programskom jeziku Rockstar:

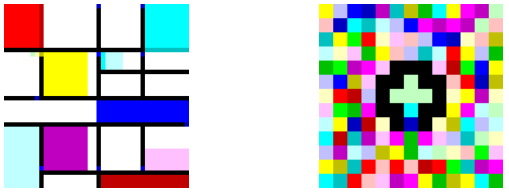
```
Put the whole of your heart into my hands
```

- Primer programa u programskom jeziku Unlambda:

```
`.!`.d`.l`.r`.o`.w`. `.,`.o`.l`.l`.e`.Hi
```

Sintaksa — Ezoterični programski jezici

- Primeri programa u programskom jeziku Piet:



Piet program koji štampa reč ”Piet” (levo) i ”Hello World” (desno).

1.3 Semantika programskih jezika

Semantika

- Semantika pridružuje značanje ispravnim konstrukcijama na nekom programskom jeziku.
- Semantika programskog jezika određuje značenje jezika, odnosno šta se dešava kada se program izvršava.
- Obično je značajno teže definisati nego sintaksu.
- Formalno zadavanje sintakse nam omogućava da automatski generišemo parser
- Slično, formalno zadavanje semantike ima svoje značajne primene

Semantika

- Kompajler prevodi kôd na mašinski kôd u skladu sa zadatom semantikom jezika.
- Tokom kompilacije, vrši se proveravanje da li postoji neka semantička greška, tj situacija koja je sintaksno ispravna ali za konkretne vrednosti nema pridruženo značenje zadatom semantikom.
- Neki aspekti semantičke korektnosti programa se mogu proveriti tokom prevođenja programa — na primer, da su sve promenljive koje se koriste u izrazima definisane i da su odgovarajućeg tipa
- Neki aspekti semantičke korektnosti se mogu proveriti tek u fazi izvršavanja programa — na primer, deljenje nulom, pristup elementima niza van granica...

Semantika

- Semantičke provere se mogu podeliti na statičke (provere prilikom kompilacije) i dinamičke (provere prilikom izvršavanja programa).
- Statički se ne može pouzdano utvrditi ispunjenost semantičkih uslova, tako da je moguće da se neke greške ne utvrde, iako prisutne, kao i da se neke greške pogrešno utvrde iako nisu prisutne i da rezultiraju nepotrebnim proverama prilikom izvršavanja programa.
- Različitim programskim jezicima odgovaraju izvršni programi koji sadrže različite nivoe dinamičkih provera ispravnosti.

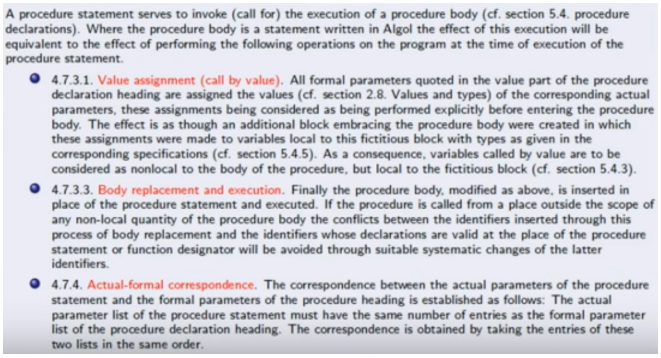
1.4 — Neformalna semantika

Neformalna semantika

- Semantika može da se opiše formalno i neformalno, često se zadaje samo neformalno.

- Uloga neformalne semantike je da programer može da razume kako se program izvršava pre njegovog pokretanja.
- Na primer, semantika naredbe `if(a<b) a++;` neformalno se opisuje sa „ukoliko je vrednost promenljive a manja od vrednosti promenljive b, onda uvećaj vrednost promenljive a za jedan"
- Programski jezici su kompleksni i zadavanje semantike programskog jezika je kompleksno

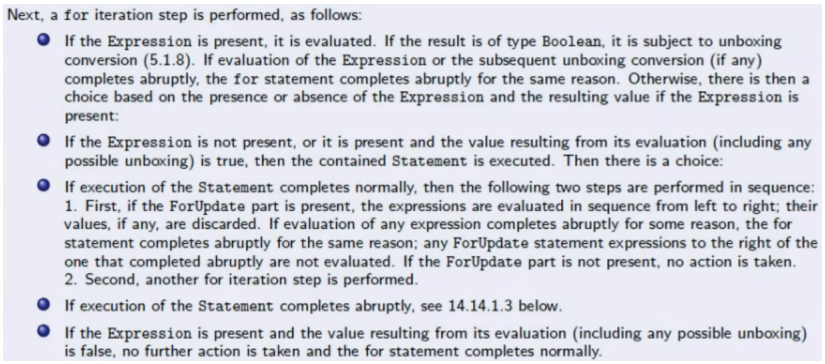
Neformalna semantika — Algol 60



A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. procedure declarations). Where the procedure body is a statement written in Algol the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement.

- 4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).
- 4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.
- 4.7.4. Actual-formal correspondence. The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

Neformalna semantika danas — Java



Next, a `for` iteration step is performed, as follows:

- If the `Expression` is present, it is evaluated. If the result is of type `Boolean`, it is subject to unboxing conversion (5.1.8). If evaluation of the `Expression` or the subsequent unboxing conversion (if any) completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the `Expression` and the resulting value if the `Expression` is present:
- If the `Expression` is not present, or it is present and the value resulting from its evaluation (including any possible unboxing) is true, then the contained `Statement` is executed. Then there is a choice:
- If execution of the `Statement` completes normally, then the following two steps are performed in sequence: 1. First, if the `ForUpdate` part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the for statement completes abruptly for the same reason; any `ForUpdate` statement expressions to the right of the one that completed abruptly are not evaluated. If the `ForUpdate` part is not present, no action is taken. 2. Second, another for iteration step is performed.
- If execution of the `Statement` completes abruptly, see 14.14.1.3 below.
- If the `Expression` is present and the value resulting from its evaluation (including any possible unboxing) is false, no further action is taken and the for statement completes normally.

Problemi sa neformalnom semantikom

- Problemi sa neformalnim opisima:
  - Dvosmislenost
  - Nekonzistentnost (kontradikcije)
  - Nedorečenost
- Nekonzistentnosti u semantici koja je neformalno zadata je teško pronaći, dok u formalno zadatim semantikama je moguće automatski proveriti konzistentnost

Kako savladti semantiku?

- Semantika programskog jezika se obično ne uči čitanjem specifikacije
- Programeri obično razvijaju svoju mentalnu sliku semantike svakodnevnim korišćenjem jezika, tj. kroz kompajler/interperter
- Znanje koje se stiče iskustvenim putem na osnovu toga šta neki programi rade je neprecizno i nepotpuno, ali je za neke obične upotrebe to u redu

Kako savladti semantiku?

- Kako se radi razvoj kompajlera?
- Da li razvoj kompajlera sme da ide na takav način?
- Da li i može da ide na takav način ako ne postoji kompajler za taj programski jezik?

Kako savladti semantiku?

- Za razvoj kompajlera neophodno je da se čita specifikacija.
- Ako je specifikacija neformalna, moguće je da će je različiti programeri različito razumeti i da će se zato različiti kompajleri ponašati drugačije.
- Ako se kompajleri različito ponašaju, kako znati koji je usklađen sa predviđenom semantikom tog programskog jezika?
- Formalnom semantikom mogu se prevazići ovi problemi, a formalna semantika se može zadati i za realne programske jezike (npr, semantika jezika ML je zadata formalno, "The definition of standard ML", MIT Press, M. Tofte, R. Milner, R. Harper)

1.5 — Formalno zadavanje semantike

Formalna semantika

- Cilj: precizno definisati značenje (ponašanje) programa
- Formalnom semantikom se definiše značenje samo sintaksno ispravnih programa
- U skladu sa time, precizna semantika je definisana implementacijom kompajlera: na primer, precizna semantika jezika C/C++ je zapravo data kompajlerom gcc ili clang
- Međutim, iako precizna, takva semantika za mnoge probleme nije upotrebljiva.
  - Na primer, recimo da nas interesuje da li su dve implementacije funkcije koja izračunava faktorijel ekvivalentne. Korišćenjem gcc-a ili clang-a, možemo da proverimo da li se te dve implementacije poklapaju na nekim ulazima, ali ne možemo da dokažemo njihovu ekvivalentnost

  - Takođe, ne može se upotrebom kompajlera, bez dodatnih objašnjenja, razumeti jezik

- Formalna semantika daje okvir za **rezonovanje** o osobinama programa

Ko rezonuje o programima?
Programer

- Da li kôd radi ono što želim? (Testiranjem možemo samo da podignemo pouzdanost da je kôd dobar, ali semantika nam govori šta se tu dešava i da li je to u skladu sa našim očekivanjima.)
- Da li je zamena koda *boljim* kodom u redu (proces refaktorisanja)? Optimizacijom od koda koji radi, često dobijamo kôd koji je brži, ali ne radi... Zamenu koda rade ne samo programeri, već i optimizacije u kompajlerima. Npr, da li je zamena `x=x+1; x=x+1;` sa `x=x+2;` u redu?

Ko rezonuje o programima?
Dizajner programskog jezika / Programer koji učestvuje u razvoju kompajlera

- Brz razvoj prototipa jezika
  - Operaciona semantika je opis apstraktne mašine koja izvršava program i ona omogućava brz razvoj prototipa
  - Ako je za neki konstrukt programskog jezika potrebno previše prostora da se formalno zapiše, onda je verovatno u pitanju loša ideja koju ne treba implementirati jer je previše kompleksna i niko neći razumeti kako treba
- Optimizacije
- Napredne implementacione tehnike (npr, funkcionalni programski jezici zahtevaju netrivijalne transformacije kako bi se prevli u kôd koji se brzo izvršava)
- Formalna semantika omogućava formalno rezonovanje o svojstvima programa

Osnovne vrste semantika
Postoji veliki broj mogućnosti zadavanja semantike programa, naredna tri su osnovne vrste semantike

- **Operaciona semantika** — Šta program radi? Odgovara izvršavanju u okviru apstraktne mašine
- **Denotaciona semantika** — Šta program znači? Matematički objekti u nekom adekvatnom semantičkom domenu (npr. funkcija iz stanja u stanje, odnosno transformacija ulaza u izlaz)
- **Aksiomatska semantika** — Koje osobine ima program? Kolekcija logičkih osobina koje obezbeđuje program (npr. preduslovi i postuslovi, prisutno u paradigmi *design by contract*)

Operaciona semantika

- Operaciona semantika opisuje kako se izračunavanje izvršava.
- Ponašanje se formalno definiše korišćenjem apstraktnih mašina, formalnih automata, tranzicionih sistema...
- U okviru ove semantike postoje strukturna operaciona semantika (small-step semantics, opisuje individualne korake izračunavanja) i prirodna operaciona semantika (big-step semantics, opisuje ukupne rezultate izračunavanja).
- Najčešće se koristi za opis i rezonovanje o imperativnim jezicima jer individualni koraci izračunavanja opisuju na koji način se menja stanje programa.

Denotaciona semantika

- Denotaciona semantika definiše značenje prevođenjem u drugi jezik, za koji se pretpostavlja da je poznata semantika
- Najčešće je taj drugi jezik nekakav matematički formalizam
- Povezivanje svakog dela programskog jezika sa nekim matematičkim objektom kao što je broj ili funkcija: svaka sintaksna definicija se tretira kao objekat na koji se može primeniti funkcija koja taj objekat preslikava u matematički objekat koji definiše značenje.

Denotaciona semantika

- Dodeljivanjem značenja delovima programa dodeljuje se značenje celokupnom programu, tj semantika jedne programske celine definisana je preko semantike njenih poddelova. Ova osobina denotacione semantike naziva se **kompozitivnost**.
- Koristi se za definisanje semantike funkcionalnih programskih jezika — funkcionalno programiranje zasniva se na pojmu matematičkih funkcija i izvršavanje programa svodi se na evaluaciju funkcija.
- Analiziranje programa se svodi na analiziranje matematičkih objekata, što olakšava formalno dokazivanje semantičkih svojstava programa.

Aksiomatska semantika

- Aksiomatska semantika omogućava viši nivo apstrakcije i udaljavanje od konkretne semantike programskog jezika
- Aksiomatska semantika zasniva se na Horovoj logici
- Horova trojka `{P} S {Q};` gde se `{P}` naziva preduslov a `{Q}` postuslov, opisuje kako izvršavanje dela koda menja stanje izračunavanja:
  - ako je ispunjen preduslov `{P}`,
  - ako se izvršavanje komande `S` završava za dato stanje
  - onda će `{Q}` da važi u stanju u kojme se `S` završilo
- Na ovaj način se mogu dokazati razna interesantna svojstva programa koji se razmatra

Formalno zadavanje semantike

Više na temu formalnog zadavanja semnatike:

- Uvod u semantiku programskih jezika: https://youtu.be/nOxGHUXLD_A
- Operaciona semantika: https://youtu.be/QSCWv6zzGAI
- Denotaciona semantika: https://youtu.be/e4-AFlZRK6k
- Aksiomatska semantika: https://youtu.be/HHuC9sjH7kY

1.6 Pragmatika

Pragmatika programskog jezika

- Pragmatika programskog jezika se bavi načinima na koje je nameravno da se jezik praktično koristi, odnosno na koji način konstrukti jezika mogu da se upotrebljavaju da bi se ostvarili različiti željeni ciljevi
- Pragmatika obuhvata posledice realizacije i načine upotrebe osnovnih koncepata programskih jezika, koji uključuju, na primer, imena, promenljive, doseg, životni vek, tipove podataka, kontrolu toka, potprograme, modularnost, upravljanje memorijom, prevođenje, izvršavanje

Pragmatika programskog jezika

- Na primer, razmotrimo naredbu dodele. `promenljiva = izraz;`
- Sintaksno, naredba dodele obuhvata promenljivu i izraz koji su razdvojeni operatorom =
- Semantički, promenljiva označava memorijsku lokaciju, a izraz označava izračunavanje vrednosti na osnovu sadržaja memorije.
- Naredba dodele semantički izvršava evaluaciju izraza i upisivanje izračunate vrednosti na odgovarajuću memorijsku lokaciju

Pragmatika programskog jezika

- Pragmatika govori na koji način naredba dodele može da se upotrebi.
- Postoji puno mogućnosti:
  - da se postavi privremena promenljiva za vrednost izraza koji će biti potreban više puta,
  - da se iskomunicira vrednost iz jednog dela programa drugom,
  - da se izmeni deo strukture podataka,
  - da se postavi naredna vrednost promenljive u nekom iterativnom izračunavanju,
  - ...
- Dakle, pragamtika govori šta je neki koncept i na koji način se može koristiti

2 Imena, promenljive, povezanost

2.1 Imena

Imena

- Ime — string koji se koristi za predstavljanje nečega (promenljive, konstante, operatora, tipova...)
- Prvi programski jezici imali su imena dužine jednog karaktera (po uzoru na matematičke promenljive). Fortran I je to promenio dozvoljavajući šest karaktera u imenu.
- Fortran 95 i kasnije verzije Fortrana dozvoljavaju 31 karakter u imenu, C99 nema ograničenja za interna imena, ali samo prvih 63 je značajno. Eksterna imena u C99 (ona koja su definisana van modula i o kojima mora da brine linker) imaju ograničenje od 31 karaktera. Imena u Javi, C# i Adi nemaju ograničenja dužine i svi karakteri su značajni. C++ ne zadaje limit dužine imena, ali implementacije obično zadaju.

Imena

- Imena u većini programskih jezika imaju istu formu — slova, cifre i podvlaka
- Podvlaka se sve češće zamenjuje kamiljom notacijom
- U nekim jezicima imena moraju da počnu specijalnim znacima (npr PHP ime mora da počne sa $)
- Imena su najčešće *case sensitive*, što može da stvara probleme

Imena

- Specijalne reči se korsite da učine program čitljivijim — imenuju akcije koje treba da se sprovedu ili sintaksno odvajaju delove naredbi i programa. U većini jezika specijalne reči su rezervisane reči koje ne mogu da budu predefinisane od strane programera.
- Ključne reči su najčešće rezervisane reči, ali ne moraju to da budu, kao na primer kontestno zavisne ključne reči
- Nije dobro ako jezik ima preveliki broj rezervisanih reči
- Na primer, COBOL ima oko 300 rezervisanih reči, u koje spadaju i `count`, `length`, `bottom`, `destination`...

2.2 Promenljive

Promenljive

- Promenljiva: apstrakcija memorijskih jedinica (ćelija)
- Karakteristke promenljivih: **ime, adresa, tip, vrednost, doseg, životni vek**

2.3 – Ime, adresa, tip, vrednost, doseg, životni vek

Ime promenljive

- Ime promenljive je imenovanje memorijske lokacije
- Imena promenljivih su najčešća imena u programu, ali **nemaju sve promenljive imena** (tj nemaju sve meorijske lokacije imena, na primer, memorijske lokacije eksplicitno definisane na hipu kojima se pristupa preko pokazivača)

Adresa

- Adresa promenljive: adresa fizičke memorije koja je pridružena promenljivoj za skladištenje podataka.
- Promenljiva može imati različite fizičke lokacije prilikom istog izvršavanja programa (npr različiti pozivi iste funkcije rezultuju različitim adresama na steku). Adresa se često naziva l-vrednost
- Moguće je da postoje više promenljivih koje imaju istu adresu — aliasi
- Aliasi pogoršavaju čitljivost programa i čine verifikaciju programa težom.
- Aliasi: unije u C/C++-u, dva pointera koji pokazuju na istu memorijsku lokaciju, dve reference, pointer i promenljiva...
- Aliasi se u mnogim jezicima kreiraju kroz parametre poziva potprograma

Tip

- Tip promenljive određuje opseg **vrednosti** koje promenljiva može da ima kao i **operacije** koje se mogu izvršiti za vrednosti tog tipa
  - osnovni tipovi,
  - niske,
  - korisnički definisani prebrojivi tipovi (`enum`, `subrange`),
  - nizovi,
  - pokazivači
  - reference
  - asocijativni nizovi (grupisani po ključu),
  - strukture,
  - torke,
  - liste,
  - unije,
  - ...

Vrednost

- Vrednost promenljive je sadržaj odgovarajuće memorijske ćelije, često se naziva r-vrednost
- Da bi se pristupilo r-vrednosti mora najpre da se odredi l-vrednost, što ne mora da bude jednostavno, jer zavisi od pravila dosega.

Doseg

- Doseg određuje deo programa u kojem je vidljivo neko ime
- Pravila dosega mogu da budu veoma jednostavna (kao u skript jezicima) ili veoma kompleksna
- Doseg ne određuje nužno životni vek objekta

Životni vek

- Životni vek obično odgovara jednom od tri osnovna mehanizma skladištenja podataka
- **Statički objekti** — koji imaju životni vek koji se prostire tokom celog rada programa (npr globalne promenljive)
- **Objekti na steku** — živoni vek koji se alocira i dealocira LIFO principom, obično zajedno sa pozivom i završetkom rada podprograma.
- **Objekti na hipu** — koji se alociraju i dealociraju proizvoljno od strane programera, implicitno ili eksplicitno, i koji zahtevaju opštiji i skuplji sistem za upravljanje skladištenjem (memorijom)

2.4 Povezivanje

Povezivanje

- Povezivanje uspostavlja odnos između imena sa onim što to ime predstavlja.
- Vreme povezivanja je vreme kada se ova veza uspostavlja
- Rano vreme povezivanja — veća efikasnost, kasno vreme povezivanja — veća fleksibilnost.

Povezivanje

- Vreme povezivanja može biti (vreme — primer)
  - vreme dizajna programskog jezika (osnovni konstrukti, primitivni tipovi podataka...)
  - vreme implementacije jezika (veličina osnovnih tipova podataka...)
  - vreme programiranja (algoritmi, strukture podataka, imena promenljivih...)

- vreme kompiliranja (preslikavanje konstrukata višeg nivoa u mašinski kôd)
- vreme linkovanja (kada se ime iz jednog modula odnosi na objekat definisan u drugom modulu)
- vreme učitavanja (povezivanje objekata sa fizičkim adresama u memoriji)
- vreme izvršavanja (povezivanje promenljivih i njihovih vrednosti)

3 Kontrola toka i tipovi

3.1 Kontrola toka

Kontrola toka

- Kontrola toka definiše redosled izračunavanja koje računar sprovodi da bi se ostvario neki cilj.
- Postoje različiti mehanizmi određivanja kontrole toka:
  1. Sekvenca — određen redosled izvršavanja
  2. Selekcija — u zavisnosti od uslova, pravi se izbor (`if`, `if-else`, `switch`, `case`)
  3. Iteracija — fragment koda se izvršava više puta (`while`, `for`, `foreach`, `repeat-until`, `do-while`)
  4. Potprogrami su apstrakcija kontrole toka: one dozvoljavaju da programer sakrije proizvoljno komplikovan kôd iza jednostavnog interfejsa (procedure, funkcije, rutine, metodi, podrutine, korutine)

Kontrola toka

- Postoje različiti mehanizmi određivanja kontrole toka:
  5. Rekurzija — definisanje izraza u terminima samog izraza (direktno ili indirektno)
  6. Konkurentnost — dva ili više fragmenta programa mogu da se izvršavaju u istom vremenskom intervalu, bilo paralelno na različitim procesorima, bilo isprepletano na istom procesoru
  7. Podrška za rad sa izuzecima — fragment koda se izvršava očekujući da su neki uslovi ispunjeni, ukoliko se desi suportno, izvršavanje se nastavlja u okviru drugog fragmenta za obradu izuzetka
  8. Nedeterminizam — redosled izvršavanja se namerno ostavlja nedefinisan, što povlači da izvršavanje proizvoljnim redosledom treba da dovede do korektnog rezultata

3.2 Sistem tipova

Tipovi podataka

- Programski jezici moraju da organizuju podatke na neki način
- Tipovi pomažu u dizajniranju programa, proveri ispravnosti programa i u utvrđivanju potrebne memorije za skladištenje podataka
- Mehanizmi potrebni za upravljanjem podacima nazivaju se sistem tipova

Sistem tipova

- Sistem tipova obično uključuje
  - skup predefinisanih osnovnih tipova (npr `int`, `string`...)
  - mehanizam građenja novih tipova (npr `struct`, `union`)
  - mehanizam kontrolisanja tipova
    - pravila za utvrđivanje ekvivalentnosti: kada su dva tipa ekvivalentna?
    - pravila za utvrđivanje kompatibilnosti: kada se jedan tip može zameniti drugim?
    - pravila izvođenja: kako se dodeljuje tip kompleksnim izrazima?
  - pravila za proveru tipova (statička i dinamička provera)

Tipovi

- Jezik je tipiziran ako precizira za svaku operaciju nad kojim tipovima podataka može da se izvrši
- Jezici niskog nivoa, kao što je asembler i mašinski jezici nisu tipizirani jer se svaka operacija izvršava nad bitovima fiksne širine

Tipovi

Na primer, razmotrimo operator +

- U programskom jeziku C, operator + može da se primeni nad celobrojnim tipom, nad realnim tipom, ali ne može da se primeni nad niskama karaktera:

```
int a=3, b=4, c=a+b;

float f1=3.0, f2=4.0, f3=f1+f2;

char s1[10]="niska1", s2[10]="niska2", s3[20]=s1+s2;
error: invalid operands to binary + (have ‘char *’ and ‘char *’)
```

Tipovi

- Operator + u programskom jeziku Python može da se primeni nad celobrojnim i relanim vrednostima, ali i nad stringovima

```
a = 3
b = 5
c = a + b
print(c)
```

```
a = 3.0
b = 4.0
c = a + b
print(c)
```

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

Ali ne može se primeniti nad stringom i celobrojnom vrednošću

```
a = "Hello"
b = 5
c = a + " " + b
TypeError: can only concatenate str (not "int") to str
```

Tipovi

- Tipiziranje može da bude *slabo* i *jako*
  - Kod jako tipiziranih jezika izvođenje operacije nad podacima pogrešnog tipa će izazvati grešku
  - Slabo tipizirani jezici izvršavaju implicitne konverzije ukoliko nema poklapanja tipova
  - Neki jezici ne dozvoljavaju implicitno ali dozvoljavaju eksplicitno kastovanje tipova
- Implicitne konverzije skrivaju moguće greške i u modernim programskim jezicima se najčešće izbegavaju

Tipovi

Na primer, programski jezik C je slabo tipiziran i implicitnim konverzijama dozvoliće izračunavanje narednih izraza

```
int a = 5;
float b = 5.5;
float f = a + b; /* f = 8.5; */
int i = a + b;   /* i = 8;   */
```

Tipovi

Međutim, programski jezik Rust je jako tipiziran i za naredni program će prijaviti grešku

```
fn main() {
    let a = 5;
    let b = 5.5;
    let k = a + b;
    println!("{}", k);
}
```

```
error[E0277]: cannot add a float to an integer
 --> src/main.rs:4:16
  |
4 |     let k = a + b;
  |               ^ no implementation for `{integer} + {float}`
  |
  = help: the trait `std::ops::Add<{float}>`
          is not implemented for `{integer}`
```

Tipovi

Programski jezik Python ne dozvoljava sabiranje stringa i broja, tj ne vrši implicitno kastovanje. Ali, ova vrsta sabiranja je dozvoljena uz eksplicitno kastovanje.

```
s1 = 'aaa'
s2 = 'bbb'
i = 100
f = 0.25
# s = s1 + i
# TypeError: ...
```

```
s = s1 + '_' + str(i) + '_' + s2 + '_' + str(f)
print(s)
# aaa_100_bbb_0.25
```

Tipovi

- Važan je trenutak kada se radi provera tipova
  - Za vreme prevođenja programa — statičko tipiziranje (C, Rust)
  - Za vreme izvršavanja programa — dinamičko tipiziranje (Python)
- Statičko tipiziranje omogućava otkrivanje grešaka u fazi kompilacije tako da se u fazi izvršavanja greške sa tipovima ređe dešavaju. S druge strane, statičko određivanje tipova je dosta restriktivno.
- Dinamičko određivanje tipova daje veću fleksibilnost programeru, ali zato je teže za debagovanje i može lakše da dovede do grešaka u fazi izvršavanja.

Tipovi

- Za statičko određivanje tipova važi
  - Jako tipizirani jezici imaju snažan sistem tipova koji omogućava otkrivanje grešaka i koji ne dopušta da greške prođu u fazu izvršavanja (npr Rust)
  - Slabo tipizirani jezici imaju sistem tipova koji dopušta da greške prođu u fazu izvršavanja (npr C)

4 Prevođenje i izvršavanje

4.1 Kompilacija vs interpretiranje

Kompilacija vs interpretiranje

- Kompilirani jezici se prevode u mašinski kôd koji se izvršava direktno na procesoru računara — faze prevođenja i izvršavanja programa su razdvojene.
- U toku prevođenja, vrše se razne optimizacije izvršnog koda što ga čini efikasnijim.
- Jednom preveden kôd se može puno puta izvršavati, ali svaka izmena izvornog koda zahteva novo prevođenje.

Kompilacija vs interpretiranje

- Interpretirani jezici se prevode naredbu po naredbu i neposredno zatim se naredba izvršava — faze prevođenja i izvršavanja nisu razdvojene već su međusobno isprepletene.
- Rezultat prevođenja se ne smešta u izvršnu datoteku, već je za svako naredno pokretanje potrebna ponovna analiza i prevođenje.
- Sporiji, ali prilikom malih izmena koda nije potrebno vršiti analizu celokupnog koda.
- Hibridni jezici — kombinacija prethodna dva.

Kompilacija vs interpretiranje

- Teorijski, svi programski jezici mogu da budu i kompilirani i interpretirani.
- Moderni programski jezici najčešće pružaju obe mogućnosti, ali u praksi je za neke jezike prirodnije koristiti odgovarajući pristup.
- Nekada se u fazi razvoja i testiranja koristi interpretirani pristup, a potom se generiše izvršni kôd kompilacijom.
- Prema tome, razlika se zasniva pre na praktičnoj upotrebi nego na samim karakteristikama jezika.

4.2 Izvršavanje

Izvršavanje

- Svaka netrivijalna implementacija jezika višeg nivoa intenzivno koristi rantajm biblioteke
- Rantajm sistem se odnosi na skup biblioteka od kojih implementacija jezika zavisi kako bi program mogao ispravno da se izvršava.
- Neke bibliotečke funkcije rade jednostavne stvari (npr podrška aritmetičkim funkcijama koje nisu implementirane u okviru hardvera, kopiranje sadržaja memorije...) dok neke rade komplikovanije stvari (npr upravljanje hipom, rad sa baferovanim I/O, rad sa grafičkim I/O...)

Izvršavanje

- Neki jezici imaju veoma male rantajm sisteme (npr. C)
- Neki jezici koriste rantajm sisteme intenzivno
- Virtuelne mašine u potpunosti skrivaju hardver arhitekture nad kojom se program izvršava
- Npr, JAVA, Scala, Kotlin koriste JVM
- Jezici .Net platforme koriste CLR (engl. *Common Language Runtime*)

5 Pitanja i literatura

Pitanja

- Koja su osnovna svojstva programskih jezika?
- Koji formalizam se koristi za opisivanje sintakse programskog jezika?
- Šta definiše semantkika programskog jezika?
- Koji su formalni okviri za definisanje semantike programskih jezika?
- Šta je ime?
- Šta je povezivanje?
- Koja su moguća vremena povezivanja?

Pitanja

- Šta je doseg?
- Šta je kontrola toka?
- Koji su mehanizmi određivanja kontrole toka?
- Šta je sistem tipova i šta on uključuje?
- Šta je tipiziranje i kakvo tipiziranje postoji?
- Kada se radi provera tipova?
- Koja je razlika između kompiliranja i interpretiranja?
- Šta je rantajm sistem?

Literatura

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages (Pragmatic Programmers), 2010. by Bruce A. Tate
- Semantics with Applications: A Formal Introduction. Hanne Riis Nielson, Flemming Nielson. John Wiley & Sons, Inc. http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf

Literatura

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Deo materijala je preuzet od prof Dušana Tošića, iz istoimenog kursa.