Osnove formalne semantike programskih jezika

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Isidora Đurđević, Ana Stanković,
Milica Đurić
isidoradjurdjevic.100@gmail.com, anastankovic167@gmail.com,
mdjuric55@gmail.com

1. maj 2017.

Sažetak

Semantika ima posebnu ulogu u teoriji i proučavanju značenja programskih jezika. Cilj ovog rada je da pokaže koja je uloga formalnog zadavanja semantike programskih jezika i osnovne primene formalne semantike. Biće definisana tri osnovna pristupa semantike: operaciona, denotaciona i aksiomatska semantika i objašnjene njihove uloge. Svaki od pristupa je potkrepljen primerom koji prikazuje primenu te semantike u računarstvu.

Ključne reči: semantika programskih jezika, formalna semantika, operaciona semantika, denotaciona semantika, značenje programa.

Sadržaj

1 Uvod

Vođenje konverzacije ne bi bilo moguće bez znanja o tome šta koja reč koju koristimo predstavlja. U našoj glavi stoje jasne neformalne definicije svake reči koje znamo i pomoću kojih povezujemo objekte sa njihovim značenjem. U lingvistici, nauci o jeziku, svaka reč dobija svoju formalnu definiciju, a pravila formiranja rečenica su jako rigorozna, odnosno, postoje jasne odredbe šta je ispravno, a šta neispravno. Ove formalne definicije i pravila omogućavaju onima koji se susreću sa jezikom po prvi put da ga lakše nauče i razumeju kako bi mogli da ga koriste. Slično, ovo može biti primenjeno i na programske jezike.

Kao što je izučavanje prirodnog jezika podeljeno na izučavanje strukture onoga čime se nešto izražava, tj. *sintakse* i izučavanje značenja onoga što je izraženo, tj. *semantike*, tako se i proučavanje programskog jezika može podeliti na proučavanje njegovog formiranja i njegovog značenja. Osnovni zadatak sintakse programskih jezika je da omogući formiranje korektnih programa iz ugla strukture izraza. Izrazi poput $x++; function(x, y);$ su sintaksno ispravni, ali šta oni predstavljaju, koje je njihovo značenje? Sintaksa nema odgovor na ovo pitanje jer ona nije povezana sa značenjem ili ponašanjem programa u toku izvođenja. To je zadatak ***semantike programskih jezika***. Međutim, sintaksa mora biti definisana pre semantike jer se značenje može pridružiti samo ispravnim izrazima jezika. Definisanje sintakse je obično lakši posao od definisanja semantike.

Semantika može da se opiše formalno i neformalno, gde je češće neformalno opisivanje, odnosno opisivanje prirodnim jezikom što može biti neprecizno. Zato se u narednim glavama bavimo formalnim opisom semantike koja bi trebalo da otkloni sve nepreciznosti i da dâ potpunu definiciju programskog jezika. Biće opisani neki od formalnih pristupa kao što su *operaciona, denotaciona i aksiomatska* semantika. Svakom od pristupa će se prići površinski zbog same složenosti formalnog definisanja, a čitaocu se ostavlja da proširi svoje znanje u naznačenoj literaturi.

2 Formalna semantika

Kao što je već napomenuto, semantika se može definisati formalno i neformalno. Pri neformalnom definisanju koristimo prirodan jezik. Na primer, izraz

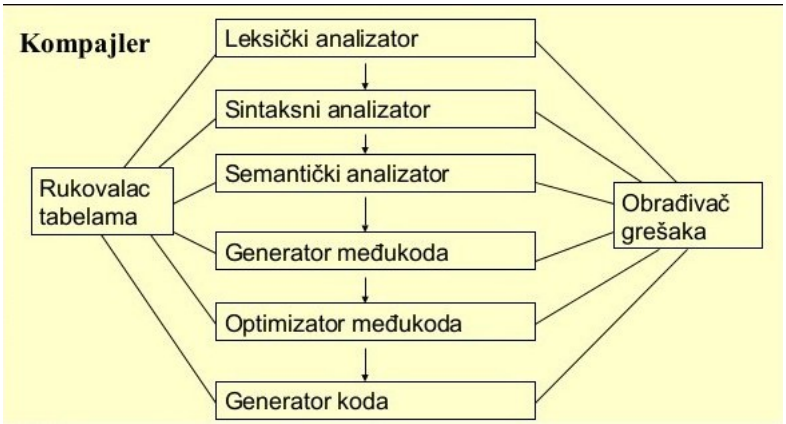
$$while(x > 0)\ \{x - -;\}$$

može se opisati rečima : "sve dok je vrednost promenljive x veća od nule umanjuj vrednost promenljive x za jedan". Međutim, da li je to jednostavno uraditi za svaki deo programskog koda ili za svaki programski jezik?

Teško je pisati vrlo precizne definicije neformalnim jezikom, obično postane komplikovano za razumeti i predugačko za neke kompleksne programske jezike što dovodi do loše napisanih programa jer se stvaralac programskog jezika i onaj ko ga koristi ne razumeju. To je jedan od razloga zašto se uvodi *formalna semantika*, što znači precizna semantika.

Pri učenju novog programskog jezika, programer bi trebalo da uči i samu logiku jezika. Neki vid opisa logike programskog jezika olakšava programerima pisanje jakih i stabilnih programa. Ovaj opis nam može pružiti, takođe, formalna semantika.

Još jedan od razloga zašto se uvodi formalna semantika jeste to što ona čini osnovu formalne verifikacije programa, tj. deo je interpretera i kompilatora. Na slici 1 dat je prikaz strukture kompilatora.



Slika 1: Struktura kompilatora [**?**]

Kompleksnost uvođenja formalne semantike daje više načina pristupa njenom definisanju. Mi ćemo se fokusirati na tri pristupa:

1. **Operaciona semantika**
   Značenje programa predstavlja tok koraka izvršavanja programa sa datim ulazom.
2. **Denotaciona semantika**
   Značenje programa je matematička funkcija koja prevodi ulaz u izlaz programa.
3. **Aksiomatska semantika**
   Značenje programa je ono što može biti dokazano o njemu koristeći aksiome.

U sledećim poglavljima, svaka od ovih formalnih semantika će biti opisana šire.

2.1 Operaciona semantika

Operaciona semantika je način davanja značenja programskim jezicima kroz matematičku reprezentaciju. Svrha operacione semantike je da opiše *kako* se izvršavaju programi, ne samo koji su rezultati izvršavanja tog programa. Preciznije, ona treba da opiše kako se stanja menjaju tokom izvršavanja naredbe programa. U okviru ove semantike postoje:

- *Prirodna semantika* (ili veliki koraci (eng. *big-step semantics*)): svrha joj je da opiše kako su *ukupni rezultati* izvršavanja dobijeni.
- *Strukturna operaciona semantika* (ili mali koraci (eng. *small-step semantics*)): svrha joj je da opiše kako se *svaki korak* izvršava.

[**?**].

Jedna od osnovnih karakteristika imperativnih jezika je da ovi jezici imaju implicitno stanje i da se izvršavanje programa svodi na postepeno

menjanje tog stanja izvođenjem pojedinačnih naredbi [**?**]. Takođe, redosled naredbi je bitan [**?**]. S obzirom da sve ove karakteristike zadovoljava operaciona semantika, možemo reći da će ova semantika najviše odgovarati imperativnim jezicima.

Ponašanje se formalno može definisati korišćenjem apstraktnih mašina, formalnih automata, tranzicionih sistema... U daljem tekstu ćemo prikazati primer korišćenja apstraktne mašine za definiciju, kao i upotrebu tranzicionih sistema.

Za svaki programski jezik možemo definisati realnu ili apstraktnu mašinu, na kojoj ćemo pratiti promene stanja koje proizvode pojedini konstrukti programskog jezika. Ovako definisana mašina biće interpretator određenog programskog jezika. Naravno, za ovo nije dobro izabrati realnu mašinu, jer bi semantičke definicije bile vezane za jednu konkretnu mašinu, a to znači i za sve specifičnosti određene arhitekture računara. Zato je izbor apstraktne mašine pogodniji. Pokazaćemo ovo na jednom primeru. Posmatrajmo brojački programski ciklus u jeziku Pascal:

for i:= početak to kraj do
.
.
.

Značenje ovakvog zapisa ne može biti lako razumljivo. Stvarni mehanizam izvršavanja ciklusa je skriven za korisnika. Naravno, oni koji poznaju razne zapise programskih ciklusa mogu lako pretpostaviti značenje gore navedenog zapisa. Međutim, i za njih će biti teško da odgovore, šta će se dogoditi ako je vrednost promenljive *početak* veća od vrednosti promeljive *kraj*?

Zamislimo sada da naša apstraktna mašina raspolaže instrukcijama dodele, uslovnog i bezuslovnog prelaska. Tada bi se gore navedeni ciklus mogao zapisati instrukcijama apstraktne mašine u obliku:

```
                i:=početak
ciklus:     if i >kraj then goto izlaz
                    .
                    .
                    .
                i:=i+1
              goto ciklus
izlaz:          ...
```

Sada je jasno značenje programskog ciklusa, pa i to da se telo ciklusa neće izvršiti ako je vrednost promenljive *početak* veća od vrednosti promenljive *kraj*.

Ovo objašnjenje nam daje apstraktnu definiciju *kako* se program izvršava na mašini. Bitno je primetiti da jeste zaista apstrakcija: ignorišemo detalje kao što je upotreba registara i adresa za promenljive. Dakle, operaciona semantika je nezavisna od arhitekture mašine i načina implementacije.

Uvedimo pojam konfiguracije.

Definicija 2.1 *Konfiguracija je par* $\langle$ *c, ω* $\rangle$*, gde c predstavlja naredbu i pripada skupu naredbi C, dok ω predstavlja ulaznu vrednost za naredbu S, to jest početno stanje. Intuitivno, konfiguracija* $\langle$ *c, ω* $\rangle$ *predstavlja neko trenutno stanje izvršavanja programa, gde ω predstavlja trenutno stanje programa, a c predstavlja sledeću naredbu koja će se izvršiti.*

Opišimo sada značenje koje se predstavlja na isti način za obe vrste operacione semantike, prirodne i strukturno operacione, *tranzicionim sistemom* (eng. *transition system*) na primeru rada *While* petlje. Ona će imati dve vrste konfiguracija [**?**]:

- $\langle$ S, s $\rangle$ - Naredba S će biti izvršena od stanja s.
- s' predstavlja završno stanje.

Tranziciona relacija (eng. *transition relation*) će onda opisivati kako izgleda tok izvršavanja.

2.1.1 Prirodna semantika

Prirodna semantika je apstrakcija. U njoj je bitna veza između početnog i završnog stanja izvršavanja. Tada će tranziciona relacija prikazivati vezu između početnog stanja i završnog stanja za svaku naredbu. Tranziciju ćemo obeležavati kao:

$$\langle\ S,\ s\ \rangle \rightarrow s'$$

Intuitivno ovo znači da izvršavanje programa S sa ulaznim stanjem s će se završiti i rezultujuće stanje će biti s'. Drugi deo formule je samo delimično definisan, neophodno je naglasiti da je moguće da se program S izvrši i da rezultujuće stanje bude neko drugo stanje različito od s', kao i da je moguće da se program S uopšte ne izvrši. Pravo značenje programa S bi tada bilo:

$$\{(s,\ s') : \langle S,\ s\rangle \rightarrow s'\}$$

pa nam semantička pravila omogućavaju da generišemo skup tačnih to jest istinitih iskaza, gde su svi ostali iskazi netačni po definiciji.[**?**] Definicija za $\rightarrow$ je prikazana preko pravila datih u tabeli 1. *Pravilo* generalno ima formu

$$\frac{\langle S_1,s_1\rangle \rightarrow s'_1,\ldots,\langle S_n,s_n\rangle \rightarrow s'_n}{\langle S,s\rangle \rightarrow s'}$$

gde S_1, ..., S_n nazivamo neposrednim konstituentima (eng. *immediate constituents*) od S. Pravilo se sastoji iz određenog broja *premisa* (nalaze se iznad linije) i jednog *zaključka* (nalazi se ispod linije).

Pravilo takođe može imati određeni broj *uslova* (nalaze se sa desne strane linije) koji moraju biti ispunjeni kako bi se primenilo pravilo. Pravilo sa praznim skupom premisa se naziva *aksiom.* [**?**]

Kada se koriste aksiomi i pravila da se izvede tranzicija $\langle$ *S, s* $\rangle \rightarrow s'$, to jest da se dokaže iskaz, dobija se *stablo izvođenja. Koren* stabla izvođenja je upravo $\langle$ *S, s* $\rangle \rightarrow s'$, dok su listovi instance aksioma. Unutrašnji čvorovi su zaključci instanciranih pravila i njihova neposredna deca će biti odgovarajuće premise. Stablo izvođenja se naziva *jednostavnim stablom* ako je instanca aksioma, inače se naziva *kompozitnim stablom.*

Primer 2.1 *(z := x, x := y); y := z*
Neka s_0 *bude stanje koje mapira sve promenljive osim x i y u 0 i ima* s_0 $x = 5$ *i* s_0 $y = 7$*. Tada dobijamo sledeće stablo izvođenja:*

Tablica 1: Prirodna semantika za While petlju

$[ass_{ns}]$	$\langle x := a, s\rangle \rightarrow s[x \mapsto A[[a]]s]$
$[skip_{ns}]$	$\langle skip, s\rangle \rightarrow s$
$[comp_{ns}]$	$\frac{\langle S_1,s\rangle \rightarrow s', \langle S_2,s'\rangle \rightarrow s''}{\langle S_1;S_2,s\rangle \rightarrow s''}$
$[if^{tt}_{ns}]$	$\frac{\langle S_1,s\rangle \rightarrow s'}{\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle \rightarrow s'}\ if\ B[[b]]s\ =\ tt$
$[if^{ff}_{ns}]$	$\frac{\langle S_2,s\rangle \rightarrow s'}{\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle \rightarrow\ s'}\ if\ B[[b]]s\ =\ ff$
$[while^{tt}_{ns}]$	$\frac{\langle S,s\rangle \rightarrow s', \langle\ while\ b\ do\ S,\ s'\rangle \rightarrow\ s''}{\langle\ while\ b\ do\ S,\ s\rangle \rightarrow\ s''}\ if\ B[[b]]s\ =\ ff$
$[while^{ff}_{ns}]$	$\langle\ while\ b\ do\ S,\ s\rangle \rightarrow\ s\ if\ B[b]s\ =\ ff$

$$\frac{\dfrac{\langle z{:=}x,s_0\rangle \rightarrow s_1 \quad \langle x{:=}y,s_1\rangle \rightarrow s_2}{\langle z{:=}x,x{:=}y,s_0\rangle \rightarrow s_2} \quad \langle y{:=}z,s_2\rangle \rightarrow s_3}{\langle (z{:=}x;x{:=}y);y{:=}z,s_0\rangle \rightarrow s_3}$$

Posmatrajmo problem konstruisanja stabla izvođenja za datu naredbu S i stanje s. Najbolji pristup ovakvom problemu jeste konstruisati stablo od korena *nagore*. Dakle, počinjemo pronalaskom aksioma ili pravila sa zaključkom gde se leva strana slaže sa konfiguracijom $\langle\ S,\ s\ \rangle$. Imaćemo dva slučaja:

- Ako je u pitanju aksiom i ako su uslovi aksioma ispunjeni onda možemo da zaključimo završno stanje i konstrukcija stabla izvođenja je gotova.
- Ako je u pitanju pravilo, sledeći korak je pokušati konstruisati stablo izvođenja za premise datog pravila. Kad se ovaj deo odradi, neophodno je proveriti da li su uslovi pravila ispunjeni i tek onda možemo zaključiti završno stanje $\langle\ S,\ s\ \rangle$.

Više o ovoj temi se može pronaći u [**?**, **?**].

2.1.2 Strukturna operaciona semantika

Prirodna semantika nekada nije pogodna za obimnu analizu, tada se uglavnom koristi *strukturna operaciona semantika* (skraćeno SOS), kojom je moguće opisati imperativne programe kao i neke složene slučajeve, kao što su na primer, semantika pokazivača i semantika višenitne obrade ili čak baratati sa ,,goto" naredbama. U SOS-u ćemo tranzicionu relaciju zapisivati kao:

$$\langle\ S,\ s\ \rangle \Longrightarrow \gamma$$

ovo treba razmatrati kao *prvi korak* izvršavanja programa S u stanju s koji vodi do stanja γ [**?**].
Možemo razlikovati dva slučaja za γ:

1. $\gamma = \langle\ S',\ s'\ \rangle$: izvršavanje programa S sa ulaznim stanjem s *nije završeno*, i ostatak izračunavanja će biti izraženo srednjom konfiguracijom $\langle\ S',\ s'\ \rangle$.
2. $\gamma = s'$: izvršavanje programa S sa ulaznim stanjem s se završilo sa završnim stanjem s'.

U slučaju da resultat izvršavanja $\langle$ *S', s'* $\rangle$ nije dostupan kažemo da je γ *zaglavljena* (eng. *stuck*) konfiguracija i da nema sledećih tranzicija. *Značenje* programa P za ulazno stanje s je skup *završnih* stanja (kao i zaglavljenih konfiguracija) koji mogu biti izvršavani u proizvoljnom redosledu u konačnom broju koraka.

Primer 2.2 *Posmatrajmo ponovo slučaj While-a.*
Prva dva aksioma $[ass_{sos}]$ *i* $[skip_{sos}]$ *su nepromenjena jer se oba izraza izvršavaju u potpunosti u jednom koraku.*

- $[ass_{sos}]$ $\quad \langle\ x := a,\ s\ \rangle \Longrightarrow s[x \mapsto A[a]]$
- $[skip_{sos}]$ $\quad \langle\ skip,\ s\ \rangle \Longrightarrow s$

Druga dva pravila $[comp^1_{sos}]$ *i* $[comp^2_{sos}]$ *za programe* $S_1;S_2$ *govore da izvršavanje počinje prvim korakom iz* S_1 *i ulaznim stanjem s. Tada postoje dva moguća završna stanja u zavisnosti od dva slučaja:*

- $[comp^1_{sos}]$ *- Izvršavanje programa* S_1 *nije završeno. Tada je neophodno izvršiti taj program pre početka izvršavanja programa* S_2*. U ovom slučaju prvi korak* $\langle$ *S, s* $\rangle$ *je srednja konfiguracija* $\langle$ S_1*', s'* $\rangle$ *i sledeća konfiguracija je* $\langle$ S_1*';*S_2*,s'* $\rangle$*.*
  $[comp^1_{sos}]$ $\quad \dfrac{\langle S_1,s\rangle \Longrightarrow \langle S_1',s'\rangle}{\langle S_1;S_2,s\rangle \Longrightarrow \langle S_1';S_2,s'\rangle}$
- $[comp^2_{sos}]$ *- Izvršavanje programa* S_1 *je završeno. Tada možemo početi sa izvršavanjem programa* S_2*. U slučaju da je rezultat izvršavanja programa* S_1 *sa ulaznim stanjem s stanje s' tada je sledeća konfiguracija* $\langle$ S_2*, s'*$\rangle$
  $[comp^2_{sos}]$ $\quad \dfrac{\langle S_1,s\rangle \Longrightarrow s'}{\langle S_1;S_2,s\rangle \Longrightarrow \langle S_2,s'\rangle}$

Prvi korak uslovnog grananja počinje sa ispitivanjem istinitosne vrednosti izraza b-a, zatim grananjem u zavisnosti od ishoda:

- $[if^{tt}_{sos}]$ $\langle$ *if b then* S_1 *else* S_2*, s* $\rangle \Longrightarrow \langle$ S_1*, s* $\rangle$ $\qquad$ *if* $B[b]s = tt$
- $[if^{ff}_{sos}]$ $\langle$ *if b then* S_1 *else* S_2*, s* $\rangle \Longrightarrow \langle$ S_2*, s* $\rangle$ $\qquad$ *if* $B[b]s = ff$

Prvi korak kod ,,while" petlje jeste odrediti uslov kada se izlazi iz petlje, zatim u sledećem koraku izvršavanja treba proveriti da li je ispunjen taj uslov i da li je samim tim moguće nastaviti izvršavanje petlje.

$[while_{sos}]$ $\langle$ *while b do S,s* $\rangle \Longrightarrow \langle$ *if b then (S; while b do S) else skip,s* $\rangle$

Više o navedenom možete pronaći u [**?**] i podstičemo čitaoca da pogleda navedene izvore.

2.2 Denotaciona semantika

Nastala 1960. godina od strane Kristofera Strejčija (eng. *Christopher Strachey*) i njegove istraživačke grupe na Oksfordu [**?**], *denotaciona semantika* predstavlja jednu vrstu reakcije na operacionu semantiku za koju se smatra da sadrži puno informacija. Naziv je dobila po engleskoj reči označiti (eng. *denote*) jer pridružuje značenja sintaksnim definicijama jezika. Alternativno, može se nazivati i *matematička semantika* zbog njene okrenutosti matematičkim formalizmima pri definisanju ove formalne semantike. Jedan način definisanja denotacione semantike je dat u sledećoj definiciji.

Definicija 2.2 *Pristup formalizaciji semantike konstruisanjem matematičkih objekata koji opisuju značenje jezika naziva se* ***denotaciona semantika*** *[**?**].*

Dok se u operacionoj semantici vodilo računa o koracima izvršavanja, u denotacionoj to postaje nebitno. Na primer, značenje izraza $(15 + 3) * (2 + 2)$ jeste 72 i ne treba obraćati pažnju na unutrašnja izračunavanja. Bitan je samo efekat koji izvršavanje programa proizvodi, odnosno odnos između početnog i završnog stanja programa. Za posmatranje ovog efekta potrebno je uočiti odnos između sintakse i semantike programskog jezika.

Ideja ove semantike je da poveže svaki deo programskog jezika sa nekim matematičkim objektom kao što je broj ili funkcija. Odavde se jasno vidi da je potrebno raščlaniti programski jezik na sintaksne delove (to nam pruža apstraktna sintaksa) i svakom delu dodeliti značenje. Svaka sintaksna definicija se tretira kao objekat na koji se može primeniti funkcija koja taj objekat preslikava u matematički objekat koji definiše značenje [**?**]. Dodeljivanjem značenja delovima programa dodeljuje se značenje celokupnom programu što nam govori o najvažnijem aspektu denotacione semantike.

Definicija 2.3 *Semantika jedne programske celine definisana je preko semantike njenih poddelova. Ova osobina denotacione semantike naziva se* ***kompozitivnost****.*

Ovo znači da ukoliko se zameni jedan deo programske celine sa delom koji ima isto značenje, neće se promeniti značenje cele programske celine. Gore pomenuti izraz je imao semantičku vrednost 72, a to isto značenje ima i izraz $(16+2)*(2+2)$. To znači da se semantika izraza nije promenila iako su zamenjeni delovi izraza. Nije došlo do promene jer $15 + 3$ ima isto značenje kao i $16 + 2$. Treba se još pozabaviti dodeljivanjem semantičke vrednosti delovima programske celine.

Nekim sintaksnim delovima programa je lako dodeliti semantičku vrednost. Takvi su brojevi ili aritmetički operatori jer oni već imaju svoje matematičko značenje. Ali neke sintaksne definicije poput rekurzije ili goto naredbe je teško videti kroz matematičko značenje. Daćemo primer definisanja denotacione semantike aritmetičkih izraza, dok se o rekurziji ili nekim naprednijim primerima može pročitati više u [**?**].

Potrebno je prvo definisati apstraktnu sintaksu aritmetičkih izraza. Neka su podržani samo prirodni brojevi i od aritmetičkih operatora +. Ovo znači da će semantička vrednost nekog izraza biti prirodan broj. Primer definicije apstraktne sintakse ovakvih aritmetičkih izraza dat je u nastavku.

Sintaksni domeni i pravila:

$B : Broj$ B je nenegativan broj
$C : Cifra$ C je cifra 0,1...,9
$I : Izraz$
$Broj ::== Cifra|BrojCifra$
$Cifra ::== 0|1|2|3|4|5|6|7|8|9$
$Izraz ::== Broj|Izraz + Izraz$

Sledeći korak jeste definisanje matematičkih objekata koji će predstavljati semantičke vrednosti. Ti matematički objekti nazivaju se **semantički domeni**. Njihova kompleksnost zavisi od toga koliko je kompleksan programski jezik kojem dajemo značenje. U našem jednostavnom primeru, kao što je već rečeno, semantička vrednost može biti samo prirodan broj.

Semantički domeni

$N = 0, 1, 2, 3, ...$ skup prirodnih brojeva

Posle uvedenih objekata, treba uvesti neke matematičke funkcije koje će davati značenje prethodno uvedenim sintaksnim definicijama. Takve funkcije se nazivaju **funkcije značenja** (eng. *meaning functions*). Definicije funkcija koje su potrebne za naš jednostavan primer su date u nastavku.

Funkcije značenja

$povezibn : B \to N$ unarna funkcija - povezuje broj sa N
$povezicn : C \to N$ unarna funkcija - povezuje cifru sa N
$semantika : I \to N$ unarna funkcija - povezuje izraz sa N
$plus : N \times N \to N$ binarna funkcija plus - isto što i +
$pom : N \times N \to N$ binarna funkcija pom - isto što i *
$povezicn[[0]] = 0, ..., povezicn[[9]] = 9$
$povezibn[[C]] = povezicn[[C]]$
$povezibn[[BC]] = plus(pom(10, povezibn[[B]]), povezibn[[C]])$
$semantika[[B]] = povezibn[[B]]$
$semantika[[I1 + I2]] = plus(semantika[[I1]], semantika[[I2]])$

Primetimo da su korišćene zagrade [[,]] koje imaju ulogu da razdvoje semantički deo od sintaksnog dela. U okviru zagrada nalazi se sintaksni deo definicija. Primer koji oslikava korišćenje denotacione semantike je dat u nastavku.

Primer 2.3 *Pronaći značenje izraza 2+32+61.*

Rešenje:

semantika[[2+32+61]] *= plus(semantika[[2+32]],semantika[[61]])*
= plus(plus(semantika[[2]],semantika[[32]]),povezibn[[61]])
= plus(plus(2,32),61)
= plus(2+32,61)
= plus(34+61) = 34+61 = 95

jer je:

semantika[[2]] *= povezibn[[2]] = povezicn[[2]] = 2*

semantika[[32]] *= povezibn[[32]]*
= plus(pom(10,povezibn[[3]]),povezibn[[2]])
=plus(pom(10,povezicn[[3]]),povezicn[[2]])
*=plus(pom(10,3),2) = plus(10*3,2)*
= plus(30,2) = 30+2 = 32

semantika[[61]] *= povezibn[[61]]*
=plus(pom(10,povezibn[[6]]),povezibn[[1]])
=plus(pom(10,povezicn[[6]]),povezicn[[1]])
*=plus(pom(10,6),1) = plus(10*6,1)*
= plus(60,1) = 60+1 = 61

S obzirom na to da se funkcionalno programiranje zasniva na pojmu matematičkih funkcija i da se izvršavanje programa svodi na evaluaciju funkcija [**?**], denotaciona semantika je najbolja za definisanje semantike funkcionalnih programskih jezika.

Prednost denotacione semantike je u tome što apstrahuje kako se programi izvršavaju. Analiziranje programa se svodi na analiziranje matematičkih objekata, što olakšava stvar. Denotaciona semantika ima veliku primenu u konstrukciji programa za prevođenje [**?**].

2.3 Aksiomatska semantika

Za nastanak i razvoj *aksiomatske semantike* su zaslužni pre svega Robert Flojd (eng. *Robert Floyd*), Toni Hor (eng. *Antony Hoare*) i Edsger Dijkstra (hol. *Edsger Dijkstra*). Zasniva se na matematičkoj logici pa je kao takva apstraktnija od denotacione i operacione sematike. Ova semantika značenje programa zasniva na tvrdnjama o vezama koje ostaju iste svaki put kad se program izvrši [**?**]. Aksiomatska semantika razvija metode za proveru korektnosti programa. Za svaku kontrolnu strukturu i komandu se definišu logički izrazi. Ovi izrazi se nazivaju **tvrđenja** (eng. *assertions*) i u njima se zadaju ograničenja za promenljive koja se javljaju u tim kontrolnim strukturama i komandama.

Tvrđenja su data u obliku Horovih trojki:

{P}c{Q}

Definicija 2.4 ***Horova trojka*** $\{P\}C\{Q\}$ *opisuje kako izvršavanje dela koda menja stanje izračunavanja ako je ispunjen preduslov (eng.* precondition*)* $\{P\}$*, izvršavanje komande C vodi do postuslova (eng.* postcondition*)* $\{Q\}$ *[**?**] .*

Preduslov je logički izraz u kome se definišu ograničenja promenljivih pre izvršavanja komande, a postuslov definiše ograničenja promenljivih posle izvršavanja komande. Horove trojke se drugačije nazivaju i *parcijalna ispravnost specifikacije* (eng. *partial correctness specification*) . Ali one ne mogu da osiguraju da će se program završiti pa se zbog toga i nazivaju "parcijalnim".

Parcijalne ispravnosti specifikacija biće određene sistemom zaključivanja koji se sastoji iz skupa aksioma i pravila. Za sve konstruktore jednostavnog imperativnog programskog jezika, Horova logika obezbeđuje aksiome i pravila izvođenja [**?**]. Aksiomatski sistem za parcijalnu ispravnost je dat u Tabeli 2 ispod.

Tablica 2: Aksiomatski sistem za parcijalnu ispravnost

$[ass_p]$	$\{P[x \to [\![A]\!]\}\ x := a\ \{P\}$
$[skip_p]$	$\{P\}$ skip $\{P\}$
$[comp_p]$	$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\{R\}}$
$[if_p]$	$\frac{\{B[\![b]\!] \wedge P\}\ S_1\ \{Q\},\ \{\neg B[\![b]\!] \wedge P\}\ S_1\ \{Q\}}{\{P\}\ if\ b\ then\ S_1\ else\ S_2\ \{Q\}}$
$[while_p]$	$\frac{\{B[\![b]\!] \wedge P\}\ S\ \{P\}}{\{P\}\ while\ b\ do\ S\ \{\neg B[\![b]\!] \wedge P\}}$
$[cons_p p]$	$\frac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}$ if $P \Rightarrow P'$ and $Q \Rightarrow Q'$

Pored parcijalne ispravnosti specifikacija, imamo i *potpunu ispravnost naredbe* (eng. *total correctness statements*) koja osigurava da će se program završiti dok god preduslov važi.

Osim dokazivanja korektnosti programa i algoritama, uloga aksiomatske sematike je i dokazivanje ispravnosti harverdskih opisa (ili nalaženje bagova), proširena statička provera (npr. provera granice niza), dokumentacija programa i interfejsa [**?**] .

Preduslovi i postuslovi mogu se smatrati interfejsom ili ugovorom između programa i njegovih klijenata. Oni pomažu korisnicima da razumeju šta program treba da proizvede bez potrebe da shvati kako se program izvršava. Tipično, programeri ih pišu kao komentare za funkcije i funkcionišu kao dokumentacija i olakšavaju održavanje programa. Takve specifikacije su posebno korisne za bibliotečke funkcije za koje izvorni kod često nije dostupan korisnicima [**?**].

Način funkcionisanja ove semantike možemo prikazati u primeru sa faktorijalom koji sledi (primer je preuzet iz knjige [**?**]).

Primer 2.4 *Izračunati faktorijal*

$$\{x = n\}$$
$$y := 1; \textbf{while } \neg(x = 1) \textbf{ do } (y := x*y;\ x := x-1)$$
$$\{\ y=n!\ \ \text{and } n > 0\ \}$$

n je u primeru specijalna promenljiva koja se naziva logička promenljiva i koja, za razliku od programskih promenljivih, ne sme se pojaviti ni u jednoj naredbi koja se izvršava u programu i njena vrednost će uvek biti ista. Njena uloga je da pamte inicijalne vrednosti programskih promenljivih. Zapisali smo preduslov da promenljiva n ima jednaku vrednost kao i x u početnom stanju (tj. nego što program sa faktorijalom krene da se izvršava). S obzirom da program neće promeniti vrednost promenljive n, postuslov y=n! će izraziti da konačna vrednost y će biti jednaka faktorijalu početne vrednosti promenljive x, kada se izvršavanje programa završi.

3 Zaključak

Prilikom izučavanja programskih jezika treba obratiti pažnju kako na sintaksu, tako i na semantiku. Primena formalnih opisa semantike zahteva matematičko znanje i apstraktno razmišljanje, a kada je ono omogućeno, onda poznavanje osnova formalne semantike programskih jezika omogućava programerima da lakše shvate tok i rezultat izvršavanja programa.

Rad se zasniva samo na osnovama formalne semantike i služi da uvede i zainteresuje čitaoca za jednu od veoma kompleksnih tema programiranja. Dalje istraživanje bi išlo u dubinu, odnosno, trebalo bi istražiti kako se neki kompleksniji fragmenti programskih kodova semantički definišu iz sva tri ugla - operacionog, denotacionog i aksiomatskog. Takođe, mogao bi se pružiti uvid u primenu formalne semantike pri konstruisanju kompilatora.