Interpreter za parcijalno rekurzivne funkcije

Eterović, Marko

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Marko Eterović

INTERPRETER ZA PARCIJALNO REKURZIVNE FUNKCIJE

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Majci i ocu, za neprestanu podršku i ljubav kroz čitav život.
I Žani, uz koju je pisanje ovih stranica ljepše.

Sadržaj

Uvod

Za brojevnu funkciju $f\colon \mathbb{N}^k \to \mathbb{N}$ kažemo da je **izračunljiva** ako postoji algoritam za njeno izračunavanje. Ako uzmemo $f(x, y) = 5x + y^2$ kao primjer jedne brojevne funkcije, onda je vrlo jednostavno opisati pripadni algoritam za izračunavanje vrijednosti $f(x, y)$, za bilo koje unaprijed zadane $x, y \in \mathbb{N}$. Funkcija f se očito sastoji od osnovnih matematičkih operacija *zbrajanja, množenja* i *potenciranja*, koje se trebaju izvršiti u određenom redoslijedu. Znamo da je množenje zapravo pokrata za *ponovljeno zbrajanje,* dok je potenciranje pokrata za *ponovljeno množenje.* Zbog činjenice da je pojam „ponovljeno" izraziv u jeziku algoritama (recimo, pomoću petlji), dolazimo do sljedećeg zaključka: „*Ako postoji algoritam za zbrajanje, onda postoje algoritmi za množenje i potenciranje, pa posebno i algoritam za računanje vrijednosti funkcije* f." Naravno da algoritam za zbrajanje prirodnih brojeva postoji, ali i on ovisi o izračunljivosti nekih još jednostavnijih funkcija. To nas navodi da aksiomatski definiramo **inicijalne funkcije** — vrlo jednostavne funkcije čija je izračunljivost neupitna — te na njima „gradimo" sve kompliciranije izračunljive funkcije.

Definirat ćemo *tri osnovne operacije* na funkcijama odnosno relacijama: **kompoziciju**, **primitivnu rekurziju** i **minimizaciju**. Svaka od tih operacija ima određenu algoritamsku interpretaciju, što nam omogućuje da zadržimo poveznicu između funkcijskog jezika i ostalih uobičajenih (imperativnih) programskih jezika. Primjerice, kompozicija će odgovarati *slijednom* izvršavanju naredbi. U gornjem primjeru, zbog prioriteta operacija, želimo prvo izračunati vrijednosti $5x$ i y^2, kako bismo ih nakon toga mogli zbrojiti. Pokazuje se da je kompozicija idealna operacija za specifikaciju poretka u kojem se izračunavaju vrijednosti funkcija, jer funkciju f možemo zapisati kao $f(x, y) = \mathtt{add}(\mathtt{mul}(5, x), \mathtt{pow}(y, 2))$ — pri čemu računamo funkciju $\mathtt{add}$ tek nakon što su poznate vrijednosti oba njena argumenta.

Parcijalno rekurzivne funkcije su sve brojevne funkcije koje su nastale primjenom osnovnih operacija na inicijalnim funkcijama.

U ovom ćemo radu prvo detaljnije iznijeti teoriju parcijalno rekurzivnih funkcija, a nakon toga ćemo opisati proces izrade **interpretera** za parcijalno rekurzivne funkcije uz pomoć **vepra** (*framework* za interpretaciju programa koji je razvio V. Čačić). Interpreter omogućava korisniku definiranje funkcija i relacija nastalih raznim kombinacijama inicijalnih funkcija i osnovnih operacija (koje uključuju tri prije navedene, ali i neke druge često korištene operacije), kao i njihovo korištenje u kasnijim definicijama.

Poglavlje 1

Parcijalno rekurzivne funkcije

1.1 Osnovni pojmovi i oznake

Za bilo koji $k \in \mathbb{N}_+$, preslikavanje $f\colon \mathbb{N}^k \to \mathbb{N}$ zovemo **brojevna funkcija** (ponekad ćemo je zvati samo *funkcija*). Brojevna funkcija je *totalna* ako joj je domena čitav $\mathbb{N}^k$; skup svih totalnih funkcija označavamo s $\mathtt{Tot}$. Broj k je *mjesnost* funkcije f i označava dimenziju njene domene (u kojoj se nalaze uređene k-torke koje često označavamo s $\vec{x}^k$). Oznakom f^k naglašavamo da je f jedna k-mjesna funkcija. S $\otimes^k$ označavamo praznu funkciju mjesnosti k. Njena domena je prazan skup, odnosno $\otimes^k(\vec{x})$ nema vrijednost ni za jedan $\vec{x}^k \in \mathbb{N}^k$. Za relaciju $R^k \subseteq \mathbb{N}^k$ kažemo da je izračunljiva ako je njena *karakteristična funkcija* $\chi_R\colon \mathbb{N}^k \to \{0,1\}$ izračunljiva.

1.2 Inicijalne funkcije

Kao što je naglašeno u uvodu, kako bismo stvarali kompliciranije izračunljive funkcije od jednostavnijih, potrebne su nam vrlo jednostavne *inicijalne* funkcije čiju izračunljivost prihvaćamo aksiomatski u okviru funkcijskog modela računanja. Kao dodatni argument u korist njihove izračunljivosti, iskoristit ćemo programski jezik Python kao intuitivno prihvatljiv model opće izračunljivosti.

Definicija 1.2.1. *Neka je $k \in \mathbb{N}_+$. Za funkciju F^k kažemo da je **Python-izračunljiva** ako postoje Python-funkcija f^k (napisana u programskom jeziku Python) i broj $l < k$ tako da za sve $\vec{y}^k = (\vec{x}^l, y_{l+1}, \ldots, y_k)$ poziv $f(y_{l+1}, \ldots, y_k, *x)$ završi ako i samo ako je $\vec{y}^k \in \mathcal{D}_F$, i vraća upravo $F(\vec{y})$.*

Definicija 1.2.2. *Inicijalne funkcije su:*

- *nulfunkcija Z^1, s pravilom preslikavanja $Z(x) := 0,\ \forall x \in \mathbb{N}$;*

- *sljedbenik* Sc^1*, s pravilom preslikavanja* $\mathrm{Sc}(x) := x + 1,\ \forall x \in \mathbb{N}$;
- *za svaki* $k \in \mathbb{N}_+$*, za svaki* $n \in \{1, \ldots, k\}$*,* n*-ta* k*-mjesna koordinatna projekcija* I_n^k*, s pravilom preslikavanja* $\mathrm{I}_n(\vec{x}^k) = \mathrm{I}_n(x_1, x_2, \ldots, x_k) = x_n$.

Propozicija 1.2.3. *Svaka inicijalna funkcija je Python-izračunljiva.*

Dokaz. Za svaku inicijalnu funkciju ćemo napisati njoj ekvivalentnu funkciju u Pythonu koja ju računa. Nulfunkcija i sljedbenik su trivijalni, a koordinatne projekcije ćemo promatrati kao poseban slučaj. Problem je u tome što n u izrazu $\mathrm{I}_n(\vec{x}^k) = x_n$ predstavlja dio naziva funkcije, a ne argument te funkcije. Zbog toga ne postoji jedna ekvivalentna Python-funkcija, već više njih: po jedna ta svaki mogući n. To ćemo riješiti tako da definiramo tzv. *callable* klasu kojoj u konstruktor prosljeđujemo broj $n \geq 1$:

```
1 def Z(x):
2     return 0
3 def Sc(x):
4     return x + 1
5 class I:
6     def __init__(self, n):
7         self.n = n
8     def __call__(self, *x):
9         return x[self.n - 1]
```

Program 1.2.1: Inicijalne funkcije

Primjerice, za $n = 2$ i $\vec{x}^3 = (0, 5, 3)$, izraz $\mathrm{I}_2(\vec{x}^3)$ bismo mogli izračunati ovako:

```
1 I_2 = I(2)
2 I_2(0,5,3)  # 5
```

Program 1.2.2: Poziv inicijalne funkcije

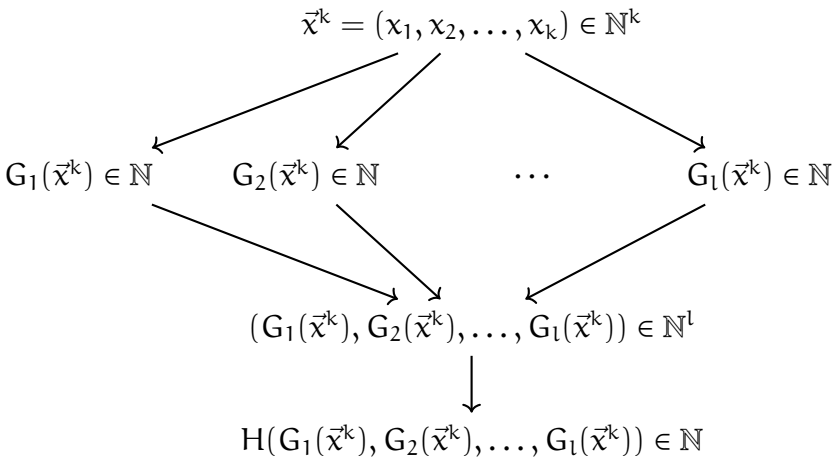
□

Za brojevnu funkciju F^{k+2}, vrijednost te funkcije u točki $(\vec{x}^k, y, z) \in \mathbb{N}^{k+2}$ označavamo s $\mathrm{F}(\vec{x}^k, y, z)$. Pritom smatramo da su parametri y i z u trenutnom okruženju „važniji" od parametara $\vec{x}^k = (x_1, \ldots, x_k)$, pa ih pišemo odvojeno od k-torke $\vec{x}$ koju zovemo *kontekst*. U svrhu usklađenosti sa sintaksom Pythona, u nastavku rada ćemo kontekst $\vec{x}$ u Python-kōdu prikazivati kao zadnji argument funkcije (`*x` označava da funkcija `F` može primiti proizvoljno mnogo argumenata) kao u prethodnom dokazu, dok ćemo se na svim ostalim mjestima držati uobičajene konvencije iz [3] — konteksta kao prvog (možda višemjesnog) argumenta funkcije.

1.3 Kompozicija

Kompozicija je najjednostavnija od triju osnovnih operacija spomenutih u uvodu, ali njena precizna definicija sadrži nekoliko tehničkih detalja.

Zbog činjenice da su brojevne funkcije H^l i G^k preslikavanja s višemjesnom domenom i jednomjesnom kodomenom, jasno je da oznaka $H(G(\vec{x}^k))$ ne bi bila dovoljno precizna. Naime, $G(\vec{x}^k)$ je prirodni broj, što znači da bi mjesnost l funkcije H morala biti jednaka 1. Kako bismo osigurali da vanjska funkcija H također može primiti više argumenata, komponirat ćemo je s (potencijalno) više funkcija odjednom, u oznaci $H \circ (G_1, G_2, \ldots, G_l)$. Za svaku *koordinatnu* funkciju G_i zasebno računamo vrijednost $G_i(\vec{x}^k)$ te tako dobivamo l argumenata koji su potrebni za poziv funkcije H. Ovakvim pristupom smo povećali broj funkcija koje trebamo navesti u kompoziciji, ali smo zato osigurali da sve te funkcije zaista mogu biti brojevne (jer svaka vraća po jedan prirodni broj), te da je rezultirajuća kompozicija $H \circ (G_1, G_2, \ldots, G_l) \colon \mathbb{N}^k \to \mathbb{N}$ također brojevna.



Dijagram 1.1. Prikaz računanja vrijednosti kompozicije s argumentima $\vec{x}^k$.

Drugi problem koji trebamo razriješiti tiče se načina *evaluacije* funkcija koje nisu totalne. Konkretno, potrebno je odabrati između *lijene* i *marljive* evaluacije te sukladno tom odabiru definirati kompoziciju. Pogledajmo razliku ta dva pristupa na primjeru funkcije (kompozicije) $I_1 \circ (Sc, \otimes)$.

Lijena evaluacija se zasniva na principu da je za računanje vrijednosti neke funkcije potrebno prethodno izračunati samo one vrijednosti njenih argumenata koje se stvarno pojavljuju u njenom pravilu preslikavanja, dok se ostali argumenti zanemaruju. U pravilu preslikavanja funkcije I_1 se pojavljuje samo prvi argument, pa bi lijena evaluacija izgledala ovako:

$$I_1(Sc(x), \otimes(x)) = Sc(x) = x + 1, \quad \text{za bilo koji } x \in \mathbb{N}.$$

Nasuprot tome, marljiva evaluacija nalaže prvo izračunati vrijednosti *svih* argumenata funkcije, pa tek onda izračunati vrijednost te funkcije. Budući da $\otimes(x)$ nikad nema vrijednost, marljivom evaluacijom možemo zaključiti da je domena funkcije $I_1 \circ (Sc, \otimes)$ prazan skup, odnosno

$$I_1(Sc(x), \otimes(x)) \text{ nema vrijednost ni za koji } x \in \mathbb{N}.$$

U skladu s [3], u ovom radu ćemo izabrati **marljivu** evaluaciju i po tom principu računanja izraditi interpreter.

Definicija 1.3.1. *Neka su* $k, l \in \mathbb{N}_+$ *i* $G_1^k, G_2^k, \ldots, G_l^k$ *i* H^l *funkcije. Za funkciju* F^k *s domenom*

$$\mathcal{D}_F := \left\{ \vec{x} \in \bigcap\nolimits_{i=1}^{l} \mathcal{D}_{G_i} \;\middle|\; (G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x})) \in \mathcal{D}_H \right\} \tag{1.1}$$

i pravilom preslikavanja

$$F(\vec{x}) := H(G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x})), \ \forall \vec{x} \in \mathcal{D}_F \tag{1.2}$$

kažemo da je dobivena ***kompozicijom*** *iz funkcija* $G_1^k, G_2^k, \ldots, G_l^k$ *i* H_l,
i pišemo $F = H \circ (G_1, G_2, \ldots, G_l)$.

Skup funkcija $\mathcal{F}$ *je* ***zatvoren na kompoziciju*** *ako za sve* $k, l \in \mathbb{N}_+$,
za sve $G_1^k, G_2^k, \ldots, G_l^k, H^l \in \mathcal{F}$, *vrijedi* $H \circ (G_1, G_2, \ldots, G_l) \in \mathcal{F}$.

Dakle, ako je $F = H \circ (G_1, G_2, \ldots, G_l)$ neka kompozicija i ako je $\vec{x} \in \mathbb{N}^k$, izraz $F(\vec{x})$ ima vrijednost ako je $\vec{x}$ unutar *svih* domena unutarnjih funkcija $\mathcal{D}_{G_i}$ i ako je pripadna l-torka $(G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x}))$ unutar domene $\mathcal{D}_H$ vanjske funkcije H.

U sljedećem primjeru pokazujemo da se bilo koja konstantna funkcija može dobiti kompozicijom iz inicijalnih funkcija.

Primjer 1.3.2. $C_3^2 := Sc \circ Sc \circ Sc \circ Z \circ I_1^2$. *Za sve* $(x, y) \in \mathbb{N}^2$ *računamo:*

$$C_3^2(x, y) = (Sc \circ Sc \circ Sc \circ Z)(I_1^2(x, y)) \overset{1.2.2}{=} (Sc \circ Sc \circ Sc \circ Z)(x) \overset{1.2.2}{=} (Sc \circ Sc \circ Sc)(0) \overset{1.2.2}{=} 3.$$

Vidljivo je da se kompozicijom iz inicijalnih funkcija još uvijek mogu dobiti samo vrlo jednostavne funkcije. Zato je potrebno uvesti nove (kompliciranije) operacije koje će znatno proširiti skup funkcija koje promatramo.

1.4 Primitivna rekurzija

Sljedeća operacija koju uvodimo je *primitivna rekurzija*. Ona je ekvivalentna *petljama* u standardnim programskim jezicima, i to onima čiji je broj ponavljanja unaprijed poznat (npr. ograničena `for`-petlja). Primitivna rekurzija će nam omogućiti da u funkcijskom modelu izračunavanja definiramo osnovne matematičke operacije poput zbrajanja, (ograničenog) oduzimanja i množenja.

Definicija 1.4.1. *Neka je* $k \in \mathbb{N}_+$ *te neka su* G^k *i* H^{k+2} *totalne funkcije. Za funkciju* F^{k+1} *definiranu s*

$$F(\vec{x}, 0) := G(\vec{x}), \ \forall \vec{x} \in \mathbb{N}, \tag{1.3}$$

$$F(\vec{x}, Sc(y)) := H(\vec{x}, y, F(\vec{x}, y)), \ \forall y \in \mathbb{N}, \ \forall \vec{x} \in \mathbb{N}, \tag{1.4}$$

kažemo da je dobivena ***primitivnom rekurzijom*** *iz funkcija* G *i* H.

Skraćeno pišemo $F := G \operatorname{PR} H$ *i smatramo da operator* PR *ima niži prioritet od* $\circ$.
Skup funkcija $\mathcal{F}$ *je* ***zatvoren na primitivnu rekurziju*** *ako za svaki* $k \in \mathbb{N}_+$,
za sve $G^k, H^{k+2} \in \mathcal{F} \cap \mathrm{Tot}$, *vrijedi* $G \operatorname{PR} H \in \mathcal{F}$.

Izraz (1.3) zovemo *inicijalizacija* ili *početni uvjet* primitivne rekurzije, dok (1.4) zovemo *korak* ili *tijelo* primitivne rekurzije.

Definicija 1.4.1 isprva izgleda komplicirano, pa ju je najbolje objasniti na jednostavnom primjeru; definirat ćemo funkciju za zbrajanje preko primitivne rekurzije i onda opisati proces izračunavanja zbroja nekih konkretnih prirodnih brojeva.

Primjer 1.4.2. *Funkcija za zbrajanje dva prirodna broja je zadana s* $\mathtt{add}^2 = I_1^1 \operatorname{PR} Sc \circ I_3^3$, *odnosno:*

$$\mathtt{add}(x, 0) = I_1^1(x) = x, \ \forall x \in \mathbb{N}, \tag{1.5}$$

$$\mathtt{add}(x, Sc(y)) = (Sc \circ I_3^3)(x, y, \mathtt{add}(x, y)) = Sc(\mathtt{add}(x, y)), \ \forall x, y \in \mathbb{N}. \tag{1.6}$$

Lako je vidjeti da je prethodni primjer usklađen s definicijom 1.4.1; inicijalizacija i korak su redom zadani s $G^1 = I_1^1$ i $H^3 = Sc \circ I_3^3$. Preostalo je samo uvjeriti se da ovako zadana funkcija $\mathtt{add}$ ispravno zbraja prirodne brojeve: u tu svrhu izračunajmo vrijednost $\mathtt{add}(5, 3)$.

Kada pogledamo pravila (1.5) i (1.6), možemo zaključiti da postoje dva razumna pristupa kojima možemo „raspisati" izračunavanje izraza $\mathtt{add}(5, 3)$.

Prvi pristup je da krenemo od izraza $\mathtt{add}(5, 3)$ i raspisujemo ga po pravilu (1.6) dok ne dođemo do izraza $\mathtt{add}(5, 0)$, čija nam je vrijednost poznata po pravilu (1.5):

$$\begin{aligned}\mathtt{add}(5,3) &\overset{(1.6)}{=} Sc(\mathtt{add}(5,2)) \overset{(1.6)}{=} Sc(Sc(\mathtt{add}(5,1)) \\ &\overset{(1.6)}{=} Sc(Sc(Sc(\mathtt{add}(5,0)))) \overset{(1.5)}{=} Sc(Sc(Sc(5))) \\ &\overset{1.2.2}{=} Sc(Sc(6)) \overset{1.2.2}{=} Sc(7) \overset{1.2.2}{=} 8.\end{aligned}$$

To je klasični *rekurzivni* pristup u modernim programskim jezicima; točne vrijednosti izraza $\mathtt{add}(5, 3)$, $\mathtt{add}(5, 2)$ i $\mathtt{add}(5, 1)$ nam nisu poznate u trenutku kada na njih naiđemo, ali ih svejedno raspisujemo po pravilu (1.6) jer „vjerujemo" da ćemo u nekom trenutku doći do nekog početnog uvjeta koji će imati unaprijed zadanu vrijednost (u ovom slučaju $\mathtt{add}(5, 0) = 5$). Jednom kada stignemo do početnog uvjeta, vraćamo se unatrag

i računamo sve pozive (funkcije Sc) koji su bili „na čekanju". U Pythonu bi to izgledalo ovako:

```
1 def add(x, y):
2     if y == 0: return x
3     return Sc(add(x, y-1))
```

Program 1.4.1: Rekurzivni pristup

Ipak, ovaj pristup nije u potpunosti u skladu s definicijom primitivne rekurzije. Naime, u definiciji 1.4.1 podrazumijevamo da je inicijalizacija $G(x) = add(x, 0) = x$ ujedno i *polazna vrijednost* od koje mora krenuti svako izračunavanje, i da se za *svaku iduću* vrijednost $add(x, Sc(y))$, gdje je $y \in \mathbb{N}$, moraju izračunati *sve prethodne* vrijednosti $add(x, 1), \ldots, add(x, y)$, jedna po jedna (prateći pravilo (1.6)):

$$add(5, 0) \overset{(1.5)}{=} 5, \tag{1.7}$$

$$add(5, 1) \overset{(1.6)}{=} Sc(add(5, 0)) \overset{(1.7)}{=} Sc(5) = 6, \tag{1.8}$$

$$add(5, 2) \overset{(1.6)}{=} Sc(add(5, 1)) \overset{(1.8)}{=} Sc(6) = 7, \tag{1.9}$$

$$add(5, 3) \overset{(1.6)}{=} Sc(add(5, 2)) \overset{(1.9)}{=} Sc(7) = 8. \tag{1.10}$$

Ovo je *primitivno rekurzivni* pristup; ne izvodimo prave rekurzivne pozive funkcija, već samo pamtimo posljednju izračunatu vrijednost i uz pomoć nje računamo iduću. Sada je mnogo jasnija poveznica između primitivne rekurzije i programskih petlji, što se još bolje vidi u sljedećem kōdu:

```
1 def add(x, y):
2     z = x
3     for i in range(y):
4         z = Sc(z)
5     return z
```

Program 1.4.2: Primitivno rekurzivni pristup

Ako se prisjetimo da su rekurzivni funkcijski pozivi memorijski puno zahtjevniji od iteriranja kroz petlju, jasno je da bismo vrijednosti poput $add(1, 10000)$ mnogo lakše izračunali primitivno rekurzivnim pristupom.

Pojasnimo još i značenje pojedinih argumenata funkcija u izrazima (1.3) i (1.4). Kao što smo već rekli, uređenu k-torku $\vec{x}$ koja se pojavljuje kao argument u svim pozivima (F, G i H) zovemo *kontekst*. To je unaprijed poznat konačni niz vrijednosti koji se ne mijenja tijekom izračuna primitivne rekurzije, nego se samo pojavljuje u pravilima preslikavanja tih funkcija. Primitivnu rekurziju provodimo po *zadnjem* argumentu funkcije F označenom s y. Naposljetku, $F(\vec{x}, y)$ kao zadnji argument funkcije H reflektira činjenicu da je za izračun vrijednosti $F(\vec{x}, Sc(y))$ potrebna prethodno izračunata vrijednost $F(\vec{x}, y)$.

Jasno je da smo funkciju $\mathtt{add} = I^1_1 \operatorname{PR} Sc \circ I^3_3$ dobili određenom kombinacijom inicijalnih funkcija, kompozicije i primitivne rekurzije. Iduća propozicija nam garantira da smo takvim postupkom sačuvali izračunljivost.

Propozicija 1.4.3. *Skup Python-izračunljivih funkcija je zatvoren na kompoziciju i primitivnu rekurziju.*

Dokaz. Radi ilustracije, tvrdnju za kompoziciju ćemo dokazati za fiksni $l = 3$ (sasvim je jasno kako bi dokaz izgledao za druge l). Neka je $k \in \mathbb{N}_+$ te G^k_1, G^k_2, G^k_3 i H^3 Python-izračunljive funkcije. To znači da postoje Python-funkcije (označimo ih s `G_1`, `G_2`, `G_3` i `H`) koje računaju redom brojevne funkcije G^k_1, G^k_2, G^k_3 i H^3. Tada njihovu kompoziciju $F := H \circ (G_1, G_2, \ldots, G_l)$ računa Python-funkcija `F` zadana s

```
def F(*x):
    return H(G_1(*x), G_2(*x), G_3(*x))
```

Program 1.4.3: Python-funkcija za kompoziciju

Pretpostavimo sada da su zadane Python-izračunljive totalne funkcije G^k i H^{k+2}. To znači da postoje Python-funkcije `G` i `H` koje računaju brojevne funkcije G^k i H^{k+2}. Sada po uzoru na definiciju 1.4.1 i kōd za funkciju `add`, možemo napisati Python-funkciju `F` koja računa brojevnu funkciju dobivenu primitivnom rekurzijom $F := G \operatorname{PR} H$:

```
def F(y, *x):
    z = G(*x)
    for i in range(y):
        z = H(i, z, *x)
    return z
```

Program 1.4.4: Python-funkcija za primitivnu rekurziju

Navedeni kōd će biti vrlo relevantan u idućem poglavlju, jer ćemo upravo po uzoru na njega „ugraditi" primitivnu rekurziju u interpreter. □

Propozicija 1.2.3 nam garantira da su inicijalne funkcije izračunljive, pa po propoziciji 1.4.3 zaključujemo da su funkcije $Sc \circ I^3_3$ i $\mathtt{add} = I^1_1 \operatorname{PR} Sc \circ I^3_3$ također izračunljive. Iduća definicija daje službeni naziv svim funkcijama koje možemo dobiti ovakvim postupkom.

Definicija 1.4.4. *Skup **primitivno rekurzivnih** funkcija je najmanji skup funkcija koji sadrži sve inicijalne funkcije te je zatvoren na kompoziciju i primitivnu rekurziju.*

Sljedeća karakterizacija nam daje lakši način da za konkretne funkcije utvrdimo da su primitivno rekurzivne. Njen dokaz se može naći u [3].

Propozicija 1.4.5. *Neka je* F *brojevna funkcija. Tada je* F *primitivno rekurzivna ako i samo ako se* F *može dobiti iz inicijalnih funkcija pomoću konačno mnogo primjena kompozicije i primitivne rekurzije.*

Primjenom prethodne propozicije možemo zaključiti da je $\mathtt{add} = \mathrm{I}_1^1 \,\mathbf{PR}\, \mathrm{Sc} \circ \mathrm{I}_3^3$ primitivno rekurzivna. Također, ta propozicija nam omogućava da, kad stvaramo nove funkcije uz pomoć kompozicije i primitivne rekurzije, više ne moramo kretati od inicijalnih funkcija, već da koristimo funkcije za koje smo prije utvrdili da su primitivno rekurzivne.
U skladu s time, konstruirajmo funkcije za množenje i potenciranje.

Primjer 1.4.6. *Funkcija za množenje zadana je s* $\mathtt{mul}^2 = \mathsf{Z} \,\mathbf{PR}\, \mathtt{add}^2 \circ (\mathrm{I}_1^3, \mathrm{I}_3^3)$*, odnosno:*

$$\mathtt{mul}(x, 0) = \mathsf{Z}(x) = 0, \ \forall x \in \mathbb{N}, \tag{1.11}$$

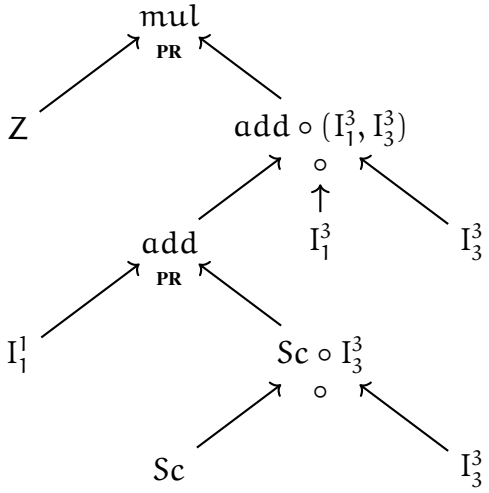
$$\mathtt{mul}(x, \mathrm{Sc}(y)) = \mathtt{add}(x, \mathtt{mul}(x, y)), \ \forall x, y \in \mathbb{N}, \tag{1.12}$$

dok je funkcija za potenciranje zadana s $\mathtt{pow}^2 = \mathrm{C}_1^1 \,\mathbf{PR}\, \mathtt{mul}^2 \circ (\mathrm{I}_1^3, \mathrm{I}_3^3)$*, odnosno:*

$$\mathtt{pow}(x, 0) = \mathrm{C}_1^1(x) = 1, \ \forall x \in \mathbb{N} \tag{1.13}$$

$$\mathtt{pow}(x, \mathrm{Sc}(y)) = \mathtt{mul}(x, \mathtt{pow}(x, y)), \ \forall x, y \in \mathbb{N}. \tag{1.14}$$

Na idućoj ilustraciji možemo vidjeti postupak kojim smo iz inicijalnih funkcija (listovi) i preko operacija kompozicije i primitivne rekurzije (bridovi) došli do funkcije $\mathtt{mul}$ (korijen).



Dijagram 1.2. Stablasti prikaz funkcije $\mathtt{mul}$.

Degenerirana primitivna rekurzija

Definicija 1.4.1 ima jedan nedostatak: funkcija F^{k+1} dobivena primitivnom rekurzijom iz funkcija G^k i H^{k+2} mora biti *barem dvomjesna* (zbog $k \in \mathbb{N}_+$). To nas uvelike ograničava u stvaranju novih korisnih primitivno rekurzivnih funkcija. Primjerice, zasad uopće ne možemo definirati funkciju za oduzimanje $\mathtt{sub}$, jer se ona očito zasniva na iterativnom dekrementiranju (kao što se $\mathtt{add}$ zasniva na iterativnom inkrementiranju). Funkcija koja umanjuje prirodni broj za jedan bi očito trebala biti *jednomjesna,* a to znači da je ne možemo konstruirati pomoću definicije 1.4.1.

Idućom propozicijom zaobilazimo taj problem i pokazujemo da jednomjesne funkcije također mogu biti primitivno rekurzivne.

Propozicija 1.4.7. *Neka je $a \in \mathbb{N}$ i H^2 primitivno rekurzivna funkcija. Tada je primitivno rekurzivna i funkcija F^1, zadana s*

$$F(0) := a, \tag{1.15}$$

$$F(x+1) := H(x, F(x)), \ \forall x \in \mathbb{N}. \tag{1.16}$$

Dokaz. Definirajmo funkciju $F(x,y) := F(y)$ za sve $x, y \in \mathbb{N}$. Imajmo na umu da su F^2 i F^1 različite funkcije, jer imaju različite domene odnosno mjesnosti. Iz zapisa

$$F(x,0) = F(x) = a = C_a^1, \tag{1.17}$$

$$F(x, Sc(y)) = F(Sc(y)) = H(y, F(y)) = H(y, F(x,y)) =: H(x, y, F(x,y)), \tag{1.18}$$

vidimo da je F^2 dobivena (pravom) primitivnom rekurzijom iz primitivno rekurzivnih funkcija $G^1 := C_a^1$ i $H^3 := H^2 \circ (I_2^3, I_3^3)$, pa je primitivno rekurzivna. Sada primijetimo da jednakost $F(y) = F(x,y)$ vrijedi za bilo koji x, pa uzmimo $x = 0 = Z(y)$. Primitivna rekurzivnost funkcije F^1 sada slijedi iz zapisa $F^1 = F^2 \circ (Z, I_1^1)$. □

Definicija 1.4.8. *Neka je $a \in \mathbb{N}$, H^2 primitivno rekurzivna funkcija i F^1 funkcija zadana izrazima (1.15) i (1.16). Kažemo da je F^1 dobivena* ***degeneriranom primitivnom rekurzijom*** *i pišemo $F^1 := a \ \mathrm{PR}\ H^2$.*

Propozicija 1.4.7 nam garantira da je funkcija F^1 iz prethodne definicije primitivno rekurzivna. Štoviše, možemo i eksplicitno zapisati izraz

$$F = a \ \mathrm{PR}\ H^2 := (C_a^1 \ \mathrm{PR}\ H^2 \circ (I_2^3, I_3^3)) \circ (Z, I_1^1) \tag{1.19}$$

iz kojeg slijedi primitivna rekurzivnost funkcije F^1.

Sada napokon možemo definirati funkciju *prethodnik* uz pomoć degenerirane primitivne rekurzije. Želimo nešto oblika $pd(x) = x - 1$ (ovo je neslužbeni zapis jer oduzimanje nije brojevna funkcija), ali problem je slučaj kada je $x = 0$. Naime, kodomena svake brojevne funkcije je po definiciji jednaka $\mathbb{N}$, pa ne možemo dopustiti $pd(0) = -1$.

To ćemo zaobići tako da definiramo $\mathtt{pd}(0) = 0$, što onda mijenja naš neslužbeni zapis funkcije prethodnik u $\mathtt{pd}(x) := \max\{x - 1, 0\}$.

Slični problem pojavit će se i za funkciju $\mathtt{sub}$; ako oduzimamo veći broj od manjeg, ne želimo dobiti negativni broj. Dakle, definirat ćemo *ograničeno oduzimanje* sa $\mathtt{sub}(x, y) := \max\{x - y, 0\}$.

Primjer 1.4.9. *Funkcija prethodnik* $\mathtt{pd}^1$ *je definirana s*

$$\mathtt{pd}(0) = 0, \tag{1.20}$$
$$\mathtt{pd}(y + 1) = y, \ \forall y \in \mathbb{N}, \tag{1.21}$$

dok je funkcija za ograničeno oduzimanje $\mathtt{sub}^2$ *definirana s*

$$\mathtt{sub}(x, 0) = x, \ \forall x \in \mathbb{N}, \tag{1.22}$$
$$\mathtt{sub}(x, \mathtt{Sc}(y)) = \mathtt{pd}(\mathtt{sub}(x, y)), \ \forall x, y \in \mathbb{N}. \tag{1.23}$$

Simbolički, $\mathtt{pd} = 0 \operatorname{PR} I_1^2$ *i* $\mathtt{sub} = I_1^1 \operatorname{PR} \mathtt{pd} \circ I_3^3$. *Dakle,* $\mathtt{pd}$ *je primitivno rekurzivna po propoziciji 1.4.7, pa je onda i* $\mathtt{sub}$ *primitivno rekurzivna po propoziciji 1.4.5.*

Primjer 1.4.10. *Funkcija* $\mathtt{factorial}^1$ *je definirana s*

$$\mathtt{factorial}(0) := 1, \tag{1.24}$$
$$\mathtt{factorial}(y + 1) := \mathtt{mul}(\mathtt{Sc}(y), \mathtt{factorial}(y)), \ \forall y \in \mathbb{N}, \tag{1.25}$$

pa je primitivno rekurzivna po propoziciji 1.4.7.

Za kraj ovog odjeljka pokazat ćemo kako se utvrđuje primitivna rekurzivnost nekih jednostavnih relacija. Prisjetimo se, kažemo da je relacija R^k **primitivno rekurzivna** ako je njena karakteristična funkcija χ_R primitivno rekurzivna.

Primjer 1.4.11. *Relacija* $\mathbb{N}_+$ *je primitivno rekurzivna jer je njena karakteristična funkcija zadana s* $\chi_{\mathbb{N}_+} = 0 \operatorname{PR} C_1^2$.

Dvomjesna relacija „strogo veće", u oznaci $>$, *definirana je s* $(x, y) \in (>)$ *ako i samo ako* $x > y$. *Uočimo da je* $x > y$ *ako i samo ako je broj* $\mathtt{sub}(x, y)$ *pozitivan, pa vrijedi* $\chi_> = \chi_{\mathbb{N}_+} \circ \mathtt{sub}$. *Iz toga slijedi da je* $>^2$ *primitivno rekurzivna relacija.*

Jednostavnom zamjenom argumenata možemo doći i do karakteristične funkcije za relaciju „strogo manje", $\chi_< = \chi_> \circ (I_2^2, I_2^1)$. *Dakle, i relacija* $<^2$ *je primitivno rekurzivna.*

1.5 Minimizacija

Što ako domena funkcije nije čitavi $\mathbb{N}^k$? Do sada smo promatrali samo *totalne* izračunljive funkcije, ali jasno je da bi i funkcije poput $\otimes^k$ trebale biti izračunljive — svako izračunavanje koje nikad ne stane (ni za jedan ulaz) predstavlja jedno izračunavanje funkcije $\otimes^k$. Primjerice, funkcija $\otimes^3$ je svakako Python-izračunljiva, jer je računa Python-funkcija `prazna`, definirana s

```
1 def prazna(x, y, z):
2     while True: pass
```

Program 1.5.1: Prazna funkcija u Pythonu

Dakle, očito postoji algoritam za praznu funkciju, ali ga ne možemo prikazati u funkcijskom modelu računanja bez uvođenja nove operacije čiji će rezultat (ponekad) biti *parcijalna* funkcija (čija domena je pravi podskup od $\mathbb{N}^k$).

Ta nova operacija, *minimizacija,* odgovarat će *neograničenim petljama* koje stanu u trenutku kad je ispunjen neki uvjet. Zato minimizaciju možemo zamišljati kao `while` petlju s negiranim uvjetom: `while not` *Uvjet*`:...`

Definicija 1.5.1. *Neka je* $k \in \mathbb{N}_+$ *i* R^{k+1} *relacija. Za funkciju* F^k *definiranu s*

$$\mathcal{D}_F := \{\vec{x} \in \mathbb{N}^k \mid \exists y R(\vec{x}, y)\}, \tag{1.26}$$

$$F(\vec{x}) := \min\{y \in \mathbb{N} \mid R(\vec{x}, y)\}, \ \forall \vec{x} \in \mathcal{D}_F, \tag{1.27}$$

kažemo da je dobivena ***minimizacijom*** *relacije* R. *Pišemo* $F^k := \mu R^{k+1}$, *ili točkovno* $F(\vec{x}) :\simeq \mu y R(\vec{x}, y)$. *Skup funkcija* $\mathcal{F}$ *je* ***zatvoren na minimizaciju*** *ako za svaki* $k \in \mathbb{N}_+$, *za svaki* $R^{k+1} \subseteq \mathbb{N}^{k+1}$, $\chi_R \in \mathcal{F}$ *povlači* $\mu R \in \mathcal{F}$.

Za razliku od operacija na funkcijama koje smo do sada promatrali, naglasimo da je minimizacija *operacija na relaciji* čiji je rezultat *funkcija.* Uz oznake kao u definiciji 1.5.1, $F(\vec{x})$ je jednako *najmanjem* $y \in \mathbb{N}$ za koji je $(k+1)$-torka $(\vec{x}, y)$ u relaciji R^{k+1}.

Uvođenjem minimizacije napokon smo došli do skupa funkcija koji je tema ovog rada i za koji ćemo izraditi interpreter.

Definicija 1.5.2. *Skup* ***parcijalno rekurzivnih*** *funkcija je najmanji skup funkcija koji sadrži sve inicijalne funkcije te je zatvoren na kompoziciju, na primitivnu rekurziju i na minimizaciju.*

Propozicija 1.5.3. *Skup Python-izračunljivih funkcija zatvoren je na minimizaciju.*

Dokaz. Neka je $k \in \mathbb{N}_+$ i R^{k+1} Python-izračunljiva relacija. To znači da postoji Python-funkcija χ_R koja računa karakterističnu funkciju relacije R. Stoga minimizaciju $F^k := \mu R^{k+1}$ računa Python-funkcija `F` definirana s

```
def F(*x):
    y = 0
    while not χR(y, *x):
        y += 1
    return y
```

Program 1.5.2: Python-funkcija za minimizaciju

□

Sljedeći teorem osigurava da su sve funkcije koje smo do sada promatrali, i sve funkcije koje ćemo od sada promatrati, doista izračunljive u Python-modelu izračunavanja.

Teorem 1.5.4. *Svaka parcijalno rekurzivna funkcija je Python-izračunljiva.*

Dokaz. Skup svih Python-izračunljivih funkcija sadrži sve inicijalne funkcije po propoziciji 1.2.3 i zatvoren je na kompoziciju, primitivnu rekurziju i minimizaciju po propozicijama 1.4.3 i 1.5.3. Budući da je skup svih parcijalno rekurzivnih funkcija *najmanji* skup s tim svojstvima, zaključujemo da je on podskup skupa svih Python-izračunljivih funkcija. □

1.6 Tehnike za rad s primitivno rekurzivnim funkcijama

Ovaj odjeljak ćemo posvetiti raznim operacijama (izvedenima pomoću osnovnih) koje će nam biti korisne u interpreteru. Za početak ćemo uvesti osnovne logičke operatore na relacijama i dokazati da oni čuvaju primitivnu rekurzivnost.

Napomena 1.6.1. *Zbog jednostavnije sintakse, od sada ćemo često koristiti uobičajene pokrate za matematičke operacije umjesto funkcija:*

$$x + y \quad \textit{umjesto} \quad \mathtt{add}(x, y), \tag{1.28}$$
$$x \dot{-} y \quad \textit{umjesto} \quad \mathtt{sub}(x, y), \tag{1.29}$$
$$x \cdot y \quad \textit{umjesto} \quad \mathtt{mul}(x, y). \tag{1.30}$$

Propozicija 1.6.2. *Neka je* $k \in \mathbb{N}_+$ *te* R^k *i* P^k *primitivno rekurzivne relacije. Tada su primitivno rekurzivne i relacije zadane s*

$$R^c(\vec{x}) :\Longleftrightarrow \neg R(\vec{x}), \tag{1.31}$$
$$(R \cap P)(\vec{x}) :\Longleftrightarrow R(\vec{x}) \wedge P(\vec{x}), \tag{1.32}$$
$$(R \cup P)(\vec{x}) :\Longleftrightarrow R(\vec{x}) \vee P(\vec{x}). \tag{1.33}$$

Dokaz. Potrebno je pokazati da su karakteristične funkcije χ_{R^c}, $\chi_{R\cap P}$ i $\chi_{R\cup P}$ primitivno rekurzivne. Svaku od tih funkcija ćemo prikazati kao kompoziciju drugih funkcija za koje znamo da su primitivno rekurzivne. U cijelom dokazu uzmimo proizvoljni $\vec{x} \in \mathbb{N}^k$.

Vrijednost $\chi_{R^c}(\vec{x})$ mora biti suprotna (u smislu istinitosti) vrijednosti $\chi_R(\vec{x})$. Ako se prisjetimo da brojevna funkcija $x \mapsto 1 \dot{-} x$ preslikava 0 u 1 i 1 u 0, zaključujemo da je tražena karakteristična funkcija jednaka $\chi_{R^c}(\vec{x}) = 1 \dot{-} \chi_R(\vec{x})$.

Konjunkciju možemo vrlo lako riješiti ako iskoristimo činjenicu da množenje dvaju faktora iz skupa $\{0, 1\}$ rezultira jedinicom ako i samo ako su oba faktora jedinice (a inače rezultira nulom). Dakle, $\chi_{R\cap P}(\vec{x}) = \chi_R(\vec{x}) \cdot \chi_P(\vec{x})$.

Za disjunkciju ćemo iskoristiti ekvivalenciju $R(\vec{x}) \vee P(\vec{x}) \iff \neg(\neg R(\vec{x}) \wedge \neg P(\vec{x}))$ kako bismo uz pomoć karakterističnih funkcija za negaciju i konjunkciju dobili rješenje: $\chi_{R\cup P}(\vec{x}) = 1 \dot{-} (1 \dot{-} \chi_R(\vec{x})) \cdot (1 \dot{-} \chi_P(\vec{x}))$. □

Korolar 1.6.3. *Relacije nestrogog uređaja $\leq$ i $\geq$ te relacija jednakosti $=$ su primitivno rekurzivne.*

Dokaz. Tvrdnje slijede iz primjera 1.4.11 i (funkcijskih) jednakosti

$$\chi_{\leq} = \chi_{>^c}, \tag{1.34}$$

$$\chi_{\geq} = \chi_{<^c}, \tag{1.35}$$

$$\chi_{=} = \chi_{(\leq)\cap(\geq)}. \quad \square \tag{1.36}$$

Teorem o grananju

Do sada smo u funkcijskoj paradigmi uspjeli ostvariti slijedno izvršavanje naredbi te ograničene i neograničene petlje. Ako malo razmislimo, *uvjetno izvršavanje naredbi* je jedino što nam nedostaje da bi funkcijski jezik imao sva obilježja jednostavnog programskog jezika.

Definicija 1.6.4. *Neka su $k, l \in \mathbb{N}_+$, neka su G_0^k, G_1^k,...,G_l^k funkcije, a R_1^k, R_2^k,...,R_l^k u parovima disjunktne relacije iste mjesnosti. Za funkciju F^k definiranu s*

$$\mathcal{D}_F := \bigcup_{i=0}^{l} (\mathcal{D}_{G_i} \cap R_i), \quad \text{gdje je } R_0 := \left(\bigcup_{i=1}^{l} R_i \right)^c, \tag{1.37}$$

$$F(\vec{x}) := \begin{cases} G_1(\vec{x}), & R_1(\vec{x}) \\ G_2(\vec{x}), & R_2(\vec{x}) \\ \quad\vdots & \\ G_l(\vec{x}), & R_l(\vec{x}) \\ G_0(\vec{x}), & \text{inače} \end{cases} \quad \text{za sve } \vec{x} \in \mathcal{D}_F, \tag{1.38}$$

kažemo da je dobivena ***grananjem*** *iz* ***grana*** $G_0, G_1, \ldots, G_l$ *i* ***uvjeta*** $R_1^k, R_2^k, \ldots, R_l^k$. *Simbolički pišemo* $F := \text{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0\}$.

Cilj nam je dokazati *teorem o grananju*, koji kaže da je funkcija F iz prethodne definicije primitivno rekurzivna ako su sve G_i i R_i primitivno rekurzivne funkcije odnosno relacije. Naglasimo da ćemo taj teorem dokazati samo za totalne funkcije, tako da uvijek zahtijevamo da uvjet G_0 bude eksplicitno zadan.

Prvo dokažimo dva pomoćna rezultata koja ćemo koristiti u dokazu teorema o grananju.

Lema 1.6.5. *Za svaki* $k \in \mathbb{N}_+$, *funkcije* $\mathtt{add}^k$ *i* $\mathtt{mul}^k$, *zadane s*

$$\mathtt{add}(x_1, x_2, \ldots, x_k) := x_1 + x_2 + \cdots + x_k, \tag{1.39}$$

$$\mathtt{mul}(x_1, x_2, \ldots, x_k) := x_1 \cdot x_2 \cdots x_k, \tag{1.40}$$

primitivno su rekurzivne.

Dokaz. Tvrdnju za $\mathtt{add}^k$ ćemo dokazati matematičkom indukcijom po k.

- Baza: za $k = 1$ je $\mathtt{add}^1 = I_1^1$ primitivno rekurzivna jer je inicijalna.
- Pretpostavka: neka je $\mathtt{add}^k$ primitivno rekurzivna za neki $k \in \mathbb{N} \setminus \{0, 1\}$.
- Korak: iz jednakosti

$$\mathtt{add}(x_1, x_2, \ldots, x_k, x_{k+1}) = \mathtt{add}(\mathtt{add}(x_1, x_2, \ldots, x_k), x_{k+1}), \tag{1.41}$$

  korištenjem primjera 1.4.2 i pretpostavke indukcije zaključujemo da je $\mathtt{add}^{k+1}$ primitivno rekurzivna.

Tvrdnju za $\mathtt{mul}^k$ dokazujemo potpuno analogno, uz doslovnu zamjenu $\mathtt{add}$ i $\mathtt{mul}$. □

Propozicija 1.6.6. *Neka su* $k, l \in \mathbb{N}_+$ *te* $R_1^k, R_2^k, \ldots, R_l^k$ *primitivno rekurzivne relacije iste mjesnosti. Tada su* $\bigcap_{i=1}^l R_i$ *i* $\bigcup_{i=1}^l R_i$ *također primitivno rekurzivne.*

Dokaz. Primijetimo da vrijedi jednakost $\chi_{\bigcap_{i=1}^l R_i} = \chi_{R_1} \cdot \chi_{R_2} \cdots \chi_{R_l}$, a lema 1.6.5 osigurava primitivnu rekurzivnost funkcije $\mathtt{mul}^l$, pa je $\chi_{\bigcap_{i=1}^l R_i}$ primitivno rekurzivna.

Za proizvoljan $\vec{x} \in \mathbb{N}^k$, $\vec{x}$ je u relaciji $\bigcup_{i=1}^l R_i$ ako i samo ako je $\vec{x}$ u barem jednoj relaciji R_i. Dakle, ako zbrojimo vrijednosti svih karakterističnih funkcija χ_{R_i}, onda je dovoljno provjeriti je li dobiveni broj pozitivan. Sve u svemu, iz jednakosti $\chi_{\bigcup_{i=1}^l R_i} = \chi_{\mathbb{N}_+} \circ (\chi_{R_1} + \chi_{R_2} + \cdots + \chi_{R_l})$ i leme 1.6.5 (za funkciju $\mathtt{add}^l$) zaključujemo da je $\chi_{\bigcup_{i=1}^l R_i}$ primitivno rekurzivna. □

Teorem 1.6.7. *(Teorem o grananju — međusobno disjunktni uvjeti): Neka su* $k, l \in \mathbb{N}_+$, *neka su* $G_0^k, G_1^k, \ldots, G_l^k$ *primitivno rekurzivne funkcije, a* $R_1^k, R_2^k, \ldots, R_l^k$ *u parovima disjunktne primitivno rekurzivne relacije iste mjesnosti. Tada je i funkcija* $F := \text{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0\}$ *također primitivno rekurzivna.*

Dokaz. Relacija $R_0 := \left(\bigcup_{i=1}^{l} R_i\right)^c$ je primitivno rekurzivna zbog propozicije 1.6.6 i primjera 1.6.2. Neka je $i \in \{0, 1, \ldots, l\}$. Iz izraza

$$\chi_{R_i}(\vec{x}) \cdot G_i(\vec{x}) = \begin{cases} G_i(\vec{x}), & R_i(\vec{x}) \\ 0, & \text{inače,} \end{cases} \tag{1.42}$$

i činjenice da je proizvoljni $\vec{x} \in \mathbb{N}^k$ uvijek u točno jednoj relaciji R_i (jer su R_i po parovima disjunktne, a R_0 je definirana tako da skupi sve one koji nisu ni u jednoj od $R_1, \ldots, R_l$), možemo doći do sljedećeg zapisa funkcije F:

$$F = \chi_{R_0} \cdot G_0 + \chi_{R_1} \cdot G_1 + \ldots + \chi_{R_l} \cdot G_l. \tag{1.43}$$

Drugim riječima, za svaki $\vec{x} \in \mathbb{N}^k$ postoji (točno jedan) $i \in \{0, 1, \ldots, l\}$ takav da je $F(\vec{x}) = G_i(\vec{x})$, i to je upravo onaj i takav da vrijedi $R_i(\vec{x})$. Funkcija F iz zapisa (1.43) je primitivno rekurzivna primjenom primjera 1.4.6 (za funkciju mul^2) i leme 1.6.5 (za funkciju add^l). □

Sljedeći rezultat nam pokazuje da iz teorema 1.6.7 možemo izbaciti uvjet o međusobnoj disjunktnosti relacija $R_1, R_2, \ldots, R_l$ ako na neki način *poredamo* provjere pripadnosti nekog $\vec{x}$ relacijama R_i (primjerice, redom kojim su relacije indeksirane).

Definicija 1.6.8. *Neka su* $k, l \in \mathbb{N}_+$, *neka su* $G_0^k, G_1^k, \ldots, G_l^k$ *funkcije, a* $R_1^k, R_2^k, \ldots, R_l^k$ *relacije iste mjesnosti. Definirajmo relacije* $P_1^k, P_2^k, \ldots, P_l^k$ *na sljedeći način:*

$$P_1 := R_1, \text{ te } P_i := R_i \setminus \bigcup_{j=1}^{i-1} R_j, \text{ za svaki } i \in \{2, \ldots, l\}. \tag{1.44}$$

Za funkciju F^k *definiranu s*

$$\mathcal{D}_F := \bigcup_{i=0}^{l} (\mathcal{D}_{G_i} \cap R_i), \quad \text{gdje je } R_0 := \left(\bigcup_{i=1}^{l} R_i\right)^c, \tag{1.45}$$

$$F(\vec{x}) := \begin{cases} G_1(\vec{x}), & P_1(\vec{x}) \\ G_2(\vec{x}), & P_2(\vec{x}) \\ & \vdots \\ G_l(\vec{x}), & P_l(\vec{x}) \\ G_0(\vec{x}), & \text{inače} \end{cases} \quad \text{za sve } \vec{x} \in \mathcal{D}_F, \tag{1.46}$$

kažemo da je dobivena ***uređenim grananjem*** *iz* ***grana*** $G_1, \ldots, G_l, G_0$
i ***uvjeta*** $R_1^k, R_2^k, \ldots, R_l^k$. *Simbolički pišemo* $F := \text{if}[R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0]$.

Teorem 1.6.9. *(Teorem o uređenom grananju): Neka su* $k, l \in \mathbb{N}_+$, *neka su* G_0^k, G_1^k,...,G_l^k *primitivno rekurzivne funkcije, a* R_1^k, R_2^k,...,R_l^k *primitivno rekurzivne relacije iste mjesnosti. Tada je i funkcija* $F := \text{if}[R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0]$ *također primitivno rekurzivna.*

Dokaz. Dokažimo prvo tvrdnju da operacija skupovne razlike čuva primitivnu rekurzivnost. Naime, ako su A i B primitivno rekurzivne relacije iste mjesnosti, onda je i $C := A \setminus B$ primitivno rekurzivna relacija, jer vrijedi $\chi_C = \chi_A \cdot \chi_{B^c}$.

Sada definiramo konačni niz P_1, P_2,...,P_l relacija kao u definiciji 1.6.8:

$$P_1 := R_1, \text{ te } P_i := R_i \setminus \bigcup_{j=1}^{i-1} R_j, \text{ za svaki } i \in \{2, \ldots, l\}. \tag{1.47}$$

Svaka relacija P_i je primitivno rekurzivna po propoziciji 1.6.6 i prethodnoj tvdnji o skupovnoj razlici. Prema propozicijama 1.6.6 i 1.6.2, primitivno rekurzivna je i relacija $P_0 := \left(\bigcup_{i=1}^{l} P_i\right)^c = \left(\bigcup_{i=1}^{l} R_i\right)^c$ (zadnja jednakost se može dokazati indukcijom po l). Nadalje, $P_1, P_2, \ldots, P_l$ su u parovima disjunktne, pa prema teoremu 1.6.7 zaključujemo da je funkcija $F = \text{if}\{P_1 : G_1, P_2 : G_2, \ldots, P_l : G_l, G_0\}$ primitivno rekurzivna. □

Propozicija 1.6.10. *Skup Python-izračunljivih funkcija zatvoren je na uređeno grananje.*

Dokaz. Kao i ranije, tvrdnju ćemo dokazati za $l = 2$, a jasno je kako bi dokaz izgledao za neki drugi l. Neka su G_0^k, G_1^k i G_2^k Python-izračunljive funkcije, a R_1^k i R_2^k Python-izračunljive relacije iste mjesnosti. Tada su Python-izračunljive i karakteristične funkcije χ_{R_i}, za $i \in \{1, 2\}$, pa funkciju $F = \text{if}[R_1 : G_1, R_2 : G_2, G_0]$ računa Python-funkcija

```
1 def F(*x):
2     if χ_R1(x): return G1(x)
3     elif χ_R2(x): return G2(x)
4     else: return G0(x)
```

Program 1.6.1: Python-funkcija za uređeno grananje

Zahvaljujući teoremu 1.6.9, funkciju F možemo shvatiti kao uvjetnu naredbu `if/elif/else` u kojoj je uređen redoslijed provjere uvjeta. □

Brojeća funkcija

Funkcija F iz teorema o grananju jedan je primjer „posebne" operacije koja će biti dio sintakse interpretera iako se njena primitivna rekurzivnost ne vidi odmah iz njenog zapisa $\text{if}[R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0]$. Naime, interpreter će podržavati neke operacije

čiju primitivnu rekurzivnost garantiraju određeni teoremi i propozicije koje smo dokazali. Jedna takva je i *brojeća funkcija.* Za dane $\vec{x} \in \mathbb{N}^k$ i $y \in \mathbb{N}$, ona *broji* za koliko brojeva $i \in \{0, 1, \ldots, y-1\}$ vrijedi $R(\vec{x}, i)$.

Prije formalne definicije brojeće funkcije i dokaza njene primitivne rekurzivnosti, dokažimo jednu pomoćnu lemu.

Lema 1.6.11. *Neka je* $k \in \mathbb{N}_+$ *i* G^k *primitivno rekurzivna funkcija. Tada je primitivno rekurzivna i funkcija* F^k *zadana s*

$$F(\vec{x}, y) := \sum_{i<y} G(\vec{x}, i). \tag{1.48}$$

Dokaz. Iz zapisa

$$F(\vec{x}, 0) := 0, \tag{1.49}$$

$$F(\vec{x}, y+1) := F(\vec{x}, y) + G(\vec{x}, y), \ \forall y \in \mathbb{N} \tag{1.50}$$

je vidljivo da je F dobivena primitivnom rekurzijom iz funkcija Z i $\mathtt{add}^2 \circ (I^3_3, G \circ (I^3_1, I^3_2))$, pa je i sama primitivno rekurzivna. □

Propozicija 1.6.12. *Neka je* $k \in \mathbb{N}_+$ *i* R^k *primitivno rekurzivna relacija. Tada je funkcija* F^k *zadana s*

$$F(\vec{x}, y) := \operatorname{card}\{i \in \mathbb{N} \mid i < y \wedge R(\vec{x}, i)\} \tag{1.51}$$

također primitivno rekurzivna. Skraćeno pišemo $F(\vec{x}, y) := (\#i < y)\ R(\vec{x}, i)$.

Dokaz. Budući da je karakteristična funkcija χ_R primitivno rekurzivna, tvrdnja slijedi iz zapisa

$$F(\vec{x}, y) = (\#i < y)\ R(\vec{x}, i) = \sum_{i<y} \chi_R(\vec{x}, i), \tag{1.52}$$

primjenom leme 1.6.11. □

Propozicija 1.6.13. *Skup Python-izračunljivih funkcija zatvoren je na brojenje.*

Dokaz. Neka je R^k Python-izračunljiva relacija. Tada je njena karakteristična funkcija Python-izračunljiva, odnosno postoji Python-funkcija χ_R takva da za sve $\vec{x}^{k-1}$ i $i \in \mathbb{N}$, poziv `χR(i,*x)` vraća `True` ako vrijedi $R(\vec{x}, i)$, a `False` inače. Kako se u Pythonu vrijednosti `False` i `True` aritmetički ponašaju kao brojevi 0 i 1, proizlazi da brojeću funkciju $F(\vec{x}, y) := (\#i < y)\ R(\vec{x}, i)$ računa Python-funkcija

```
1 def F(y,*x):
2     z = 0
3     for i in range(y): z += χR(i, *x)
4     return z
```

Program 1.6.2: Python-funkcija za brojenje

□

Ograničena minimizacija

Kao što smo prethodno naveli, operacija primitivne rekurzije odgovara petljama s unaprijed poznatim brojem ponavljanja. S druge strane, minimizacija odgovara neograničenim petljama, a funkcije dobivene minimizacijom primitivno rekurzivnih relacija nisu nužno primitivno rekurzivne. Pokažimo sada da je dovoljno *ograničiti* takvu minimizaciju kako bi rezultirajuća funkcija bila primitivno rekurzivna.

Definicija 1.6.14. *Neka je* $k \in \mathbb{N}_+$ *te* R^k *relacija. Za funkciju* F^k *definiranu s*

$$F(\vec{x}, y) := \mu i(i < y \to R(\vec{x}, i)) =: (\mu i < y)R(\vec{x}, i). \tag{1.53}$$

kažemo da je dobivena ***ograničenom minimizacijom*** *relacije* R.

Napomena 1.6.15. *Funkcija* F *iz prethodne definicije ima vrijednost za sve* $(\vec{x}, y) \in \mathbb{N}^k$. *Naime, ako postoji* $i \in \{0, 1, \ldots, y-1\}$ *takav da vrijedi* $R(\vec{x}, i)$, *onda postoji i najmanji takav* i, *te je vrijednost* $F(\vec{x}, y)$ *jednaka tom najmanjem* i. *Ako takav* i *ne postoji, onda je* $F(\vec{x}, y) = y$.

Propozicija 1.6.16. *Neka je* $k \in \mathbb{N}_+$ *te* R^k *primitivno rekurzivna relacija. Tada je i funkcija* F^k, *dobivena ograničenom minimizacijom relacije* R, *također primitivno rekurzivna.*

Dokaz. Vrijednost $F(\vec{x}, y)$ odredit ćemo matematičkom indukcijom po y:

- Baza: za $y = 0$ vrijedi $F(\vec{x}, 0) = 0$, jer je antecedens kondicionala iz definicije 1.6.14 lažan, pa je kondicional istinit.
- Pretpostavka: neka je izračunana vrijednost $z := F(\vec{x}, y)$ za neki $y \in \mathbb{N}_+$. Zbog napomene 1.6.15 vrijedi $z \leq y$.
- Korak: za određivanje vrijednosti $F(\vec{x}, Sc(y))$ razmotrimo slučajeve:
  - Ako je $z < y$, to znači da postoji $i < y < y + 1$ takav da vrijedi $R(\vec{x}, i)$, i z je najmanji takav i, pa je očito $F(\vec{x}, Sc(y)) = z$. Napomenimo da ovdje očito vrijedi $R(\vec{x}, z)$.
  - Ako je $z = y$, to znači da ne postoji $i < y$ takav da vrijedi $R(\vec{x}, i)$. Zanima nas vrijedi li $R(\vec{x}, y)$. Ako vrijedi, onda je $F(\vec{x}, Sc(y)) = y = z$. Inače, nismo našli nijedan $i < y + 1$ takav da vrijedi $R(\vec{x}, i)$, pa je $F(\vec{x}, Sc(y)) = Sc(y)$ u skladu s napomenom 1.6.15.

Dakle, po principu matematičke indukcije zaključujemo kako, za dani $y \in \mathbb{N}$, vrijednost $F(\vec{x}, Sc(y))$ može biti jednaka prethodno izračunanoj vrijednosti $F(\vec{x}, y)$ ili sljedbeniku

$Sc(y)$, ovisno o vrijednosti $R(\vec{x}, F(\vec{x}, y))$. Stoga, neka je H^{k+1} funkcija zadana s:

$$H(\vec{x}, y, z) = \begin{cases} z, & R(\vec{x}, z) \\ Sc(y), & \text{inače} \end{cases}. \tag{1.54}$$

Ona je primitivno rekurzivna po teoremu 1.6.7. Iz zapisa $F = C_0^{k-1}$ **PR** H (odnosno 0 **PR** H ako je $k = 1$) zaključujemo da je i F primitivno rekurzivna. □

1.7 Primjeri primitivno rekurzivnih funkcija

Za kraj ovog poglavlja demonstrirat ćemo da su još neke zanimljive matematičke funkcije i relacije primitivno rekurzivne. Ograničena minimizacija nam omogućava da koristimo isti mehanizam kao u običnoj minimizaciji (iteracija kroz petlju sve dok se ne ispuni određeni uvjet) i da kao rezultat dobijemo *primitivno* rekurzivnu funkciju. Sve što trebamo je *predvidjeti* najveći mogući broj iteracija kroz petlju za neki zadani algoritam i u skladu s tim ograničiti minimizaciju.

Pogledajmo pseudokōd algoritma za cjelobrojno dijeljenje, za sada ne pazeći na ograničavanje broja iteracija petlje.

```
1 def div(x,y):
2     k = 0
3     while True:
4         if k * y > x: break
5         k += 1
6     return k - 1
```

Program 1.7.1: Funkcija za cjelobrojno dijeljenje

Algoritam se zasniva na uzastopnoj provjeri vrijednosti $k \cdot y$, za $k \in \{0, 1, \ldots\}$, sve dok ta vrijednost ne postane *veća* od x. Takvim postupkom smo pronašli *najmanji* takav k, ali nama treba *najveći* k takav da je $k \cdot y$ *manje* od x, pa kao rezultat algoritma vraćamo $k - 1$.

Primijetimo da je vrlo jednostavno naći gornju ogradu za broj iteracija petlje. Naime, ako pretpostavimo da je $y > 0$, onda je jasno da će izraz $k \cdot y$ biti veći od x (u najgorem slučaju) za $k = x + 1$.

Klasičan problem dijeljenja s nulom se javlja i ovdje; ako je $y = 0$, onda $k \cdot y$ nikada neće biti veće od x. Srećom, u skladu s napomenom 1.6.15, dovoljno je definirati funkciju za dijeljenje uz pomoć ograničene minimizacije i ona će uvijek imati vrijednost (u ovom slučaju *definiramo* $div(x, 0) := x$).

Primjer 1.7.1. *Funkcija za cjelobrojno dijeljenje* div^2 *zadana je s*

$$div(x, y) = pd\Big((\mu i \leq x)(i \cdot y > x)\Big). \tag{1.55}$$

Funkcija $\mathtt{div}^2$ *je primitivno rekurzivna jer je jednaka kompoziciji funkcije* $\mathtt{pd}$ *s funkcijom koja je nastala ograničenom minimizacijom primitivno rekurzivne relacije (koja je primitivno rekurzivna prema propoziciji 1.6.16).*

Uz pomoć cjelobrojnog dijeljenja sada možemo relativno lako doći i do funkcije $\mathtt{mod}^2$. Za $x \in \mathbb{N}$ i $y \in \mathbb{N}_+$, iskoristit ćemo teorem o dijeljenju s ostatkom da dobijemo jedinstvene $k \in \mathbb{N}$ i $l \in \{0, 1, \ldots, y-1\}$, takve da vrijedi $x = k \cdot y + l$. Ta jednakost je ekvivalentna s $l = x \dot{-} (k \cdot y)$, gdje je $k = \mathtt{div}(x, y)$ rezultat cjelobrojnog dijeljenja brojeva x i y.

Primjer 1.7.2. *Funkcija za ostatak prilikom cjelobrojnog dijeljenja* $\mathtt{mod}^2$*, zadana s*

$$\mathtt{mod}(x, y) = x \dot{-} \Big(\mathtt{div}(x, y) \cdot y\Big), \tag{1.56}$$

također je primitivno rekurzivna. Primijetimo da je $\mathtt{mod}(x, 0) = x$*, što je uobičajena konvencija u teoriji brojeva.*

Primjer 1.7.3. *Djeljivost je primitivno rekurzivna relacija, jer je zadana s*

$$y \mid x \iff \mathtt{mod}(x, y) = 0. \tag{1.57}$$

Primjer 1.7.4. *Skup* $\mathbb{P}$ *svih prostih brojeva zadan je s*

$$x \in \mathbb{P} \iff (\#\, d \leq x)(d \mid x) = 2. \tag{1.58}$$

Primjenom propozicije 1.6.12 i primjera 1.7.3 zaključujemo da je $\mathbb{P}$ *primitivno rekurzivan skup (odnosno primitivno rekurzivna jednomjesna relacija).*

Propozicija 1.7.5. *Strogo rastući niz* $(p_i)_{i \in \mathbb{N}}$ *svih prostih brojeva je primitivno rekurzivan.*

Dokaz. Konstruirat ćemo funkciju $\mathtt{prime}^1$, koja svakom prirodnom broju n pridružuje n-ti prost broj p_n. Definirat ćemo je degeneriranom primitivnom rekurzijom, u koraku koristeći funkciju $\mathtt{nextprime}^1$ koja za bilo koji prirodni broj n vraća najmanji prosti broj veći od n.

U definiciji funkcije $\mathtt{nextprime}$ koristit ćemo ograničenu minimizaciju. Provjeravat ćemo vrijedi li uvjet $q \in \mathbb{P}$, za neke $q \in \mathbb{N}$. Jasno, želimo da vrijedi $\mathtt{nextprime}(p) > p$, pa je nepotrebno provjeravati uvjet $q \in \mathbb{P}$ za $q \leq p$. Nažalost, petlja u (ograničenoj) minimizaciji po definiciji kreće od 0, pa uvjetu $q \in \mathbb{P}$ moramo dodati uvjet $q > p$.

Kako ograničiti minimizaciju? Ako se sjetimo dokaza da prostih brojeva ima beskonačno mnogo, onda je moguće doći do vrlo velike (ali konačne) gornje ograde. Naime, ako pretpostavimo da je $p_n \in \mathbb{P}$ *najveći* prosti broj, onda možemo zaključiti da je i $p_0 \cdot p_1 \cdots p_n + 1$ prost, jer taj broj zasigurno nije djeljiv ni sa jednim p_i. Problem je što

ne znamo je li umnožak $p_0 \cdot p_1 \cdots p_n$ primitivno rekurzivan, jer upravo pokušavamo dokazati primitivnu rekurzivnost niza $(p_i)_{i\in\mathbb{N}}$. Zato ćemo uzeti još veću gornju ogradu za koju znamo da je primitivno rekurzivna: $\mathtt{factorial}(p_n) + 1$.

Sve u svemu, funkcija $\mathtt{nextprime}^1$ zadana je s

$$\mathtt{nextprime}(p) \coloneqq (\mu q \leq p! + 1)(q \in \mathbb{P} \wedge q > p), \tag{1.59}$$

pa je primitivno rekurzivna primjenom primjera 1.7.4 i propozicije 1.6.16.

Funkcija $\mathtt{prime}^1$ sada je zadana s

$$\mathtt{prime}(0) \coloneqq 2, \tag{1.60}$$

$$\mathtt{prime}(n+1) \coloneqq \mathtt{nextprime}(\mathtt{prime}(n)). \tag{1.61}$$

Dakle, $\mathtt{prime}$ je dobivena degeneriranom primitivnom rekurzijom iz konstante 2 i primitivno rekurzivne funkcije $\mathtt{nextprime} \circ I^2_2$, pa je po propoziciji 1.4.7 i sama primitivno rekurzivna. □

Poglavlje 2

Interpreter

2.1 Uvod u interpretaciju programa i opis problema

Interpretacija programa je proces pretvorbe izvornog kōda programa u nekom jeziku u *rezultate* (definirane pravilima jezika) koji trebaju nastati izvršavanjem tog kōda. Ilustrirajmo prethodnu rečenicu na primjeru interpretera za parcijalno rekurzivne funkcije koji želimo izraditi.
Želimo omogućiti izvršavanje programa poput:

```
1 add(x,0) := x
2 add(x,Sc(y)) := Sc(add(x,y))
3 add(5,3)
```

Program 2.1.1: Jednostavan program

U skladu s primjerom 1.4.2, prva dva retka reprezentiraju *definiciju* funkcije $\mathtt{add}$ primitivnom rekurzijom, dok zadnji redak označava *izračunavanje vrijednosti* $\mathtt{add}(5,3)$. Želja nam je da izvršavanje ovakavog kōda rezultira ispisom broja 8 na standardni izlaz.

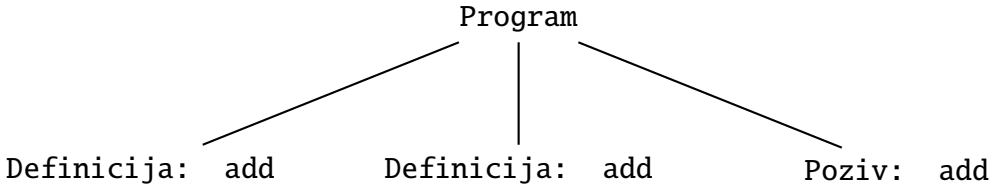
Napisat ćemo program u Pythonu koji će pročitati neki ulazni tekst (primjerice, onaj u programu 2.1.1), napraviti određenu *analizu* tog teksta i onda *izvršiti* sve naredbe u skladu s rezultatima te analize. Primijetimo da je ponašanje funkcije $\mathtt{add}$ u potpunosti definirano kōdom koji interpretiramo, što znači da naš Python-program mora imati (među ostalim) ugrađen mehanizam za detekciju i izračunavanje vrijednosti funkcija koje su definirane primitivnom rekurzijom. Dakle, nećemo ugraditi zbrajanje u naš interpreter samo zato što ga smatramo jednostavnom operacijom, već ćemo omogućiti korisniku da sam definira *bilo koju* parcijalno rekurzivnu funkciju, počevši samo od inicijalnih funkcija.

Interpretacija programa se sastoji od tri glavna koraka koja ćemo sada detaljno opisati:

- **Leksička analiza** je proces pretvorbe ulaznog teksta (izvornog kōda) u konačni niz **tokena** — najmanjih relevantnih jedinica od kojih se kōd sastoji. Naime, ne želimo interpretirati program *znak-po-znak,* već *pojam-po-pojam.* Primjerice, u liniji `add(x,0) := x`, vidimo da niz od tri znaka `add` predstavlja jedan pojam (u ovom slučaju je to naziv funkcije). Samim time, prirodno je slova `a`, `d` i `d` spojiti u jednu cjelinu — token sadržaja `add`. *Tokeniziranjem* čitavog izraza `add(x,0) := x` dobivamo tokene sljedećih sadržaja:

  `'add'`, `'('`, `'x'`, `','`, `'0'`, `')'`, `':='`, `'x'`.

- **Sintaksna analiza** (ili **parsiranje**) je proces pretvorbe konačnog niza tokena u **apstraktno sintaksno stablo** (**AST**) — stablasti prikaz hijerarhijske strukture kōda. Korijen tog stabla je najopćenitiji pojam (najčešće čitav *program* kao apstraktni pojam), dok svaki niži čvor predstavlja specifičnije dijelove kōda. Uočimo da se program 2.1.1 sastoji od tri *naredbe* — što je još uvijek vrlo općenit pojam, ali manje općenit od pojma programa. Prve dvije naredbe definiraju funkciju `add`, dok treća poziva tu funkciju s nekim konkretnim brojevima. Pojednostavljeni AST za program 2.1.1 (sa samo dvjema najvišim razinama) izgleda ovako:



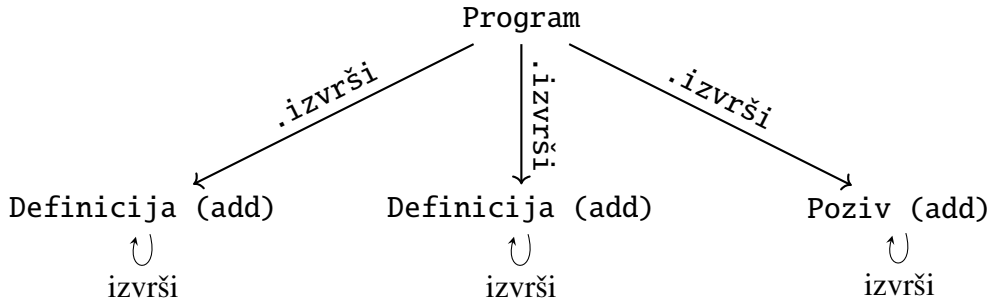
  Primjer sintaksno neispravnog izraza je `add(x,)`, zato što nakon zareza očekujemo naziv neke varijable, broj ili poziv funkcije, a ne zatvorenu zagradu. Napomenimo da je ovo *sintaksna* greška, a ne *leksička* — izraz `add(x,)` se može ispravno rastaviti na tokene, ali njihov poredak nije u skladu s pravilima jezika.

- **Semantička analiza** je proces ispitivanja ispravnosti kōda na višoj razini od same sintaksne ispravnosti. U ovoj fazi se ispituje je li kōd napisan *smisleno* i u skladu s pravilima i ograničenjima programskog jezika. Primjerice, izraz `add(x,0) := y` je sintaksno ispravan, ali semantički nije. Naime, s desne strane jednakosti se mogu pojavljivati samo one varijable koje su navedene kao argumenti funkcije s lijeve strane. Drugim riječima, `y` je nepoznata varijabla u tom kontekstu.

  Semantičku analizu ćemo u praksi spojiti s vjerojatno najzanimljivijom fazom interpretacije — **izvršavanjem** programa. Naime, proces izvršavanja interpretiranih naredbi započet ćemo odmah nakon generiranja AST-a, odnosno pokrenut ćemo svaki program koji je sintaksno ispravan. Ako u programu postoji semantička greška, ona

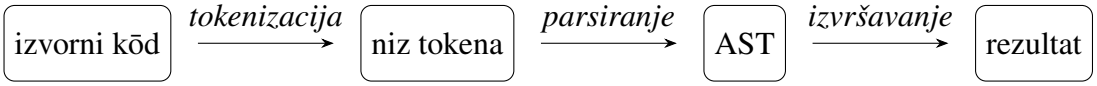
će se pojaviti tijekom izvođenja (*runtime error*) te će se izvršavanje programa prekinuti.

Izvršavanje programa se svodi na obilazak AST-a čitavog programa redoslijedom *postorder*, uz izvršavanje svakog pojedinog čvora.



Dijagram 2.2. Izvršavanje AST-a.

Sve korake interpretacije programa možemo prikazati sljedećim dijagramom:



Dijagram 2.3. Koraci interpretacije programa.

Kao što smo ranije naveli, koristit ćemo *framework* **vepar** za interpretaciju programa. To je modul napisan u Pythonu, s mnoštvom funkcija, klasa i operatora koji olakšavaju proces interpretacije programa i osiguravaju da korisnik ne mora (previše) misliti o detaljima niske razine tijekom pisanja interpretera. U ovom radu se nećemo baviti implementacijom *vepra*, već ćemo detaljno opisati kako se njime koristi na jednom konkretnom projektu poput ovog interpretera. U tom smislu, ovo poglavlje može poslužiti kao *tutorial* za *vepar*.

Kad je riječ o organizaciji tijeka izrade interpretera, postoje dva pristupa koja bismo mogli izabrati. Prvi pristup je da odmah definiramo sva pravila jezika i sve vrste naredbi koje bi na kraju trebale biti podržane, i da onda počnemo kōdirati interpreter tako da istovremeno pišemo kōd za svaku vrstu naredbe. Iako ovaj pristup na prvi pogled izgleda prihvatljiv jer je dobro organiziran (i ima paralele s procesom izrade softvera *waterfall*), u stvarnosti je prilično nepraktičan. Naime, gotovo sigurno bismo se izgubili u nizu grešaka stvorenih jer pokušavamo *parsirati* sve vrste naredbi istovremeno, još i prije nego što smo sigurni da nam osnovne naredbe funkcioniraju.

Zato ćemo izabrati drugi, *iterativni* pristup. Prvo ćemo napraviti minimalni funkcionalni primjer, po uzoru na program 2.1.1, a nakon toga ćemo ga postupno nadograđivati novim funkcionalnostima. Ako pažljivo pogledamo program 2.1.1, primijetit ćemo da su

u njemu prisutne operacije kompozicije i primitivne rekurzije. Prvu iteraciju ćemo pisati s ciljem da nam radi *bilo koja* funkcija definirana primitivnom rekurzijom (a ne samo `add`). Iako ćemo često referencirati program 2.1.1 (zbog njegove jednostavnosti), imajmo na umu činjenicu da interpreter mora biti u stanju ispravno prepoznati i kompliciranije izraze. Primjerice, ako pretpostavimo da smo napisali kōd takav da se program 2.1.1 ispravno izvršava, htjeli bismo da program

```
1 add(x,0) := x
2 add(x,Sc(y)) := Sc(add(x,y))
3 mul(x,0) := 0
4 mul(x,Sc(y)) := add(x,mul(x,y))
5 f(x,y) := add(x,mul(x,y))
6 mul(3,add(5,3))
7 f(2,2)
```

Program 2.1.2: Složeniji program

ispiše 24 i 6 bez ikakvog modificiranja kōda interpretera vezanog za primitivnu rekurziju (odnosno samo uz dodavanje kōda koji bi ispravno interpretirao *direktnu* definiciju funkcije `f`). To izgleda zahtjevno, ali zapravo samo moramo paziti da dio kōda koji prepoznaje primitivnu rekurziju bude ovisan isključivo o izrazima koji predstavljaju inicijalizaciju i korak.

2.2 Tipovi tokena

Za rad s *veprom* potrebno je unijeti modul `vepar` (precizno, sva imena iz tog modula) u neko svoje okruženje za rad u Pythonu, što činimo naredbom

```
1 from vepar import *
```

Program 2.2.1: Početak rada s *veprom*

Kako bismo mogli stvarati tokene različitih tipova, potrebno je definirati vrste tokena koje postoje u našem jeziku. Od sada ćemo za tokene koristiti oznaku `VRSTA_TOKENA'sadržaj'`, gdje `sadržaj` predstavlja podstring ulaza od kojeg se token sastoji.

Promatranjem programa 2.1.1 i 2.1.2, možemo zaključiti da postoje četiri kategorije u koje bismo mogli svrstati tokene:

- Tokeni koji se sastoje od **jednog** (fiksnog) znaka, kao što su `OTV'('`, `ZATV')'` i `ZAREZ','`.
- Tokeni koji se sastoje od **više** (fiksnih) znakova, kao što je `DEF_FUN':='`.

- Tokeni koji predstavljaju **imena** funkcija ili varijabli, kao što su `IME'add'` i `IME'x'`.
- Tokeni koji predstavljaju **prirodne brojeve**, kao što su `BROJ'0'` i `BROJ'42'`.

Primijetimo da su tokeni tipa `IME` i `BROJ` u određenom smislu posebni — njihov sadržaj nije uvijek isti, već ovisi o tome kako je korisnik odlučio nazvati funkcije ili varijable, odnosno koje je brojeve odlučio koristiti.

Pogledajmo kako bismo napisali enumeraciju svih vrsta tokena.

```
1  class T(TipoviTokena):
2      DEF_FUN = ':='
3      OTV, ZATV, ZAREZ = '(),'
4      class IME(Token):
5          def izvrši(self, memorija, funkcije):
6              return memorija[self]
7      class BROJ(Token):
8          def izvrši(self, memorija, funkcije):
9              return int(self.sadržaj)
```

Program 2.2.2: Tipovi tokena

Oznaka `TipoviTokena` predstavlja praznu enumeraciju, što znači da je klasa `T` također enumeracija (jer nasljeđuje od nje). Tokene s fiksnim sadržajem (*literale*) smo definirali na vrhu enumeracije, a složenije tokene `IME` i `BROJ` smo napisali kao potklase. Više o metodi `izvrši` reći ćemo nešto kasnije.

2.3 Lekser

Lekser je funkcija koja provodi proces tokenizacije — pretvara ulazni string u konačni niz tokena. Postoji mnogo pomoćnih alata unutar *vepra* koji pojednostavljuju pisanje leksera. On se uvijek piše kao jedna `for`-petlja koja iterira po cijelom ulaznom stringu čitajući znakove jedan po jedan. Unutar nje imat ćemo slučajeve (u obliku naredbe `if/elif/else`) koji će stvarati (i odašiljati) određene vrste tokena.

```
@lexer
def lekser(ulaz):
    for znak in ulaz:
        if znak.isspace():
            ulaz.zanemari()
        elif znak == '/':
            ulaz >> '/':
            ulaz.pročitaj_do('\n')
            ulaz.zanemari()
        elif znak == ':':
            ulaz >> '='
            yield ulaz.token(T.DEF_FUN)
        elif znak.isalpha() or znak == '_':
            ulaz * {str.isalnum, '_'}
            yield ulaz.literal_ili(T.IME)
        elif znak.isdecimal():
            ulaz.prirodni_broj(znak)
            yield ulaz.token(T.BROJ)
        else:
            yield ulaz.literal(T)
```

Program 2.3.1: Lekser

Dekorator `@lexer` signalizira *vepru* da smo na tom mjestu definirali funkciju koja predstavlja lekser. Ta funkcija je najčešće Pythonov *generator* koji odašilje niz tokena uz pomoć ključne riječi `yield`. Argument `ulaz` je *string* koji predstavlja izvorni kōd našeg jezika. Njemu *vepar* pridruži korisna svojstva koja prate koji dio ulaza je do sada pročitan i koji se dio trenutno čita. Isto tako, omogućeno nam je korištenje raznih funkcija i operatora na ulaznom stringu. Slijedi opis nekih često korištenih mogućnosti:

- `ulaz.zanemari()` zanemaruje sve što je pročitano od zadnjeg odaslanog tokena.
- `ulaz >> c` čita idući znak i zahtijeva da je on jednak `c`, u suprotnom javlja grešku.
- `ulaz >= c` provjerava je li idući znak jednak `'c'` — ako jest, onda ga pročita i vrati istinu, a inače ga ne pročita i vrati laž.
- `ulaz > c` provjerava je li idući znak jednak `c` i ovisno o tome vraća istinu odnosno laž (nikada ne čita znak).
- `ulaz * uvjet` čita ulazni string sve dok je uvjet istinit.
- `ulaz.pročitaj_do(c)` čita ulazni string sve dok ne stigne do znaka `c`. Toj funkciji možemo proslijediti i istinosne argumente `uključivo` i `više_redova` koje dodatno određuju krajnju granicu čitanja. U programu 2.3.1 smo ovu funkciju iskoristili kako bismo zanemarili jednolinijske *komentare*, označene s `'//'`.
- `ulaz.prirodni_broj(znak)` čita prirodan broj počevši od znamenke `znak`.
- `yield` je ključna riječ koja odašilje token željene vrste parseru (ili na konzolu) za dalju obradu. Možemo je koristiti u kombinaciji sa sljedećim funkcijama:

- `ulaz.token(T.TIP)` vraća token tipa `TIP` čiji je sadržaj jednak dotad pročitanom stringu.
- `ulaz.literal(T)` vraća literal čiji je sadržaj jednak dotad pročitanom stringu, a tip mu je određen kao literal s tim sadržajem — ako takav literal ne postoji, funkcija diže izuzetak.
- `ulaz.literal_ili(T.TIP)` provjerava postoji li u enumeraciji neki literal sa sadržajem koji se podudara s trenutno pročitanim znakom (ili stringom) — ako postoji, onda vrati taj literal, a inače vrati token tipa `TIP`.

Lekser možemo testirati tako da ga pozovemo s tekstom programa 2.1.1.

```
1 ulaz = """add(x,0) := x
2 add(x,Sc(y)) := Sc(add(x,y))
3 add(5,3)
4 """
5 lekser(ulaz)
```

Program 2.3.2: Testiranje leksera

Dobit ćemo detaljan ispis stvorenih tokena, zajedno s njihovim sadržajima i pozicijama u tekstu.

```
1 Redak 0, stupac 1 - redak 1, stupac 3: IME'add'
2 Redak 1, stupac 4      : OTV'('
3 Redak 1, stupac 5      : IME'x'
4 Redak 1, stupac 6      : ZAREZ','
5 ...
```

Program 2.3.3: Ispis testa leksera

2.4 Beskontekstna gramatika

Beskontekstna gramatika je skup pravila kojima se može generirati bilo koji beskontekstni jezik. Svako pravilo ima lijevu i desnu stranu — lijevo se nalazi varijabla koju mijenjamo u niz varijabli ili tipova tokena koji se nalaze na desnoj strani. Prilikom pisanja tih pravila, sadržaj konkretnih tokena nas neće zanimati, već samo njihov tip. Naime, kao što smo ranije pojasnili, sintaksna analiza ne može detektirati semantičke pogreške, poput poziva nedefinirane funkcije ili korištenja nedefinirane varijable, već može samo provjeriti jesu li tipovi tokena napisani ispravnim redoslijedom. Vidjet ćemo da je pisanje beskontekstne gramatike bitan dio interpretacije bilo kojeg jezika, zato što parser pišemo u skladu s pravilima gramatike — svako pravilo unutar beskontekstne gramatike će odgovarati jednoj metodi parsera.

Prilikom prikazivanja beskontekstnih pravila, pridržavat ćemo se konvencije **EBNF** (*extended Backus–Naur form*), po kojoj se svi literali zamjenjuju njihovim sadržajima — primjerice, pišemo `:=` umjesto `DEF_FUN`. Iz estetskih razloga, izostavit ćemo navodnike oko sadržaja literala, jer mislimo da neće doći do zabune (iz konteksta će biti jasno što su varijable, a što sadržaji literala). Varijable ćemo pisati malim, a složene tipove tokena velikim slovima.

Pogledajmo kako bi mogla izgledati beskontekstna gramatika koja generira programe 2.1.1 i 2.1.2. Svaki od tih programa se sastoji od više naredbi, a svaka naredba je ili definicija funkcije ili evaluacija nekog izraza. Također, primijetimo da je složenost izraza na lijevoj i desnoj strani tokena `DEF_FUN':='` različita. Naime, izraz na lijevoj strani mora sadržavati ime funkcije s argumentima koji mogu biti varijable, brojevi ili poziv funkcije `Sc` (s nekom varijablom), dok izraz na desnoj strani može biti varijabla, broj ili poziv neke funkcije s argumentima koji ponovno mogu biti varijable, brojevi ili pozivi funkcija. Dakle, očito je da će se beskontekstna pravila za lijevu i desnu stranu tokena `DEF_FUN` razlikovati — na desnoj strani moramo omogućiti neograničeno ugnježđivanje izraza, dok na lijevoj strani dopuštamo samo jednostavne izraze.

Sve u svemu, beskontekstna gramatika je dana s

$$\texttt{program} \to \texttt{naredba} \mid \texttt{program naredba} \tag{2.1}$$
$$\texttt{naredba} \to \texttt{definicija} \mid \texttt{izraz} \tag{2.2}$$
$$\texttt{definicija} \to \texttt{IME(lijeve\_varijable) := izraz} \tag{2.3}$$
$$\texttt{lijeve\_varijable} \to \texttt{lijeva} \mid \texttt{lijeva, lijeve\_varijable} \tag{2.4}$$
$$\texttt{lijeva} \to \texttt{IME} \mid \texttt{BROJ} \mid \texttt{Sc(IME)} \tag{2.5}$$
$$\texttt{desne\_varijable} \to \texttt{izraz} \mid \texttt{izraz, desne\_varijable} \tag{2.6}$$
$$\texttt{poziv} \to \texttt{IME(desne\_varijable)} \tag{2.7}$$
$$\texttt{izraz} \to \texttt{IME} \mid \texttt{BROJ} \mid \texttt{poziv} \tag{2.8}$$

Recimo još nešto o raznim vrstama (s obzirom na razinu apstrakcije) čvorova apstraktnog sintaksnog stabla. *Listovi* (čvorovi bez djece) su konkretne strukture koje predstavljaju najjednostavnije pojmove od kojih se kōd sastoji. Jedini listovi u apstraktnom stablu našeg interpretera će biti tokeni `IME` i `BROJ`, odnosno jedini složeni tokeni u programu 2.2.2.

Suprotni pojam od listova su (potpuno) *apstraktni* čvorovi AST-a. To su pojmovi koji nemaju konkretne instance tijekom izvršavanja stabla, već služe isključivo za pregledniju organizaciju hijerarhije čvorova. Primjer jednog apstraktnog čvora je `naredba`. Po pravilu 2.2, ona može biti ili `definicija` ili `izraz`, što znači da nas nikad neće zanimati kako se izvršava generički pojam poput `naredbe` (jer postoje konkretniji pojmovi čije je izvršavanje lakše definirati).

Primjeri treće vrste čvorova su `definicija` i `poziv` — oni ne moraju nužno biti listovi, ali su svejedno *dovoljno konkretni* pojmovi da bi se njihovo izvršavanje moglo jasno definirati. Primjerice, izvršavanje `definicije` funkcije bi trebalo zapisati tu funkciju (odnosno njenu definiciju) negdje u memoriju, a izvršavanje `poziva` funkcije bi trebalo pronaći tu funkciju u memoriji, pozvati ju sa zadanim argumentima i vratiti vrijednost te funkcije. Ovakve pojmove ćemo od sada zvati **čvorovi**, uz napomenu da i listove smatramo čvorovima. U kasnijim odjeljcima ćemo definirati još čvorova (poput minimizacije, brojeće funkcije i grananja) čija će parcijalna rekurzivnost biti garantirana rezulatatima iz prvog poglavlja.

Jasno je da svaka grana AST-a mora završiti listovima. Primjerice, ako pogledamo kompoziciju `Sc(Sc(Sc(Sc(...))))`, očito je da ćemo u nekom trenutku morati prekinuti ugnježđivanje (ako želimo dobiti smislen izraz) i napisati nešto *jednostavnije* od poziva funkcije `Sc` (ili neke druge funkcije). Drugim riječima, svako ugnježđivanje u nekom trenutku moramo prekinuti listovima; poziv `Sc(Sc(Sc(Sc(2))))` ili definicija `f(x) := Sc(Sc(x))` su neki od primjera.

2.5 Parser

Lekserom smo iz izvornog kōda uspjeli stvoriti konačan niz tokena. Sada je taj niz tokena potrebno pretvoriti u apstraktno sintaksno stablo. Ono će u kōdu biti reprezentirano kao jedna instanca klase koja predstavlja korijen stabla — u našem slučaju će to biti klasa `Program`. Ta instanca će sadržavati podatke o svim čvorovima koji su na razini neposredno ispod nje. Drugim riječima, instanca klase `Program` sadrži listu instanci nekih drugih klasa koje predstavljaju *djecu* korijena stabla. S druge strane, neki čvorovi (poput `Definicije`) zbog svog semantičkog značenja imaju fiksan broj djece, pa ih ne moraju nužno pamtiti u listi, već ih mogu pamtiti kao varijable članice različitih imena. Stablo se širi (na dva upravo opisana načina) sve dok svaka grana ne završi listovima (koji su također instance klasa).

Apstraktno sintaksno stablo generirat ćemo uz pomoć *parsera* — klase (nazvat ćemo je `P`) koja prolazi redom po tokenima i od njih stvara instance čvorova AST-a. Klasa `P` nasljeđuje *veprovu* klasu `Parser` — njene glavne funkcionalnosti ćemo opisati u ovom odjeljku.

Gotovo svi programski jezici se parsiraju na način da se *čitav* izvorni kōd parsira zajedno. Naime, u većini jezika postoje petlje (u klasičnom smislu) i blokovi naredbi, što znači da je svaki redak kōda iznimno ovisan o retcima iznad.

```
# izvorni_kod je ulazni string u nekom jeziku
ast = P(izvorni_kod)
```

Program 2.5.1: Uobičajeni način parsiranja

U funkcijskom jeziku to nije slučaj! Ako pogledamo program 2.1.2, možemo zaključiti da su reci *gotovo neovisni* jedan o drugome. Primjerice, da bi se redak `add(x,Sc(y)) := Sc(add(x,y))` (koji predstavlja korak definicije funkcije `add` primitivnom rekurzijom) ispravno parsirao, jedino je bitno da se prethodno parsirao redak koji predstavlja inicijalizaciju funkcije `add`. Općenito, ispravno parsiranje redaka ovisi *samo* o tome jesu li prethodno definirane sve funkcije koje se u tom retku pozivaju (osim eventualno funkcije koju definiramo baš u tom retku).

Zbog svega navedenog, iskoristit ćemo parsiranje **liniju-po-liniju**, uz napomenu da to nipošto nije standardni pristup u interpretaciji programa — koristimo ga samo zbog visoke međusobne neovisnosti linija u funkcijskom jeziku.

```
naredbe = []
for linija in izvorni_kod:
    if linija.isspace() : continue
    elif ':=' in linija: P.start = P.definicija
    else: P.start = P.izraz
    naredbe.append(P(linija))
ast = Program(naredbe)
ast.izvrši()
```

Program 2.5.2: Parsiranje liniju-po-liniju

Primijetimo da nije potrebno eksplicitno pozvati lekser — *vepar* osigurava da se lekser pozove u trenutku poziva klase `P`. Niz tokena kojeg lekser generira sprema se u člansku varijablu `stream` klase `P` i spreman je za korištenje u metodama parsera.

Pozivamo parser za svaku liniju izvornog kōda posebno, i rezultat parsiranja zapisujemo u listu `naredbe`. Ako je linija prazna, ne želimo ju parsirati. U suprotnom, provjeravamo nalazi li se podstring „`:=`” u liniji te ovisno o tome razlikujemo definiciju funkcije od evaluacije izraza. `P.start` je funkcija kojom parsiranje započinje, pa ju je potrebno postaviti na željenu funkciju (ako je ne postavimo, početna funkcija će biti ona koju smo napisali „na vrhu” klase `P`).

Provjera nalazi li se određena oznaka u retku je ujedno i najveća prednost parsiranja liniju-po-liniju. Naime, parser (barem u *vepru*) nizom tokena putuje **strogo s lijeva na desno**, bez mogućnosti vraćanja. Drugim riječima, jednom kada pročitamo neki token (operatorom >= ili >>), to čitanje ne možemo poništiti. Da smo odabrali klasičan pristup parsiranja čitavog kōda zajedno, ne bismo mogli provjeriti nalazi li se token `DEF_FUN` u retku, jer ne bismo znali gdje redak završava. U svakom trenutku, parser vidi *samo* iterator na *trenutni* token u nizu kojeg čita, pa je nemoguće predvidjeti koliko je tokena ostalo do kraja svakog retka.

Naposljetku „ručno” stvaramo korijen AST-a pozivanjem konstruktora apstraktne klase `Program`. Prethodnim programom smo pokrili pravilo (2.1) tako da smo ga napisali izvan klase koja predstavlja parser. U standardnom pristupu (program 2.5.1) to ne bismo morali učiniti, jer bi povratni tip poziva klase `P` bila jedna instanca klase `Program`.

Općenita konvencija za pisanje parsera koje ćemo se uglavnom pridržavati je sljedeća: *za svako pravilo unutar beskontekstne gramatike, u parseru postoji pripadajuća metoda koja parsira izraze po tom pravilu*. Naglasimo da ne mora nužno postojati bijekcija između pravila u beskontekstnoj gramatici i metoda parsera — često nam u kōdu trebaju razne pomoćne funkcije koje olakšavaju parsiranje, a dodavanje pripadnih pravila u gramatiku bi je učinilo nepotrebno kompliciranom.

Napišimo funkciju `P.definicija`, po uzoru na pravilo (2.3):

```
1  class P(Parser):
2      zamjena: tuple
3      def definicija(self):
4          ime = self >> T.IME
5          self >> T.OTV
6          lijeve = self.lijeve_varijable()
7          self >> T.ZATV
8          self >> T.DEF_FUN
9          if ime.sadržaj + baseString in funkcije: self.zamjena = (ime, lijeve.copy())
10         izraz = self.izraz()
11         self.zamjena = None
12         return self.definiraj_i_vrati_funkciju(ime, lijeve, izraz)
```

Program 2.5.3: Funkcija `P.definicija`

Prethodni program lijepo prikazuje glavnu ideju parsiranja — funkcije koje su visoko u hijerarhiji apstraktnog sintaksnog stabla, kao što je `P.definicija`, ne znaju parsirati izraze koji su na nižim hijerarhijskim razinama (poput onih koje generiraju varijable `lijeve_varijable` i `izraz`), ali znaju „u kojem trenutku” se takvi izrazi trebaju parsirati, pa pozovu funkcije koje su specijalizirane za njihovo parsiranje. Kōd vezan za varijablu `P.zamjena` objasnit ćemo nešto kasnije.

Kao što vidimo, u parseru također možemo koristiti operatore >, >= i >>. Oni imaju ista značenja kao i u lekseru, samo što sada djeluju na nizu tokena (a ne nizu znakova) koji se generira lekserom „u pozadini” klase `P`. Primjerice, izraz `self >= T.BROJ` čita idući token u nizu ako je on tipa `BROJ` te vraća istinosnu vrijednost. Možemo koristiti i operator ^ koji provjerava pripada li token određenom tipu — primjerice, `var ^ T.BROJ` vraća istinosnu vrijednost s obzirom na to je li varijabla `var` token tipa `BROJ`.

Na početku parsiranja svakog programa pretpostavit ćemo da su poznate jedino inicijalne funkcije. Imat ćemo globalnu listu `funkcije` koja prati koje su funkcije trenutno definirane te ćemo ju postupno povećavati nakon svake nove (ispravne) definicije funkcije.

Pomoćna funkcija `P.definiraj_i_vrati_funkciju` će imati 3 slučaja:

- Ako je zadnja lijeva varijabla `BROJ` sadržaja `0`, onda tu naredbu interpretiramo kao početni uvjet u definiciji funkcije primitivnom rekurzijom (1.3) — imenu funkcije ćemo dodati sufiks `#Base`.
- Ako je zadnja lijeva varijabla poziv funkcije `Sc`, onda tu naredbu interpretiramo kao korak u definiciji funkcije primitivnom rekurzijom (1.4) — imenu funkcije ćemo dodati sufiks `#Step`. Također, zbog lakšeg izvršavanja AST-a, zadnju lijevu varijablu ćemo pretvoriti iz poziva funkcije `Sc` u varijablu unutar tog poziva — primjerice, poziv `Sc(y)` će postati token `IME` sadržaja `y`. Ako je `f` neka funkcija dobivena primitivnom rekurzijom i ako je funkcija `f#Step` funkcija uspješno interpretirana, smatramo da je definicija funkcije `f` potpuna, odnosno funkcija `f` postaje spremna za korištenje (pozivanje) u ostatku izvornog kōda.
- U svim ostalim slučajevima, naredbu interpretiramo kao direktnu definiciju funkcije — ime funkcije ostaje nepromijenjeno.

Povratna vrijednost svakog od ovih slučajeva će biti jedna instanca klase `Definicija`. Svaka `Definicija` će sadržavati 3 bitna svojstva: ime, argumente i izraz s desne strane tokena `DEF_FUN`. U kōdu to izgleda ovako:

```
1  funkcije = ['Z', 'Sc']
2  baseString = '#Base', stepString = '#Step', prevString = '#Prev'
3  class P(Parser):
4      # ... prethodni kod
5      def definiraj_i_vrati_funkciju(self, ime, lijeve, izraz):
6          if isinstance(lijeve[-1], Token) and lijeve[-1].sadržaj == '0':
7              funkcije.append(ime.sadržaj + baseString)
8              return Definicija(Token(T.IME, ime.sadržaj + baseString), lijeve[:-1], izraz)
9          elif isinstance(lijeve[-1], Poziv) and lijeve[-1].ime.sadržaj == 'Sc':
10             funkcije.append(ime.sadržaj + stepString)
11             funkcije.append(ime.sadržaj)
12             lijeve[-1] = lijeve[-1].parametri[0]
13             lijeve.append(Token(T.IME, prevString))
14             return Definicija(Token(T.IME, ime.sadržaj + stepString), lijeve, izraz)
15         funkcije.append(ime.sadržaj)
16         return Definicija(ime, lijeve, izraz)
```

Program 2.5.4: Funkcija `P.definiraj_i_vrati_funkciju`

Napomenimo da smo u listi `funkcije` namjerno izostavili koordinatne projekcije. Njihova imena će biti `I_`n, gdje je n pozitivni prirodni broj. Budući da ih ima prebrojivo mnogo, definirat ćemo ih kao posebnu klasu (slično dokazu propozicije 1.2.3) te ćemo ih u čitavom kōdu tretirati kao poseban slučaj. Kōd vezan za koordinatne projekcije ćemo u ovom radu izostaviti zato što se svodi na nekoliko provjera odgovara li ime neke funkcije regularnom izrazu `'I_\d+'`, što bi nepotrebno skretalo pozornost s važnijih dijelova kōda.

Primijetimo jednu vrlo važnu posljedicu definicije 1.4.1 — funkcije koje predstavljaju inicijalizaciju i korak primitivne rekurzije imaju **različitu mjesnost** od same funkcije. Prisjetimo se da, za dani $k \in \mathbb{N}_+$ i funkcije G^k i H^{k+2}, funkcija $F = G \text{ PR } H$ je $(k+1)$-mjesna. Zaista, u programu 2.5.3 je vidljivo da su povratne vrijednosti funkcije `P.definicija` usklađene s tim — funkcija `#Base` ima jedan argument manje, a funkcija `#Step` jedan argument više (označen s `#Prev`) od broja argumenata s kojima ćemo inače pozivati običnu verziju te funkcije.

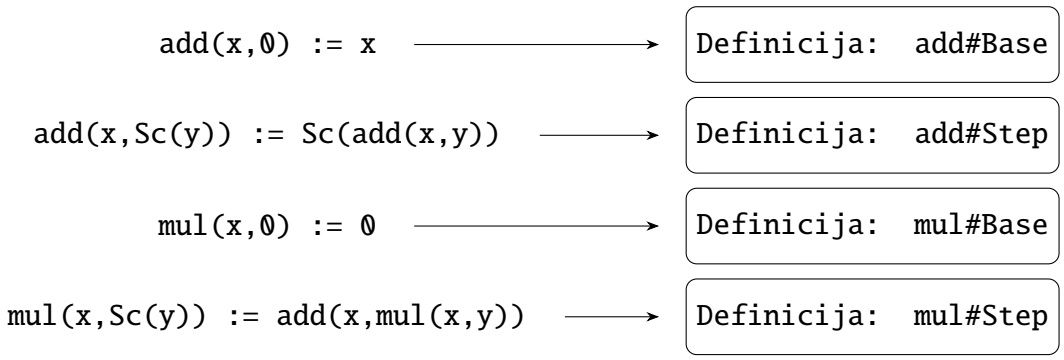
Napišimo sada kōd za beskontekstna pravila (2.6) i (2.8):

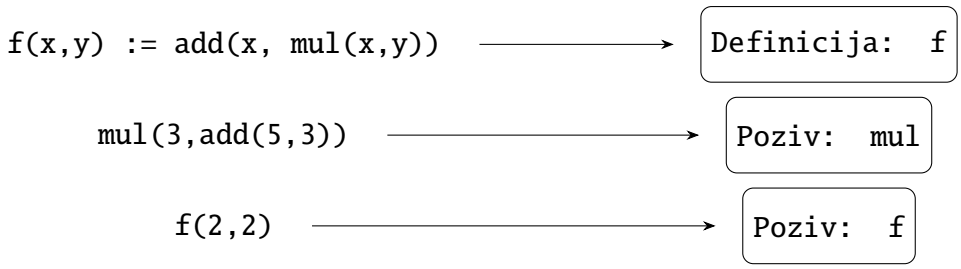
```
1  class P(Parser):
2      # ... prethodni kod
3      def desne_varijable(self):
4          desne = [self.izraz()]
5          while self >= T.ZAREZ:
6              desne.append(self.izraz())
7          return desne
8      def izraz(self):
9          if ime := self >= T.IME:
10             if self > T.OTV:
11                 return self.poziv(ime)
12             return ime
13         else: return self >> T.BROJ
```

Program 2.5.5: Funkcije `P.desne_varijable` i `P.izraz`

Funkcije iz prethodnog odsječka programa tipične su za interpretaciju bilo kojeg jezika — funkcija `P.desne_varijable` više puta poziva funkciju `P.izraz`, koja parsira sve dozvoljene čvorove apstraktnog sintaksnog stabla (u našem slučaju to su imena varijabli, brojevi te pozivi funkcija). Nakon što se jedan izraz uspješno parsira, provjerava se je li sljedeći token `ZAREZ` — ako jest, onda se parsira novi izraz, sve dok se nisu parsirale sve (desne) varijable. Na kraju izvođenja vraća se lista svih parsiranih varijabli.

Pogledajmo koje se instance klasa stvore u listi `naredbe` iz programa 2.5.2 kada parsiramo program 2.1.2:





Dijagram 2.4. Interpretacija programa 2.1.2.

Sada navodimo i sve ostale funkcije parsera `P`, napisane po uzoru na preostala pravila beskontekstne gramatike. Temelje se na sličnim principima koje smo do sada vidjeli, pa većinu ovih funkcija nećemo dodatno pojašnjavati.

```
1  class P(Parser):
2      # ... prethodni kod
3      def lijeve_varijable(self):
4          lijeve = [self.lijeva_varijabla()]
5          while self >= T.ZAREZ:
6              lijeve.append(self.lijeva_varijabla())
7          return lijeve
8      def lijeva_varijabla(self):
9          if ime := self >= T.IME:
10             if self > T.OTV:
11                 return self.poziv_lijeve(ime)
12             return ime
13         return self >> T.BROJ
14     def poziv_lijeve(self, ime):
15         self >> T.OTV
16         varijabla = self >> T.IME
17         self >> T.ZATV
18         return Poziv(ime, [varijabla])
19     def poziv(self, ime):
20         self >> T.OTV
21         desne = self.desne_varijable()
22         self >> T.ZATV
23         if self.zamjena is not None:
24             ime_zamjena, lijeve_zamjena = self.zamjena
25             if isinstance(lijeve_zamjena[-1], Poziv):
26                 lijeve_zamjena[-1] = lijeve_zamjena[-1].parametri[0]
27             if ime_zamjena == ime and lijeve_zamjena == desne:
28                 return Token(T.IME, prevString)
29         return Poziv(ime, desne)
```

Program 2.5.6: Ostale funkcije parsera

Objasnimo sada kōd vezan za varijablu `P.zamjena`. Prisjetimo se da je u koraku definicije primitivne rekurzije (1.4) zadnji parametar funkcije H prethodno izračunata vrijednost funkcije F. To znači da ga upravo tako moramo parsirati — kao varijablu `#Prev`, a ne kao poziv funkcije F. Dakle, u svakom koraku definicije primitivne rekurzije, svako pojavljivanje izraza $F(\vec{x}, y)$ s desne strane tokena `DEF_FUN` moramo zamijeniti tokenom `IME` sadržaja

`#Prev`. U varijabli `P.zamjena` (program 2.5.3) zapisani su ime funkcije koju trenutno parsiramo i lijeve varijable u tom kontekstu. Zato je u funkciji `P.poziv` dovoljno provjeriti podudaraju li se zapamćene varijable s trenutno parsiranim varijablama — ako da, onda vraćamo token `IME` sadržaja `#Prev`, a inače vraćamo običan poziv funkcije.

Za kraj ovog odjeljka pokazujemo još jednu korisnu *veprovu* funkciju — `prikaz`. Kada parsiramo sve linije i stvorimo instancu klase `Program` (koja predstavlja korijen AST-a), možemo pozvati funkciju `prikaz` koja će na standardni izlaz ispisati hijerarhijsku strukturu čitavog stabla. Parsirajmo program 2.1.1 i pozovimo funkciju `prikaz`:

```
# ... prethodni kod
ast = Program(naredbe)
prikaz(ast)
```

Program 2.5.7: Poziv funkcije `prikaz`

```
Program:
  naredbe = [...]:
  . Definicija:  @[Znakovi #1-kraj]
  .   ime = IME'add#Base'
  .   parametri = [...]:
  .   . IME'x'  @[Znak #5]
  .   izraz = BROJ'0'  @[Znak #13]
  . Definicija:  @[Znakovi #5-kraj]
  .   ime = IME'add#Step'
  .   parametri = [...]:
  .   . IME'x'  @[Znak #9]
  .   . IME'y'  @[Znak #14]
  .   . IME'#Prev'
  .   izraz = Poziv:  @[Znakovi #21-kraj]
  .     ime = IME'Sc'  @[Znakovi #21-#22]
  .     parametri = [...]:
  .     . IME'#Prev'
  . Poziv:  @[Znakovi #5-kraj]
  .   ime = IME'add'  @[Znakovi #5-#7]
  .   parametri = [...]:
  .   . BROJ'5'  @[Znak #9]
  .   . BROJ'3'  @[Znak #11]
```

Program 2.5.8: Ispis funkcije `prikaz`

2.6 Klase unutar AST-a

Neke metode parsera su vraćale instance klasa koje predstavljaju čvorove u apstraktnom sintaksnom stablu. U ovom odjeljku ćemo vidjeti kako se te klase implementiraju. Pogledajmo prvo kostur apstraktne klase `Program` koja predstavlja korijen AST-a:

```
class Program(AST):
    naredbe: List[AST]
    def izvrši(self):
        for naredba in self.naredbe:
            naredba.izvrši()
```

Program 2.6.1: Kostur klase `Program`

Vidimo da metoda `Program.izvrši` redom poziva metode `izvrši` svake pojedine naredbe, koje onda dalje pozivaju metode `izvrši` svoje djece. To rezultira obilaskom *postorder* po AST-u i izvršavanjem svakog pojedinog čvora. Naravno, svaki čvor u AST-u će imati posebnu implementaciju svoje funkcije `izvrši`, koja odgovara semantičkom značenju tog čvora.

U početku, prije interpretiranja naredbi u izvornom kōdu, trebalo bi omogućiti korištenje inicijalnih funkcija. Dakle, ako korisnik odluči u prvom retku napisati `Sc(5)`, ta naredba bi trebala rezultirati ispisom broja 6 u konzolu, jer je `Sc` u tom trenutku *poznata* funkcija. Očito je da svakom novom (potpunom) definicijom funkcije u izvornom kōdu stvaramo novu funkciju kojom bismo trebali proširiti skup poznatih funkcija.

Sve trenutno poznate funkcije, kao i vrijednosti varijabli poznatih u trenutnom kontekstu, ubacivat ćemo u instancu *veprove* klase `Memorija`. To je objekt koji se ponaša gotovo identično kao i Pythonov *dictionary* (rječnik), ali i sadrži neke pogodnosti koje ga čine boljom opcijom od rječnika. Primjerice, korisnik ne mora razmišljati koristi li tokene ili stringove kao ključeve u memoriji:

```
memorija = Memorija()  # prazna memorija
memorija['x'] = 5
ime = Token(T.IME, 'x')
print(memorija[ime])  # 5
print(memorija[ime.sadržaj])  # 5
```

Program 2.6.2: Primjer korištenja `Memorije`

Glavna ideja je da se instanca klase `Memorija` prosljeđuje po apstraktnom sintaksnom stablu (kao argument funkcije `izvrši`) te se postupno popunjava novim funkcijama. Naravno, vrijednosti objekta `Memorija` mogu biti *bilo što*, pa tako i instance klase `Definicija`.

Bitno je razlikovati imena funkcija od imena varijabli u danom kontekstu. Primjerice, htjeli bismo omogućiti sljedeće:

```
f(x) := Sc(x)
g(f) := f(Sc(f))
g(5) // 7
```

Program 2.6.3: Razlikovanje imena funkcija i imena varijabli

Prethodni program bi trebao bez problema shvatiti da argument `f` funkcije `g` predstavlja običnu varijablu te da ni na koji način nije povezan s funkcijom `f` koju smo ranije definirali. Također, trebao bi ispravno razlikovati tokene `IME'f'` u izrazu `f(Sc(f))` — vanjski `f` je poziv funkcije, a unutarnji je varijabla u kontekstu tog retka. To ćemo postići tako da sve funkcije `izvrši` imaju dva argumenta — jedan predstavlja varijable koje „postoje" isključivo u tom retku, a drugi predstavlja sve dosad definirane funkcije. Na ovaj način ćemo lako izbjeći moguću dvosmislenost iz prethodnog programa.

U programu 2.1.2 je vidljivo da se naredbe dijele na one koje ne ispisuju ništa (nego samo modificiraju Memoriju) i na one koje ispisuju neki rezultat. U ovom radu nećemo praviti razliku između *izvršavanja* i *izračunavanja vrijednosti* naredbi. Ako program 2.6.1 ostane nepromijenjen, onda bismo unutar svake klase trebali implementirati dvije funkcije: `vrijednost`, koja računa i vraća vrijednost te klase i `izvrši`, koja ispisuje vrijednost te klase na standardni izlaz. Ovaj pristup je potpuno objektno orijentiran, jer svaka klasa zasebno odlučuje treba li se rezultat njenog izvršavanja ispisati. Ipak, funkcije `vrijednost` i `izvrši` zapravo provode vrlo sličan postupak, a budući da se samo rezultat izvršavanja definicije funkcije ne ispisuje, u svakoj klasi ćemo implementirati samo funkciju `izvrši`, a odluku o ispisivanju rezultata ćemo prepustiti klasi `Program`. Zbog svega navedenog, modificirat ćemo program 2.6.1, a usput ćemo definirati nulfunkciju i sljedbenika.

```
1  class Program(AST):
2      naredbe: List[AST]
3      def izvrši(self):
4          funkcije = Memorija()
5          self.def_inicijalne(funkcije)
6          for naredba in self.naredbe:
7              rez = naredba.izvrši(Memorija(), funkcije)
8              if not isinstance(naredba, Definicija): print(rez)
9      def def_inicijalne(self, funkcije):
10         funkcije[Token(T.IME, 'Z')] = PythonFunction(Token(T.IME, 'Z'), Z)
11         funkcije[Token(T.IME, 'Sc')] = PythonFunction(Token(T.IME, 'Sc'), Sc)
12
13 class PythonFunction(AST):
14     ime: Token
15     izraz: AST
16     def pozovi(self, argumenti, memorija, funkcije):
17         return funkcije[self.ime].izraz(argumenti[0])
```

Program 2.6.4: Klase `Program` i `PythonFunction`

Klasom `PythonFunction` želimo naglasiti da postoji razlika između funkcija čije ponašanje definiramo izvornim kōdom i inicijalnih funkcija čije je ponašanje ugrađeno u interpreter. Djelovanje većine kompliciranijih funkcija koje definiramo počivat će na višestrukim pozivima inicijalnih funkcija — zamislimo samo koliko se puta treba pozvati funkcija `Sc` da bi se izračunalo `mul(100,100)`.

Napišimo sada glavne klase AST-a — `Definicija` i `Poziv`. Ako se naredba parsirala kao definicija funkcije, htjeli bismo da izvršavanje te naredbe spremi tu funkciju u me-

moriju. S druge strane, ako je naredba poziv funkcije, onda trebamo pronaći tu funkciju u memoriji i pozvati je s određenim ulaznim argumentima. Taj postupak ćemo detaljno opisati na primjeru izvršavanja programa 2.1.1:

- Prvo se izvršava redak `add(x,0) := x`, koji se parsirao kao `Definicija` funkcije imena `add#Base`, čiji je parametar `x` (jer nulu ne smatramo parametrom) i izraz `x`. Ovakvu funkciju (instancu klase `Definicija`) ubacujemo (s ključem `add#Base`) u varijablu `funkcije` (instancu klase `Memorija`) koja predstavlja sve dosad definirane funkcije.

- Sljedeći redak koji izvršavamo je `add(x,Sc(y)) := Sc(add(x,y))`, koji se parsirao kao `Definicija` funkcije imena `add#Step`, s parametrima `x`, `y` i `#Prev` te izrazom koji je instanca klase `Poziv` (s imenom `Sc` i parametrom `#Prev`). Budući da je ovo definicija koraka funkcije, a definicija baze je već unutar varijable `funkcije`, smatramo da je funkcija `add` ispravno definirana primitivnom rekurzijom. U varijablu `funkcije` prvo ubacujemo `add#Step`, a potom i `add`, koja sve parametre kao i `add#Step`, osim zadnjeg.

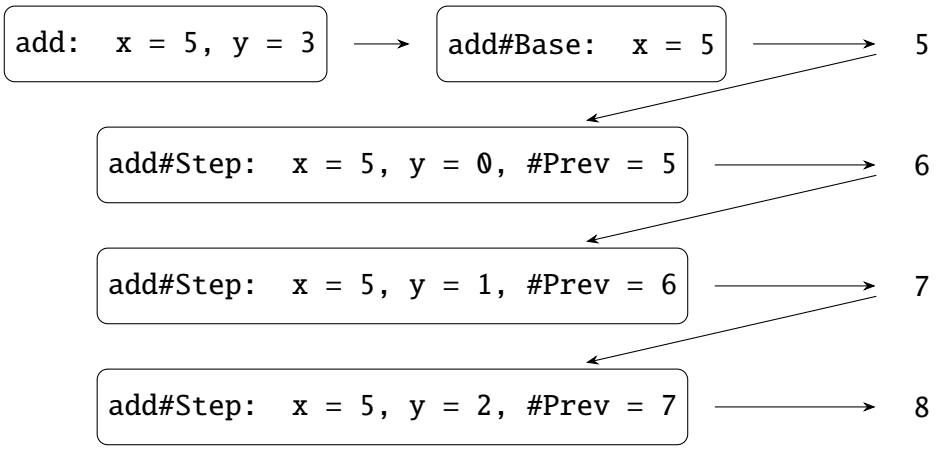
- Zadnji redak je `add(5,3)`, koji se parsirao kao `Poziv` funkcije imena `add`, s argumentima `5` i `3`. Kako bismo izvršili poziv funkcije, prvo moramo saznati vrijednosti svih njenih argumenata. Zato prvo izvršavamo svaki argument — u programu 2.2.2 vidimo da izvršavanje tokena tipa `BROJ` vraća prirodan broj čiji je dekadski zapis sadržaj tog tokena. Nakon što su nam vrijednosti arugmenata poznate, uz pomoć ključa `add` pristupamo željenoj instanci klase `Definicija` unutar varijable `funkcije` te *pozivamo* tu funkciju — zovemo funkciju `Definicija.pozovi` koja ima tri slučaja (od kojih se prvo dogodi *drugi* slučaj):

  - Ako je pozvana funkcija `#Base`, onda izvršavamo njen izraz s lokalnom memorijom. Kako bismo spojili imena lokalnih varijabli s njihovim vrijednostima, koristimo Pythonovu funkciju `zip` funkciju koja stvara iterator uređenih parova imena i vrijednosti. U našem primjeru, izvršavamo izraz `x` funkcije `add#Base` s lokalnom memorijom u kojoj se nalazi varijabla `x` vrijednosti `5`. Rezultat izvršavanja (broj `5`) je ujedno i rezultat funkcije `Definicija.poziv`.
  - Inače, ako je pozvana funkcija definirana primitivnom rekurzijom, onda provodimo postupak primitivne rekurzije. Prvo pozivamo funkciju `add#Base` s jednim argumentom manje, čije izvršavanje je opisano u prethodnom slučaju. Dobivenu vrijednost zapamtimo u varijabli `z` i koristimo je kao vrijednost parametra `#Prev` u idućoj petlji gdje provodimo korak primitivne rekurzije. Funkciju `add#Step` pozivamo s argumentima `x`, `i` te `z`, gdje je `x` jedina varijabla konteksta (jednaka `5`), a `i` varijabla po kojoj iteriramo (jednaka redom `0`, `1` i

2). Vrijednost svakog takvog poziva zapisujemo ponovno u varijablu `z`, koju na kraju iteriranja vratimo iz metode `Definicija.poziv`.

– Preostale su dvije mogućnosti: pozvana je verzija `#Step` funkcije ili je pozvana funkcija koja je definirana direktno (ne primitivnom rekurzijom). U oba slučaja postupamo na jednak način — izvršavamo izraz te funkcije s lokalnom memorijom.

Na idućem dijagramu prikazujemo proces evaluacije vrijednosti izraza `add(5,3)`, odnosno slijedno izvršavanje prethodna tri slučaja funkcije `Definicija.pozovi`.



Dijagram 2.5. Izvršavanje poziva `add(5,3)`.

U općenitom slučaju, argumenti poziva funkcije neće biti toliko trivijalni kao što su `5` i `3`. Primjerice, izvršavanje retka `add(add(3,2),add(1,2))` očito zahtijeva izvršavanje više ugniježđenih `Poziv`-a. No, kao što smo ranije naglasili, svako ugnježđivanje mora završiti listovima — u nekom trenutku se moraju evaluirati `BROJ`-evi.

```
class Definicija(AST):
    ime: Token
    parametri: List[AST]
    izraz: AST
    def izvrši(self, memorija, funkcije):
        funkcije[self.ime] = self
        if stepString in self.ime.sadržaj:
            ime = Token(T.IME, self.ime.sadržaj[:-len(stepString)])
            funkcije[ime] = Definicija(ime, self.parametri[:-1], self.izraz)
    def pozovi(self, argumenti, memorija, funkcije):
        bIme = self.ime.sadržaj + baseString, sIme = self.ime.sadržaj + stepString
        if baseString in self.ime.sadržaj:
            return self.izraz.izvrši(Memorija(zip(self.parametri, argumenti)), funkcije)
        elif stepString not in self.ime.sadržaj and sIme in funkcije:
            z = funkcije[bIme].pozovi(argumenti[:-1], memorija, funkcije)
            for i in range(argumenti[-1]):
                args = argumenti[:-1]
                args.append(i), args.append(z)
                z = funkcije[sIme].pozovi(args, memorija, funkcije)
            return z
        return self.izraz.izvrši(Memorija(zip(self.parametri, argumenti)), funkcije)
class Poziv(AST):
    ime: Token
    parametri: List[AST]
    def izvrši(self, memorija, funkcije):
        argumenti = [parametar.izvrši(memorija, funkcije) for parametar in self.parametri]
        return funkcije[self.ime].pozovi(argumenti, memorija, funkcije)
```

Program 2.6.5: Klase `Definicija` i `Poziv`

2.7 Relacije i logički izrazi

Dosadašnja verzija interpretera podržava bilo kakvu definiciju funkcije primitivnom rekurzijom, pa tako i definiciju karakteristične funkcije skupa pozitivnih prirodnih brojeva $\chi_{\mathbb{N}_+}$:

```
Positive(0) := 0
Positive(Sc(x)) := 1
Positive(0)   // 0
Positive(42)  // 1
```

Program 2.7.1: Funkcija $\chi_{\mathbb{N}_+}$

`Positive` iz prethodnog programa može se smatrati i relacijom $\mathbb{N}_+$ (a ne samo njenom karakterističnom funkcijom), ako prihvatimo konvenciju da nulu interpretiramo kao laž, a jedinicu kao istinu. Dakle, za neku relaciju R^k i za neki ulaz $\vec{x}^k$, ako poziv $R(\vec{x})$ u interpreteru rezultira ispisom nule, to znači da $\vec{x} \notin R$, dok ispis jedinice znači da $\vec{x} \in R$. To znači da su, uz ovakvu konvenciju, relacije već implementirane — to su funkcije s domenom N^k i kodomenom $\{0, 1\}$. Po uzoru na primjer 1.4.11, možemo definirati i relaciju $>$:

```
sub(x,0) := x
sub(x,Sc(y)) := pd(sub(x,y))
Greater(x,y) := Positive(sub(x,y))
Greater(2,2)  // 0
Greater(3,2)  // 1
```

Program 2.7.2: Relacija $>$

Ipak, za nešto složenije relacije zgodno je uvesti *logičke operatore*. Primijetimo da trenutna verzija interpretera omogućava „simuliranje" logičkih operatora — primjerice, negaciju možemo lagano implementirati direktnom definicijom:

```
Not(x) := sub(1, x)
Not(1)  // 0
Not(Positive(0))  // 1
```

Program 2.7.3: Funkcijska implementacija negacije

Dakle, uvođenjem logičkih operatora samo obogaćujemo sintaksu i olakšavamo korištenje interpretera, a skup funkcija koje možemo definirati ne povećavamo — za sada su to samo primitivno rekurzivne funkcije (i relacije).

Želimo uvesti logičke operatore za disjunkciju(`||`), konjunkciju (`&&`) i negaciju(`!`) te ih parsirati u skladu sa standardnim prioritetima (`!`, pa `&&`. pa `||`) logičkih operatora. Htjeli bismo omogućiti ovakve izraze:

```
Equal(x,y) := !Greater(x,y) && !Greater(y,x)
Equal(0,0)  // 1
Equal(0,42)  // 0
Greater(5,5) || Equal(5,5)  # 1
```

Program 2.7.4: Primjeri logičkih operatora

Kao što vidimo, logički izrazi bi trebali biti omogućeni na desnoj strani tokena `DEF_FUN`, ali i kao zasebni poziv. To znači da funkcija `P.izraz` iz programa 2.5.5 treba znatno proširiti opseg izraza koje je u stanju parsirati. Također, potrebno je promijeniti beskontekstno pravilo (2.8) i dodati nova pravila koja generiraju logičke izraze. Ostale promjene (dodavanje novih tipova tokena koji predstavljaju logičke operatore u klasu `T` te izmjena leksera zbog novih tokena) su dovoljno jednostavne pa ćemo ih izostaviti.

$$\texttt{izraz} \to \texttt{log\_ili} \tag{2.9}$$

$$\texttt{log\_ili} \to \texttt{log\_i} \mid \texttt{log\_i || log\_ili} \tag{2.10}$$

$$\texttt{log\_i} \to \texttt{log\_literal} \mid \texttt{log\_literal \&\& log\_i} \tag{2.11}$$

$$\texttt{log\_literal} \to \texttt{!log\_literal} \mid \texttt{(izraz)} \mid \texttt{list} \tag{2.12}$$

$$\texttt{list} \to \texttt{IME} \mid \texttt{BROJ} \mid \texttt{poziv} \tag{2.13}$$

Svaki izraz će se sada parsirati kao disjunkcija više različitih konjunkcija logičkih literala. Ako usporedimo pravila (2.8) i (2.12), primjećujemo da je varijabla `log_literal` preuzela ulogu generiranja listova. Uz listove, `log_literal` generira negaciju i logičke izraze koji se nalaze u zagradama (kojima ćemo na standardni način moći precizirati prioritet parsiranja logičkih operatora). Pogledajmo implementaciju ovih beskontekstnih pravila:

```
class P(Parser):
    # ... prethodni kod
    def izraz(self):
        return self.log_ILI()
    def log_ILI(self):
        disjunkcija = [self.log_I()]
        while self >= T.LOG_ILI: disjunkcija.append(self.log_I())
        return Log_ILI.ili_samo(disjunkcija)
    def log_I(self):
        konjunkcija = [self.log_literal()]
        while self >= T.LOG_I: konjunkcija.append(self.log_literal())
        return Log_I.ili_samo(konjunkcija)
    def log_literal(self):
        if self >= T.LOG_NE: return Log_NE(self.log_literal())
        elif self >= T.OTV:
            izraz = self.izraz()
            self >> T.ZATV
            return izraz
        else: return self.list()
```

Program 2.7.5: Parsiranje logičkih izraza

Vidimo da svaka funkcija u prethodnom programu strogo prati svako pripadno beskontekstno pravilo. Zasad ćemo izostaviti funkciju `P.list`, jer ćemo u iduća dva odjeljka definirati nove čvorove. Oni će biti složeniji od klasičnih listova, ali ih svejedno parsira ista funkcija `P.list`, jer se (sintaksno) smiju pojavljivati na istim mjestima kao i listovi.

Preostaje samo pojasniti *veprovu* metodu `ili_samo` klase AST. Primjer njenog poziva je `Log_ILI.ili_samo(disjunkcija)`, gdje je `disjunkcija` lista čvorova AST-a. Ako je duljina te liste 1, onda funkcija `ili_samo` vraća samo `disjunkcija[0]`, a inače vraća `Log_ILI(disjunkcija)`.

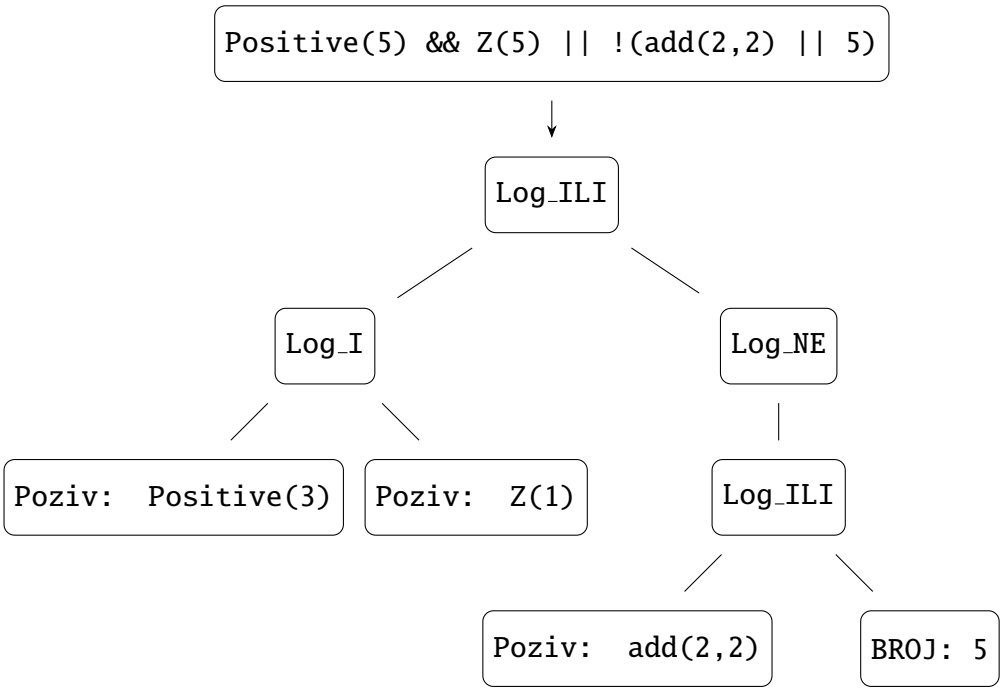
To je iznimno korisno u kontekstu našeg interpretera. Naime, zamislimo kako bi se parsirao izraz `Sc(add(x,y))` bez funkcije `ili_samo`. Dobili bismo jednu instancu klase `Log_ILI`, u kojoj se nalazi jedna instanca klase `Log_I`, u kojoj se nalazi jedna instanca klase `Poziv`. Jasno je da izraz `Sc(add(x,y))` još uvijek želimo parsirati kao `Poziv` te da ništa ne dobivamo njegovim ugnježđivanjem u dvije dodatne klase. Štoviše, bez funkcije `ili_samo` potrošili bismo znatno više memorije računala (zbog dodatnih instanca klasa) te usporili izvršavanje AST-a (zbog dodatnih poziva funkcija `izvrši`).

Prije nego što napišemo pripadne klase AST-a, uvedimo još jednu konvenciju. Budući da u interpreteru ne pravimo bitnu razliku između funkcija i relacija, htjeli bismo da logički operatori rade sa svim prirodnim brojevima, a ne samo s nulom i jedinicom. Pozitivne

prirodne brojeve u svim logičkim izrazima ćemo smatrati ekvivalentnima jedinici, odnosno svaki pozitivan broj predstavlja istinu, dok nula (kao i prije) predstavlja laž.

```
class Log_ILI(AST):
    disjunkcija: List[AST]
    def izvrši(self, memorija, funkcije):
        for konjunkcija in self.disjunkcija:
            value = konjunkcija.izvrši(memorija, funkcije)
            if value: return value
        return 0
class Log_I(AST):
    konjunkcija: List[AST]
    def izvrši(self, memorija, funkcije):
        value = 0
        for literal in self.konjunkcija:
            value = literal.izvrši(memorija, funkcije)
            if value == 0: return 0
        if len(self.konjunkcija) == 1: return value
        return 1
class Log_NE(AST):
    literal: AST
    def izvrši(self, memorija, funkcije):
        value = self.literal.izvrši(memorija, funkcije)
        if value: return 0
        return 1
```

Program 2.7.6: Klase Log_ILI, Log_I i Log_NE



Dijagram 2.6. Parsiranje kompliciranijeg logičkog izraza.

2.8 Minimizacija, brojeća funkcija i grananje

Ovaj odjeljak posvećujemo složenijim čvorovima — minimizaciji, brojećoj funkciji i grananju. Naglasimo činjenicu da je minimizacija jedina operacija koja u teorijskom smislu proširuje skup funkcija koje se mogu definirati (na skup parcijalno rekurzivnih funkcija), dok su ostale dvije operacije samo sintaksne pokrate koje olakšavaju rad s interpreterom.

Za početak, pogledajmo primjere izraza koje bismo htjeli parsirati:

```
1 div(x,y) := pd((mu z <= x) Greater(mul(z,y),x))
2 spora_identiteta(x) := (# d < x) Greater(x,d)
3 veci(x,y) := if[Greater(x,y): x, y]
4 div(7,3)  // 2
5 spora_identiteta(5)  // 5
6 veci(2,3)  // 3
7 add((mu x < 5) Greater(x,2), (# x < 5) Greater(x,0))  // 7
```

Program 2.8.1: Primjeri minimizacije, brojeće funkcije i grananja

Iskoristili smo funkciju `Greater` (definiranu u programu 2.7.2) kako bismo definirali primjere korištenja svih operacija koje planiramo implementirati. Zadnji redak programa 2.8.1 (osim što predstavlja poprilično ekstreman način zbrajanja brojeva 3 i 4) ilustrira činjenicu da se operacije koje definiramo mogu pojavljivati na *istim* mjestima kao i obični listovi AST-a. Primijetimo suptilnu razliku između minimizacija u prvom i zadnjem retku programa 2.8.1: rezultat minimizacije u prvom retku je *funkcija* `div` koja ovisi o varijablama `x` i `y`, dok je rezultat minimizacije u zadnjem retku *broj* 3, jer ta minimizacija ne ovisi ni o jednoj „vanjskoj" varijabli.

$$\texttt{list} \to \texttt{IME} \mid \texttt{BROJ} \mid \texttt{poziv} \mid \texttt{min} \mid \texttt{brojeća} \mid \texttt{grananje} \tag{2.14}$$

$$\texttt{min} \to \texttt{op\_min izraz} \tag{2.15}$$

$$\texttt{brojeća} \to \texttt{op\_br izraz} \tag{2.16}$$

$$\texttt{grananje} \to \texttt{IF[unutar\_gr]} \tag{2.17}$$

$$\texttt{op\_min} \to \texttt{MU IME} \mid \texttt{MU IME nejednakost list} \mid \texttt{(MU IME)} \mid \texttt{(MU IME nejednakost list)} \tag{2.18}$$

$$\texttt{op\_br} \to \texttt{CARD IME nejednakost list} \mid \texttt{(CARD IME nejednakost list)} \tag{2.19}$$

$$\texttt{unutar\_gr} \to \texttt{izraz: izraz, unutar\_gr} \mid \texttt{izraz} \tag{2.20}$$

$$\texttt{nejednakost} \to \texttt{<} \mid \texttt{<=} \tag{2.21}$$

Želimo omogućiti korištenje ograničene i neograničene minimizacije (pravilo (2.18)). Također, zadavanje varijable po kojoj izvršavamo minimizaciju ili brojenje ne moramo nužno

staviti u zagrade (pravila (2.18) i (2.19)), a znak nejednakosti može biti strog ili inkluzivan (pravilo (2.21)).

Prikazat ćemo samo implementacijski kōd vezan za minimizaciju, dok ćemo ostale dvije operacije izostaviti. Naime, brojeća funkcija se parsira na gotovo isti način kao ograničena minimizacija, a tehnike koje se koriste u parsiranju grananja smo već više puta vidjeli. Izvršavanje klase `Brojeća` se svodi na iterativno izvršavanje pripadne relacije i brojenje koliko je tih izvršavanja rezultiralo pozitivnim brojem. S druge strane, izvršavanje klase `Grananje` se svodi na implementaciju uređenog grananja — u onom trenutku kada izvršavanje nekog *uvjeta* rezultira pozitivnim brojem, vraćamo vrijednost pripadne *grane* (u skladu s nazivima uvedenima u definiciji 1.6.4).

Pogledajmo prvo kako ćemo pozivati funkciju `P.minimizacija`:

```
class P(Parser):
    # ... prethodni kod
    def log_literal(self):
        if self >= T.LOG_NE: return Log_NE(self.log_literal())
        elif self >= T.OTV:
            if self > T.MU: return self.minimizacija(otvorena=True)
            if self > T.CARD: return self.brojeca(otvorena=True)
            izraz = self.izraz()
            self >> T.ZATV
            return izraz
        else: return self.list()
    def list(self):
        if ime := self >= T.IME:
            if self > T.OTV:
                return self.poziv(ime)
            return ime
        if self > T.MU: return self.minimizacija(otvorena=False)
        if self > T.CARD: return self.brojeća(otvorena=False)
        if self > T.IF: return self.grananje()
        return self >> T.BROJ
```

Program 2.8.2: Funkcije `P.log_literal` i `P.list`

Zbog pravila (2.18), želimo omogućiti parsiranje uvjeta minimizacije sa ili bez zagrada. Uzmimo izraz `(mu x < 5) Greater(x,2)` kao primjer jedne minimizacije. Njegovo parsiranje će započeti pozivom funkcije `P.izraz` i nastaviti se pozivima funkcija (redom) `P.log_ILI`, `P.log_I` i `P.log_literal`. Funkcija `P.log_literal` će pročitati otvorenu zagradu, ali to *nije* ona otvorena zagrada iz pravila (2.12), već zagrada koja je dio sintakse minimizacije. Naime, uvjet `(mu x < 5)` ne smijemo parsirati kao „izraz u zagradama”, jer taj tekst sam po sebi nema cjelovito značenje, pa ga ne možemo ni smatrati izrazom (nečim što bi parsirala funkcija `P.izraz`). On dobiva značenje tek kada se nakon njega parsira neka relacija — tada to postaje čvor koji predstavlja minimizaciju.

Dakle, nakon što funkcija `log_literal` pročita otvorenu zagradu, moramo provjeriti je li sljedeći token `MU` ili `CARD` — ako jest, onda pozivamo odgovarajuću funkciju s istinitim

parametrom `otvorena`, koji signalizira funkciji da je pročitana jedna otvorena zagrada prije njenog poziva. U suprotnome, parsiramo običan izraz.

Funkcija `P.minimizacija` prvo parsira tokene `MU` i `IME`, a nakon toga provjerava je li idući token neka vrsta nejednakosti te sukladno tome stvara instancu klase `Ograničena_Minimizacija` odnosno `Neograničena_Minimizacija`. U slučaju da je pročitana inkluzivna nejednakost, varijablu koja predstavlja gornju ogradu za ograničenu minimizaciju ćemo povećati za jedan. Ako je jedna otvorena zagrada pročitana prije poziva funkcije `P.minimizacija`, onda nužno moramo pročitati i zatvorenu zagradu prije parsiranja relacije.

```
1  class P(Parser):
2      # ... prethodni kod
3      def minimizacija(self, otvorena):
4          self >> T.MU
5          varijabla = self >> T.IME
6          if nejednakost := self >= {T.MJEDNAKO, T.MANJE}:
7              plus = 1 if nejednakost ^ T.MJEDNAKO else 0
8              ograda = self.list()
9              if otvorena: self >> T.ZATV
10             relacija = self.izraz()
11             return Ograničena_Minimizacija(varijabla, plus, ograda, relacija)
12         else:
13             if otvorena: self >> T.ZATV
14             relacija = self.izraz()
15             return Neograničena_Minimizacija(varijabla, relacija)
```

Program 2.8.3: Funkcija `P.minimizacija`

Implementacije pripadnih klasa bit će vrlo slične — jedina prava razlika je (ne)ograničenost petlje. Varijablu unutar uvjeta minimizacije ćemo inicijalizirati na nulu, ubacit ćemo je u Memoriju i izvršiti parsiranu relaciju. Ponavljat ćemo taj postupak (uz inkrementiranje varijable) sve dok relacija ne vrati istinitu vrijednost (pozitivni broj). Kada se to dogodi, potrebno je obrisati varijablu iz Memorije, jer ta varijabla postoji samo u kontekstu minimizacije (a ne i u ostatku retka kojeg interpretiramo).

```
class Ograničena_Minimizacija(AST):
    varijabla: Token
    plus: int
    ograda: AST
    relacija: AST
    def izvrši(self, memorija, funkcije):
        ograda = self.ograda.izvrši(memorija, funkcije) + self.plus
        for i in range(ograda):
            memorija[self.varijabla] = i
            if self.relacija.izvrši(memorija, funkcije):
                del memorija[self.varijabla]
                return i
        return ograda

class Neograničena_Minimizacija(AST):
    varijabla: Token
    relacija: AST
    def izvrši(self, memorija, funkcije):
        i = 0
        while ...:
            memorija[self.varijabla] = i
            if self.relacija.izvrši(memorija, funkcije):
                del memorija[self.varijabla]
                return i
            i += 1
```

Program 2.8.4: Klase `Ograničena_Minimizacija` i `Neograničena_Minimizacija`

Pokažimo sada primjerom da se minimizacijom stvarno mogu definirati parcijalne funkcije unutar interpretera. Prisjetimo se: izračunavanje prazne funkcije $\otimes$ ne stane ni za jedan ulaz. Dakle, svaku funkciju (definiranu izvornim kōdom interpretera) koja za svaki ulaz tijekom izvršavanja „zapne" u beskonačnoj petlji smatramo praznom funkcijom. Sljedeći program daje nekoliko jednostavnih definicija prazne funkcije.

```
prazna1(x) := (mu y) 0
prazna2(x) := (mu y) Z(x)
prazna3(x) := (mu y) Greater(0,y)
prazna1(42)  // beskonacna petlja
```

Program 2.8.5: Prazna funkcija

2.9 Infiksni operatori

U interpreteru možemo definirati i koristiti osnovne matematičke operacije, ali je sintaksa ponekad zahtjevna za čitanje. Primjerice, ako pogledamo izraz s puno ugniježđenih poziva, poput `add(sub(x,mul(x,y)),div(x,add(y,3)))`, nije odmah jasno da on računa matematički izraz $x - x \cdot y + \lfloor \frac{x}{y+3} \rfloor$. U ovom odjeljku ćemo pokušati ublažiti ovaj problem uvođenjem *infiksnih operatora*. Uz pomoć njih ćemo također moći definirati funkcije

(mjesnosti 2), ali će njihovo pozivanje biti u sintaksnom smislu razumljivije od dosadašnjih funkcija. Pogledajmo primjer:

```
[x + 0] := x
[x + Sc(y)] := Sc([x+y])
[x - y] := sub(x,y)
[2 + 3]  // 5
[8 - [2 + 2]]  // 4
```

Program 2.9.1: Primjer zadavanja infiksnih operatora

Kao što vidimo, infiksni operatori se sastoje od uglatih zagrada unutar kojih se nalaze dva izraza razdvojena operatorom. Želimo omogućiti definiciju infiksnih operatora primitivnom rekurzijom, kao i direktnu definiciju. U slučaju definicije infiksnog operatora `[+]` primitivnom rekurzijom, možemo primijetiti sljedeću činjenicu: prethodno izračunana vrijednost `#Prev` izgleda kao poziv infiksnog operatora `[x+y]`, što znači da će parsiranje takvog izraza biti vrlo slično parsiranju običnih funkcija definiranih primitivnom rekurzijom. Također, direktnom definicijom infiksnog operatora `[-]` izjednačavamo njegovo izračunavanje s izračunavanjem izraza s desne strane tokena `DEF_FUN` (u ovom slučaju s funkcijom `sub`). Zbog svega navedenog, možemo zaključiti da je izvršavanje infiksnih operatora **ekvivalentno izvršavanju funkcija**. Dakle, potrebno je implementirati samo sintaksu (u parseru) za infiksne operatore, a ne treba raditi dodatne klase AST-a, jer će infiksni operatori biti instance klasa `Definicija` i `Poziv`.

Prvo je potrebno ispravno tokenizirati operator koji se nalazi u sredini. Dogovorno, on smije biti bilo koji znak koji nije ni slovo niti broj, a nije jednak sadržaju nekog literala. Uvest ćemo novi tip složenijeg tokena `OP` te ćemo unutar leksera na samom kraju dodati slučaj koji odašilje tokene tog tipa. Naime, ako je lekser došao do tog slučaja, onda je jasno da trenutni znak koji se čita ne može biti ni slovo niti broj.

```
class T(TipoviTokena):
    # ... prethodni kod
    class OP(Token): pass

@lexer
def lekser(ulaz):
    for znak in ulaz:
        # ... prethodni kod
        elif znak.isalpha() or znak == '_':
            ulaz * {str.isalnum, '_'}
            yield ulaz.literal_ili(T.IME)
        elif znak.isdecimal():
            ulaz.prirodni_broj(znak)
            yield ulaz.token(T.BROJ)
        else:
            yield ulaz.literal_ili(T.OP)
```

Program 2.9.2: Tokenizacija operatora

Napomenimo da sadržaji tokena tipa OP nisu fiksni (jer ih definira korisnik), pa je nužno da to bude složeni tip tokena. Za razliku od složenih tipova tokena IME i BROJ, nikad nam neće trebati eksplicitno izvršavanje tokena OP, pa potklasa OP smije biti prazna.

```
1 class P(Parser):
2     def definicija(self):
3         if self > T.UGOTV: return self.definicija_infix()
4         # ... ostatak koda je nepromijenjen
5     def list(self):
6         if self > T.UGOTV: return self.poziv_infix()
7         # ... ostatak koda je nepromijenjen
```

Program 2.9.3: Poziv funkcije `P.defincija_infix`

Ako se sadržaj tokena DEF_FUN nalazi unutar linije, onda ta linija može predstavljati definiciju obične funkcije ili definiciju infiksnog operatora. Ta dva slučaja lako možemo razlikovati ako pogledamo prvi token u liniji — ako je on otvorena uglata zagrada, onda je to definicija infiksnog operatora. Također, htjeli bismo da poziv infiksnog operatora bude jedan mogući čvor (kojeg parsira funkcija `P.list`).

$$\texttt{definicija} \to \texttt{IME(lijeve\_varijable) := izraz} \mid \texttt{definicija\_infix} \tag{2.22}$$

$$\texttt{list} \to \texttt{IME} \mid \texttt{BROJ} \mid \texttt{poziv} \mid \texttt{min} \mid \texttt{brojeća} \mid \texttt{grananje} \mid \texttt{poziv\_infix} \tag{2.23}$$

$$\texttt{definicija\_infix} \to \texttt{[lijeva OP lijeva] := izraz} \tag{2.24}$$

$$\texttt{poziv\_infix} \to \texttt{[izraz OP izraz]} \tag{2.25}$$

Jedina prava razlika između funkcija i infiksnih operatora je u njihovoj mjesnosti — funkcije koje definiramo mogu biti proizvoljne mjesnosti, dok su infiksni operatori uvijek „mjesnosti" 2. To znači da će parsiranje tih dvaju pojmova biti gotovo isto, samo što tijekom parsiranja infiksnih operatora trebamo parsirati točno dva izraza (a ne proizvoljno mnogo izraza odvojenih zarezom), po jedan sa svake strane tokena tipa OP.

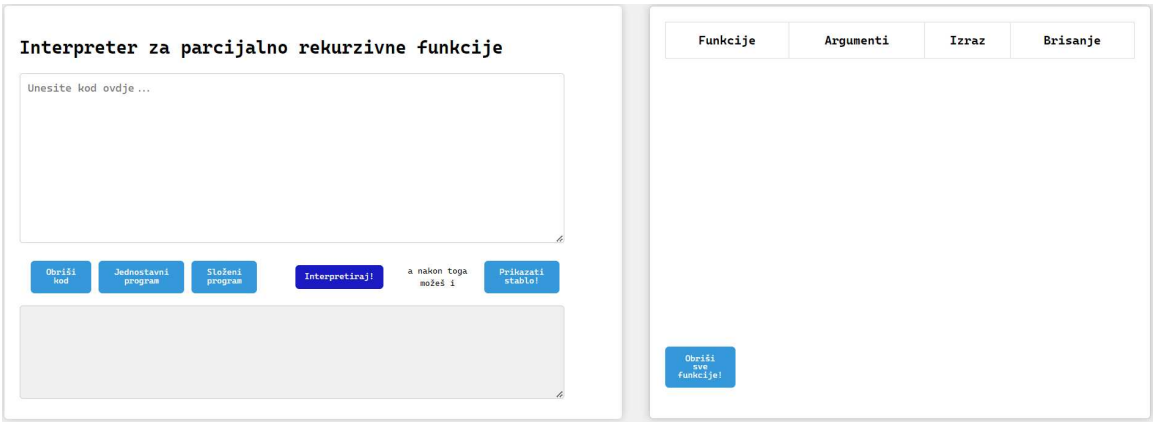
```
class P(Parser):
    # ... prethodni kod
    def definicija_infix(self):
        self >> T.UGOTV
        prvi = self.lijeva_varijabla()
        operator = Token(T.IME, (self >> T.OP).sadržaj)
        self.trenutna = operator.sadržaj
        drugi = self.lijeva_varijabla()
        self >> T.UGZATV
        self >> T.DEF_FUN
        if operator.sadržaj + baseString in funkcije:
            self.zamjena = (operator, [prvi, drugi].copy())
        izraz = self.izraz()
        self.zamjena = None
        return self.definiraj_i_vrati_funkciju(operator, [prvi, drugi], izraz)
    def poziv_infix(self):
        self >> T.UGOTV
        lijevi = self.izraz()
        operator = Token(T.IME, (self >> T.OP).sadržaj)
        desni = self.izraz()
        self >> T.UGZATV
        if self.zamjena is not None:
            ime_zamjena, lijeve_zamjena = self.zamjena
            if isinstance(lijeve_zamjena[-1], Poziv):
                lijeve_zamjena[-1] = lijeve_zamjena[-1].parametri[0]
            if ime_zamjena == operator and lijeve_zamjena == [lijevi, desni]:
                return Token(T.IME, prevString)
        return Poziv(operator, [lijevi, desni])
```

Program 2.9.4: Poziv funkcije `P.defincija_infix`

2.10 Korištenje interpretera

U ovom poglavlju su prikazani isječci kōda interpretera te su mnogobrojne manje bitne stvari izostavljene. Primjerice, izostavljene su razne naredbe `assert` koje sprječavaju korištenje 0-mjesnih funkcija te one koje provjeravaju je li funkcija pozvana s ispravnim brojem argumenata. Čitav kōd je dostupan u repozitoriju [1].

Interpreter ima i *online* verziju! Uz pomoć JavaScriptovog *framework-a* **Py-Script**, koji omogućava izvršavanje Python-programa u internetskom pregledniku, interpreter je postavljen na server Matematičkog odsjeka te je dostupan na adresi [2]. Njegovo sučelje izgleda ovako:

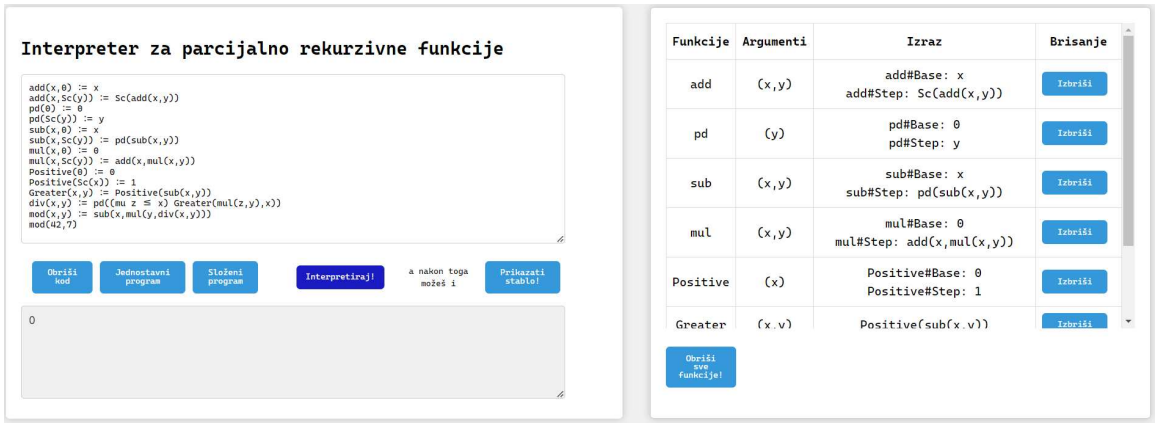


Slika 2.1: Sučelje interpretera

U programu 2.8.5 definirali smo praznu funkciju na više načina, ali to očito nije jedina parcijalna funkcija koju možemo definirati unutar interpretera. Iskoristit ćemo *online* verziju interpretera kako bismo definirali parcijalnu funkciju $even$[1], koja se na parnim brojevima ponaša kao identiteta, a na neparnima nije definirana:

$$even(x) := \begin{cases} x, & \mathrm{mod}(x,2) = 0 \\ \otimes(x), & \text{inače.} \end{cases} \tag{2.26}$$

Jasno je da će nam trebati funkcija mod[2] iz primjera 1.7.2, a samim time i sve primitivno rekurzivne funkcije i relacije od kojih se ona sastoji. Redom definiramo zbrajanje, prethodnik, oduzimanje, množenje, relacije $\mathbb{N}_+$ i $>$, dijeljenje te naposljetku ostatak pri dijeljenju:

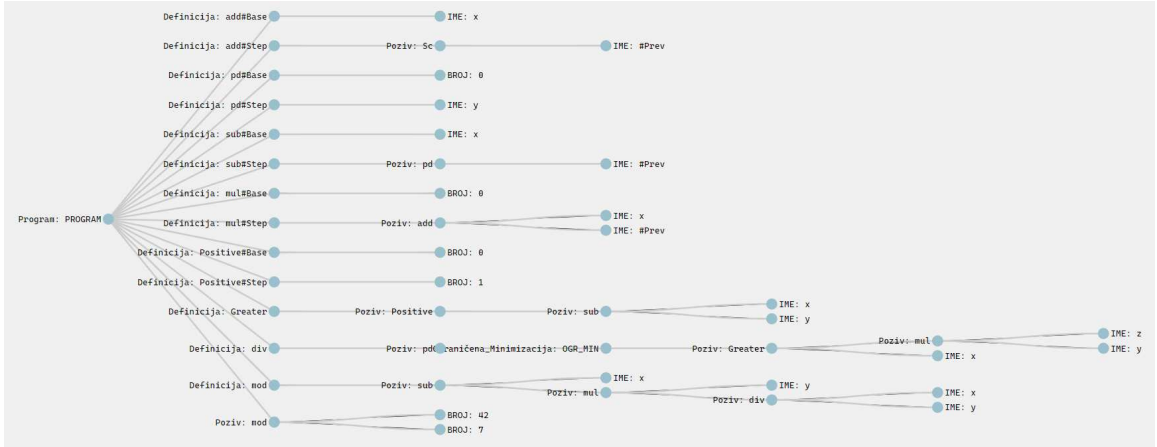


Slika 2.2: Definicija funkcije $\mathtt{mod}$[2]

Izvorni kōd funkcijskog jezika unosimo u glavni tekstualni okvir. Klikom na gumb *Interpretiraj!*, kōd se interpretira i rezultat se ispisuje u donji tekstualni okvir (u ovom slučaju $\mathtt{mod}(42, 7) = 0$).

Online verzija interpretera ima dodatnu funkcionalnost — sve ispravno definirane funkcije se *pamte* u popisu na desnoj strani sučelja, te ih možemo koristiti u idućim pokretanjima interpretera bez ponovne definicije. Preciznije, pamti se lista instanci klase `Definicija` svake ispravno interpretirane funkcije.

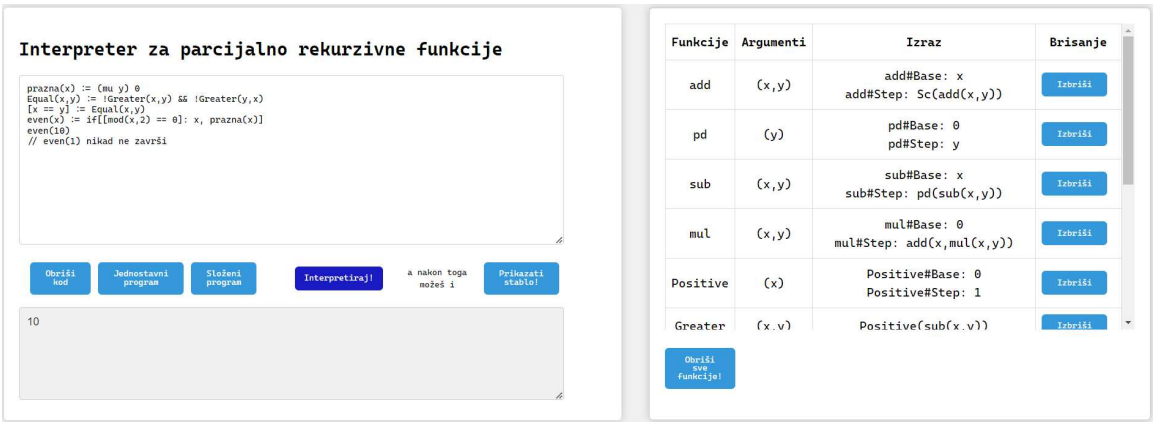
Nakon što smo interpretirali neki kōd, možemo kliknuti na gumb *Prikazati stablo!*, koji uz pomoć JavaScriptove biblioteke **d3.js** prikazuje stablo parsiranja tog kōda.



Slika 2.3: Stablo parsiranja

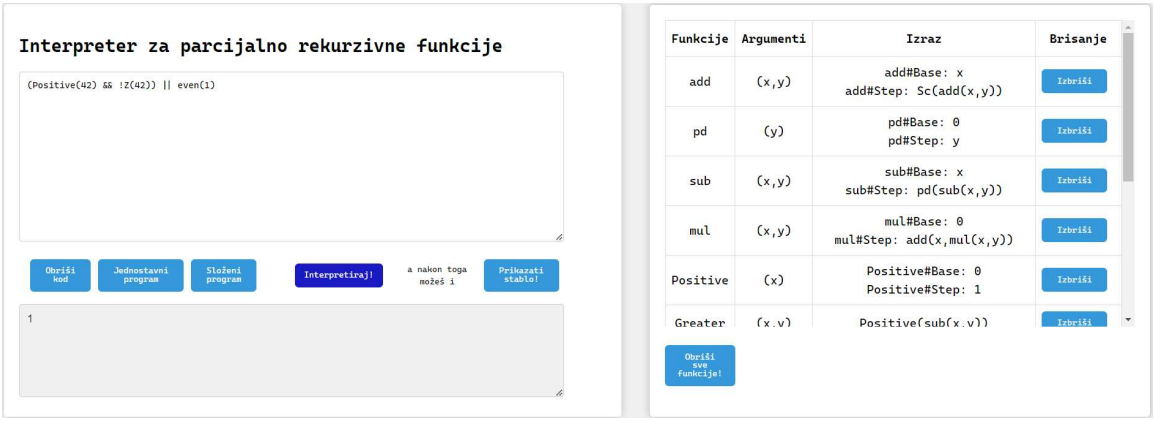
Kako bismo došli do funkcije $even$[1], trebamo iskoristiti operaciju grananja i praznu funkciju. Budući da moramo usporediti vrijednosti $\mathtt{mod}(x, 2)$ i 0, definirat ćemo i relaciju

`Equal`. Zbog jednostavnije sintakse, umjesto `Equal` koristit ćemo infiksni operator `[==]` (*online* verzija podržava infiksne operatore s više znakova).



Slika 2.4: Definicija funkcije *even*[1]

Za kraj dajemo još jedan zanimljiv primjer. Ako pogledamo program 2.7.6, možemo vidjeti da su logički operatori `||` i `&&` implementirani lijenom evaluacijom (tzv. *short-circuiting*). To znači da unutar nekih logičkih izraza možemo pozivati parcijalne funkcije u točkama gdje nisu definirane, a da svejedno ne zapnemo u beskonačnoj petlji — samo trebamo paziti da se u takvim izrazima parcijalne funkcije nikad ne počnu izračunavati.



Slika 2.5: Lijena evaluacija logičkih operatora

Bibliografija

[1] Eterović, M.: *GitHub repozitorij*, 2023. https://github.com/EteraGit/InterpreterFunkcijskogJezika.

[2] Eterović, M.: *Online interpreter*, 2023. http://web.studenti.math.pmf.unizg.hr/~eteromar/Eter.

[3] Čačić, V.: *Komputonomikon*. 2022. https://web.math.pmf.unizg.hr/~veky/izr/Komputonomikon.pdf.

Sažetak

U ovom radu proučavamo parcijalno rekurzivne funkcije te izrađujemo interpreter za njih.

U prvom poglavlju definiramo inicijalne funkcije i tri osnovne operacije (kompoziciju, primitivnu rekurziju i minimizaciju) na funkcijama i relacijama. Dokazujemo da je skup Python-izračunljivih funkcija zatvoren na te tri operacije. Definiramo neke dodatne funkcije, operacije i logičke operatore te dokazujemo njihovu primitivnu rekurzivnost.

U drugom poglavlju koristimo framework vepar kako bismo u Pythonu izradili interpreter koji podržava sve funkcije i operacije na funkcijama definirane u prvom poglavlju. Kroz poglavlje postupno dodajemo nove funkcionalnosti u interpreter i opisujemo razne mogućnosti vepra.

Summary

In this paper we study partial recursive functions and develop an intepreter for them.

In the first chapter we define initial functions and three primary operations (composition, primitive recursion and minimisation) on functions and relations. We prove that the set of all Python-computable functions is closed under those three operations. We define some other functions, operations and logical operators and prove they are primitive recursive (or that they preserve primitive recursivity).

In the second chapter we use a framework called vepar to develop an interpreter in Python which supports all functions and operations on functions defined in the first chapter. Throughout the chapter we incrementally add new functionalities to the interpreter and describe various capabilities of vepar.

Životopis

Rođen sam 11.1.1999. u Splitu, gdje sam pohađao osnovnu školu Gripe i V. gimnaziju Vladimir Nazor. Godine 2017. upisao sam Preddiplomski sveučilišni studij matematike na Matematičkom odsjeku PMF-a u Zagrebu, kojeg sam završio 2021. godine. Nakon toga upisujem Diplomski studij računarstva i matematike na istom fakultetu.

Tijekom diplomskog studija radio sam kao student u tvrtci Visage Technologies, a trenutno sam zaposlen u tvrtci AVL-AST gdje radim na razvoju softvera za razne simulacije u automobilskoj industriji.