# Interpreter za lambda-račun

**Lovnički, Sandro**

**SVEUČILIŠTE U ZAGREBU**

**PRIRODOSLOVNO–MATEMATIČKI FAKULTET**

**MATEMATIČKI ODSJEK**

Sandro Lovnički

# INTERPRETER ZA $\lambda$-RAČUN

Diplomski rad

Voditelj rada:
doc.dr.sc. Vedran Čačić

Zagreb, rujan 2018.

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. _____

*Za Petru, koja me spasila i rasplamsala vatru znanosti.*

# Sadržaj

# Uvod

U prvoj polovici 20. stoljeća jačala je želja za aksiomatizacijom temelja matematike te je, kao pokušaj rješenja tog problema, 1930-ih Alonzo Church stvorio $\lambda$-račun koji se temelji na definicijama i aplikacijama bezimenih funkcija koristeći supstitucije. Churchov učenik Alan Turing, ne znajući još za Churchov model, stvorio je apstraktne strojeve koji modeliraju izračunavanja manipulacijom simbolima na beskonačnoj traci, te na prvi pogled nemaju puno sličnosti s $\lambda$-računom. Ipak, Church i Turing zajedno su dokazali da oba modela genereiraju istu klasu funkcija — parcijalno rekurzivne funkcije.

Iako ovi modeli na kraju nisu uspjeli aksiomatizirati matematiku — što se pokazalo nemogućim, poslužili su za razvoj dvije velike grane u računarskoj znanosti: izračunljivosti i složenosti. Također, $\lambda$-račun je imao ključnu ulogu u razvoju funkcijskih programskih jezika poput Haskella.

Ovaj rad dvojake je prirode: s jedne strane ima za cilj upoznati čitatelja s osnovnim idejama $\lambda$-računa, dok s druge daje dokumentaciju konkretne implementacije jednostavnog programskog jezika *pLam* temeljenog na $\lambda$-izrazima.

U prvom dijelu upoznat ćemo sintaksu i semantiku $\lambda$-računa, te odmah uvidjeti njegov značaj u proučavanju teorije programskih jezika. Nadalje, gotovo neizbježno i sasvim prirodno, istražit ćemo ulogu $\lambda$-računa kao modela za opisivanje izračunljivosti, te vidjeti kako određeni $\lambda$-izrazi generiraju upravo klasu parcijalno rekurzivnih funkcija. Na to se odmah nadovezujemo željom za efikasnošću, te krećemo stvarati brojevne sustave koji će nam omogućiti donekle efikasan rad s interpreterom jezika koji je teorijski toliko moćan. Spomenut ćemo i klasične brojevne sustave koji su od povijesnog i/ili teorijskog značaja za ovo polje istraživanja. Također ćemo, po potrebi, uz svako poglavlje, vidjeti kako sve što smo teorijski razradili zaista funkcionira na konkretnim primjerima programa pisanih u jeziku koji smo razvili da simuliramo izvrednjavanje $\lambda$-izraza.

Nadamo se da će na kraju svaki čitatelj produbiti svoju želju za proučavanjem teorijskog računarstva, a možda i pridonijeti razvoju interpretera *pLam* pišući biblioteke $\lambda$-izraza koje još nismo implementirali, ili izgraditi još efikasniju reprezentaciju brojeva.

# Poglavlje 1

# $\lambda$-račun

$\lambda$-račun je formalni sustav za definiranje i pozivanje (apliciranje) funkcija, uključujući i rekurzivne pozive. Razvio ga je 1930-ih godina Alonzo Church kao logičku osnovu matematike, a nakon Gödelovih rezultata o nepotpunosti svakog takvog formalnog sustava, koristio ga je za proučavanje izračunljivosti. Kao posljedica toga, on i njegov učenik Alan Turing dali su 1936. godine negativan odgovor na Hilbertov *Entscheidungsproblem*[1] iz 1928. godine, tijekom čega je nastalo i ono što danas smatramo jednom od mnogih varijacija (npr. [3, 5]) Church-Turingove teze[2].

## 1.1 Osnovni pojmovi

Osnovni pojmovi koje trebamo su *$\lambda$-izrazi* te relacija ekvivalencije na njima koju zovemo *konvertibilnost*. U ovom poglavlju detaljno ćemo opisati komponente nužne za shvaćanje daljnjeg sadržaja ovog rada. Treba imati na umu da svrha ovog rada nije razrada teorijskih rezultata $\lambda$-računa, nego primjena — točnije, implementacija raznih programa pisanih isključivo koristeći $\lambda$-izraze.

**Definicija 1.1.1** ($\lambda$-izraz)**.** *$\lambda$-izrazi* su riječi nad abecedom koju čine:

- skup *varijabli* $Var$ (čije elemente pišemo: $x, y, u, v, w, \ldots$)
- apstraktor: $\lambda$
- zagrade i točka: $(, ), .$ $\triangleleft$

[1] *problem odluke*; Postoji li algoritam koji će kao ulaz primiti neku formulu pisanu jezikom logike prvog reda, te kao izlaz vratiti da li je ta formula valjana?

[2] Postoji algoritam (s neograničenim prostorno-vremenskim resursima) za izračunavanje funkcije ako i samo ako je ona $\lambda$-definabilna, tj. ako i samo ako je Turing-izračunljiva.

**Definicija 1.1.2** (Skup Λ)**.** Skup $\lambda$-izraza $\Lambda$ definiramo induktivno:

- $x \in Var \Rightarrow x \in \Lambda$
- $x \in Var \wedge M \in \Lambda \Rightarrow (\lambda x.\, M) \in \Lambda$ (**Apstrakcija**)
- $M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$ (**Aplikacija**) ◁

Navedimo sada neke notacijske napomene (Barendregt, [2]) koje će nam olakšati pisanje i razumijevanje kompliciranih $\lambda$-izraza.

- Općenito, varijable ćemo označavati malim slovima ($x, y, z, \ldots$), a $\lambda$-izraze koji nisu nužno varijable velikima ($M, N, \ldots$).
- Specijalne $\lambda$-izraze — one koje smo implementirali — pisat ćemo slovima fiksne širine, npr. `T`, `if`, `mul`. Shvaćamo ih kao globalne (okolinske) varijable i samo su pokrata za izraz koji predstavljaju. Za razliku od „čistih" varijabli iz gornje točke, ova imena mogu se sastojati od više znakova, tj. `if` $\neq$ `i f`.
- Simbol „≡" označava sintaksnu ekvivalenciju.
- Vanjske zagrade nećemo uvijek pisati, tj. $(M\ N) \equiv M\ N$.
- $\lambda\vec{x}.\, M \equiv \lambda x_1 x_2 \ldots x_n.\, M \equiv \lambda x_1.\, \lambda x_2.\, \ldots\, \lambda x_n.\, M$ ($currying$[3]).
- $M\ \vec{N} \equiv M\ N_1\ N_2\ \cdots\ N_n \equiv (\cdots\ ((M\ N_1)\ N_2)\ \cdots\ N_n)$ (grupiranje ulijevo).

**Primjer 1.1.3.** *Primjeri λ-izraza:*

$$\lambda x.\, x, \quad \lambda xy.\, x\ (\equiv \lambda x.\, \lambda y.\, x), \quad (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x),$$

$$\lambda xy.\, x\ \Big(\big(\lambda x.\, x\ (\lambda xy.\, y)\ (\lambda xy.\, x)\big)\ y\Big)\ y.$$

Kako bismo smisleno i razumno baratali $\lambda$-izrazima, potrebne su nam relacije konverzije na $\Lambda$, koje će generirati aksiomi teorije koju upoznajemo. Jedna od najvažnijih operaciju nad $\lambda$-izrazima je supstitucija uz koju se usko veže pojam slobodnih varijabli.

Za varijablu $x$ kažemo da je **slobodna** u $\lambda$-izrazu $M$ ako se ne nalazi u dosegu apstrakcije $\lambda x.$. U suprotnom kažemo da je $x$ **vezana** varijabla. Na primjer, u izrazu $M \equiv x\ (\lambda y.\, x\ y)$ varijabla $x$ pojavljuje se dva puta slobodno, a varijabla $y$ je vezana. Definirajmo to i formalno.

[3] *currying*: evaluacija djelovanja funkcije više argumenata kao niz evaluacija djelovanja funkcija jednog argumenta. (Gottlob Frege, Moses Schönfinkel, Haskell Curry)

**Definicija 1.1.4** (Slobodne varijable)**.** Skup svih *slobodnih varijabli* u $M$ označavamo s $FV(M)$ i definiramo induktivno:

$$\begin{aligned} FV(x) &= \{x\}, \\ FV(\lambda x.\, M) &= FV(M) \setminus \{x\}, \\ FV(M\, N) &= FV(M) \cup FV(N). \end{aligned}$$ ◁

**Definicija 1.1.5.** Svaki $\lambda$-izraz $M$ za koji vrijedi $FV(M) = \emptyset$ zovemo *kombinator*. ◁

Kažemo da je $N$ **supstituiran za** $x$ **u** $M$ i pišemo $M[x := N]$ ako smo sve slobodne pojave varijable $x$ u izrazu $M$ na odgovarajući način, koji ćemo objasniti u točki 1.3, zamijenili izrazom $N$.

**Definicija 1.1.6** (Teorija $\lambda$)**.** *Teorija* $\lambda$ sastoji se od formula oblika $M = N$ $(M, N \in \Lambda)$ generiranih sljedećim aksiomima i pravilima:

1. $M = M$
2. $M = N \Rightarrow N = M$
3. $M = N, N = L \Rightarrow M = L$
4. $M = N \Rightarrow M\, Z = N\, Z$
5. $M = N \Rightarrow Z\, M = Z\, N$
6. $M = N \Rightarrow \lambda x.\, M = \lambda x.\, N$
7. $(\lambda x.\, M)\, N = M[x := N]$ ($\beta$-konverzija)
8. $\lambda x.\, M\, x = M$ ($\eta$-konverzija) ◁

Dokazivost neke jednadžbe u $\lambda$ zapisujemo $\lambda \vdash M = N$ ili kraće, samo $M = N$. Ako $\lambda \vdash M = N$, kažemo da su $M$ i $N$ *konvertibilni*.

Prije nego počnemo detaljno razmatrati najvažniju binarnu relaciju u $\Lambda$, $\beta$-konverziju, upoznat ćemo specijalne binarne relacije — redukcije — nad skupom $\Lambda$, i njihova svojstva. One su alat koji jeziku $\lambda$-računa „daje život" pretvarajući (*reducirajući*) izraze jedan u drugi, poželjno kompliciranije u jednostavnije, duže u kraće. Vidjet ćemo da ono što smo dosad zvali $\beta$-konverzijom može dobiti i specijaliziraniji naziv — $\beta$-redukcija, te će upravo ona predstavljati okosnicu svih izračunavanja u $\lambda$-računu.

## 1.2 Redukcije

**Definicija 1.2.1.** Za binarnu relaciju $R$ na $\Lambda$ kažemo da je *kompatibilna* ako $(M, M') \in R$ i $Z \in \Lambda$ povlači

$$(Z\ M, Z\ M') \in R, \quad (M\ Z, M'\ Z) \in R \quad \text{i} \quad (\lambda x.\ M, \lambda x.\ M') \in R.$$

Relacija *jednakosti* na $\Lambda$ je kompatibilna relacija ekvivalencije. Relacija *redukcije* na $\Lambda$ je kompatibilna, refleksivna i tranzitivna. ◁

**Lema 1.2.2.** *$\beta$-konverzija je relacija redukcije.*

*Dokaz.* Dokaz ove tvrdnje slijedi izravno iz definicije 1.1.6. Precizno, kompatibilnost slijedi iz točaka 4, 5 i 6, refleksivnost iz točke 1, a tranzitivnost iz točke 3. □

**Definicija 1.2.3.** Neka je $R$ redukcija na $\Lambda$. Tada $R$ inducira binarne relacije $\to_R$ ((jedan) korak $R$-redukcije) , $\twoheadrightarrow_R$ ($R$-redukcija) i $=_R$ ($R$-jednakost) definirane induktivno na sljedeći način:
$\to_R$ je kompatibilno zatvorenje od R, dakle:

$$\begin{aligned}(M, N) \in R &\Rightarrow M \to_R N,\\ M \to_R N &\Rightarrow Z\ M \to_R Z\ N,\\ M \to_R N &\Rightarrow M\ Z \to_R N\ Z,\\ M \to_R N &\Rightarrow \lambda x.\ M \to_R \lambda x.\ N.\end{aligned}$$

$\twoheadrightarrow_R$ je refleksivno i tranzitivno zatvorenje od $\to_R$, dakle;

$$\begin{aligned}M \to_R N &\Rightarrow M \twoheadrightarrow_R N,\\ M &\twoheadrightarrow_R M,\\ M \twoheadrightarrow_R N, N \twoheadrightarrow_R L &\Rightarrow M \twoheadrightarrow_R L.\end{aligned}$$

$=_R$ je relacija ekvivalencije generirana s $\twoheadrightarrow_R$, dakle;

$$\begin{aligned}M \twoheadrightarrow_R N &\Rightarrow M =_R N,\\ M =_R N &\Rightarrow N =_R M,\\ M =_R N, N =_R L &\Rightarrow M =_R L.\end{aligned}$$

◁

**Napomena 1.2.4.** *Gornje relacije čitamo na sljedeći način:*
*$M \to_R N$: „M se R-reducira u N u jednom koraku"*
*$M \twoheadrightarrow_R N$: „M se R-reducira u N" ili „N je R-redukt od M"*
*$M =_R N$: „M je R-konvertibilan u N"*

S obzirom da $\beta$-redukciju koristimo kao glavni alat za evaluaciju $\lambda$-izraza, postavlja se pitanje „Kakve izraze možemo $\beta$-reducirati?" Postoje razne vrste redukcija ([1, 12]), ovisno o tome kada i kako dopuštamo izvršavanje $\beta$-redukcije, a mi ćemo reducirati izraze takozvanim *normalnim redoslijedom* (eng. *normal order*, [12]). Primijetimo da na lijevoj strani definicije $\beta$-redukcije imamo aplikaciju apstrakcije. Takav $\lambda$-izraz zovemo **redeks** i on se može $\beta$-reducirati. Također, da bi se neki $\lambda$-izraz mogao $\beta$-reducirati, ne mora nužno on sam biti redeks, već je dovoljno da sadrži redeks unutar sebe (npr. $\lambda x.\ x\ ((\lambda y.\ y)\ x) =_\beta \lambda x.\ x\ x$), dakle dopuštamo redukciju unutar apstrakcija i aplikacija.

Pojam koji se prirodno veže uz redukcije je *normalna forma*.

**Definicija 1.2.5** ($\beta$-nf)**.** Kažemo da je $\lambda$-izraz u $\beta$-normalnoj formi ako nema $\beta$-redeksa. ◁

Strategija kojom biramo $\beta$-redeks je takva da prvo reduciramo **vanjske** redekse u $\lambda$-izrazu, te od njih, ako ih je više, biramo **prvi s lijeva** (eng. *leftmost, outermost*). Zanima nas da li takva strategija biranja redeksa uvijek vodi do normalne forme ako ona postoji. Da bismo odgovorili na to pitanje, pogledajmo prvo kakvi izrazi nemaju $\beta$-normalnu formu i zašto.

**Korolar 1.2.6.** *Postoje $\lambda$-izrazi koji nemaju $\beta$-normalnu formu.*

*Dokaz.* Neka je $\texttt{omega} := (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. $\beta$-redukcijama dobivamo beskonačan niz

$$\texttt{omega} \to_\beta \texttt{omega} \to_\beta \texttt{omega} \to_\beta \cdots$$ □

Promotrimo sada $\lambda$-izraz $M := \texttt{T}\ (\lambda x.\ x)\ \texttt{omega} \equiv (\lambda xy.\ x)\ (\lambda x.\ x)\ \Big((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\Big)$. On ima $\beta$-normalnu formu $(\lambda x.\ x)$, te do nje dolazimo gore opisanom strategijom na sljedeći način (podcrtavamo redeks koji biramo):

$$\begin{aligned} M &= \underline{(\lambda xy.\ x)\ (\lambda x.\ x)}\ \Big((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\Big) = \\ &= \underline{\Big(\lambda y.\ (\lambda x.\ x)\Big)\ \Big((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\Big)} = \\ &= (\lambda x.\ x). \end{aligned}$$

Primjenom neke druge strategije, npr. biranjem desnog unutarnjeg redeksa, imali bismo

$$\begin{aligned} M &= (\lambda xy.\ x)\ (\lambda x.\ x)\ \Big(\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)}\Big) = \\ &= (\lambda xy.\ x)\ (\lambda x.\ x)\ \Big(\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)}\Big) = \\ &= \dots \end{aligned}$$

i time ne bismo došli do $\beta$-normalne forme izraza $M$, nego beskonačno dugo reducirali izraz `omega` kao u dokazu prethodnog korolara.

Možemo zaključiti da će strategija reduciranja najlijevijeg vanjskog redeksa uvijek doći do normalne forme ako ona postoji. Naime, izabrani redeks nikad ne može biti argument neke druge funkcije jer je najlijeviji u izrazu, stoga je njegovo reduciranje evaluacija funkcije. Ako ta funkcija „ignorira" neki argument (kao u gornjem primjeru), to pogotovo ne može pridonijeti pojavi beskonačnog niza redukcija. S druge strane, ako koristimo strategiju koja u nekom trenutku evaluira argument prije nego funkciju koja ga prima (kao u gornjem primjeru), možemo se naći u situaciji s (možda nepotrebnim) beskonačnim nizom redukcija.

## 1.3 Supstitucije i imena

Svaku apstrakciju zamišljamo kao funkciju koja prima neki izraz i primjenom $\beta$-redukcije vraća rezultat supstitucije tog izraza za odgovarajuću varijablu u svome tijelu. Možemo se složiti da izrazi $\lambda x.\ x$ i $\lambda y.\ y$ predstavljaju istu funkciju, tj. jednako će djelovati za svaki ulaz — $\beta$-reducirati se točno u njega; $(\lambda x.\ x)\ M =_\beta M$ i $(\lambda y.\ y)\ M =_\beta M$, za svaki $M \in \Lambda$.

**Definicija 1.3.1** ($\alpha$-ekvivalencija)**.** Za $\lambda$-izraze koji su jednaki do na preimenovanje varijabli kažemo da su *$\alpha$-ekvivalentni*, a sam postupak preimenovanja zovemo *$\alpha$-preimenovanje*. ◁

Fokusirajmo se sada na $\beta$-redukciju i njeno djelovanje. Kako bi izrazi $(\lambda x.\ M)\ N$ i $M[x := N]$ zaista bili semantički jednaki, obrađujemo detalje procesa supstitucije izraza $N$ za $x$ u $M$. Na sljedećem primjeru vidjet ćemo kako ne možemo uvijek (bez dodatnih pretpostavki) izvršiti supstituciju i zadržati semantičku jednakost.

Promotrimo apstrakciju $\lambda xy.\ x\ y$. Ona prima dva argumenta i vraća aplikaciju prvog na drugi. Primjena $\beta$-redukcije na aplikaciju $(\lambda xy.\ x\ y)\ z$, tj. zamjena varijable $x$ varijablom $z$ u tijelu apstrakcije, daje nam apstrakciju $\lambda y.\ z\ y$ koja prima jednu varijablu i primjenjuje $z$ na nju. S druge strane, ako bismo „slijepo" napravili $\beta$-redukciju na aplikaciji $(\lambda xy.\ x\ y)\ y$, dobili bismo $\lambda y.\ y\ y$, što je apstrakcija koja prima jednu varijablu i primjenjuje ju na samu sebe. Jasno, to je nepoželjno ponašanje i u tom slučaju trebamo primijeniti $\alpha$-preimenovanje prije $\beta$-redukcije. Precizno, imamo

$$(\lambda xy.\ x\ y)\ y \equiv (\lambda x.\ \lambda y.\ x\ y)\ y = (\lambda y.\ x\ y)[x := y] =_\alpha (\lambda y'.\ x\ y')[x := y] = \lambda y'.\ y\ y',$$

čime smo dobili točan rezultat, apstrakciju koja prima varijablu i primjenjuje $y$ na nju.

Valja napomenuti da ćemo kroz ostatak rada podrazumijevati da se u pozadini svih $\beta$-redukcija po potrebi događa i proces $\alpha$-preimenovanja te ga nećemo eksplicitno pisati.

Inače, postoji notacija u kojoj nema potrebe za $\alpha$-preimenovanjem koju je smislio nizozemski matematičar Nicolaas Govert de Bruijn, po kome je dobila ime. Više o vrstama notacije može se vidjeti u [7], a valja spomenuti i Barendregtovu konvenciju koja glasi slično kao što i mi zahtijevamo; da nema potrebe za eksplicitnim $\alpha$-preimenovanjem, tj. pretpostavlja se da su varijable izraza koji ulaze u redukciju imenovane upravo tako da se redukcija može izvršiti bez preimenovanja. Dakako, takva pretpostavka je samo prebacivanje problema te smo tijekom implementacije jezika *pLam* (kojim ćemo se koristiti u primjerima, a o kojem ćemo više reći u poglavlju 4) morali implementirati $\alpha$-preimenovanje.

## 1.4 Primjeri

U ovom poglavlju navest ćemo nekoliko primjera $\lambda$-izraza koji će biti od velikog značaja u daljnjim poglavljima te će biti korišteni u većini dokaza. Koristeći svojstva $\lambda$-izraza, kroz kratku raspravu definiramo logičke konstante i osnovne logičke operatore. Također ćemo i na primjeru *pLam* kodova vidjeti da se ti izrazi, primjenom $\beta$-redukcija, evaluiraju upravo kako i očekujemo.

### 1.4.1 Logički operatori

Ideja iza implementacije logičkih konstanti je sljedeća: Htjeli bismo da istina (**T**rue) od dva ponuđena izbora bira prvi, a laž (**F**alse) drugi. Vođeni time, definicije ostalih operatora slijede.

**Primjer 1.4.1** (`T`, `F`, `if`)**.**
*Istinu definiramo kao* $\mathbf{T} := \lambda xy.\ x$*, dok laž definiramo kao* $\mathbf{F} := \lambda xy.\ y$*.*
*Primijetimo da* `T` *vraća prvi argument, a* `F` *drugi pa lako možemo definirati (a i ne moramo)*[4] *operator grananja if-then-else kao* $\mathbf{if} := \lambda xyz.\ x\ y\ z$*. On prima tri argumenta i ovisno o istinitosti argumenta x vraća y, odnosno z. Detaljnije, ako je* $x \equiv$ `T`*, tada imamo* `T` $y\ z = y$*, a ako je* $x \equiv$ `F` *imamo* `F` $y\ z = z$*.*

**Primjer 1.4.2** (`not`, `and`, `or`, `xor`)**.**

$$\begin{aligned}
\mathbf{not} &:= \lambda x.\ x\ \texttt{F}\ \texttt{T},\\
\mathbf{and} &:= \lambda xy.\ x\ y\ \texttt{F},\\
\mathbf{or} &:= \lambda xy.\ x\ \texttt{T}\ y,\\
\mathbf{xor} &:= \lambda xy.\ x\ (\texttt{not}\ y)\ y.
\end{aligned}$$

[4] Izraz čiju istinitost koristimo za grananje, možemo samo aplicirati na grane $if_{\mathrm{T}}$ i $if_{\mathrm{F}}$ bez da `if` apliciramo na sva tri izraza. Ubuduće ćemo često pisati (*uvjet*) $if_{\mathrm{T}}\ if_{\mathrm{F}}$ umjesto `if` *uvjet* $if_{\mathrm{T}}\ if_{\mathrm{F}}$.

### 1.4.2 Istinosne vrijednosti u jeziku *pLam*

Sad pokazujemo gore definirane logičke izraze unutar interpretera *pLam*. Možemo primijetiti da je grčko slovo $\lambda$ kao oznaka apstrakcije zamijenjeno znakom \.

Dodatno, vidimo da unutar ljuske interpretera možemo dodjeljivati imena izrazima koristeći znak „=", koji tada postaju globalne varijable dostupne u daljnjem radu preko svog imena. Detaljnije o svim mogućnostima *pLam*-a može se pogledati u poglavlju 4, ali zasad smatramo da je korištenje jasno iz primjera.

**Primjer 1.4.3** (rad s logičkim izrazima unutar ljuske *pLam* interpretera)**.**

```
          _
         | |
    _____| |    ___   __   __
    | _ \ |__| _ \|   \/   |
    |  _/____|____\_\__/_| v1.0.0
    |_| pure λ-calculus interpreter
   ================================

pLam> T = \x y. x
pLam> F = \x y. y
pLam>
pLam> not = \x. x F T
pLam> and = \x y. x y F
pLam> or = \x y. x T y
pLam> xor = \x y. x (not y) y
pLam>
pLam> and (or F (not F)) (xor T F)
> reductions count                : 18
> uncurried β-normal form         : (λxy. x)
> curried (partial) α-equivalent: T
```

# Poglavlje 2

# Izračunljivost

Bez obzira koji model izračunavanja koristimo (RAM-programe, Turingove strojeve, $\lambda$-račun, Postove strojeve, ...), izračunljivost se bavi definiranjem određene klase funkcija unutar danog modela, što za cilj ima pokazati koliko je taj model izražajan. Ovdje se bavimo samo $\lambda$-računom, a o drugim modelima može se čitati u [4, 13, 14].

Glavni cilj ovog poglavlja je pokazati izomorfnost određenog skupa funkcija koje je moguće definirati unutar $\lambda$-računa i skupa parcijalno rekurzivnih funkcija. Skup parcijalno rekurzivnih funkcija sastoji se od svih inicijalnih funkcija te je zatvoren na kompoziciju, primitivnu rekurziju i minimizaciju. Iskažimo to i formalno u sljedećoj točki.

## 2.1 Parcijalno rekurzivne funkcije

Klasa parcijalno rekurzivnih funkcija sadrži samo *numeričke* funkcije, pa definirajmo prvo taj pojam onako kako ga shvaćamo u ovom radu. Također, neke od tih funkcija mogu biti i *parcijalne* — pojam koji ćemo također sada definirati.

**Definicija 2.1.1.** Za funkciju kažemo da je *numerička* ako joj je domena podskup od $\mathbb{N}_0^k$ za neki $k \in \mathbb{N}$, a kodomena joj je $\mathbb{N}_0$. Za numeričku funkciju $f$ kažemo da je *totalna* ako joj je domena jednaka $\mathbb{N}_0^k$. Ako numerička funkcija nije totalna, kažemo da je *parcijalna*. $\triangleleft$

Kod definicija ne nužno totalnih funkcija, umjesto simbola „=" koristimo „$\simeq$" kako bismo istaknuli moguću nedefiniranost u nekim točkama.

**Definicija 2.1.2** (Incijalne funkcije)**.** *Inicijalne funkcije* su numeričke funkcije $I_i^p$, $S$ i $Z$ definirane s:

$$\begin{aligned} I_i^p(n_1, \ldots, n_p) &= n_i, \quad 1 \leq i \leq p \quad (\textbf{Projekcija}), \\ S(n) &= n + 1 \quad (\textbf{Sljedbenik}), \\ Z(n) &= 0 \quad (\textbf{Nula}). \end{aligned}$$ ◁

Sada uvodimo pojmove *kompozicije*, *primitivne rekurzije* i *minimizacije* koji služe kao gradivne jedinice u kreiranju parcijalno rekurzivnih funkcija iz inicijalnih funkcija. Tada ćemo skup svih parcijalno rekurzivnih funkcija definirati upravo kao skup koji sadrži sve inicijalne funkcije i zatvoren je na navedene tri operacije.

**Definicija 2.1.3.** Neka je $\mathscr{P}$ skup numeričkih funkcija.

1. $\mathscr{P}$ je zatvoren na **kompoziciju** ako za svaku funkciju $\varphi$ definiranu kao

   $$\varphi(\vec{n}) \simeq \chi(\psi_1(\vec{n}), \ldots \psi_m(\vec{n})),$$

   gdje su $\chi, \psi_1, \ldots, \psi_m \in \mathscr{P}$, vrijedi i $\varphi \in \mathscr{P}$.
2. $\mathscr{P}$ je zatvoren na **primitivnu rekurziju** ako za svaku funkciju $\varphi$ definiranu kao

   $$\begin{aligned} \varphi(0, \vec{n}) &= \chi(\vec{n}), \\ \varphi(k+1, \vec{n}) &= \psi(\varphi(k, \vec{n}), k, \vec{n}), \end{aligned}$$

   gdje su $\chi, \psi \in \mathscr{P}$ totalne, vrijedi i $\varphi \in \mathscr{P}$.
3. $\mathscr{P}$ je zatvoren na **minimizaciju** ako za svaku funkciju $\varphi$ definiranu kao

   $$\varphi(\vec{n}) \simeq \mu m[\chi(\vec{n}, m) = 0],$$

   gdje je $\chi \in \mathscr{P}$ totalna, vrijedi i $\varphi \in \mathscr{P}$. ◁

Kao što smo i najavili, sad formalno definiramo skup svih parcijalno rekurzivnih funkcija.

**Definicija 2.1.4** (Parcijalno rekurzivne funkcije)**.**
Skup $\mathscr{P}$ svih *parcijalno rekurzivnih funkcija* najmanji je skup numeričkih funkcija koji sadrži sve inicijane funkcije i zatvoren je na kompoziciju, primitivnu rekurziju i minimizaciju. ◁

## 2.2 Churchovi numerali

Kako skup $\Lambda$ ne sadrži eksplicitno nikakve numerale[1], htjeli bismo odabrati $\lambda$-izraze koji će nam poslužiti za reprezentaciju prirodnih brojeva. Nadalje, želimo definirati i numeričke funkcije nad takvim numeralima. Krećemo s najpoznatijom verzijom numerala unutar $\lambda$-računa — Churchovim numeralima.

Detaljnije rezultate o Churchovim numeralima, ali i drugim brojevnim sustavima ([6, 11]), vidjet ćemo u poglavlju 3, dok ih ovdje samo definiramo u svrhu dokazivanja daljnjih rezultata vezanih uz izračunljivost.

**Definicija 2.2.1.** *n-ti Churchov numeral* $c_n$ definiramo kao

$$c_n := \lambda f x.\ f^n\ x \equiv \lambda f x.\ f\ (f\ \cdots\ (f\ x)\ \cdots).$$ ◁

Dakle, broj $n$ reprezentiramo numeralom $c_n$ koji prima $\lambda$-izraze $f$ i $x$ te vraća izraz nastao $n$-strukom primjenom $f$ na $x$.

Kao što smo spomenuli, nisu sve numeričke funkcije definirane na cijeloj svojoj domeni, te izračunavanje takvih u točkama izvan domene nikad ne stane. Kako bismo i tu opciju imali pokrivenu $\lambda$-izrazima, uvodimo sljedeću definiciju.

**Definicija 2.2.2.** Kažemo da je $M \in \Lambda$ *rješiv* ako postoji $\vec{N}$ takav da je $M\ \vec{N} = \mathtt{I}$[2]. Ako $M$ nije rješiv, kažemo da je *nerješiv*. ◁

**Lema 2.2.3.** *Svi Churchovi numerali su rješivi $\lambda$-izrazi i to za $\vec{N}$ (iz definicije)* $= \mathtt{I}\ \mathtt{I}$.

*Dokaz.* Indukcijom.
Za $c_0$ lako vidimo $c_0\ \mathtt{I}\ \mathtt{I} = (\lambda f x.\ x)\ \mathtt{I}\ \mathtt{I} = \mathtt{I}$
Za $c_{n+1}$, koristeći definiciju $c_{n+1} = \mathtt{S}_c{}^3\ c_n$ i pretpostavku indukcije imamo sljedeći niz jednakosti:

$$c_{n+1}\ \mathtt{I}\ \mathtt{I} = (\mathtt{S}_c\ c_n)\ \mathtt{I}\ \mathtt{I} = \Big(\big(\lambda n f x.\ f\ (n\ f\ x)\big)\ c_n\Big)\ \mathtt{I}\ \mathtt{I} = \big(\lambda f x.\ f\ (c_n\ f\ x)\big)\ \mathtt{I}\ \mathtt{I} =$$
$$= \big(\lambda x.\ \mathtt{I}\ (c_n\ \mathtt{I}\ x)\big)\ \mathtt{I} = \mathtt{I}\ (c_n\ \mathtt{I}\ \mathtt{I}) = (\text{pretp.}) = \mathtt{I}\ \mathtt{I} = \mathtt{I}.$$

□

[1] numeral: reprezentacija apstraknog pojma broja.
[2] Identiteta, $\mathtt{I} := \lambda x.\ x$.
[3] Sljedbenik. U točki 2.4 ćemo i dokazati rezultat $\mathtt{S}_c := \lambda n f x.\ f\ (n\ f\ x)$.

### 2.2.1 Primjeri

Interpreter *pLam* opremljen je alatom za parsiranje Churchovih numerala te ga koristi kao zadanu akciju za interpretaciju stringova koji predstavljaju prirodne brojeve zapisane arapskim brojkama.

**Primjer 2.2.4** (parsiranje prirodnih brojeva)**.**

```
pLam> 5
> reductions count                : 0
> uncurried β-normal form         : (λfx. f (f (f (f (f x)))))
> curried (partial) α-equivalent: 5
pLam>
```

Pogledajmo sad implementaciju i djelovanje osnovnih računskih operacija nad Churchovim numeralima.

**Primjer 2.2.5** (sljedbenik, prethodnik, zbrajanje, oduzimanje, množenje, potenciranje)**.**

```
pLam> Sc = \n f x. f (n f x)
pLam> Pc = \n f x. n (\g h. h (g f)) (\u. x) (\u. u)
pLam> add = \m n f x. (m f (n f x))
pLam> sub = \m n. (n Pc) m
pLam> mul = \m n f. m (n f)
pLam> exp = \m n. n m
pLam>
pLam> mul (add 2 (Sc 2)) (sub (exp 2 3) (Pc 8))
> reductions count                : 752
> uncurried β-normal form         : (λfx. f (f (f (f (f x)))))
> curried (partial) α-equivalent: 5
pLam>
```

Sjetimo se da radimo samo s prirodnim brojevima pa oduzimanje većeg od manjeg ne može dati negativan rezultat. Operacija `Pc` nad Churchovih numeralima definirana je tako da je prethodnik nule nula, a operator oduzimanja ju koristi u svojoj definiciji. Stoga, da bismo uspoređivali Churchove numerale, dovoljni su nam $\lambda$-izrazi definirani na sljedeći način.

**Primjer 2.2.6** (test za nulu, manje ili jednako, jednakost)**.**

```
pLam> T = \x y. x
pLam> F = \x y. y
pLam> and = \x y. x y F
pLam>
pLam> Pc = \n f x. n (\g h. h (g f)) (\u. x) (\u. u)
pLam> sub = \m n. (n Pc) m
pLam>
pLam> IsZc = \n. n (\x. F) T
pLam> leq = \m n. IsZc (sub m n)
pLam> eq = \m n. and (leq m n) (leq n m)
pLam>
pLam> eq 4 5
> reductions count                : 113
> uncurried β-normal form         : (λxy. y)
> curried (partial) α-equivalent: F
pLam> eq 5 5
> reductions count                : 134
> uncurried β-normal form         : (λxy. x)
> curried (partial) α-equivalent: T
pLam> eq 5 4
> reductions count                : 60
> uncurried β-normal form         : (λxy. y)
> curried (partial) α-equivalent: F
```

## 2.3 Kombinator Y

Cirkularna definicija rekurzivnih funkcija, koje će nam trebati u točki 2.4, nije dopuštena u $\lambda$-računu. Stoga moramo posegnuti za najznačajnijom idejom u teoriji programskih jezika. Konstruirat ćemo $\lambda$-izraz $\mathtt{Y}$ koji će, primijenjen na ne-rekurzivnu funkciju $f$, kreirati njenu „rekurzivnu verziju" tražeći joj fiksnu točku. Točnije, želimo da se $(\mathtt{Y}\ f)$ reducira u $(f\ (\mathtt{Y}\ f))$.

**Definicija 2.3.1** (kombinator $\mathtt{Y}$)**.** $\mathtt{Y} := \lambda f.\ \Big(\lambda x.\ f\ (x\ x)\Big)\ \Big(\lambda x.\ f\ (x\ x)\Big)$. ◁

**Propozicija 2.3.2.** $\mathtt{Y}$ *je kombinator fiksne točke, tj. za sve* $F \in \Lambda$ *vrijedi*

$$\mathtt{Y}\ F = F\ (\mathtt{Y}\ F).$$

*Dokaz.* Neka je $W := \lambda x.\ F\ (x\ x)$. Tada je

$$\mathtt{Y}\ F = W\ W = \Big(\lambda x.\ F\ (x\ x)\Big)\ W = F\ (W\ W) = F\ (\mathtt{Y}\ F).$$ □

**Napomena 2.3.3.** *Kombinatora fiksne točke ima prebrojivo mnogo.*

Pokušajmo sada definirati jedan od najpoznatijih primjera rekurzivne funkcije — funkciju koja računa faktorijel. Htjeli bismo da izraz `fakt` primijenjen na numeral $c_n$ rekurzivno računa i vraća $c_{n!}$. Krenimo s prvim i najintuitivnijim pokušajem te definirajmo $\lambda$-izraz

$$\mathtt{fact0} := \lambda n.\ \mathtt{if}\ (\mathtt{eq}\ n\ c_0)\ c_1\ \Big(\mathtt{mul}\ n\ \Big(\mathtt{fact0}\ (\mathtt{P}_c\ n)\Big)\Big).$$

Kao što smo rekli, cirkularne definicije nisu dopuštene pa zasad možemo apstrahirati `fact0` u tijelu funkcije čime dobijemo

$$\mathtt{fact1} := \lambda f n.\ \mathtt{if}\ (\mathtt{eq}\ n\ c_0)\ c_1\ \Big(\mathtt{mul}\ n\ \Big(f\ (\mathtt{P}_c\ n)\Big)\Big).$$

Ovakva definicija više nije cirkularna pa je dopustiva u $\lambda$-računu, ali izgubili smo svojstvo rekurzivnosti. No, izraz `fakt1` jako je blizu rješenja koje tražimo, što možemo vidjeti u sljedećoj lemi.

**Lema 2.3.4.** *Funkcija* $\mathtt{fact} := \mathtt{Y}\ \mathtt{fact1}$ *rekurzivno računa faktorijel, tj. za svaki* $n \in \mathbb{N}$ *vrijedi* $\mathtt{fact}\ c_n = c_{n!}$.

*Dokaz.* Primijetimo da je definicija dopustiva u $\lambda$-računu jer nije cirkularna.
Dokaz provodimo indukcijom. Za lakše čitanje dokaza, označimo $\mathtt{y} := \lambda x.\ \mathtt{fact1}\ (x\ x)$ i primijetimo da je tada $\mathtt{Y}\ \mathtt{fact1} = \mathtt{y}\ \mathtt{y} = \mathtt{fact1}\ (\mathtt{y}\ \mathtt{y})$.

- Kao bazu indukcije, provjerimo da tvrdnja vrijedi za $c_0$.

$$\begin{aligned}\texttt{fact}\ c_0 &= (\texttt{Y}\ \texttt{fact1})\ c_0 = \texttt{fact1}\ (\texttt{y}\ \texttt{y})\ c_0 = \\ &= \texttt{if}\ (\texttt{eq}\ c_0\ c_0)\ c_1\ \Big(\texttt{mul}\ c_0\ \big((\texttt{y}\ \texttt{y})\ (\texttt{P}_c\ c_0)\big)\Big) = c_1.\end{aligned}$$

- Neka tvrdnja vrijedi za neki $c_n$, tj. $\texttt{fact}\ c_n = c_{n!}$. Tada imamo (uz početak sličan kao gore)

$$\begin{aligned}\texttt{fact}\ c_{n+1} &= \cdots = \\ &= \texttt{if}\ (\texttt{eq}\ c_{n+1}\ c_0)\ c_1\ \Big(\texttt{mul}\ c_{n+1}\ \big((\texttt{y}\ \texttt{y})\ (\texttt{P}_c\ c_{n+1})\big)\Big) = \\ &= \texttt{mul}\ c_{n+1}\ \big((\texttt{y}\ \texttt{y})\ (\texttt{Pc}\ c_{n+1})\big) = \\ &= \texttt{mul}\ c_{n+1}\ (\texttt{fact}\ c_n) = \\ &= (\textit{pretp.}) = \texttt{mul}\ c_{n+1}\ c_{n!} = c_{(n+1)\cdot n!} = c_{(n+1)!}.\end{aligned}$$

Iz dokaza koraka indukcije vidimo da `fact` zaista računa na rekurzivan način. □

## 2.4 $\lambda$-definabilnost

Već nam je jasno da $\lambda$-račun reprezentira određenu klasu numeričkih funkcija (barem one koje smo definirali u primjeru 2.2.5). Pokazat će se, po rezultatu Kleeneja, da su to upravo parcijalno rekurzivne funkcije. Krenimo sad s dokazima koji će voditi tom rezultatu.

**Definicija 2.4.1** ($\lambda$-definabilna funkcija)**.** Neka je $\varphi$ numerička funkcija s $k$ argumenata. Kažemo da je $\varphi$ *$\lambda$-definabilna* ako postoji $F \in \Lambda$ takav da za sve prirodne brojeve $n_1, \ldots, n_k$ vrijede tvrdnje:

- Ako je $\varphi$ definirana u $(n_1, \ldots, n_k)$, tada $F\ c_{n_1} \cdots\ c_{n_k} = c_{\varphi(n_1,\ldots,n_k)}$.
- Ako $\varphi$ nije definirana u $(n_1, \ldots, n_k)$, tada je $F\ c_{n_1} \cdots\ c_{n_k}$ nerješiv.

Eksplicitno kažemo i da je $\varphi$ *$\lambda$-definirana* s $F$. ◁

**Lema 2.4.2.** *Inicijalne funkcije su $\lambda$-definabilne.*

*Dokaz.* Definirajmo odgovarajuće $\lambda$-izraze za nul-funkciju, projekciju i sljedbenika.

- $\mathtt{Z} := \lambda x.\, c_0$.
  Trivijalno.
- $\mathtt{I}_i^p := \lambda x_1 \ldots x_p.\, (x_1\ \mathtt{I}\ \mathtt{I}) \cdots (x_p\ \mathtt{I}\ \mathtt{I})\ x_i$.
  Projekcija nije rješiva ako i samo ako joj neki od argumenata nije rješiv.
- Neka je $f$ $\lambda$-izraz koji predstavlja funkciju sljedbenika i neka $x$ predstavlja broj nulu. Tada bi, po definiciji Churchovih numerala, $(c_n\ f\ x)$ predstavljao broj $n$ (nastao $n$ puta primjenom $f$ na $x$), a njegov sljedbenik bi tada bio $f\ (c_n\ f\ x)$.
  Dakle, $c_{n+1}\ f\ x = f\ (c_n\ f\ x)$.
  Ako sad apstrahiramo $f$ i $x$ na obje strane, imamo $\lambda f x.\, c_{n+1}\ f\ x = \lambda f x.\, f\ (c_n\ f\ x)$. Na lijevoj strani iskoristimo pravilo $\eta$-konverzije da bismo dobili $c_{n+1} = \lambda f x.\, f\ (c_n\ f\ x)$, te sad apstrahiramo $c_n$ da bismo dobili funkciju sljedbenika

  $$\mathtt{S}_c := \lambda n f x.\, f\ (n\ f\ x).$$

  Zaista,

  $$\begin{aligned}\mathtt{S}_c\ c_n &= \Big(\lambda n f x.\, f\ (n\ f\ x)\Big)\ c_n = \lambda f x.\, f\ (c_n\ f\ x) = \\ &= \lambda f x.\, f\ \Big(\big(\lambda f x.\, f\ (f\ \cdots\ (f\ x)\ \cdots)\big)\ f\ x\Big) = \\ &= \lambda f x.\, f\ \Big(f\ (f\ \cdots\ (f\ x)\ \cdots)\Big) = c_{n+1}.\end{aligned}$$ □

Iz sljedećeg niza lema slijedit će da su sve parcijalno rekurzivne funkcije $\lambda$-definabilne. Redom pokazujemo zatvorenost $\lambda$-definabilnih funkcija na kompoziciju, primitivnu rekurziju i minimizaciju.

**Lema 2.4.3.** *Skup $\lambda$-definabilnih funkcija zatvoren je na kompoziciju.*

*Dokaz.* Neka su $\chi, \psi_1, \ldots, \psi_m$ numeričke funkcije $\lambda$-definirane s $G, H_1, \ldots, H_m$. Tada je

$$\varphi(\vec{n}) \simeq \chi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

$\lambda$-definirana s

$$F \equiv \lambda \vec{x}.\, (H_1\ \vec{x}\ \mathtt{I}\ \mathtt{I}) \cdots (H_m\ \vec{x}\ \mathtt{I}\ \mathtt{I})\ \Big(G\ (H_1\ \vec{x}) \cdots (H_m\ \vec{x})\Big).$$

Primijetimo kako u ovoj definiciji, koristeći lemu 2.2.3, osiguravamo da $(F\ \vec{x})$ neće biti rješiv ako za neki $i$ $(H_i\ \vec{x})$ nije rješiv, čak i ako ga $G$ ne koristi za svoje izračunavanje. □

**Lema 2.4.4.** *Skup $\lambda$-definabilnih funkcija zatvoren je na primitivnu rekurziju.*

*Dokaz.* Neka su $\chi$ i $\psi$ totalne funkcije $\lambda$-definirane s $G$ i $H$. Tada je $\varphi$ definirana s

$$\begin{aligned}\varphi(\vec{n}, 0) &= \chi(\vec{n}),\\ \varphi(\vec{n}, k+1) &= \psi(\vec{n}, k, \varphi(\vec{n}, k))\end{aligned}$$

$\lambda$-definirana s

$$F \equiv \mathtt{Y}\left(\lambda f \vec{x} k.\ (\mathtt{IsZ}_c\ k)\ (G\ \vec{x})\left(H\ \vec{x}\ (\mathtt{P}_c\ k)\left(f\ \vec{x}\ (\mathtt{P}_c\ k)\right)\right)\right).$$

Indukcijom po $k$, vidi se da je $F\ \vec{c_n}\ k = c_{\varphi(\vec{n},k)}$. □

**Lema 2.4.5.** *Skup $\lambda$-definabilnih funkcije zatvoren je na minimizaciju.*

*Dokaz.* Neka je je $\chi$ totalna funkcija $\lambda$-definirana s $G$ i

$$\varphi(\vec{n}) \simeq \mu m[\chi(\vec{n}, m) = 0].$$

Kao pomoćnu funkciju, za neki $L \in \Lambda$ za koji vrijedi ili $L\ c_n = \mathtt{T}$ ili $L\ c_n = \mathtt{F}$ za sve $n$, definirajmo $H_L = \mathtt{Y}\left(\lambda hc.\ (L\ c)\ c\left(h\ (\mathtt{S}_c\ c)\right)\right)$. Sad minimizaciju izraza $L$ možemo zapisati kao slanje početne vrijednosti $c = c_0$ rekurzivnom evaluatoru $H_L$, tj. $\mu L := H_L\ c_0$.
Analogno, za $L = \lambda y.\ \mathtt{IsZ}_c\ (G\ \vec{x}\ y)$, izraz

$$F \equiv \lambda \vec{x}.\ \mu\Big(\lambda y.\ \mathtt{IsZ}_c\ (G\ \vec{x}\ y)\Big)$$

$\lambda$-definira $\varphi$. □

**Propozicija 2.4.6.** *Sve parcijalno rekurzivne funkcije su $\lambda$-definabilne.*

*Dokaz.* Slijedi iz prethodnih lema. □

**Teorem 2.4.7** (Kleene)**.** *Numerička funkcija je parcijalno rekurzivna ako i samo ako je $\lambda$-definabilna.*

## 2.5 Primjeri

Ovdje pokazujemo implementaciju i djelovanje izraza za konstruiranje kompozicije, primitivne rekurzije i minimizacije te izraza koje smo njihovom primjenom dobili.

Po uzoru na leme 2.4.3, 2.4.4 i 2.4.5, definiramo izraze za konstruiranje kompozicija triju mjesnosti, jednomjesne i dvomjesne primitivne rekurzije te minimizacije. Svi oni primaju izraze koji predstavljaju parcijalno rekurzivne funkcije od kojih će dobivena parcijalno rekurzivna funkcija biti građena.

U sljedećem primjeru koristit ćemo biblioteku *comp.plam* (čiji se detaljni sadržaj nalazi u točki 4.2) u kojoj smo definirali sve gore spomenute izraze (inicijalne funkcije, `I12` za stvaranje prve dvomjesne koordinatne projekcije, `C3` za stvaranje tromjesne kompozicije, `PR0` za stvaranje jednomjesne primitivne rekurzije, ...). Više o bibliotekama i njihovom korištenju biti će rečeno u točki 4.2.

**Primjer 2.5.1** (prethodnik, zbrajanje)**.**

```
pLam> :import comp
pLam>
pLam> pred = PR0 Z I12
pLam> pred 7
> reductions count                : 33
> uncurried β-normal form         : (λfx. f (f (f (f (f (f x))))))
> curried (partial) α-equivalent: 6
pLam>
pLam> add = PR1 I11 (C3 S I33)
pLam> add 2 3
> reductions count                : 129
> uncurried β-normal form         : (λfx. f (f (f (f (f x)))))
> curried (partial) α-equivalent: 5
pLam>
```

Kao primjer minimizacije, naći ćemo najmanju nultočku funkcije $f(x) = (2 - x)(3 - x)$, tj. 2. Valja primijetiti kako ne možemo zapisati $f(x) = (x - 2)(x - 3)$ jer nemamo mogućnost rada s negativnim brojevima pa bi odmah početni kandidat za rješenje minimizacije $x = 0$ dao $f(0) = 0 \cdot 0 = 0$.

Ovdje će nam osim izraza definiranih u *comp.plam* biblioteci trebati i izrazi za množenje i oduzimanje Churchovih numerala koji su definirani u biblioteci *std.plam* pa ćemo i nju uključiti u donji program.

**Primjer 2.5.2** (minimizacija)**.**

```
pLam> :import comp
pLam> :import std
pLam>
pLam> fun = \x. mul (sub 2 x) (sub 3 x)
pLam> zero = MIN1 fun
pLam>
pLam> zero
> reductions count                : 115
> uncurried β-normal form         : (λfx. f (f x))
> curried (partial) α-equivalent: 2
pLam>
```

# Poglavlje 3

# Brojevni sustavi

Već smo se susreli s Churchovim numeralima u prošlom poglavlju. Lako je uočiti da je takva definicija numerala vrlo pogodna za laku implementaciju funkcija zbrajanja, množenja i potenciranja, ali vrlo kompliciranа (i ne-efikasna) za implementaciju operacije oduzimanja. Nadahnuti tim nedostatkom, ovdje ćemo istražiti i neke druge mogućnosti implementacije numerala u skupu $\Lambda$ te najviše vremena posvetiti binarnim numeralima.

Definirajmo prvo općenite rezultate o brojevnim sustavima koji će nam biti oslonac tijekom implementacije novog sustava.

**Definicija 3.0.1.** *Brojevni sustav* je niz $d = d_0, d_1, \ldots$ kojeg čine kombinatori, takav da postoje neki $\lambda$-izrazi $\mathtt{S}_d$ i $\mathtt{isZ}_d$ takvi da za sve $n \in \mathbb{N}$ vrijedi

$$\mathtt{S}_d\ d_n = d_{n+1},$$
$$\mathtt{IsZ}_d\ d_0 = \mathtt{T}, \quad \mathtt{IsZ}_d\ d_{n+1} = \mathtt{F}.$$

◁

**Definicija 3.0.2.** Neka je $d$ brojevni sustav.

(I) Numerička funkcija $\varphi : \mathbb{N}^p \rightharpoonup \mathbb{N}$ je *$\lambda$-definabilna s obzirom na $d$* ako

$$(\exists F \in \Lambda)\ (\forall n_1, \ldots, n_p \in \mathbb{N})\ (F\ d_{n_1} \cdots\ d_{n_p} = d_{\varphi(n_1,\ldots,n_p)}).$$

(II) Brojevni sustav $d$ je *adekvatan* ako i samo ako su sve parcijalno rekurzivne funkcije $\lambda$-definabilne s obzirom na $d$.

◁

**Propozicija 3.0.3.** *Neka je d brojevni sustav. d je adekvatan ako i samo ako*

$$(\exists P_d \in \Lambda)\ (\forall n \in \mathbb{N})\ (P_d\ d_{n+1} = d_n).$$

*Dokaz.* Pokazat ćemo samo jedan smjer.

$\Rightarrow$ Ako je $d$ adekvatan, tada su po definiciji sve parcijalno rekurzivne funkcije $\lambda$-definabilne s obzirom na $d$. $P_d$ (prethodnik) je parcijalno rekurzivna funkcija. Preciznije, prethodnik definiramo kao primitivnu rekurziju incijalnih funkcija $Z$ i $I_1^2$.

$\Leftarrow$ Može se vidjeti u [2], *Propozicija 6.4.3*. □

## 3.1 Adekvatnost Churchovih numerala

Kao što smo vidjeli, definicije većine osnovnih računskih operacija s Churchovim numeralima nisu komplicirane. Kako bismo pokazali adekvatnost, trebat nam funkcija prethodnika koja nije jednostavna niti na prvi pogled, a ni idejno, te je stvarala velike probleme čak i Alonzu Churchu koji je gotovo odustao od ideje da je prethodnik $\lambda$-definabilan.

Ipak, presudan je tu bio još jedan od njegovih uspješnih studenata — Stephen Kleene, koji se početkom 1932. godine dosjetio kako definirati prethodnik u stomatološkoj ordinaciji, tijekom vađenja umnjaka! [8]

**Korolar 3.1.1.** $(c_n)_{n\in\mathbb{N}}$ *je adekvatan brojevni sustav.*

*Dokaz.* U točki 2.4 pokazali smo da je sljedbenik za Churchove numerale definiran kao $\lambda$-izraz

$$S_c := \lambda nfx.\ f\ (n\ f\ x),$$

Izvedimo sad i $\lambda$-izraze za prethodnika te test za nulu.

Kako je Churchov numeral $c_n$ definiran kao $n$-struka primjenu funkcije, htjeli bismo da prethodnik za primljeni numeral $c_n$ vraća numeral s jednom manje primjenom funkcije. U $\lambda$-računu nemamo mogućnost uklanjanja aplikacije pa ćemo morati kreirati prethodnik krečući od nule. Htjeli bismo neki $\lambda$-izraz $F$ koji bi supstituiran za $f$ u Churchovom numeralu „prikrio" prvu primjenu $f$ na $x$. Promotrimo izraze

$$\begin{aligned} X &:= \lambda u.\ x, \\ F &:= \lambda gh.\ h\ (g\ f), \\ E &:= \lambda k.\ k\ (\lambda u.\ u), \end{aligned}$$

te primijetimo kako se reduciraju aplikacije izraza $F$ na $X$:

$$\begin{aligned}
F\ X &= \Big(\lambda gh.\ h\ (g\ f)\Big)\ X = \lambda h.\ h\ (X\ f) = \lambda h.\ h\ \Big((\lambda u.\ x)\ f\Big) = \lambda h.\ h\ x,\\
F\ (F\ X) &= \Big(\lambda gh.\ h\ (g\ f)\Big)\ (\lambda h.\ h\ x) = \lambda h.\ h\ \Big((\lambda h.\ h\ x)\ f\Big) = \lambda h.\ h\ (f\ x),\\
&\vdots\\
F^k\ X &= \cdots = \lambda h.\ h\ (f^{k-1}\ x).
\end{aligned}$$

Također, za svaki $k > 0$ imamo

$$\begin{aligned}
E\ (F^k\ X) &= \Big(\lambda k.\ k\ (\lambda u.\ u)\Big)\ \Big(\lambda h.\ h\ (f^{k-1}\ x)\Big) = \Big(\lambda h.\ h\ (f^{k-1}\ x)\Big)\ (\lambda u.\ u) =\\
&= (\lambda u.\ u)\ (f^{k-1}\ x) = f^{k-1}\ x.
\end{aligned}$$

Dakle, prethodnik bismo definirali tako da primljeni numeral apliciramo na $F$ i $X$ te na kraju primijenimo $E$ kojim „izvučemo" potrebni pod-izraz koji želimo pisati u tijelo apstrakcije. Ne moramo eksplicitno koristiti gore definirani izraz $E$, pa prethodnik (kao što smo već i vidjeli), definiramo kao

$$\mathtt{P}_c := \lambda nfx.\ n\ F\ X\ (\lambda u.u) \equiv \lambda nfx.\ n\ \Big(\lambda gh.\ h\ (g\ f)\Big)\ (\lambda u.\ x)\ (\lambda u.\ u)$$

Provjerimo da zadovoljava svojstva koja očekujemo tako da $\mathtt{P}_c\ c_0$ raspišemo detaljno, a ostale slučajeve koristeći pokrate.

$$\begin{aligned}
\mathtt{P}_c\ c_0 &= \Big(\lambda nfx.\ n\ \Big(\lambda gh.\ h\ (g\ f)\Big)\ (\lambda u.\ x)\ (\lambda u.\ u)\Big)\ (\lambda fx.\ x) =\\
&= \lambda fx.\ (\lambda fx.\ x)\ \Big(\lambda gh.\ h\ (g\ f)\Big)\ (\lambda u.\ x)\ (\lambda u.\ u) =\\
&= \lambda fx.\ (\lambda x.\ x)\ (\lambda u.\ x)\ (\lambda u.\ u) =\\
&= \lambda fx.\ (\lambda u.\ x)\ (\lambda u.\ u) =\\
&= \lambda fx.\ x = c_0.
\end{aligned}$$

$$\begin{aligned}
\mathtt{P}_c\ c_{n+1} &= \Big(\lambda nfx.\ n\ F\ X\ (\lambda u.u)\Big)\ (\lambda fx.\ f^{n+1}\ x) =\\
&= \lambda fx.\ (\lambda fx.\ f^{n+1}\ x)\ F\ X\ (\lambda u.\ u) =\\
&= \lambda fx.\ F^{n+1}\ X\ (\lambda u.\ u) =\\
&= \lambda fx.\ \Big(\lambda h.\ h\ (f^n\ x)\Big)\ (\lambda u.\ u) =\\
&= \lambda fx.\ (\lambda u.\ u)\ (f^n\ x) =\\
&= \lambda fx.\ f^n\ x = c_n.
\end{aligned}$$

Kod definiranja testa za nulu, također iskorištavamo definiciju Churchovih numerala kao $n$-struke primjene funkcije $f$ na $x$. Htjeli bismo numeral (koji prima dvije varijable) aplicirati na dva izraza takva da onaj koji supstituiramo za $f$ bude funkcija koja uvijek vraća laž, a onaj koju supstituiramo za $x$ da bude istina. Lako je vidjeti da sljedeća funkcija zadovoljava tražena svojstva.

$$\texttt{IsZ}_c := \lambda n.\ n\ (\lambda x.\ \texttt{F})\ \texttt{T}$$

Ovime je pokazano da Churchovi numerali čine adekvatan brojevni sustav. □

## 3.2 Barendregtovi numerali

Postoje i brojevni sustavi u kojima funkcija prethodnika ne mora biti tako kompliciranа kao u prethodnoj točki. Hendrik Pieter Barendregt zaslužan je za numerala temeljene na uzastopnom sparivanju [10]. Da bismo ih definirali, prvo nam trebaju definicije $\lambda$-izraza za reprezentaciju i stvaranje uređenog para dvaju $\lambda$-izraza.

**Definicija 3.2.1.** *Uređeni par* dva $\lambda$-izraza $M$ i $N$ definiramo kao $\lambda$-izraz $\lambda p.\ p\ M\ N$, a dva $\lambda$-izraza *sparujemo* funkcijom $\texttt{pair} = \lambda xyp.\ p\ x\ y$. ◁

**Definicija 3.2.2.** $n$-ti *Barendregtov numeral* $h_n$ definiramo rekurzivno s

$$\begin{aligned} h_0 &:= \texttt{I}, \\ h_{n+1} &:= \texttt{pair F}\ h_n. \end{aligned}$$ ◁

Kao što možemo vidjeti iz definicije, Barendregtovi numerali također imaju svojstvo da za reprezentaciju broja $n$ trebamo $p(n)$ simbola, gdje je $p$ neki linearni polinom. Pokazat ćemo da se može puno bolje, koristeći binarni zapis u točki 3.3.

**Korolar 3.2.3.** $(h_n)_{n\in\mathbb{N}}$ *je adekvatan brojevni sustav.*

*Dokaz.* Definirajmo

$$\texttt{S}_h := \lambda x.\ \texttt{pair F}\ x, \quad \texttt{P}_h := \lambda x.\ x\ \texttt{F}, \quad \texttt{IsZ}_h := \lambda x.\ x\ \texttt{T}$$

i provjerimo da zadovoljavaju svojstva koja očekujemo od sljedbenika, prethodnika i testa za nulu. Lako se vidi da vrijedi

$$\texttt{S}_h\ h_n = (\lambda x.\ \texttt{pair F}\ x)\ h_n = \texttt{pair F}\ h_n = h_{n+1},$$

$$\begin{aligned} \texttt{P}_h\ h_{n+1} &= (\lambda x.\ x\ \texttt{F})\ (\texttt{pair F}\ h_n) = (\texttt{pair F}\ h_n)\ \texttt{F} = (\lambda p.\ p\ \texttt{F}\ h_n)\ \texttt{F} = \\ &= \texttt{F F}\ h_n = (\lambda xy.\ y)\ \texttt{F}\ h_n = h_n. \end{aligned}$$

(primijetimo da je $\mathtt{P}_h\ h_0 = (\lambda x.\ x\ \mathtt{F})\ \mathtt{I} = \mathtt{I}\ \mathtt{F} = \mathtt{F}$).
Nadalje, test za nulu ima sljedeće djelovanje:

$$\mathtt{IsZ}_h\ h_0 = (\lambda x.\ x\ \mathtt{T})\ \mathtt{I} = \mathtt{I}\ \mathtt{T} = \mathtt{T},$$
$$\mathtt{IsZ}_h\ h_{n+1} = (\lambda x.\ x\ \mathtt{T})\ (\mathtt{pair}\ \mathtt{F}\ h_n) = (\mathtt{pair}\ \mathtt{F}\ h_n)\ \mathtt{T} = (\lambda p.\ p\ \mathtt{F}\ h_n)\ \mathtt{T} = \mathtt{T}\ \mathtt{F}\ h_n = \mathtt{F}.$$

Dakle, Barendregtovi numerali također čine adekvatan brojevni sustav. □

## 3.3 Binarni numerali

Iako postoje mnoge mogućnosti kako definirati binarnu reprezentaciju brojeva u $\lambda$-računu, ovdje ćemo govoriti o onoj koja je implementirana u sklopu jezika *pLam* koji opisujemo u zadnjem poglavlju. Takva reprezentacija odlikuje se vrlo intuitivnom implementacijom računskih operacija, upravo onako kako smo navikli izvršavati ih nad nizovima bitova u računalu. Niz „bitova" od kojih će biti sačinjena naša reprezentacija numerala, definirati ćemo posebnom vrstom liste koju sada uvodimo.

**Definicija 3.3.1.** *Listu s eksplicitnim krajem* proizvoljnih $\lambda$-izraza $M_1, M_2 \ldots, M_k$ definiramo kao uzastopnu primjenu sparivanja;

$$\mathtt{pair}\ M_1\ (\mathtt{pair}\ M_2\ (\cdots\ (\mathtt{pair}\ M_k\ \mathtt{end})\ \cdots)),$$

gdje je `end` poseban $\lambda$-izraz različit od svih $M_i$, $i \in \{1, \ldots, k\}$. ◁

Prvom elementu para pristupamo funkcijom $\mathtt{fst} := \lambda p.\ p\ \mathtt{T}$, a drugom elementu para, *ostatku liste*, funkcijom $\mathtt{snd} := \lambda p.\ p\ \mathtt{F}$. Primijetimo kako su ovi izrazi $\alpha$-ekvivalentni izrazima $\mathtt{isZ}_h$ i $\mathtt{P}_h$.

Kako bismo omogućili operacije na binarnim numeralima ne-fiksne duljine, trebati će nam odgovarajuća oznaka za kraj liste (`end`). Elemente liste, bitove, ćemo prikazivati $\lambda$-izrazima za istinu ($\mathtt{T} \equiv \lambda xy.\ x$) i laž ($\mathtt{F} \equiv \lambda xy.\ y$) pa ta oznaka za kraj ne može biti nijedan od tih izraza, ali želimo da bude dovoljno jednostavna kako ne bi predstavljala prevelik teret tijekom redukcija.

Definiramo $\lambda$-izraz $\mathtt{end} := \lambda e.\ \mathtt{T}$ kao oznaku za kraj liste, te test za njega $\mathtt{isEnd} := \lambda e.\ e\ (\lambda xy.\ \mathtt{F})$. Pokažimo valjanost testa za kraj liste uz pretpostavku da su sve implementacije binarnih numerala ispravne, tj. `isEnd` uvijek koristimo za provjeru da li je **ostatak** liste (`snd` apliciran na par) `end`, pa je jedina druga opcija da je taj ostatak oblika $\mathtt{pair}\ R\ Q$, za neke $R, Q \in \Lambda$.

**Korolar 3.3.2.**

$$\mathtt{isEnd}\ M = \begin{cases} \mathtt{T}, & M \equiv \mathtt{end} \\ \mathtt{F}, & M \textit{ je par} \end{cases}$$

*Dokaz.* Pokažimo prvo slučaj kad je $M \equiv \texttt{end}$. Tada imamo

$$\begin{aligned}\texttt{isEnd end} &= \Big(\lambda e.\ e\ (\lambda xy.\ \texttt{F})\Big)\ \texttt{end}\\ &= \texttt{end}\ (\lambda xy.\ \texttt{F}) = (\lambda e.\ \texttt{T})\ (\lambda xy.\ \texttt{F}) = \texttt{T}.\end{aligned}$$

Za par $M$ proizvoljnih $\lambda$-izraza $R$ i $Q$ imamo

$$\begin{aligned}\texttt{isEnd}\ M &= \Big(\lambda e.\ e\ (\lambda xy.\ \texttt{F})\Big)\ M = M\ (\lambda xy.\ \texttt{F})\\ &= (\texttt{pair}\ R\ Q)\ (\lambda xy.\ \texttt{F}) = (\lambda p.\ p\ R\ Q)\ (\lambda xy.\ \texttt{F}) = (\lambda xy.\ \texttt{F})\ R\ Q = \texttt{F}.\end{aligned}$$ □

Sada možemo definirati binarne numerale.

**Definicija 3.3.3.** *n-ti binarni numeral* $bin_n$ definiramo kao

$$bin_n := \texttt{pair}\ b_0\ (\texttt{pair}\ b_1\ (\texttt{pair}\ b_2\ (\cdots\ (\texttt{pair}\ b_{k-1}\ \texttt{end})\cdots))),$$

gdje su $b_i, i \in \{0, 1, 2, \ldots, k-1\}$ elementi skupa $\{\texttt{T}, \texttt{F}\}$ koje zovemo *bitovi*, te vrijedi da je

$$\widehat{b_{k-1}} \cdots \widehat{b_2}\widehat{b_1}$$

binarni prikaz broja $n$, gdje su $\widehat{\texttt{F}} = 0$ i $\widehat{\texttt{T}} = 1$.
*Duljinu* binarnog numerala definiramo kao broj bitova u njegovom prikazu. ◁

Primijetimo ključnu razliku između ove implementacije i zapisa binarnih brojeva na koji smo navikli: poredak bitova. Ovdje značajnost bitova raste s lijeva na desno, tj. od početka liste do kraja. To je tako da bismo lakše implementirali prolazak po bitovima i smanjili količinu posla kod rukovanja numeralima različitih duljina, tj. njihovim poravnanjima.

Definirajmo sad nad ovakvim numeralima operacije koje će nam trebati za dokaz adekvatnosti brojevnog sustava kojeg oni čine.

### 3.3.1 Test za nulu

Iako binarnu reprezentaciju nule možemo definirati samo kao end, definiramo ju kao (pair F end) što će se pokazati korisnijim u daljnjim rezultatima.

Kako bismo znali predstavlja li dana binarna reprezentacija nulu, moramo proći po svim bitovima. Ako naiđemo na T bit, vraćamo laž, a ako smo stigli do kraja liste, vraćamo istinu;

$$\texttt{IsZ}_{bin} := \texttt{Y}\Big(\lambda f.\ \lambda x.\ (\texttt{isEnd}\ x)\ \texttt{T}\ \Big((\texttt{fst}\ x)\ \texttt{F}\ \Big(f\ (\texttt{snd}\ x)\Big)\Big)\Big).$$

### 3.3.2 Zbrajanje

Želimo funkciju `addB` koja prima dva argumenta i vraća njihov zbroj. Također, želimo da zbrajanje radi i na različitim duljinama numerala pribrojnika. Cilj nam je rekurzivno istovremeno prolaziti po bitovima oba pribrojnika, od najmanje značajnog do najznačajnijeg, te u tom procesu kreirati listu bitova koja će predstavljati zbroj. Taj rekurzivni proces omogućit će nam kombinator `Y` koji smo uveli u 2.3. Gruba ideja algoritma je sljedeća:

1. ako smo u oba numerala došli do kraju liste, vrati `end`
2. inače vrati uređeni par (*zbroj trenutnih bitova*, *rezultat algoritma na ostacima numerala*)

Drugi korak ovog algoritma nije toliko jednostavan u slučaju kad želimo omogućiti zbrajanje pribrojnika različitih duljina, i sastoji se od više grananja koja će uskoro biti detaljnije objašnjena.

Spomenimo prvo funkciju koja računa bit prijenosa (eng. *carry*):

$$\texttt{carryA} \;:=\; \lambda xyc.\; \texttt{or}\;(\texttt{and}\;x\;y)\;\Big(\texttt{and}\;(\texttt{or}\;x\;y)\;c\Big).$$

Kao pomoć u grananju, promotrimo izraz

$$\texttt{forkA} \;:=\; \lambda xyc.\; \Big(\texttt{and}\;(\texttt{isEnd}\;x)\;(\texttt{isEnd}\;y)\Big)\;E\;\Big((\texttt{isEnd}\;x)\;E1\;\Big((\texttt{isEnd}\;y)\;E2\;C\Big)\Big).$$

Naime, provjere da li su $x$ ili $y$ kraj liste mogu nas odvesti u 4 slučaja $(E, E1, E2, C)$ koje smo gore pisali radi lakšeg čitanja implementacije funkcije `forkA`, a sad ih detaljno opisujemo. S obzirom da ćemo raditi operacije nad 3 vrijednosti, radi jednostavnosti definiramo $\texttt{xor3} := \lambda xyc.\; \texttt{xor}\;(\texttt{xor}\;x\;y)\;c$.

- $E \equiv c\;(\texttt{pair T end})\;\texttt{end}$
  - gotovi smo i samo još treba provjeriti da li je ostao *carry* te u tom slučaju vratiti uređeni par izraza `T` i `end`, a inače samo `end`.
- $E1 \equiv \texttt{pair}\;\Big(\texttt{xor3 F}\;(\texttt{fst}\;y)\;c\Big)\;\Big(f\;\texttt{end}\;(\texttt{snd}\;y)\;\Big(\texttt{carryA F}\;(\texttt{fst}\;y)\;c\Big)\Big)$
  - u prvom pribrojniku smo došli do kraja (imao je manje bitova) pa nastavimo stavljati `F` kao njegove preostale bitove.
- $E2 \equiv \texttt{pair}\;\Big(\texttt{xor3}\;(\texttt{fst}\;x)\;\texttt{F}\;c\Big)\;\Big(f\;(\texttt{snd}\;x)\;\texttt{end}\;\Big(\texttt{carryA}\;(\texttt{fst}\;x)\;\texttt{F}\;c\Big)\Big)$
  - u drugom pribrojniku smo došli do kraja (imao je manje bitova) pa nastavimo stavljati `F` kao njegove preostale bitove.

- $C \equiv \texttt{pair}\left(\texttt{xor3}\ (\texttt{fst}\ x)\ (\texttt{fst}\ y)\ c\right)\left(f\ (\texttt{snd}\ x)\ (\texttt{snd}\ y)\left(\texttt{carryA}\ (\texttt{fst}\ x)\ (\texttt{fst}\ y)\ c\right)\right)$
  - nijedan pribrojnik nije na kraju, pa uzimamo prvi sljedeći bit iz oba.

Funkcija koja nam omogućuje rekurzivan prolazak po bitovima oba pribrojnika istovremeno tada je $\lambda$-definirana kao

$$\texttt{addB0} := \texttt{Y}\ (\lambda f.\ \texttt{forkA}).$$

Sad još samo trebamo staviti da je bit prijenosa jednak F na početku algoritma te je konačno željena funkcija zbrajanja $\lambda$-definirana kao

$$\texttt{addB} := \lambda xy.\ \texttt{addB0}\ x\ y\ \texttt{F}.$$

### 3.3.3 Oduzimanje

S obzirom da radimo s binarnom reprezentacijom koja je slična onoj u modernim računalima, oduzimanje nije ništa drugo no malo modificirano zbrajanje.

Ipak, implementacija oduzimanja zahtijeva veću pažnju kako bi rezultat imao jedinstveni prikaz u formi iz definicije 3.3.3, tj. trebamo dodatno brinuti o micanju nepotrebnih F bitova s kraja liste. Također, po uzoru na implementaciju prethodnika iz druga dva spomenuta brojevna sustava, htjeli bismo da rezultat oduzimanja većeg binarnog numerala od manjeg bude nula.

**Korekcije**

Definirajmo prvo $\lambda$-izraz trim koji prima binarni numeral s mogućim proizvoljnim brojem **samo** F bitova na kraju liste, te vraća binarni numeral bez njih. Primijetimo kako „proizvoljan broj samo F bitova" možemo detektirati $\lambda$-izrazom $\texttt{IsZ}_{bin}$, stoga ćemo rekurzivno proći po ulaznom binarnom numeralu kreirajući njegovu kopiju te stati i spariti end čim provjera nule na ostatku numerala vrati istinu;

$$\texttt{trim} := \texttt{Y}\left(\lambda f x.\ \left(\texttt{IsZ}_{bin}\ (\texttt{snd}\ x)\right)\left(\texttt{pair}\ (\texttt{fst}\ x)\ \texttt{end}\right)\left(\texttt{pair}\ (\texttt{fst}\ x)\ \left(f\ (\texttt{snd}\ x)\right)\right)\right).$$

Nadalje, kako bismo obilježili situaciju oduzimanja većeg od manjeg binarnog numerala, uvodimo specijalan $\lambda$-izraz $\texttt{flagNeg} = \lambda xy.\ \texttt{T}$ koji ćemo u tom slučaju spariti nakon zadnjeg bita i prije izraza end u listi koja predstavlja rezultat oduzimanja. Provjeru da li je element liste jednak flagNeg radimo $\lambda$-izrazom $\texttt{isFlagNeg} = \lambda z.\ z\ \texttt{F}\ \texttt{F}$, a provjeru da li cijela lista sadrži flagNeg, tj. da li je rezultat oduzimanja negativan, radimo $\lambda$-izrazom isNeg rekurzivno prolazeći po $\lambda$-izrazu koji je nastao oduzimanjem dva numerala, te utvrđujući da li postoji flagNeg;

$$\texttt{isNeg} = \texttt{Y}\left(\lambda f x.\ (\texttt{isEnd}\ x)\ \texttt{F}\left(\left(\texttt{isFlagNeg}\ (\texttt{fst}\ x)\right)\ \texttt{T}\ \left(f\ (\texttt{snd}\ x)\right)\right)\right).$$

Pokažimo sljedećim korolarom da `isNeg` nikada neće krivo detektirati da li je rezultat oduzimanja negativan, što izravno ovisi o djelovanju izraza `isFlagNeg`.

**Korolar 3.3.4.**

$$\texttt{isFlagNeg}\ M = \begin{cases} \texttt{T}, & M \equiv \texttt{flagNeg} \\ \texttt{F}, & M \in \{\texttt{F}, \texttt{T}\} \end{cases}$$

*Dokaz.* Primijetimo prvo da ne trebamo pokazivati druge opcije (npr. `isFlagNeg end`) jer provjeravati ćemo samo bitove numerala i detektirati ćemo `end` prije nego uđemo u provjeru `isFlagNeg`.

$$\begin{aligned} \texttt{isFlagNeg flagNeg} &= (\lambda z.\ z\ \texttt{F F})\ \texttt{flagNeg} = \texttt{flagNeg F F} = (\lambda xy.\ \texttt{T})\ \texttt{F F} = \texttt{T}, \\ \texttt{isFlagNeg F} &= (\lambda z.\ z\ \texttt{F F})\ \texttt{F} = \texttt{F F F} = (\lambda xy.\ y)\ \texttt{F F} = \texttt{F}, \\ \texttt{isFlagNeg T} &= (\lambda z.\ z\ \texttt{F F})\ \texttt{T} = \texttt{T F F} = (\lambda xy.\ x)\ \texttt{F F} = \texttt{F}. \end{aligned}$$

□

Na temelju istinitosti provjere `isNeg`, gradimo $\lambda$-izraz `makeZ` koji vraća ili binarni numeral za nulu ili nepromijenjeni ulaz;

$$\texttt{makeZ} = \lambda x.\ (\texttt{isNeg}\ x)\ \mathit{bin}_0\ x.$$

**Oduzimanje (nastavak)**

Krenimo sad sa samim algoritmom za oduzimanje dva binarna numerala, krećući, kao i kod zbrajanja, s funkcijom za *carry*:

$$\texttt{carryS} = \lambda xyc.\ \texttt{or}\ \Big(\texttt{and}\ (\texttt{not}\ x)\ y\Big)\ \Big(\texttt{and}\ \Big(\texttt{not}\ (\texttt{xor}\ x\ y)\Big)\ c\Big).$$

Zatim, grananje omogućujemo izrazom koji je po konstrukciji sličan kao `forkA` za grananje kod zbrajanja, ali dodatnu pažnju posvećujemo praćenju da li se nalazimo u situaciji kad je umanjenik manji od umanjitelja koju detektiramo ako smo na kraju umanjenika, ali ne i na kraju umanjitelja ili ako smo na kraju oba numerala, ali nam je ostao *carry* bit. Tada „označavamo" rezultat gore definiranim $\lambda$-izrazom `flagNeg`.

$$\texttt{forkS}\ :=\ \lambda xyc.\ \Big(\texttt{and}\ (\texttt{isEnd}\ x)\ (\texttt{isEnd}\ y)\Big)\ E\ \Big((\texttt{isEnd}\ x)\ E1\ \Big((\texttt{isEnd}\ y)\ E2\ C\Big)\Big).$$

gdje su $E$, $E1$, $E2$, $C$ definirani s

- $E \equiv c$ (`pair flagNeg end`) `end`
  - došli smo do kraja oba numerala (u istom trenutku) pa u ovisnosti o istinitosti *carry* bita $c$ vraćamo uređeni par izraza `flagNeg` i `end` ili samo `end`.

- $E1 \equiv \texttt{pair flagNeg end}$
  - došli smo do kraja umanjenika (ali ne i umanjitelja) pa je ovo slučaj kad je umanjenik manji od umanjitelja po broju bitova te onda sigurno i po vrijednosti, stoga vraćamo uređeni par izraza `flagNeg` i `end`.
- $E2 \equiv \texttt{pair}\left(\texttt{xor3}\ (\texttt{fst}\ x)\ \texttt{F}\ c\right)\left(f\ (\texttt{snd}\ x)\ \texttt{end}\left(\texttt{carryS}\ (\texttt{fst}\ x)\ \texttt{F}\ c\right)\right)$
  - u umanjitelju smo došli do kraja (imao je manje bitova) pa nastavimo stavljati `F` kao njegove preostale bitove.
- $C \equiv \texttt{pair}\left(\texttt{xor3}\ (\texttt{fst}\ x)\ (\texttt{fst}\ y)\ c\right)\left(f\ (\texttt{snd}\ x)\ (\texttt{snd}\ y)\left(\texttt{carryS}\ (\texttt{fst}\ x)\ (\texttt{fst}\ y)\ c\right)\right)$
  - nijedan operand nije na kraju, pa uzimamo prvi sljedeći bit iz oba.

Sad slično kao i kod zbrajanja, definiramo $\lambda$-izraz `subBf` za oduzimanje binarnih numerala prvo definirajući rekurzivni prolazak te mu dajući `F` kao početni bit prijenosa;

$$\begin{aligned}\texttt{subB0} &:= \texttt{Y}\ (\lambda f.\ \texttt{forkS}),\\ \texttt{subBf} &:= \lambda xy.\ \texttt{subB0}\ x\ y\ \texttt{F}.\end{aligned}$$

Dodatni znak „f" u imenu ovog izraza za oduzimanje nam simbolizira brzo (eng. *fast*) i nesigurno oduzimanje jer primijetimo da ne brinemo za izgled rezultata. Zato definiramo i dva dodatna, sigurnija, $\lambda$-izraza za oduzimanje binarnih numerala — oduzimanje s „podrezivanjem" (eng. *trim*) te sigurno (eng. *safe*) oduzimanje;

$$\begin{aligned}&\texttt{subBt} := \lambda xy.\ \texttt{trim}\ (\texttt{subBf}\ x\ y)\\ &\texttt{subBs} := \lambda xy.\ \texttt{trim}\ (\texttt{makeZ}\ (\texttt{subBf}\ x\ y))\end{aligned}$$

**Teorem 3.3.5.** *$(bin_n)_{n\in\mathbb{N}}$ je adekvatan brojevni sustav.*

*Dokaz.* Definirajmo $\lambda$-izraze za sljedbenika, prethodnika i test za nulu.
Test za nulu već smo definirali s

$$\texttt{IsZ}_{bin} := \texttt{Y}\left(\lambda f.\ \lambda x.\ (\texttt{isEnd}\ x)\ \texttt{T}\left((\texttt{fst}\ x)\ \texttt{F}\left(f\ (\texttt{snd}\ x)\right)\right)\right).$$

Funkciju sljedbenika definiramo jednostavno kao zbrajanje s $bin_1$,

$$\texttt{S}_{bin} := \lambda x.\ \texttt{addB}\ x\ bin_1,$$

a funkciju prethodnika kao oduzimanje s $bin_1$,

$$\texttt{P}_{bin} := \lambda x.\ \texttt{subBs}\ x\ bin_1.$$

□

## 3.4 Primjeri

Ovdje navodimo primjere $\lambda$-izraza za reprezentaciju i manipulaciju binarnim numeralima. Prvo napomenimo olakšavajuću značajku jezika *pLam* s obzirom na unos binarnih numerala, slično kao i za Churchove numerale.

**Napomena 3.4.1.** *Interpreter pLam prepoznaje i parsira binarne numerale zapisane kao* `<int>b`*, npr.* `6b` *će se parsirati u* $bin_6$*. Također, pLam prepoznaje $\lambda$-izraze koji su binarni numerali do uključivo* $bin_{2047}$*.*

**Primjer 3.4.2** (parsiranje i prepoznavanje binarnih numerala)**.**

```
pLam> 0b
> reductions count                : 2
> uncurried β-normal form         : (λp.((p (λxy. y)) (λexy.x)))
> curried (partial) α-equivalent: 0b
pLam>
pLam> 6b
> reductions count                : 6
> uncurried β-normal form        : (λp.((p (λxy. y)) (λp.((p
(λxy. x)) (λp.((p (λxy. x)) (λexy.x)))))))
> curried (partial) α-equivalent: 6b
pLam>
pLam> 2048b
> reductions count                : 24
> uncurried β-normal form        : (...)
> curried (partial) α-equivalent: (λp. ((p F) 1024b))
pLam>
```

U sljedeća dva primjera pokazujemo primjere operacija zbrajanja i oduzimanja nad binarnim i Churchovim numeralima. Svi potrebni izrazi koje koristimo za operacije nad binarnim numeralima definirani su u biblioteci *binary.plam*, a za operacije nad Churchovim numeralima u *std.plam*. Obratimo pozornost na broj redukcija prilikom izračunavanja oduzimanja.

**Primjer 3.4.3** (zbrajanje i oduzimanje binarnih numerala)**.**

```
pLam> :import binary
pLam>
pLam> addB 17b 15b
> reductions count                : 1411
> uncurried β-normal form         : (...)
> curried (partial) α-equivalent: 32b
pLam>
pLam> subBf 17b 15b
> reductions count                : 908
> uncurried β-normal form         : (...)
> curried (partial) α-equivalent: (λp. ((p F) (λp. ((p T)
(λp. ((p F) (λp. ((p F) 0b)))))))
pLam>
pLam> subBt 17b 15b
> reductions count                : 3035
> uncurried β-normal form         : (...)
> curried (partial) α-equivalent: 2b
pLam>
pLam> subBs 15b 17b
> reductions count                : 4692
> uncurried β-normal form         : (λp.((p (λxy. y)) (λexy.x)))
> curried (partial) α-equivalent: 0b
```

U sljedećem primjeru ćemo radi preglednosti pokazati samo broj redukcija. Treba ovdje imati na umu i da 1 redukcija ne znači 1 vremenska jedinica (vrijeme izvršavanja redukcije ovisi o veličini i broju izraza koje treba supstituirati).

Po pitanju samog vremena izvršavanja oduzimanja, operacije nad binarnim numeralima još su puno brže (od Churchovih) nego li se čini samo iz podatka o broju redukcija.

Važno je primijetiti da broj redukcija oduzimanja Churchovih numerala raste linearno s veličinom numerala, što postaje pogubno za rad s malo većim brojevima, dok broj redukcija pri oduzimanju binarnih numerala raste tek s brojem bitova većeg numerala.

**Primjer 3.4.4** (usporedba oduzimanja binarnih i Churchovih numerala)**.**

```
pLam> :import std
pLam>
pLam> sub 100 1
> reductions count                : 209
pLam>
pLam> sub 100 7
> reductions count                : 1397
pLam> sub 100 8
> reductions count                : 1588
pLam> sub 100 9
> reductions count                : 1777
pLam> sub 100 10
> reductions count                : 1964
pLam>
pLam> sub 100 30
> reductions count                : 5284
```

```
pLam> :import binary
pLam>
pLam> subBf 100b 1b
> reductions count                : 1549
pLam>
pLam> subBf 100b 7b
> reductions count                : 1748
pLam> subBf 100b 8b
> reductions count                : 1607
pLam> subBf 100b 9b
> reductions count                : 1564
pLam> subBf 100b 10b
> reductions count                : 1526
pLam>
pLam> subBf 100b 30b
> reductions count                : 1555
```

# Poglavlje 4

# Jezik *pLam*

U zadnjem poglavlju ovog rada navodimo dokumentaciju interpretera *pLam* za $\lambda$-račun. Sam interpreter napisan je u programskom jeziku Haskell čijom strukturom se nećemo baviti, već ćemo se fokusirati na mogućnosti našeg interpretera koji je sam za sebe dovoljno bogat i zanimljiv za proučavanje.

Nadamo se da će čitatelj uživati u otkrivanju svijeta izračunavanja koji je stvoren zaista *from scratch* unutar ovog jezika čiji naziv simbolizira pojam „čistog" $\lambda$-računa (*eng.* ***pure Lam**bda Calculus*).

Čitav izvorni kod kao i upute za njegovo korištenje, svi primjeri iz ovog rada i mnoštvo drugih, mogu se naći u GitHub-repozitoriju[1] autora.

## 4.1 Jezik

Atomarni elementi koje parsiramo su $\lambda$-izrazi koji su definirani kao u 1.1.2, uz dodatak *globalnih varijabli* (stringovi koji služe samo kao pokrata za $\lambda$-izraze koje imenuju, dakle smatramo ih $\lambda$-izrazima) i zamjenu znaka $\lambda$ znakom \.

Uneseni simboli interpretiraju se liniju po liniju, gdje svaka linija predstavlja semantičku cjelinu, svaku od kojih detaljno opisujemo sljedećom definicijom.

**Definicija 4.1.1** (Sintaksa i semantika linija unosa)**.** Moguće linije su:

- $\langle$*string*$\rangle$ = $\langle\lambda$-*izraz*$\rangle$ (**Definicija** globalne varijable)
- $\langle\lambda$-*izraz*$\rangle$ (**Prikaz** $\beta$-nf $\lambda$-izraza)
- :d $\langle\lambda$-*izraz*$\rangle$ (**Detaljan prikaz** koraka redukcije do $\beta$-nf)

[1] https://github.com/sandrolovnicki/lambda-calculus-interpreter

- :import ⟨*string*⟩ (**Uključivanje** $\lambda$-izraza iz biblioteke imena <*string*>.*plam* u trenutni radni prostor)
- −−⟨*string*⟩ (**Komentari** su sve što počinje s „−−")
- :run ⟨*string*⟩ (**Pokretanje** gotovog programa zapisanog u datoteci <*string*>.*plam*)
- :print ⟨*string*⟩ (**Ispis** proizvoljnog stringa)
- (**Prazna linija**)
- :quit (**Izlaz** iz ljuske *pLam* interpretera) ◁

Većinu ovih linija već smo sreli u ranijim primjerima pa ćemo sad na jednom primjeru pokazati one koje nismo, a bibliotekama i gotovim programima ćemo posvetiti još dva potpoglavlja.

**Primjer 4.1.2** (komentari, detaljan prikaz redukcija, izlaz)**.**

```
pLam> -- definirajmo logicke konstante i operator I
pLam> T = \x y. x
pLam> F = \x y. y
pLam> I = \x y. x y F
pLam>
pLam> -- detaljni prikaz redukcija do beta normalne forme
pLam> :d I T F
- type reduction option (a-auto, m-manual, [DEFAULT-fast]): a
-- 0: ((I T) F)
-- 1: ((λy. ((T y) F)) F)
-- 2: ((T F) F)
-- 3: ((λy. F) F)
-- 4: F
--- no beta redexes.
> reductions count                : 4
> uncurried β-normal form         : (λxy. y)
> curried (partial) α-equivalent: F
pLam> :quit
Goodbye!
```

## 4.2 Biblioteke

U primjerima tijekom ovog rada definirali smo razne $\lambda$-izraze koje bismo voljeli često koristiti, a da ih ne moramo uvijek iznova definirati. Tako su definicije $\lambda$-izraza vezanih uz logiku (primjer 1.4.3) spremljene u biblioteku *booleans.plam*, operatori za konstrukciju parcijalno rekurzivnih funkcija u *comp.plam*, te aritmetički operatori nad Churchovim i binarnim numeralima u *arithmetic.plam* i *binary.plam*. Dodatno, postoji i biblioteka *std.plam* koja se sastoji od sadržaja biblioteka *booleans.plam* i *arithmetic.plam* te služi kao osnova za prosječnog korisnika.

Svi $\lambda$-izrazi definirani u nekoj biblioteci *ime_biblioteke.plam* dodaju se u skup trenutnih globalnih varijabli naredbom `:import ime_biblioteke`. Sada pokazujemo sadržaj svih („glavnih") biblioteka koje su trenutno na raspolaganju.

```
------------------------------------------
--              STANDARD LIBRARY        --
------------------------------------------

---------------- BOOLEANS -----------------
T = \x y. x
F = \x y. y
not = \x. x F T
and = \x y. x y F
or = \x y. x T y
xor = \x y. x (not y) y
if = \p x y. p x y

--------------- ARITHMETIC ----------------
Sc = \n f x. f (n f x)
Pc = \n f x. n (\g h. h (g f)) (\u. x) (\u. u)
add = \m n f x. (m f (n f x))
sub = \m n. (n Pc) m
mul = \m n f. m (n f)
exp = \m n. n m

isZc = \n. n (\x. F) T
leq = \m n. isZc (sub m n)
eq = \m n. and (leq m n) (leq n m)
```

Listing 4.1: Biblioteka std.plam

```
  ----------------------------------------
  ||*          ___ ___  __  __ ___       *||
  ||          / __| _ \|  \/  | _ \        ||
  ||          \___|___/|_\__/_|  _/        ||
  ||                          |_|          ||
  |\     ------------------------------    /|
  ========================================
  |                                        |
  |   library for                          |
  |  defining partial recursive functions  |
  |                                        |
  +----------------------------------------+

--========================================--
              BASIC FUNCTIONS
--========================================--
T = \x y. x
F = \x y. y
P = \n f x. n (\g h. h (g f)) (\u. x) (\u. u)
isZ = \n. n (\x. F) T

--========================================--
           FIXED POINT COMBINATOR
--========================================--
Y = \f. (\x. f(x x)) (\x. f(x x))

--========================================--
             INITIAL FUNCTIONS
--========================================--
Z = \x. 0
S = \n f x. f (n f x)
I11 = \x. x
I12 = \x y. x
I22 = \x y. y
I13 = \x y z. x
I23 = \x y z. y
I33 = \x y z. z
```

```
--==========================================--
                  COMPOSITION
--==========================================--
C1 = \g h. \x. g (h x)
C2 = \g h. \x y. g (h x y)
C3 = \g h. \x y z. g (h x y z)

--==========================================--
              PRIMITIVE RECURSION
--==========================================--
PR0 = \g h. Y (\f k. (isZ k) g (h (P k) (f (P k))))
PR1 = \g h. Y (\f x y. (isZ y) (g x) (h x (P y) (f x (P y))))

--==========================================--
                  MINIMIZATION
--==========================================--
MIN1 = \g. (Y (\h x. (isZ (g x)) x (h (S x)))) 0
MIN2 = \g. (Y (\h x y. (isZ (g x y)) y (h x (S y)))) 0
```

Listing 4.2: Biblioteka comp.plam

```
------------------------------------------
--              BINARY LIBRARY          --
--======================================--

---------------- BOOLEANS ----------------
T = \x y. x
F = \x y. y
not = \x. x F T
and = \x y. x y F
or = \x y. x T y
or3 = \x y z. or (or x y) z
xor = \x y. x (not y) y
xor3 = \x y z. xor (xor x y) z
------------------------------------------
```

```
-- pair operations
pair = \x y p. p x y
fst = \p. p T
snd = \p. p F

-- this represents the end of the list
end = \e. T
isEnd = \e. e (\x y. F)
-- this marks the negative result of subtraction
flagNeg = \x y. T
isFlagNeg = \z. z F F

-- Y combinator
Y = \f. (\x. f(x x)) (\x. f(x x))

----------------------------
--     ADEQUACY - IsZbin     --
--=========================--
-- test for zero
IsZbin = Y (\f. \x. (isEnd x) T ((fst x) F (f (snd x))))


--------------------
--     ADDITION     --
--================--
carryA = \x y c. or (and x y) (and (or x y) c)

-- variant size addition branching
forkA = \x y c. (and (isEnd x) (isEnd y)) (c (pair T end) end)
((isEnd x) (pair (xor3 F (fst y) c) (f end (snd y) (carryA F
(fst y) c))) ((isEnd y) (pair (xor3 (fst x) F c) (f (snd x) end
(carryA (fst x) F c))) (pair (xor3 (fst x) (fst y) c)
(f (snd x) (snd y) (carryA (fst x) (fst y) c)))))

addB0 = Y (\f. forkA)
addB = \x y. addB0 x y F
-----------------------------------------------
```

```
------------------------
--      SUBTRACTION     --
--====================--
carryS = \x y c. or (and (not x) y) (and (not (xor x y)) c)

-- variant subtraction branching
forkS = \x y c. (and (isEnd x) (isEnd y)) (c (pair flagNeg end)
end) ((isEnd x) (pair flagNeg end) ((isEnd y) (pair (xor3
(fst x) F c) (f (snd x) end (carryS (fst x) F c)))
(pair (xor3 (fst x) (fst y) c) (f (snd x) (snd y)
(carryS (fst x) (fst y) c)))))

-- trim trailing F bits
trim = Y (\f. \x. (IsZbin (snd x)) (pair (fst x) end)
(pair (fst x) (f (snd x))))

-- whether a binary numeral has flagNeg indicator
isNeg = Y (\f. \x. (isEnd x) F ((isFlagNeg (fst x))
T (f (snd x))))
makeZ = \x. (isNeg x) 0b x

subB0 = Y (\f. forkS)
subBf = \x y. subB0 x y F
subBt = \x y. trim (subBf x y)
subBs = \x y. trim (makeZ (subBf x y))
-----------------------------------------


-----------------------------
--     ADEQUACY - Sbin, Pbin     --
--===========================--
-- successor
Sbin = \x. addB x 1b

-- predecessor
Pbin = \x. subBs x 1b
```

Listing 4.3: Biblioteka binary.plam

## 4.3 Programi

Spremanje niza naredbi kao programa, te njihovo pokretanje, također je omogućeno u našem interpreteru. Sve funkcionalnosti (barem one koje imaju smisla), tj. ključne riječi i naredbe funkcioniraju jednako kao i u interaktivnom radu interpretera.

Linije unutar programa spremljenog u *ime_programa.plam* izvršavaju se redom kojim su napisane, a program se pokreće naredbom `:run ime_programa`.

Unutar GitHub repozitorija, nalazi se direktorij `examples/` u kojem su kao programi spremljeni primjeri iz prva 3 poglavlja rada. Pokažimo pokretanje koda primjera 1.4.3.

**Primjer 4.3.1** (pokretanje spremljenog programa)**.**

```
          _
         | |
    _____| |    ___   __   __
    | _ \ |__| _ \|   \/   |
    |  _/____|____\_\__/_| v1.0.0
    |_| pure λ-calculus interpreter
   ================================

pLam> :run examples/1.4.3
================================
< and (or F (not F)) (xor T F)
================================
> reductions count              : 18
> uncurried β-normal form       : (λxy. x)
> curried (partial) α-equivalent: T
pLam>
```

# Bibliografija


[1] M. Abadi i dr. „Explicit Substitutions". (Svibanj 1991).

[2] Hendrik Pieter Barendregt. *The Lambda Calculus*. 1984.

[3] Udi Boker i Nachum Dershowitz. „The Church-Turing Thesis over Arbitrary Domains".

[4] Vedran Čačić. *Izračunljivost (predavanja)*.

[5] Nachum Dershowitz i Evgenia Falkovich. „A Formalization and Proof of the Extended Church-Turing Thesis".

[6] Benedetto Intrigila. „Some Results on Numerical Systems in $\lambda$-Calculus". *Notre Dame Journal of Formal Logic* 35.4 (1994).

[7] Fairouz Kamareddine. „Reviewing the Classical and the de Bruijn Notation for lambda-calculus and Pure Type Systems". *Journal of Logic and Computation* (svibanj 2001).

[8] Stephen C. Kleene. „Origins of Recursive Function Theory". *Annals of this history of computing* 3 (1981), str. 52–67.

[9] Sandro Lovnički. „Vremenski polinomna invarijantnost $\lambda$-računa". (Rujan 2018).

[10] Ian Mackie. „Linear Numeral Systems". *Journal of Automated Reasoning* (2018).

[11] Torben Æ. Mogensen. „An investigation of compact and efficient number representations in the pure lambda calculus".

[12] Peter Sestoft. „Demonstrating Lambda Calculus Reduction".

[13] Mladen Vuković. *Izračunljivost*. 2015.

[14] Mladen Vuković. *Složenost Algoritama*. 2018.

# Sažetak

Ukratko, u ovom radu upoznali smo čitatelja s osnovama $\lambda$-računa — jednog od modela za proučavanje izračunavanja koji je ujedno poslužio i kao temelj teorije funkcijskih programskih jezika. Također, bavimo se numeralima — reprezentacijama brojeva u $\lambda$-računu, postojećim brojevnim sustavima poput Churchovih numerala te binarnim brojevnim sustavom koji smo samostalno definirali. Kako bismo mogli evaluirati velike i kompliciranе $\lambda$-izraze, stvorili smo i programski jezik *pLam* — interpreter za čisti (*eng.* ***p****ure*) **Lam**bda račun s kojim možemo raditi interaktivno, a i pisati programe koje možemo spremiti kao tekstualne datoteke s ekstenzijom *.plam* te kasnije pokretati. U radu nastojimo popratiti svako poglavlje relevantnim primjerima kodova pisanih u jeziku *pLam*.

# Summary

In this paper, we introduce the reader to basics of $\lambda$-calculus — one of the models for studying computation which served as the foundation for the theory of functional programming languages. Furthermore, we deal with numerals — representations of numbers in $\lambda$-calculus, existant numeral systems like Church numerals and binary numerals which we defined ourselves. In order to evaluate large and complex $\lambda$-expressions, we created a programming language *pLam* — interpreter for **p**ure **Lam**bda calculus, enabling interactive mode, as well as writting programs, saving them as text files with *.plam* extension and running them later on. Throughout this paper, in each chapter we try to give relevant code examples written in *pLam*.

# Životopis

Dana 4. kolovoza 1992. godine, rođen sam u Koprivnici te cijelo svoje djetinjstvo živim u obližnjem selu Novigrad Podravski, gdje pohađam osnovnu školu „prof. Blaž Mađer". 2007. godine upisujem prirodoslovno-matematičku gimnaziju „Fran Galović" u Koprivnici. U ljeto 2009. godine selim s obitelji u Koprivnicu gdje završavam gimnaziju i u jesen 2011. godine upisujem Preddiplomski sveučilišni studij Matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu.

Ubrzo se zapošljam u Otvorenom učilištu Algebra kao predavač za matematiku te držim pripreme za državnu maturu i pojedinačne instrukcije. 2016. godine upisujem Diplomski sveučilišni studij Računarstva i matematike. Iste godine, zapošljavam se i kao programer u Lapisu gdje radim na IPTV-softveru za hotele.

2017. godine s dvojicom kolega sudjelujem na državnom sveučilišnom natjecanju u programiranju. Iste godine, u sklopu Lapisa, radim na izgradnji softverskog rješenja za gaming arene i otvaranju dviju arena u Zagrebu te jedne u Mostaru.

Tijekom zadnjih nekoliko godina radio sam (kako sam, tako u timu) na mnogo većih programskih projekata; otkrivanje egzoplaneta konvolucijskim neuronskim mrežama, simulacija rješavanja Rubikove kocke, prepoznavanje glasovnih naredbi, razvoj raznih mobilnih aplikacija, ... te proučavam teorijsko računarstvo, točnije — modele izračunavanja i njihovu ekstenzionalnu ekvivalenciju, na kom području pišem rad [9].