Математичка гимназија

МАТУРСКИ РАД

- из програмирања -

Ламбда рачун. Функционално програмирање

Аутор:
Ивана Ђуровић IVц

Ментор:
Милош Арсић

Београд, јун 2022.

Садржај

1

Увод

Нема сумње у томе да је програмирање једна од, ако не и најпопуларнија област науке у данашњем свијету. Данашње генерације одрастају окружене различитим технологијама и тиме, код многих се од малена појави радозналост и жеља за даљим проучавањем технолошких принципа. Ту људи обично крену са проучавањем различитих императивних програмских језика, гдје се строго фокусирају на принципе тих језика. Али можемо се запитати: да ли је императивно програмирање увијек примјењиво у свим ситуацијама, и ако јесте, да ли понекад можда постоји нешто што може бити практичније од њега?

У првој половини прошлог вијека, у свијету математике и информатике се дошло до неких револуционарних открића, међу којима је и тзв. **ламбда рачун**, који мијења дотадашњи поглед на функције и на то шта можемо радити са њима. Нешто касније, људи су увидјели да се један овакав систем можда може примјенити и у свијету информатике, и тиме су отворена врата **функционалног програмирања**. Оно се испоставља изузетно ефективним видом програмирања, због тога што поједностављује рјешења неких иначе комплексних проблема, и притом олакшава програмеру посао због генералне организованости која је карактеристична за функционално програмирање.

У овом раду, циљ је био да се читалац уведе у један нови свијет размишљања иза неких програмерских принципа, који вјероватно нису имали прилику да виде у императивном програмирању. Кроз једну причу о ламбда рачуну, читалац ће имати прилику да се детаљније упозна са тиме како је тај формални рачунски систем конципиран, као и да се упозна са многим новим појмовима. Касније, уз кодове који су писани у модерном програмском језику Scala, видјећемо шта је то заправо функционално програмирање, и зашто нам је оно као декларативни вид програмирања потребно у данашњем свијету. Моћи ћемо да видимо и како су тачно ламбда рачун и функционално програмирање

повезани, односно гдје се у функционалном програмирању заправо јављају принципи из ламбда рачуна.

У другој глави, читалац ће моћи детаљно да се упозна да ламбда рачуном кроз низ дефиниција и примјера.

У трећој глави, кроз програмски језик Scala илустроваћемо неке основне принципе у функционалном програмирању.

2

Ламбда рачун

2.1 Увод у λ рачун

Тридесетих година прошлог вијека, амерички математичар и логичар Алонзо Черч уводи тзв. **ламбда рачун**, који представља формални систем који се бави рачуном над функцијама и обрадом истих. Исте године, енглески математичар и логичар Алан Тјуринг дефинише концепт **Тјурингове машине**, која је заправо математички модел за једну апстрактну машину која би манипулисала симболима на некој траци по скупу одређених правила. Након оба открића, Тјуринг успјева да 1937. године докаже еквиваленцију између њих двоје по дефиницијама класа функција. И ламбда рачун и Тјурингова машина су се касније показали као изузетно значајни модели у развоју теорије програмских језика - по принципу Тјурингове машине данас раде различити императивни програмски језици, као и асемблерски језици, док су функционални програмски језици базирани на ламбда рачуну. Сам ламбда рачун се често назива најмањим универзалним програмским језиком на свијету, и своју примјену налази у различитим сферама.

У овом поглављу, читалац ће имати прилику да се боље упозна са правилима по којима функционише ламбда рачун, као и како се он може касније повезати са програмирањем.

2.2 λ-изрази

Да бисмо читаоцу приближили идеју ламбда записа, посматраћемо прво једноставну функцију која гласи:

$$f(x) = x,$$

која се може записати и у сљедећем облику:

$$f\colon x \mapsto x.$$

У ламбда нотацији, ми тежимо томе да избјегнемо коришћење назива функције, као што је у овом примјеру назив наше функције f. Да бисмо постигли то, ми ћемо ову функцију записати на сљедећи начин:

$$f \coloneqq \lambda x.x.$$

Овакав запис можемо тумачити на сљедећи начин: функција $\lambda x.x$ за свако x ($\lambda \boldsymbol{x}.x$) враћа исто то x ($\lambda x.\boldsymbol{x}$). То x могу бити неке конкретне вриједности, али зато могу бити и функције; штавише, можемо за x узети и саму функцију f. Ознака λ у запису представља ознаку која указује на почетак дефиниције неке функције, па бисмо запис $\lambda x.x$ могли да прочитамо као „дефинишемо функцију која за свако x, враћа исто то x".

Може се поставити питање како се могу ламбда записом записати функције са више аргумената. С обзиром да се у ламбда рачуну користе функције само са по једним аргументом, функције са по више аргумената се своде на тзв. **Каријеве функције** [10], које своје име носе по америчком математичару Хаскелу Карију. Наиме, **Каријевим процесом** се узима функција са више аргумената, и од ње се прави низ функција са по једним аргументом по сљедећем принципу:

$$f(x,y,z) = F \qquad h = g(x) \qquad r = h(y) \qquad F = r(z).$$

Одавде добијамо да функцију f заправо можемо записати у сљедећем облику:

$$f(x,y,z) = g(x)(y)(z)$$

што представља у суштини низ функција од којих свака зависи од по само једног параметра. Примјер једне Каријеве функције у ламбда рачуну је $\lambda y.(\lambda x.xy)$, која узима y, за које враћа другу функцију која узима x и враћа неку вриједност која зависи од x и y редом.

Сада, када читалац има основну слику о томе како функционише ламбда синтакса, дефинисаћемо шта је то ламбда израз.

Дефиниција 2.2.1. Претпоставимо да нам је дат бесконачан низ израза v_0, v_1, $\cdots$, које се зову промјенљиве, и коначан, бесконачан, или празан низ израза које се зову атомске константе, које се притом разликују од промјенљивих. Скуп израза који се зову **λ-изрази** је дефинисан на сљедећи начин:

1. Све промјенљиве и атомске константе су λ-изрази (овакви изрази се зову атоми),

2. Ако су A и B λ-изрази, онда је (AB) исто λ-израз (овакав израз се назива примјена, или апликација),

3. Ако је A било какав λ-израз, и x било која промјенљива, онда је $(\lambda x.A)$ исто λ-израз (овакав израз се назива апстракција).

4. Ламбда изрази се добијају коначном примјеном корака 1, 2 и 3.

Важно је напоменути двије ствари: за изразе облика $(\lambda x.M)$, рећи ћемо да је M тијело λ-израза, као и да се обично великим латиничним словима означавају λ-изрази, док се промјенљиве означавају малим латиничним словима.

Када смо већ споменули нотацију, могли бисмо да се осврнемо и на то како се заграде понашају у оваквим изразима и када смијемо да их занемаримо у изразима. Рецимо да имамо коначно много ламбда израза $M_1, M_2, ..., M_n$. Тада важи:

$$((...(((M_1M_2)M_3)M_4)...))M_n \equiv M_1M_2M_3M_4...M_n$$

па се може рећи да у овој ситуацији важи асоцијативност са лијеве стране. Супротно од овога, за коначно много промјенљивих $x_1, x_2, ..., x_n$ важи сљедеће:

$$\lambda x_n(\lambda x_{n-1}(...(\lambda x_1.A)...)) \equiv \lambda x_n x_{n-1}...x_1.A.$$

2.2.1 Интерпретација ламбда израза

У самом уводу о ламбда изразима имали смо коју ријеч о тумачењу неких елементарних функција. Сада ћемо уопштити нека генерална правила за интерпретацију λ-израза.

Многи ламбда изрази, иако су технички дозвољени по Дефиницији 2.2.1, се не интерпретирају из практичних разлога. Остали ламбда изрази се интерпретирају по сљедећем принципу:

- Ако је A функција или оператор, онда се (AB) тумачи као резултат примјене A на аргумент B.

- Израз $(\lambda x.A)$ се тумачи као функција или оператор, чија се вриједност за аргумент B рачуна тако што се x замијени за B у A.

Примјер 1. Посматрајмо израз $\lambda x.xy$. Можемо га протумачити као функцију која за неко x даје израз xy, који заправо представља примјену x на аргумент y. Ако бисмо овај израз примјенили на A, добићемо сљедеће:

$$(\lambda x.xy)A = Ay$$

тј. добијамо израз гдје се A примјењује на y.

Примјер 2. Посматрајмо сљедеће ламбда изразе:

а) $(\lambda y.(\lambda x.(xy)))$,

б) $(\lambda x.(\lambda y.(x(yz))))$.

Можемо да пробамо да се ослободимо заграда у овим изразима, као и да их интерпретирамо по претходно задатим правилима.

а) По правилима о ослобађању од заграда у λ-изразима, добијамо да важи сљедеће: $(\lambda y.(\lambda x.(xy))) = \lambda y.(\lambda x.(xy)) = \lambda yx.(xy) = \lambda yx.xy$. Тај израз можемо интерпретирати на сљедећи начин: за неке промјенљиве x и y, добијамо израз xy, који је заправо примјена x на y.

б) $(\lambda x.(\lambda y.(x(yz)))) = \lambda x.(\lambda y.(x(yz))) = \lambda xy.(x(yz)) = \lambda xy.x(yz)$. Не можемо се ослободити посљедњег пара заграда, с обзиром да у том случају не важи асоцијативност са десне стране. Што се тиче тумачења, израз можемо интерпретирати на сљедећи начин: за неке промјенљиве x и y, добијамо израз $x(yz)$, који се даље може протумачити као резултат примјене x на резултат примјене y на z.

2.3 Слободне и везане промјенљиве

Да бисмо могли прецизно да дефинишемо појам слободних и везаних промјенљивих, прво ћемо увести појам опсега ламбда израза.

Дефиниција 2.3.1. За одређено појављивање израза $(\lambda x.M)$ у изразу P, појављивање M се назива опсег појављивања λx са лијева.

Примјер 3. Посматрајмо сљедећи израз:

$$P \equiv (\lambda y.yx(\lambda x.yz(\lambda y.x)))v.$$

Опсег појављивања првог λy са лијеве стране у изразу P је $yx(\lambda x.yz(\lambda y.x))$, опсег појављивања λx у P је $yz(\lambda y.x)$, док је опсег појављивања крајњег λy са десне стране израза P само x.

Сада када је читалац упознат са појмом опсега, увешћемо појам слободних и везаних промјенљивих.

Дефиниција 2.3.2 (Слободне и везане промјенљиве). Појављивање промјенљиве x у изразу P назива се:

- везаним ако је у опсегу појављивања λx у изразу P,

- везаним и обавезујућим ако и само ако је x у λx,
- слободним ако није везано.

Примјер 4. Посматрајмо сљедећи израз:

$$P \equiv v(\lambda x.xy)(\lambda yz.zy).$$

Овај израз може се поистовијетити са изразом $P \equiv v(\lambda x.xy)(\lambda y(\lambda z.zy))$, те стога можемо закључити сљедеће:

$$P \equiv v(\lambda x.xy)(\lambda y(\lambda z.zy)).$$

Промјенљиве које су назначене плавом бојом су слободне промјенљиве, црвене су везане и обавезујуће, док су зелене везане али нису обавезујуће.

Сада када знамо шта су то слободне и везане промјенљиве, можемо увести појам скупа слободних промјенљивих, и објаснити како се он понаша.

Дефиниција 2.3.3. Скуп свих слободних промјенљивих у λ-изразу P се означава са $FV(P)$ (*free variables*), и за њега важе сљедећа правила:

1. Ако је x промјенљива, онда важи да је $FV(x) = \{x\}$;
2. Ако је x промјенљива, а P ламбда израз, важи да је $FV(\lambda x.P) = FV(E)\backslash\{x\}$;
3. Ако су A и B ламбда изрази, онда важи да је $FV(AB) = FV(A) \cup FV(B)$.

2.4 Супституције

Читаоцу појам супституције може бити већ познат из области математичке логике, гдје су оне представљале замјену промјенљивих термима у некој клаузи. Принцип по којем супституције функционишу у ламбда рачуну је поприлично сличан ономе у логици, уз одређене додатке због специфичности принципа по којима функционише ламбда рачун. Да бисмо прецизније дефинисали супституције, увешћемо сљедећу дефиницију са скупом правила:

Дефиниција 2.4.1 (Супституције). За било које A,B,x, тумачимо $[A/x]B$ као резултат супституције (тј. замијене) сваког појављивања промјенљиве x у B са A, као и мијењање везаних промјенљивих да се не би десила преклапања промјенљивих. Прецизна дефиниција супституције састоји се од настојећег скупа правила [3]:

а) $[A/x]x \equiv A$;

б) $[A/x]a \equiv a$, гдје је a атом, и важи $a \not\equiv x$;

ц) $[A/x](MN) \equiv ([A/x]M[A/x]N)$;

д) $[A/x](\lambda x.B) \equiv \lambda x.B$;

е) $[A/x](\lambda y.B) \equiv \lambda y.B$, ако $x \notin FV(B)$ и $x \not\equiv y$;

ф) $[A/x](\lambda y.B) \equiv \lambda y.[A/x]B$, ако $x \in FV(B)$, $y \notin FV(A)$ и $x \not\equiv y$;

г) $[A/x](\lambda y.B) \equiv \lambda z.[A/x][z/y]B$, ако $x \in FV(B)$, $y \in FV(A)$ и $x \not\equiv y$

Примјер 5. Посматрајмо ламбда израз $f(\lambda x.xy)\lambda z.xyz$. Када бисмо на њега примјенили супституцију $[g/x]$, добијамо сљедеће:

$$\begin{aligned}
[g/x](f(\lambda x.xy)\lambda z.xyz) &= \\
([g/x]f[g/x](\lambda x.xy)[g/x](\lambda z.xyz)) &= && \text{Из ц)} \\
(f[g/x](\lambda x.xy)[g/x](\lambda z.xyz)) &= && \text{Из б)} \\
(f(\lambda x.xy)[g/x](\lambda z.xyz)) &= && \text{Из д)} \\
f(\lambda x.xy)(\lambda z.gyz) && \text{Из ф) јер } x \in FV(xyz).
\end{aligned}$$

Супституције нам користе за дефинисање конверзија, које су изузетно важан апарат у ламбда рачуну.

2.5 Конверзије

2.5.1 α-конверзија

Како бисмо заобишли могућност преклапања назива различитих промјењљивих приликом редуковања ламбда израза, врши се преименовање промјењљивих. Тај поступак се назива α-конверзија.

Дефиниција 2.5.1. Нека је $(\lambda x.E)$ неки ламбда израз. Алфа конверзија овог израза представља сваку замјену везаних промјењљивих x у изразу E за неку другу промјењљиву t:

$$\lambda x.E \xrightarrow{\alpha} \lambda t.E[x = t],$$

а за два израза, међу којима се од једног може добити други алфа конверзијом, кажемо да важи алфа еквиваленција између њих, и то обиљежавамо на сљедећи начин:

$$P \equiv_\alpha P'.$$

Иако је сам принцип једноставан за разумјевање, само дефинисање је поприлично комплексно, и захтјева добар математички апарат и разумјевање ламбда супституција [8][9].

2.5.2 β-конверзија

Дефиниција 2.5.2. Ако је x промјенљива, а ако су E и F ламбда изрази, онда се трансформација:

$$(\lambda x.E)F \xmapsto{\beta} [F/x]E$$

назива бета конверзија (или бета редукција), а за два израза, међу којима се од једног може добити други бета конверзијом, кажемо да је први β-скраћен до другог, и то обиљежавамо на сљедећи начин:

$$P \rhd_\beta P'$$

Сада ћемо увести тзв. **Черч-Росерову теорему** коју нећемо доказивати, али која ће нам користити у каснијим примјерима и задацима.

Теорема 2.1 (**Черч-Росерова теорема**)**.** Два ламбда израза су еквивалентна ако се низом α и β конверзија они могу свести на исти израз.

Када смо већ дефинисали појам бета редукције, можемо да уведемо и појам редекса и нормалне форме.

Дефиниција 2.5.3. Нека је x промјенљива и нека су E и F ламбда изрази. Онда се израз $(\lambda x.E)F$ назива **редекс**.

Дефиниција 2.5.4. Ламбда израз је у **нормалној форми** ако не садржи ниједан подизраз који је редекс.

Одатле имамо да су изрази $(\lambda xy.z)$, $xy(zy)$ и $\lambda x.xx$ у нормалној форми, а $(\lambda x.xx)(\lambda x.xx)$ није.

Да би читалац лакше могао да разумије појам α и β конверзије, као и редекса и нормалне форме, издвојићемо сљедећи примјер.

Примјер 6. Покушајмо да нађемо нормалну форму сљедећих израза ако она постоји:

а) $(\lambda x.x)(\lambda x.xx)(\lambda x.xa)$

б) $(\lambda x.xxx)(\lambda x.xxx)$

Приликом рјешавања ових примјера, примјењиваћемо наше знање о груписању заграда у ламбда рачуну, као и алфа и бета конверзији.

а)

$$
\begin{aligned}
(\lambda x.x)(\lambda x.xx)(\lambda x.xa) &\equiv_\alpha \\
(\lambda x.x)(\lambda x.xx)(\lambda y.ya) &\rhd_\beta && x \Rightarrow_\alpha y \\
(\lambda x.xx)(\lambda y.ya) &\rhd_\beta && [(\lambda x.xx)/x] \\
(\lambda y.ya)(\lambda y.ya) &\rhd_\beta && [(\lambda y.ya)/x] \\
(\lambda y.ya)a &\rhd_\beta && [(\lambda y.ya)/y] \\
aa & && [\mathrm{a}/y]
\end{aligned}
$$

б)

$$
\begin{aligned}
(\lambda x.xxx)(\lambda x.xxx) &\rhd_\beta \\
(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) &\rhd_\beta && [(\lambda x.xxx)/x] \\
(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) &\rhd_\beta && [(\lambda x.xxx)/x] \\
\ldots && \text{Не постоји нормална форма}
\end{aligned}
$$

Ове двије конверзије су нам корисне из очигледних разлога - дају нам могућност да на потенцијално краћи и љепши начин прикажемо неки ламбда израз, и тиме нам упрошћава рад над истим. Ово ће нам се показати заправо корисним када будемо видјели како дефинишемо неке функције са реалном примјеном, конкретно у областима аритметике и логике.

2.6 Аритметика

До сад смо имали прилику да се упознамо са ламбда изразима и правилима везаним за рад над њима, али још увијек нисмо видјели како можемо заправо једну функцију попут $f(x) = x + 5$ записати користећи ламбда запис. Стога, да бисмо могли да радимо неке елементарне рачунске операције, увешћемо појам **нумерала**, који ће нам заправо представљати ламбда запис за ненегативне цијеле бројеве. Нумерале записујемо и дефинишемо на сљедећи начин:

$$
\begin{aligned}
&\overline{0} \equiv \lambda fx.x \\
&\overline{1} \equiv \lambda fx.f(x) \\
&\overline{2} \equiv \lambda fx.f(f(x)) \\
&\overline{3} \equiv \lambda fx.f(f(f(x))) \\
&\ldots \\
&\overline{n} \equiv \lambda fx.\underbrace{f(f(\ldots(f(x))\ldots))}_{f \text{ се понавља } n \text{ пута}}.
\end{aligned}
$$

Врло често, да би се избјегао претходни запис нумерала $\overline{n}$, користи се исти запис у сљедећем облику: $\lambda fx.f^n(x)$.

Да се подсјетимо, упознати смо са двије елементарне функције које смо раније спомињали: функцију идентитета, која за свако x враћа исто то x, и која се записује као $\lambda x.x$, као и константну функцију (коју нисмо експлицитно спомињали, али се прожимала кроз наше примјере), која за било коју вриједност x враћа неку константу c, и која се ламбда записом обиљежава са $\lambda x.c$.

Када знамо све ово, можемо кренути са дефинисањем неких основних функција које ће нам помоћи да схватимо како се овај систем користи у свијету аритметике.

Кренимо прво са функцијом *Sljedbenik* која за нумерал $\overline{n}$ враћа нумерал $\overline{n+1}$. Запишимо то на сљедећи начин:

$$Sljedbenik \quad \overline{n} \equiv \overline{n+1}$$

или

$$Sljedbenik \quad \lambda fx.f^n(x) \equiv \lambda fx.f^{n+1}(x) \equiv \lambda fx.f^n(f(x)) \tag{2.1}$$

На основу (2.1) можемо лако уочити да су нумерали $\overline{n}$ и $\overline{n+1}$ поприлично слични - једино се разликују у томе што се онамо гдје се у запису нумерала $\overline{n}$ налази само промјенљива x, у нумералу $\overline{n+1}$ се налази израз $f(x)$, тј. fx. Одатле можемо закључити да важи сљедеће:

$$\begin{aligned}
&\overline{n}f(fx) \equiv \\
&(\lambda fx.f^n(x))f(fx) \equiv \\
&(\lambda x.f^n(x))(fx) \equiv \\
&f^n(fx) \equiv \\
&f^{n+1}(x)
\end{aligned}$$

тј. да се примјеном израза f и (fx) редом на нумерал $\overline{n}$ добија тијело ламбда израза нумерала $\overline{n+1}$. То значи да бисмо од једног нумерала добили нумерал за један већи, морали бисмо да користимо ламбда израз *Sljedbenik* који ће изгледати овако:

$$Sljedbenik \quad \equiv \lambda nfx.nf(fx)$$

$$Sljedbenik \quad \overline{n} \equiv \lambda fx.\overline{n}f(fx) \equiv \lambda fx.f^{n+1}(x) \equiv \overline{n+1}$$

По овом сличном поступку се могу наћи и остале функције: видимо који низ трансформација морамо извршити да бисмо од једног израза добили нешто што личи на други израз, и на основу тога дефинишемо тражену функцију.

Исто тако, можемо наћи функције *Zbir* и *Proizvod* које редом враћају збир и производ два нумерала. Запишимо то на сљедећи начин:

$$\begin{aligned} Zbir &\quad \overline{n}\,\overline{m} \equiv \overline{n+m} \\ Proizvod &\quad \overline{n}\,\overline{m} \equiv \overline{nm}. \end{aligned}$$

Из дефиниције нумерала, имамо да важи сљедеће:

$$\begin{aligned} &\overline{n+m} \equiv \lambda fx.f^{n+m}(x) \equiv \lambda fx.f^n(f^m(x)) \\ &\overline{nm} \equiv \lambda fx.f^{nm}(x) \equiv \lambda fx.\underbrace{f^n(f^n(...(f^n(x))...))}_{f^n \text{ се понавља } m \text{ пута}} \end{aligned}$$

а из ламбда записа нумерала $\overline{n}$ и $\overline{m}$ знамо да важи:

$$\begin{aligned} &\overline{n}\ f\ x \equiv f^n(x) \\ &\overline{m}\ f\ x \equiv f^m(x). \end{aligned}$$

Из ових ствари можемо лако закључити да важи сљедеће:

$$\begin{aligned} &\overline{n}\ f\ (\overline{m}\ f\ x) \equiv && \overline{m}\ (\overline{n}\ f)\ x \equiv \\ &\overline{n}\ f\ (f^m(x)) \equiv && \overline{m}\ (\lambda x.(f^n(x)))\ x \\ &f^n(f^m(x)) \equiv && \lambda x.(\lambda x.(f^n(x)))((\lambda x.(f^n(x)))(...((\lambda x.(f^n(x)))(x))...))\ x \equiv \\ &f^{n+m}(x) && f^{nm}(x) \end{aligned}$$

на основу чега можемо закључити да се *Zbir* и *Proizvod* могу дефинисати на сљедећи начин:

$$\begin{aligned} Zbir &\equiv \lambda nmfx.nf(mfx) \\ Proizvod &\equiv \lambda nmfx.m(nf)x. \end{aligned}$$

Исто овако, могу се наћи и функције за рецимо одређивање степена нумерала, као и многе друге, али њих нећемо проћи у овом раду, већ их остављамо читаоцу на размишљање.

2.6.1 Рекурзија

Појам рекурзије је генерално свима познат из области математике и информатике, а у ламбда рачуну, она нам може помоћи у многим, конкретно аритметичким, проблемима. Да бисмо могли да примјенимо рекурзију у ламбда рачуну, морамо увести појам уређеног пара нумерала, који се дефинише на сљедећи начин:

$$(\overline{m}, \overline{n}) \equiv \lambda fgx.f^m(g^n(x)).$$

Тако су рецимо:

$$(\overline{0},\overline{0}) \equiv \lambda fgx.x$$
$$(\overline{1},\overline{0}) \equiv \lambda fgx.f(x)$$
$$(\overline{1},\overline{1}) \equiv \lambda fgx.f(g(x))$$
$$...$$

Да бисмо могли да користимо уређене парове нумерала, морамо дефинисати неколико основних операција над њима. Прво ћемо дефинисати оператор *Upari* која за нумерале $\overline{m}$ и $\overline{n}$ даје пар нумерала $(\overline{m},\overline{n})$. Слично *Zbir*-у, можемо лако закључити да бисмо упарили $\overline{m}$ и $\overline{n}$, на $\overline{m}$ морамо примјенити f и $(\overline{n}\ g\ x)$, па имамо:

$$\overline{m}\ f\ (\overline{n}\ g\ x) \equiv\ \overline{m}\ f\ g^n(x) \equiv\ f^m(g^n(x))$$

те се стога наша функција *Upari* може дефинисати као:

$$Upari\ \equiv\ \lambda mnfgx.mf(ngx).$$

Такође, требају нам неки оператори који ће нам за уређени пар нумерала вратити први, односно други нумерал. Назваћемо те операторе *Prvi* и *Drugi*.

$$Prvi\ (\overline{m},\overline{n}) \equiv \overline{m}$$
$$Drugi\ (\overline{m},\overline{n}) \equiv \overline{n}.$$

Ако посматрамо дефиницију уређеног пара нумерала, и упоредимо је са дефиницијом нумерала, можемо уочити да можемо добити једну од двије координате уређеног пара уколико једну од промјенљивих f и g у апстракцији замјенимо функцијом $(\lambda x.x)$. Одатле добијамо сљедеће:

$$(\overline{m},\overline{n})\ f\ (\lambda x.x)\ x \equiv f^m$$
$$(\overline{m},\overline{n})\ (\lambda x.x)\ g\ x \equiv g^n.$$

Стога, *Prvi* и *Drugi* ћемо дефинисати на сљедећи начин:

$$Prvi\ \equiv\ \lambda Pfx.Pf(\lambda x.x)x$$
$$Drugi\ \equiv\ \lambda Pgx.P(\lambda x.x)gx.$$

Сада када смо увели ове операторе, можемо дати неке примјере за то гдје се у ламбда рачуну може примјенити рекурзија. Један такав примјер је дефинисање оператора *Prethodnik*, који насупрот *Sljedbenik*-у, за нумерал $\overline{n}$ враћа нумерал

$\overline{n-1}$. Да бисмо дошли до *Prethodnika*-а, користићемо помоћни оператор $Pom1$ који ради сљедеће:

$$Pom1 \quad (\overline{m}, \overline{n}) \equiv (\overline{m+1}, \overline{m}).$$

Тако ћемо моћи да $Pom1$ примјењујемо n пута на нумерал $(\overline{0}, \overline{0})$, и тиме бисмо добили нумерал $(\overline{n}, \overline{n-1})$:

$$(\overline{0}, \overline{0}) \xmapsto{Pom1} (\overline{1}, \overline{0}) \xmapsto{Pom1} (\overline{2}, \overline{1}) \xmapsto{Pom1} ... \xmapsto{Pom1} (\overline{n}, \overline{n-1}).$$

Ако кажемо да је неко P уређени пар нумерала, од њега $Pom1$ прави нумерал по сљедећем принципу:

$$Pom1 \quad P \equiv Upari\ (Sljedbenik\ (Prvi\ P))\ (Prvi\ P).$$

Одатле, $Pom1$ можемо дефинисати користећи претходно уведене функције на сљедећи начин:

$$Pom1 \equiv \lambda P.Upari\ (Sljedbenik\ (Prvi\ P))\ (Prvi\ P).$$

Као што смо раније рекли, нама је циљ да добијемо претходник нумерала $\overline{n}$, који ћемо добити из уређеног пара нумерала $(\overline{n}, \overline{n-1})$, до којег ћемо доћи на сљедећи начин:

$$Pom1\ (\overline{0}, \overline{0}) \equiv (\overline{1}, \overline{0})$$
$$Pom1\ (Pom1\ (\overline{0}, \overline{0})) \equiv Pom1\ (\overline{1}, \overline{0}) \equiv (\overline{2}, \overline{1})$$
$$...$$
$$\underbrace{Pom1\ (Pom1\ (...(Pom1\ (\overline{0}, \overline{0}))...))}_{Pom1 \text{ се појављује } n \text{ пута}} \equiv (\overline{n}, \overline{n-1})$$

тј. на нумерал $\overline{n}$ ћемо примјенити $Pom1$ и $(\overline{0}, \overline{0})$:

$$\overline{n}\ Pom1\ (\overline{0}, \overline{0}) \equiv (\lambda f x.f(f(...(f(x))...)))\ Pom1\ (\overline{0}, \overline{0}) \equiv (\overline{n}, \overline{n-1}).$$

С обзиром да се нама тражи претходник нумерала $\overline{n}$, од добијеног уређеног пара нумерала, ми ћемо тражити други нумерал. Из свега овога добијамо да ће израз за оператор *Prethodnik* изгледати овако:

$$Prethodnik \equiv \lambda n.\ Drugi\ (n\ Pom1\ (\overline{0}.\overline{0}))$$

тј. ако замјенимо $Pom1$ за раније добијени израз:

$$Prethodnik \equiv \lambda n.\ Drugi\ (n\ (\lambda P.Upari\ (Sljedbenik\ (Prvi\ P))\ (Prvi\ P))\ (\overline{0}.\overline{0})).$$

Други примјер примјене рекурзије у ламбда рачуну јесте оператор $Faktorijel$, који за неки нумерал $\overline{n}$ враћа нумерал $\overline{n!}$:

$$Faktorijel \;\; \overline{n} \;\equiv\; \overline{n \cdot (n-1) \cdot ... \cdot 2 \cdot 1} \;\equiv\; \overline{n!}.$$

У овом случају, користићемо помоћни оператор $Pom2$ који функционише по сљедећем принципу:

$$Pom2 \;\; (\overline{m}, \overline{n}) \;\equiv\; (\overline{mn}, \overline{n+1}).$$

Можемо увидјети да када примјенимо n пута на уређени пар нумерала $(\overline{1}, \overline{1})$, добијамо уређени пар нумерала облика $(\overline{n!}, \overline{n+1})$:

$$(\overline{1}, \overline{1}) \overset{Pom2}{\longmapsto} (\overline{1}, \overline{2}) \overset{Pom2}{\longmapsto} (\overline{2}, \overline{3}) \overset{Pom2}{\longmapsto} (\overline{6}, \overline{4}) \overset{Pom2}{\longmapsto} ... \overset{Pom2}{\longmapsto} (\overline{n!}, \overline{n+1}).$$

Ако је P неки уређени пар нумерала, од њега $Pom2$ прави нови нумерал на сљедећи начин:

$$Pom2 \;\; P \;\equiv\; Upari(Proizvod\ (Prvi\ P)\ (Drugi\ P))(Sljedbenik\ (Drugi\ P)).$$

Одавде можемо закључити да се $Pom2$ може записати као:

$$Pom2 \;\equiv \lambda P.Upari(Proizvod\ (Prvi\ P)\ (Drugi\ P))(Sljedbenik\ (Drugi\ P)).$$

На исти начин на који смо добили израз $Prethodnik$, сада ћемо тражити да се оператор $Pom2$ изврши n пута над уређеним паром нумерала $(\overline{1}, \overline{1})$, и тако добијамо $(\overline{n!}, \overline{n+1})$:

$$Pom2\ (\overline{1}, \overline{1}) \;\equiv\; (\overline{1}, \overline{2})$$
$$Pom2\ (Pom2\ (\overline{1}, \overline{1})) \;\equiv\; Pom2\ (\overline{1}, \overline{2}) \;\equiv\; (\overline{2}, \overline{3})$$
$$...$$
$$\underbrace{Pom2\ (Pom2\ (...(Pom2\ (\overline{1}, \overline{1}))...))}_{Pom2 \text{ се појављује } n \text{ пута}} \;\equiv\; (\overline{n!}, \overline{n+1})$$

те нам ламбда израз за $Faktorijel$ изгледа овако:

$$Faktorijel \;\equiv\; \lambda n.Prva\ (n\ Pom2\ (\overline{1}, \overline{1}))$$
$$Faktorijel \;\equiv\; \lambda n.Prva\ (n\ (\lambda P.Upari(Proizvod\ (Prvi\ P)\ (Drugi\ P))$$
$$(Sljedbenik\ (Drugi\ P)))\ (\overline{1}, \overline{1})).$$

Рекурзија се може примјенити у многим другим сличним проблемима, као што су различите суме или производи. У свима, идеја је иста као у претходна два примјера: налази се нека помоћна функција која врши неке трансформације између уређених парова нумерала, и онда те трансформације примјењујемо одређени број пута.

2.7 Логика

Иако смо то могли донекле да примјетимо у аритметици, ламбда рачун има велику примјену у програмирању управо кроз његову примјену у логици, с обзиром да се програмирање врло често своди на комбинацију елементарних логичких правила. Упознати смо са чињеницом да се све у свијету информатике може свести на низове који се састоје од 0 и 1, и на разматрање када је нешто тачно, односно нетачно. Као што смо у аритметици дефинисали ненегативне цијеле бројеве преко нумерала, тако у логици можемо дефинисати ламбда изразе за тачно и нетачно, односно $TRUE$ и $FALSE$ на сљедећи начин:

$$TRUE \equiv \lambda xy.x \qquad\qquad FALSE \equiv \lambda xy.y.$$

У ламбда рачуну можемо записати и if изразе као што бисмо то урадили у класичном програмирању, и то радимо на сљедећи начин:

$$if\ m\ then\ n\ else\ k \equiv m\ n\ k$$

тј. ако је испуњен израз m, онда важи n, а иначе важи k. На овај начин можемо дефинисати основне логичке операције, које ћемо дефинисати у овом поглављу.

Дефинишимо прво оператор AND који враћа резултат конјункције два израза:

$$AND\ p\ q \equiv (p \wedge q)$$
$$AND\ TRUE\ TRUE \equiv TRUE$$
$$AND\ FALSE\ TRUE \equiv FALSE$$
$$AND\ TRUE\ FALSE \equiv FALSE$$
$$AND\ FALSE\ FALSE \equiv FALSE.$$

Ако радимо конјункцију од израза p и q редом, знамо да ако је p тачно, морамо да посматрамо вриједност израза q, а иначе ћемо одмах добити нетачно. Стога, AND можемо записати као:

$$AND \equiv \lambda pq.p\ q\ FALSE.$$

На сличан начин, посматрајмо оператор OR, који враћа резултат дисјункције два израза:

$$OR\ p\ q \equiv (p \vee q)$$
$$OR\ TRUE\ TRUE \equiv TRUE$$
$$OR\ FALSE\ TRUE \equiv TRUE$$
$$OR\ TRUE\ FALSE \equiv TRUE$$
$$OR\ FALSE\ FALSE \equiv FALSE.$$

По сличном принципу као за AND, ако имамо нека два израза p и q редом над којима вршимо дисјункцију, знамо да ако је p испуњено, онда треба да добијемо тачно, а иначе провјеравамо вриједност израза q. Стога, добијамо да се OR може записати на сљедећи начин:

$$OR \equiv \lambda pq.p\ TRUE\ q..$$

У математичкој логици, негација представља промјену тачности неког израза. У ламбда рачуну, негацију бисмо дефинисали као оператор NOT користећи претходни метод на сљедећи начин:

$$NOT \equiv \lambda p.p\ FALSE\ TRUE$$

јер ако је p тачан, треба да добијемо вриједност $FALSE$, а иначе $TRUE$.

Користећи негацију, можемо увести еквиваленцију и ексклузивну дисјункцију на сљедећи начин:

$$EQUIV \equiv \lambda pq.p\ q\ (NOT\ q)$$
$$XOR \equiv \lambda pq.p\ (NOT\ q)\ q$$

.

То можемо објаснити на сљедећи начин, ако посматрамо израз p у еквиваленцији, за тачну вриједност p-а, тражимо да ли је q тачно, а иначе да ли је q нетачно, док за ексклузивну дисјункцију важи обрнуто.

Користећи ове функције, можемо касније дефинисати комплексније логичке изразе и тиме прећи на логички начин размишљања у ламбда рачуну.

2.8 Комбинатори

Комбинаторе можемо дефинисати као ламбда изразе који у себи немају слободних промјенљивих. Идеја комбинаторне логике јесте да се сви ламбда изрази низом трансформација доведу до коначне комбинације комбинатора, не остављајући за собом ни једну слободну промјенљиву. Данас постоји дефинисани скуп комбинатора који се углавном користе у ламбда рачуну, а ми ћемо проћи само најбитније:

Симбол	Назив	Дефиниција
I	Идентитет	$\lambda x.x$
K	$Kestrel/TRUE$	$\lambda xy.x$
S	Супституција	$\lambda abc.ac(bc)$
C	$Cardinal$	$\lambda fab.fba$
M	$Mockingbird$	$\lambda x.xx$
KI	$Kite/FALSE$	$\lambda xy.y$
B	$Bluebird$	$\lambda fab.f(ab)$

Интересантна чињеница је да се данас, поред кратких симбола, оне углавном називају по именима птица, јер је Хаскел Кари, један од зачетника комбинаторне логике, био велики обожаватељ птица, док име Мојсеја Шејнфинкела, другог зачетнинка комбинаторне логике, има значење "лијепа птица".

Када већ причамо о комбинаторима, можемо увести и појам комбинаторног израза.

Дефиниција 2.8.1. Претпоставимо да имамо бесконачан низ израза $v_0, v_1, \ldots$ које се зову промјенљиве, и коначан низ израза који се зову атомске константе, уз три основна комбинатора, I, K и S. Комбинаторним изразом називамо оне који испуњавају сљедећа правила:

1. Све промјенљиве, атомске константе и основни комбинатори I, K и S су комбинаторни изрази.

2. Ако су A и B комбинаторни изрази, онда је и (AB) комбинаторни израз.

Комбинаторна логика поприлично наличи класичном ламбда рачуну, само немамо више апстракције, и кроз изразе се углавном прожимају I, K и S. Међутим, један од принципа који се појављује у комбинаторној логици, али не и у класичном ламбда рачуну, јесте тзв. слаба редукција. Идејно, иста је β-конверзији, једина разлика је што слаба редукција има другачију синтаксу.

Дефиниција 2.8.2 (Слаба редукција). Било који комбинаторни израз облика IX, KXY и $SXYZ$ се називају слабим редексима. Сама слаба редукција се обиљежава са $\triangleright_w$ и дефинише на сљедећи начин:

$$IX \triangleright_w X \qquad KXY \triangleright_w X \qquad SXYZ \triangleright_w XZ(YZ).$$

У складу са овим, постоји и слаба нормална форма, као и теорема која тврди да су два израза еквивалентна ако се низом слабих редукција могу свести на исту нормалну форму.

О комбинаторима, аритметици и логици имаћемо више ријечи у сљедећој глави, гдје ћемо моћи да видимо реалну примјену једног оваквог формалног система рачунања.

3

Функционално програмирање

3.1 Увод у функционално програмирање

Функционално програмирање можемо дефинисати као програмску парадигму која се заснива на евалуацији израза, који су састављени од функција које немају промјенљиво стање и самим тим споредне ефекте. **Споредни ефекти** су све појаве које доводе до тога да функција за одређени скуп вриједности може вратити различити скуп рјешења. То могу бити измјене промјенљивих, односно типова промјенљивих, учитавање података из конзоле или фајлова, штампање података у конзоли, итд.

Основна идеја функција у функционалном програмирању јесте да се оне третирају као објекти прве класе, тј. да свака функција може бити параметар неке функције, може бити враћена као вриједност неке функције, или се пак може додијелити некој промјенљивој у једначини. Ово подсјећа на ламбда рачун, гдје смо раније дефинисали неке операције над неким другим функцијама. Касније ћемо имати прилику да уочимо још паралели између ламбда рачуна и функционалног програмирања, јер као што смо већ раније навели, инспирација за развиће ове програмске парадигме стоји иза научних сазнања до којих су дошли Алонзо Черч и Алан Тјуринг у првој половини прошлог вијека.

Иако је као вид декларативног програмирања функционално програмирање знатно мање популарно од императивног, данас се често користи у науци и разним сферама индустрије. Од функционалних програмских језика, најпознатији су: LISP, Haskell, Elixir, Erlang, Scheme,..., док сљедећи језици, иако нису обавезно функционални, такође имају имплементације које омогућавају рад у функционалном програмирању: C++11, C#, Kotlin, PHP, Python, Java, Scala,...

Ми ћемо у овом раду користити програмски језик Scala, јер је генерално лак за разумијевање, и има доста сличности са Java језиком.

3.2 Увод у Scala програмски језик

Прије него што се даље удубимо у функционално програмирање, могли бисмо да кажемо пар ријечи о програмском језику у којем ћемо писати кодове који се налазе у раду, као и о синтакси која се у њему користи.

За читаоце који би хтјели да се опробају у Scala језику, постоји више опција. Једна од опција је да се, умјесто преузимања и намјештања неких нових интегрисаних развојних окружења, код пише у веб-прегледачу на сајту: `https://scastie.scala-lang.org/pEBYc5VMT02wAGaDrfLnyw`. За оне који више воле да раде у неком IDE-у, могу то учинити у Eclipse-у или у Visual Studio Code-у уз додатну инсталацију Scala Metals библиотеке. Постоји опција и да се код куца у Command Prompt-у, уз додатну инсталацију Java 8 виртуалне машине. Више информација о опцијама за програмирање у програмском језику Scala налазе се на сљедећем линку: `https://docs.scala-lang.org/getting-started/index.html`.

У овом програмском језику имамо два начина на које можемо програмирати: можемо у терминалу писати једнолинијске команде које се извршавају одмах по притиску Enter-а на тастатури, или можемо написати вишелинијски код у неком фајлу, који ћемо послије компајловати и пустити програм да изврши оно што смо тражили.

Размотримо најприје прву опцију, с обзиром да је лакша за разумијевање. Када отворимо Scala у терминалу, можемо уносити различите аритметичке и логичке изразе, као и if селекције. Погледајмо примјер како се у Scala језику израчунава вриједност једног аритметичког израза:

```
scala> 3 + 4 / 3 - 1 * 2
res0: Int = 2
```

Као што можемо примјетити, програм је добијену вриједност аутоматски додијелио у неку фиксну промјенљиву res0, интерпретирао којег типа мора бити, и одштампао добијену вриједност. Ако ми желимо да уведемо промјенљиве, можемо то учинити коришћењем кључних ријечи var и val:

```
scala> val Rezultat1 = 2 * 4 + 3 / 4
val Rezultat1: Int = 8

scala> var Rezultat2 = 9 / 4 + 3 * 4 / 8
val Rezultat2: Int = 3
```

Разлика између var и val јесте у томе што var декларише промјенљиве у правом смислу те ријечи - оне могу послије да мијењају вриједности, за разлику од оних које су дефинисане са val, које заправо представљају константне вриједности, и не могу да се мијењају након декларације:

```
scala> Rezultat2 = 5
// mutated Rezultat2

scala> println(Rezultat2)
5

scala> Rezultat1 = 2
               ^
        error: reassignment to val

scala> println(Rezultat1)
8
```

Иначе, Scala може да ради над сљедећим скупом типова података: Byte, Boolean, Char, Short, Int, Long, Float, Double. Битно је знати да Double чува прецизност до 15 цифара, као и да се максималне вриједности сваког типа података може наћи помоћу команде MaxValue.
Уколико желимо да укључимо неку од библиотека са додатним функцијама, то радимо преко команде import на сљедећи начин:

```
scala> import scala.math._
import scala.math._
// biblioteka sa matematickim funkcijama
```

Слично Java и C/C++ језицима, у функционалном програмирању можемо користити сљедеће операторе: $+=$, $-=$, $*=$, $/=$, $\%=$ у аритметичким изразима, &&, ||, ! у логичким изразима, као и релационе симболе $>$, $<$, $>=$, $<=$, $==$, $!=$. Такође имамо и while, do while и for петље. Ту су још многи други оператори и функције, које ћемо увести када нам буду требали. Као што бисмо то иначе радили у другим програмским језицима, и у Scala језику можемо се користити са if изразима:

```
scala> val P = 4
val P: Int = 4

scala> val IfCase = if (P >= 5) "yes" else "no"
val IfCase: String = no

scala> val IfCase2 = if ((P >= 2) && (P <= 3)) "First Case"
                     else if ((P > 3) && (P <=5)) "Second Case"
                     else "Third Case"
val IfCase2: String = Second Case
```

Такође, можемо дефинисати функције и методе, који су два поприлично слична појма, чију ћемо фундаменталну разлику објаснити у неком од наредних поглавља. За сада ћемо само рећи да се методе дефинишу уз кључну ријеч def, док се функције дефинишу независно. Погледајмо како се врши дефинисање једне функције и једног метода у сљедећем примјеру:

```
//funkcija
scala> val Reciprocan = (x : Double) => 1 / x
val Reciprocan: Double => Double = $Lambda$1094/0x000000010077b040@7
9a1f0a1

scala> Reciprocan(2)
val res3: Double = 0.5

//metoda
scala> def Zbir(x : Int, y : Int): Int = x + y
def Zbir(x: Int, y: Int): Int

scala> println(Zbir(1,3))
4
```

Као што можемо примјетити у синтакси функције и методе из претходног примјера, (x : Double) => 1 / x указује на то да се у наведеној функцији за неку реалну вриједност x враћа њему реципрочна вриједност, док (x : Int, y : Int): Int указује на то да нам дата метода за неке цијелобројне вриједности x и y враћа неку цијелобројну вриједност. У методама понекад не морамо експлицитно нагласити који тип података мора да врати, те смо наведену методу могли да уведемо само као Zbir(x : Int, y : Int).
Када желимо да дефинишемо методу која нам не враћа никакву конкретну вриједност, користићемо израз Unit. Суштински, подсјећа на void тип података у другим програмским језицима, као што су Java или рецимо C/C++.

```
scala> def Stampaj(ime: String): Unit={println("Moje ime je "+ime)}
def Stampaj(ime: String): Unit

scala> val test = Stampaj("Ivana")
Moje ime je Ivana
val test: Unit = ()
```

Уколико желимо да пишемо код у фајлу са .scala екстензијом, наш код ће имати структуру сличну кодовима у језику као што је рецимо Java. Све што смо до сад споменули од уграђених функција и оператора можемо користити у регуларном коду, уз неке додатке као што су класе и објекти.
Класе нам служе за дефинисање наших типова података, заједно са скупом функција, конструктора и деструктора уско везаних за њих. Да бисмо дефинисали класу, користимо кључну ријеч class, поред којег уписујемо произвољно име класе над којом радимо, као и скуп параметара класе са њиховим типовима.

```
class Tacka (x: Int, y: Int){
    def Prva(): Unit = { println("Prva koordinata : " + x) }
    def Druga(): Unit = { println("Druga koordinata : " + y) }
}
```

Уколико у коду желимо да имамо неки податак са типом који смо задали класом, користићемо кључну ријеч new као у сљедећем примјеру:

```
val tacka1 = new Tacka(3,0)
// x = 3, y = 0
val tacka2 = new Tacka(x = 2)
// y nece imati konkretnu dodijeljenu vrijednost
val tacka3 = new Tacka(y = 9)
// x nece imati konkretnu dodijeljenu vrijednost
val tacka4 = new Tacka
// ni x ni y nece imati konkretne dodijeljene vrijednosti
```

Конструкторе можемо додати тако што ћемо у класи за неки аргумент поред његовог типа дописати знак = и конкретну почетну вриједност коју желимо да податак има.
Објекте можемо дефинисати као класе које не могу да се инстанцирају, тј. не могу се позивати са кључном ријечју new и не дефинишу тип података. Методе које су написане у оквиру објеката се могу позивати било гдје у коду, и третирају се као статички методи у Java језику. Они се иначе дефинишу уз ријеч object, као у сљедећем примјеру:

```
object PrimjerObjekta{
    ...
}
```

Да би наш код функционисао како треба, у барем једном од објеката у програму мора се наћи метода main(args: Array[String]), у којој се заправо извршава позивање свих споредних метода, као и повезивање са осталим класама и објектима. Стога, један једноставан програм у овом програмском језику би изгледао овако:

```
object Program1{
    def main(args: Array[String]) = {
        val a = 2
        val b = 5
        println("Zbir brojeva "+ a +" i "+ b +" iznosi "+ (a+b))
        // a + b = 2 + 5 = 7
    }
}
```

За оне који желе да се више информишу о Scala језику и његовој синтакси и функцијама, то може учинити на званичном сајту овог програмског језика: https://www.scala-lang.org/. Сада када смо дали једно елементарно појашњење принципа по којима функционише овај уникатан програмски језик, можемо да се вратимо на причу о функционалном програмирању, гдје ћемо ту и тамо укључити неки илустративни примјер у Scala коду.

3.3 Чисте функције. Референцијална транспарентност

Као што смо рекли у уводу, функционално програмирање ради само са оним функцијама које не зависе од неких страних промјењљивих фактора које иначе називамо споредним или непожељним ефектима. Функције које немају никакве споредне ефекте се називају **чистим функцијама** [5][6]. Још један начин на који бисмо могли описати чисте функције јесте као функције које се за било коју вриједност која им је додијељена увијек пресликавају у тачно одређену вриједност (односно свака вриједност из домена функције је мапирана на тачно дефинисану и непромјењљиву вриједност). Примјер једне нечисте функције јесте Random() из Scale, која генерише насумичне бројевне вриједности на предефинисаном интервалу цијелих бројева. Посматрајмо примјер једне чисте функције коју ћемо записати у облику метода Proizvod:

```
scala> def Proizvod(x: Int, y: Int):Int = (x * y)
Proizvod: (x: Int, y: Int)Int

scala> val P = Proizvod(2,6)
P = 12
```

Попут онога што смо већ рекли, за вриједности 2 и 6 увијек ће нам метода Proizvod вратити вриједност 12. Стога, не бисмо погријешили ако бисмо свако појављивање Proizvod(2,6) замијенили са бројем 12. Ова особина да се чиста функција са одређеним скупом параметара може увијек замијенити са њеном вриједношћу, без икаквог утицаја на програм, се назива **референцијална транспарентност** [6]. Ова особина значајно олакшава debug-овање програма, и рјешавање комплексних проблема у коду, а поготово оних кодова који се састоје од строго чистих функција.

3.4 Анонимне функције

У нашем коду, може се десити да неку функцију желимо искористити само једном, и тиме желимо заобићи дефинисање нове функције са именом, листом аргумената са њиховим параметрима, типом података који функција треба да врати, као и само тијело функције. То можемо постићи коришћењем **анонимних функција**, које су заправо функције без додијељеног имена.

```
// standardna funkcija
def Sljedbenik(n: Int) = { n + 1 }

//anonimna funkcija
(n: Int) => n + 1
```

У овом примјеру, можемо видјети стандарндну и анонимну имплементацију функције која враћа сљедбеник неког броја. Да смо писали код, и да нам се баш у том тренутку јавила потреба да искористимо ову функцију, не бисмо били у обавези да креирамо нову функцију са именом, ако је нећемо користити у остатку кода [7].
Анонимне функције се могу повезати и са изразима у ламбда рачуну, гдје ни једна такорећи функција није имала име, и није се позивала по имену, већ се памтила по операцијама које извршава, аргументима које узима, и вриједности коју враћа.

3.5 Функције као објекти прве класе

Прије него што зађемо у дубљу причу, дефинишимо објекте прве класе.

Дефиниција 3.5.1. Објекат прве класе јесте ентитет који може да трпи све операције које се могу примјенити на друге конкретне вриједности. То обухвата:

1. прослеђивање аргумента функције,

2. враћање из функције,

3. додјељивање некој промјенљивој.

С обзиром да смо већ раније рекли да су функције у функционалном програмирању објекти прве класе, то значи да можемо вршити све претходно наведене операције над њима. Како би читалац могао боље разумјети како се ово разликује од објектно-оријентисаног програмирања, прођимо кроз неколико примјера.

3.5.1 Прослеђивање функције као аргумента функције

Да бисмо могли да уведемо примјер за прослеђивање функција као аргумената функција, уведишћемо појам листа у Scala програмском језику.
Листе у Scali се дефинителы на сљедећи начин:

```
scala> val Lista: List[Int] = List(1, 9, 7, 4, 8, 2, 3)
val Lista: List[Int] = List(1, 9, 7, 4, 8, 2, 3)

scala> val Imena: List[String] = List("Ivana", "Iva", "Ivan")
val Imena: List[String] = List(Ivana, Iva, Ivan)
```

Уз листе, имамо и низ различитих функција које могу користити за обраду листа, као што су head (враћа главу листе), tail (враћа све елементе листе осим главе), reverse (обрће чланове листе),... Међутим, ми ћемо се тренутно фокусирати на једну другу функцију која се зове map, која враћа листу након што изврши неку предефинисану операцију над свим функцијама.

```
object Program2{
    def main(args: Array[String]) = {
        val a : List[Int] = List (1, 2, 3, 4, 5)

        val b : List[Int] = a.map(x => x + 2)

        println(b) // (3, 4, 5, 6, 7)
    }
}
```

Као што смо раније навели, функција map прво позива неку другу функцију као свој аргумент, и затим, помоћу те функције, она извршава мијењање вриједности чланова листе. У овом примјеру, функција која је била прослеђена као аргумент је била функција која увећава неки број за 2, и самим тим, у нашој листи ће се измијенити бројевне вриједности и постаће List(3, 4, 5, 6, 7).

3.5.2 Враћање функције из функције

У овом примјеру, користићемо функцију која враћа име и презиме неке особе.

```
scala> def ImeIPrezime(Ime: String) = (Prezime: String) =>
{ Ime + " " + Prezime }
def ImeIPrezime(Ime: String): String => String

scala> val ime = ImeIPrezime("Ivana")
val ime: String => String = $Lambda$1068/0x0000000100755840@
d512c1

scala> ime("Djurovic")
val res0: String = Ivana Djurovic
```

Методу ImeIPrezime дефинисали смо као функцију која за неки параметар Ime враћа анонимну функцију која узима параметар Prezime и враћа комбинацију та два стринга. Затим, ime дефинишемо као претходно наведену анонимну функцију са унешеним параметром Ime=”Ivana”. На крају, само позивамо функцију ime са стрингом ”Djurovic” као параметром, и тиме добијамо тражену комбинацију.

3.5.3 Додјељивање функције некој промјенљивој

У сљедећем примјеру, моћи ћемо да видимо један једноставан примјер додијеле функције некој промјенљивој.

```
scala> def Proizvod(x: Int, y: Int) = (x * y)
def Proizvod(x: Int, y: Int): Int

scala> var d = Proizvod _
var d: (Int, Int) => Int = $Lambda$1086/0x0000000100770040@
30259c61
 // donja crta ukazuje na to da ignorisemo parametre

scala> d(4,5)
val res3: Int = 20
```

Као што можемо уочити, након што извршимо додјелу функције без параметра некој промјенљивој d, онда d можемо третирати као функцију, и позивати је као да је Proizvod. Ово смо могли уочити и у претходним примјерима, али овај мало боље илуструје сам овај принцип.

3.6 Функције вишег реда

Функције вишег реда (higher-order functions) су оне функције које узимају друге функције као аргументе или враћају функције. Њих смо несвјесно користили у претходним примјерима, попут функције map коју смо споменули у 3.4.1, или као методе ImeIPrezime коју смо увели у 3.4.2. Важно је напоменути да немају сви програмски језици имплементацију функција вишег реда, као рецимо Java. Зато је Scala много бољи и флексибилнији језик за функционално програмирање од Jave.

Можемо се запитати зашто су нам функције вишег реда толико значајне? Да бисмо дали одговор на ово питање, погледајмо сљедећи програм:

```
object Program3{

    def main(args: Array[String]) = {

        val Niz: Array[Int] = Array(1, 2, 3, 4, 5)
        for(i <- 0 to (Niz.length - 1)) {
            Niz(i) += 2
        }
        for(i <- 0 to (Niz.length - 1)) {
            println(Niz(i)) // (3, 4, 5, 6, 7)
        }
    }
}
```

У њему смо увели низ са цијелобројним вриједностима (низови се дефинишу

кључном ријечју Array[Type], гдје је Type тип података у низу). Након тога, прошли смо кроз низ, и у њему сваки број повећали за 2, и на крају смо одштампали новодобијене вриједности низа. С обзиром да имамо неку операцију која ће се извршити над свим члановима низа, нама би било лакше да напишемо неку методу која би над неким низом извршила низ операција које бисмо ми дефинисали. Стога, наш програм можемо написати и на сљедећи начин:

```
object Program4{

    def MapArray(ar: Array[Int], f: Int => Int) = {
        for(i <- 0 to (ar.length - 1)) {
            ar(i) = f(ar(i))
        }
    }

    def main(args: Array[String]) = {

        val Niz: Array[Int] = Array(1, 2, 3, 4, 5)

        MapArray(Niz, x => (x + 2))

        for(i <- 0 to (Niz.length - 1)) {
            println(Niz(i)) // (3, 4, 5, 6, 7)
        }
    }
}
```

Као што се може уочити, у коду смо по узору на функцију map дефинисали методу MapArray, која итерира кроз низ, и над сваким чланом низа врши неку операцију. Касније, ту методу позивамо у методи main, и тиме она ради исто оно што смо радили у претходном коду. Иако се комплексност програма није промијенила, други код је сада читљивији од првог, јер умјесто компликација са for петљама у main-у, можемо обавити један позив методе. Такође, ако будемо имали неки други низ, или неку другу операцију коју будемо хтјели да извршимо над члановима неког низа, можемо само искористити методу MapArray без писања нових for-ова. Увођењем функција вишег реда, поједностављује се структура кода и апстрахује се имплементација неких комплекснијих алгоритама.

Овдје можемо уочити једну паралелу између ламбда рачуна и функционалног програмирања. У ламбда рачуну, све смо третирали као функције, и наш рад се сводио строго на рад над њима. Постојање функција вишег реда нам отвара ту исту могућност у програмирању - оно што би иначе могле само бити неке конкретне вриједности, сада постају функције.

3.7 Каријеве функције

Подсјетимо се прије свега шта су то Каријеве функције - то су функције које узимају више параметара један по један и враћају низ функција, које затим настављају да узимају параметре док не дођу до неке конкретне вриједности. Функције са више аргумената се доводе до Каријевих функција поступком који се назива Каријев поступак. Сада ћемо погледати како можемо извршити тај поступак у Scali.
Погледајмо сљедећи код:

```
object Program5{

    def KlasicnaFunkcija (x: Int, y: Int) = {
        x + y * y
    }

    def main(args: Array[String]) = {

        val f = KlasicnaFunkcija(2, 3) // f = 2 + 3 * 3 = 11
        println(f)
    }

}
```

Помоћу методе KlasicnaFunkcija дефинисали смо оно што бисмо у математици записали као функцију $f(x, y) = x + y^2$. Оно што ми желимо да имамо јесте да добијемо функцију $h(x)(y)$ за коју важи сљедеће: $h(x)(y) = (h(x))(y) = f(x, y)$. То значи да ћемо морати да дефинишемо методу која ће представљати функцију вишег реда, и то на сљедећи начин:

```
object Program6 {

    def KarijevaFunkcija (x: Int) = (y: Int) => {
        x + y * y
    }

    def main(args: Array[String]) = {

        val f = KarijevaFunkcija(2)(3) // f = 11
        println(f)

        val g = KarijevaFunkcija(4)(_)
        //nismo unijeli drugu promjenljivu
        println(g(5)) // g(5) = 4 + 5 * 5 = 29
    }
}
```

Овдје видимо имплементацију Каријеве функције која ради исто оно што смо хтјели да ради и метода из претходног примјера. У main методи можемо уочити два начина позивања методе KarijevaFunkcija: промјенљивој f смо одмах додијелили вриједност Каријеве функције са оба параметра, док смо промјенљивој

g додијелили заправо функцију која би се у математичком облику записала на сљедећи начин: $f(4, y) = g_4(y) = 4 + y^2$. Послије g позивамо као функцију у println() и њеној промјенљивој додијељујемо вриједност 5.

3.8 Рекурзија

Један од битнијих принципа у функционалном програмирању су **рекурзије**, које представљају функције које позивају саме себе. За примјер рекурзивне функције, користићемо Фибоначијев низ:

```
object Program7 {

    def Fibonaci (n: Int): Int = {
        if (n == 1 || n == 2) 1
        else Fibonaci(n - 2) + Fibonaci(n - 1)
    }

    def main(args: Array[String]) = {

        for(i <- 1 to 10) println(Fibonaci(i))
        // 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
    }

}
```

Главни проблем са рекурзијама јесте у томе што се може десити да оне створе превише позива, који ће притом попунити стек, што може да доведе до онога што је иначе у програмирању познато као "stack overflow". Поставља се питање како можемо избјећи ово, а одговор се налази у једном изузетно корисном концепту познатом као репна рекурзија, који је уско повезан уз функционално програмирање.

3.8.1 Репна рекурзија

Репне рекурзије су оне рекурзивне функције код којих се рекурзивни позив обавља на самом крају функције. У оваквим рекурзивним позивима немамо проблема са претрпавањем стека, с обзиром да нема потребе за памћењем претходног стања функције, јер се повратна вриједност функције рачуна само као позив самој себи. Да бисмо ово лакше схватили, погледајмо сљедећи примјер:

```
import scala.annotation.tailrec
// importuje se biblioteka za tailrec - repne rekurzije

object Program8
{
    def Fibonaci(n: Int): Int =
    {
        // @tailrec - pocetak repne rekurzije
        // a1 - prethodno stanje u Fibonacijevom nizu
        // a2 - trenutno stanje u Fibonacijevom nizu
        @tailrec def FibPom(n: Int, a1: Int, a2: Int): Int =
        {
            if (n == 1)
                a1 + a2
            else
                FibPom(n - 1, a2, a1 + a2)
        }
        FibPom(n, 1, 0)
    }


    def main(args:Array[String]) = {

        for(i <- 1 to 10) println(Fibonaci(i))
    }
}
```

Да бисмо извршили репну рекурзију, морамо одрадити позив рекурзивне функције унутар функције која ће нам дати крајњи резултат. Такође, можемо видјети да смо тиме сто сваки пут памтимо вриједности претходног и тренутног члана Фибоначијевог низа загарантовали да нећемо имати непотребно трпање стека [11].

4

Закључак

Приликом проучавања ове теме посљедњих неколико мјесеци, аутор је имао прилику да се упозна са једном другачијом и изузетно интересантном облашћу, која повезује двије научне области које му најдраже: информатику и математику. Кроз овај рад, он је покушао да читаоцу приближи своја сазнања, колико уз дефиниције и теорију коју је сматрао битном, толико и уз примјере. Кроз дио рада о ламбда рачуна, читалац је имао прилику да се упозна са тиме шта су ламбда изрази и каква је њихова синтакса, шта су то алфа и бета конверзије, шта су то комбинатори, као и како се ламбда изрази могу практично користити у областима аритметике и логике. У глави о функционалном програмирању, видјели смо један релативно нов програмски језик који је прављен по идејама ламбда рачуна, и кроз њега смо показали шта су то чисте функције, зашто су нам битне анонимне функције и функције вишег реда, како програмски можемо имплементирати Каријеве функције, као и како функционишу рекурзије.

С обзиром да је тема овог рада градиво које се традиционално не проучава за вријеме средње школе, аутор се трудио да рад конципира тако да буде прилагођен четворогодишњем средњошколском образовању из програмирања. Програми који се налазе у раду намјерно су били једноставнији, јер је за циљ имао више да илуструје идеје иза функционалног програмирања, за разлику од писања неких комплексних кодова и функционалних програма. Тиме је хтио да читалац има прилику да идејно разумије функционално програмирање, и са тим знањем да може послије сам да има прилику да истражи како се пишу неки комплекснији чисто функционални програми.

Аутору је изузетно драго што је за матурски рад имао овако занимљиву тему, која га је додатно подстакла да се интересује за програмирање. За крај, он се жели захвалити свом ментору Милошу Арсићу, који му је помогао приликом израде овог рада са различитим материјалима и усмјерењима.

Литература

[1] H. Barendregt, E. Barensen, *Introduction to Lambda Calculus*, Revised edition December 1998, March 2000, https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf

[2] R. Rojas, *A Tutorial Introduction to the Lambda Calculus*, FU Berlin, WS-97/98, https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

[3] H. Barendregt, J. R. Hindley, J. P. Seldin, *Lambda-Calculus and Combinators*, Cambridge, 1986, https://www.cin.ufpe.br/~djo/files/Lambda-Calculus%20and%20Combinators.pdf

[4] P. Chisuano, R. Bjarnason, *Functional Programming in Scala*, Manning, September 2014, https://www.manning.com/books/functional-programming-in-scala

[5] N. Raychaudhuri, E. Barensen, *Scala in Action*, Manning 2013, https://www.manning.com/books/scala-in-action

[6] Tour of Scala: https://docs.scala-lang.org/tour/basics.html

[7] Functional Programming - Scala: https://www.learningjournal.guru/article/scala/functional-programming/

[8] CMCS 312: Programming Languages, Lecture 3: Lambda Calculus (Syntax, Substitution, Beta Reduction), Acar & Ahmed, 17 January 2008 https://home.ttic.edu/~pl/classes/CMSC336-Winter08/lectures/lec3.pdf

[9] Kevin Sookocheff, Alpha Conversion https://sookocheff.com/post/fp/alpha-conversion/

[10] Милена Вујошевић Јаничић, Програмске парадигме, Математички факултет, Универзитет у Београду http://www.programskijezici.matf.bg.ac.rs/ppR/2019/predavanja/fp/funkcionalno_programiranje.pdf

[11] Recursion in Scala https://www.geeksforgeeks.org/recursion-in-scala/