МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета

Програмирање и програмски језици

на тему

Ламбда рачун и функционално програмирање

уз додатни пројекат

израда компајлера у програмском језику Haskell

Ученик:

Никола Јовановић, $\mathrm{IV_d}$

Ментор:

Иван Чукић

Београд, мај 2015.

САДРЖАЈ

1 Увод **4**

2 Ламбда рачун **6**

2.1 Дефиниција 7
2.2 Везане и слободне промењиве 8
2.3 λ-редукција 9
2.3.1 β-редукција 10
2.3.2 α-редукција 10
2.3.3 η-редукција 11
2.4 Аритметика и логика 11
2.5 Рекурзија 14

3 Функционално програмирање и програмски језик Haskell **16**

3.1 Прве функције 17
3.2 Систем типова 19
3.2.1 Листе и n-торке 21
3.2.2 Преглед типова и класа типова 22
3.3 Лења евалуација 23
3.4 Рекурзија и тражење узорака 24
3.5 Каријеве функције и функције вишег реда 25
3.5.1 map и filter 26

4 Имплементација компајлера у Haskell-у **30**

4.1 Структура пројекта 31
4.2 Улазни језик PV 33
4.3 Лексичка анализа 35
4.3.1 Регуларни изрази 35
4.3.2 Коначни аутомати 36

4.3.3 Имплементација лексера 37
4.4 Синтаксна анализа 39
4.4.1 Имплементација парсера 41
4.5 Виртуелна стек машина 43
4.6 Генерисање кода 48
5 Закључак **51**
Литература **53**

Глава 1

Увод

„To iterate is human, to recurse divine." - Л. Питер Дојч

При помену појмова **програмирање** и **програмски језик** прва асоцијација је у већини случајева строго у вези са неким императивним језиком. Овакво размишљање је логично: императивни програмски језици као што су Java или C/C++ су данас заиста најзаступљенији, како у образовању, тако и у индустрији. У периоду школовања програмирање се, кад се јавља, јавља увек у облику императивних језика. Тек на факултету, или у завршној години средње школе, ђаци могу да се сусретну са неким декларативним језиком, који по доласку из императивног света делује неинтуитивно и чудно. Постоји хипотеза да је итеративни начин размишања природнији од рекурзивног, па се и при причи о израчунљивости и почецима рачунара најчешће говори о **Тјуринговој машини**, а тзв. **ламбда рачун**, еквивалентан модел израчунљивости којим су инспирисани функционални језици, ретко и да се помиње. Са друге стране, декларативни језици, а нарочито функционални, често као један од својих главних адута истичу интуитивност и блискост кода програмеру. Последица тога су кратки, концизни програми, који се пишу брзо и уз јако мало дебаговања.

Управо због тих квалитета полако почињу да бивају заступљени неки од тих језика (Haskell, Lisp, Erlang, ML, R...) или чак хибридни језици, који обједињују више различитих парадигми (F#, OCaml, Scala...). Многе светски познате компаније почињу да користе функционалне језике, а најзначајније примере представљају компаније као што су Facebook, Google, Intel и Microsoft које користе Haskell и OCaml за одређене делове својих пројеката [12, 13]. Одличан пример је и компанија Jane Street која се бави финансијама и трговином на берзи. Тај посао носи са собом велики ризик, и један баг у коду може да кошта фирму милионе долара. Како би смањио број багова и осигурао тачност кодова, Jane Street је

одлучио да искористи предности функционалног програмирања, тј. чињеницу да већина грешака буде *„ухваћена”* при самој компилацији, и користи OCaml за цело своје пословање [14]. Такође је важно поменути и језик Erlang, који је нашао примену у телекомуникацијама где замењује популарне императивне језике.

У последње време, идеје функционалног програмирања за које се показало да су корисне налазе своје место и у императивним језицима. Најпопуларнији концепт који бива имплементиран су тзв. *„ламбде”*, анонимне функције инспирисане λ рачуном. C++, Java, C#, PHP и многи други језици пружају подршку за ламбде у својим најновијим стандардима. Сам C++ одавно подржава неке идеје функционалног програмирања кроз заглавља <functional> и <algorithm>, као и генеричко програмирање *(„templates”)*, а стандард C++11 имплементира и облике **инференције типова** и **map-filter** филозофије. Оба концепта су инспирисана функционалним програмирањем, и биће објашњена у глави 3.

Циљ овог рада је да пружи увод у начине размишљања и идеје иза ламбда рачуна и функционалног програмирања и покаже одређене предности у односу на *„стандардно”* размишљање. Овај рад садржи и практичан део, имплементацију компајлера у програмском језику Haskell. Сврха имплементационог дела је да на конкретном примеру демонстрира предности функционалног програмирања.

Друга глава представља увод у ламбда рачун и његове принципе, као теоријску подлогу функционалног програмирања. У трећој глави ће бити уведено функционално програмирање кроз програмски језик Haskell , док се четврта глава фокусира на имплементациони део овог рада, израду компајлера у Haskell -у.

Глава 2

Ламбда рачун

„A language that doesn't affect the way you think about programming is not worth knowing."
- Алан Ј. Перлис

Ламбда (λ) рачун [1] је формални систем настао тридесетих година двадесетог века у циљу проучавања особина израчунљивих функција. Први га је дефинисао **Алонзо Черч**[1] , исте године када је **Алан Тјуринг**[2] дефинисао модел који зовемо **Тјурингова машина** [7]. Нешто касније је, упркос очигледној различитости ова два модела, доказана њихова еквиваленција по питању класа функција које дефинишу и формулисана је **Черч-Тјурингова теза**, хипотеза која тврди да се неформална дефиниција израчунљивости (функција је израчунљива ако може да је, игноришући ресурсе, израчуна човек користећи папир и оловку) поклапа са Тјуринг-израчунљивошћу (класа функција израчунљивих користећи Тјурингову машину), па самим тим и са λ-израчунљивошћу (функције израчунљиве уз помоћ λ рачуна). Рад ове двојице научника представља значајан искорак на пољу теоријских рачунарских наука.

Сам λ рачун је систем који описује израчунавање базиран на функцијама и примени истих, и има значајну примену у теорији доказа и теорији програмских језика. Ова глава има за циљ да формално дефинише λ рачун и објасни механизме по којима он функционише, уз посебан осврт на аритметику, логику и рекурзију, примере који демонстрирају моћ овог наизглед простог модела. Неки принципи функционалних програмских језика су директно наслеђени из λ рачуна,

[1] Алонзо Черч (1903 - 1995), амерички математичар и логичар. Пружио велики допринос развоју математичке логике и теоријских рачунарских наука.

[2] Алан Тјуринг (1912 - 1954), британски информатичар, математичар, и логичар. Најпознатији по свом раду на пољу теорије израчунљивости.

па ова глава служи и као својеврсан увод у функционално програмирање, коме ће пажња бити посвећена у наредној глави.

2.1 Дефиниција

Централни концепт λ рачуна су **изрази.** Изрази се дефинишу рекурзивно на следећи начин:

$$\begin{aligned} < izraz > ::= & \quad < ime > | < funkcija > | < aplikacija > \\ < funkcija > ::= & \ \lambda < ime > . < izraz > \\ < aplikacija > ::= & \quad < izraz >< izraz > \end{aligned}$$

Једина два симбола која се користе у λ рачуну су 'λ ' и '.', уз заграде где је то неопходно. Апликација је асоцијативна са леве стране, па се ако није друкчије назначено заградама низ израза евалуира на овај начин:

$$E1E2E3E4 \equiv E1(E2(E3(E4))))$$

У горњој дефиницији «име» представља ознаку променљиве, и за потребе овог рада ће бити из скупа малих латичних слова.

Функције почињу знаком λ и третирају се као анонимне тј. немају имена, што значи да се пре сваке примене функције налази њена дефиниција . Пример функције:

$$\lambda x.x$$

У овом примеру ознака x након λ означава аргумент функције, а израз након тачке означава тело функције, у овом случају такође x, па је ово функција идентитета. Функције имају два важна својства:

- Функције су **вредности прве класе**, што значи да могу да буду и улаз и излаз за друге функције.
- Све функције су функције **једног аргумента**. Функције више аргумената се представљају у облику **Каријевих функција**[3].

[3] Хаскел Кари (1900 - 1982), амерички математичар и логичар, по коме је ова метода добила име. По њему је такође назван и програмски језик Haskell којим ћемо се касније бавити.

Пример Каријеве функције:

$$\lambda x.(\lambda y.z)$$

Овај пример представља функцију која узима аргумент x, и враћа функцију која узима аргумент y и враћа z. Понављањем поступка илустрованог овим примером се функција са произвољним бројем агумената може представити у облику Каријеве функције. Како оваква нотација може прилично да закомпликује запис сложенијих функција користићемо алтернативни начин записа функције са више аргумената:

$$\lambda xy.z$$

Ова функција је еквивалента претходној, али је запис концизнији.

Аргумент функције може бити било који израз, а сам процес апликације примењује функцију на аргумент. Осврнимо се на један пример:

$$(\lambda x.x)y$$

Горепоменуту функцију идентитета примењујемо на аргумент y. Приметимо да су у овом примеру неопходне заграде, јер без њих израз не би имао исто значење. Апликација се евалуира супституцијом; заменом вредности аргумента у телу функције. У овом примеру се врши супституција $[y/x]$, тј. x добија вредност y:

$$\lambda(x.x)y = [y/x]x = y$$

У λ рачуну сва имена су локална, дакле важе само у дефиницији у којој се налазе. У изразу

$$(\lambda x.x)(\lambda x.x)$$

x из прве и x из друге функције су потпуно независни. Исти израз би се могао записати и овако:

$$(\lambda x.x)(\lambda y.y)$$

Из овог примера видимо да имена аргумената у функцијама немају никакво посебно значење осим да опишу начин на који се функција примењује.

2.2 Везане и слободне промењљиве

Размотримо пример:

$$(\lambda x.x)(\lambda y.yx)$$

Каже се да λ оператор **везује** промењљиву ако се она појављује у телу функције коју дефинише (као нпр. $\exists$ или $\forall$ у логици). У овом примеру, x у телу првог

подизраза је везано за прво λ, а y у телу другог подизраза је везано за друго λ, па кажемо да су промењљиве x и y **везане.** Промењљиву x у телу другог израза не везује ни један λ оператор па кажемо да је x **слободна** промењљива. Напоменули смо да су сва имена локална, па је сасвим валидно да у овом примеру x буде везано у једном, а слободно у другом подизразу.

Дефинишимо сада формално појам везане промењљиве.
За промењљиву x важи:

- x је слободно у x.
- x је слободно у $\lambda y.E$, ако је $x \neq y$ и x је слободно у изразу E.
- x је слободно у изразу $E1E2$ ако је слободно у бар једном од два подизраза.

Са друге стране, x је везано у једном од ова два случаја:

- x је везано у изразу $\lambda y.E$ ако је $x = y$ или ако је везано у изразу E
- x је везано у изразу $E1E2$ ако је везано у бар једном од два подизраза.

Напоменимо да је одређена промењљива везана за λ оператор који јој је најближи, тј у изразу

$$\lambda x.(\lambda x.xy)$$

промењљиву x везује други λ оператор.

2.3 λ-редукција

λ-редукција (λ-конверзија) је процес којим се долази до еквивалентног λ израза коришћењем три редукционе операције:

- α-редукција: мења имена везаних промењљивих
- β-редукција: извршава апликацију функција
- η-редукција: редукује користећи идеју екстензионалности

Све три операције ће бити описане у наредним поглављима.

Доказано је да редослед којим се редукције врше не утиче на резултат, и ова тврдња је позната као **Черч-Розерова теорема**. Последица теореме је да сваки израз има највише једну **нормалну форму** тј. формулацију израза такву да није могуће извршити ни једну β-редукцију.

2.3.1 β-редукција

Главни принцип β-редукције тврди да се свака апликација $\lambda(x.T)A$, где је T тело функције а A аргумент, може упростити супституцијом.

Објашњење супституције *(замена вредности аргумента у телу функције)* дато у 2.2 није потпуно прецизно. Дакле, при процесу апликације функције $\lambda x.E$ на аргумент y, само она појављивања x у изразу E **у којима је** x **слободна** бивају замењена вредношћу аргумента y. Супституција не утиче на променљиве са истим именом које су везане за λ оператор неког подизраза. Применимо функцију

$$\lambda x.(\lambda x.xy)$$

на аргумент z:

$$(\lambda x.(\lambda x.xy))z = \lambda x.(\lambda x.xy)[z/x] = (\lambda x.xy)$$

Пошто је x у подизразу везано за свој λ оператор, а y је слободна променљива, апликација ове функције на z нема ефекта. Да је у дефинциији функције аргумент функције био y, а не x, y би постала везана и била замењена у супституцији.

$$(\lambda y.(\lambda x.xy))z = \lambda y.(\lambda x.xy)[z/y] = (\lambda x.xz)$$

Овај пример може да представља и проблем. Шта би се десило да смо функцију применили на аргумент x?

$$(\lambda y.(\lambda x.xy))x = \lambda y.(\lambda x.xy)[x/y] = (\lambda x.xx)$$

Примећујемо да долази до колизије, јер након супституције слободна променљива y постаје везана. Свака супституција која прави колизије тог типа није валидна. Такви проблеми се могу решити при редукцији процесом α-редукције, који ће бити описан у наредном поглављу.

2.3.2 α-редукција

α-редукција (α-супституција) користи својства напоменута у 2.1, локалност имена и чињеницу да имена немају посебно значење, тако што трансформише израз $\lambda x.E$ у $\lambda y.E'$, где је E' израз E у коме су сва слободна појављивања променљиве x замењена са y. Алфа редукција успешно решава проблем из претходног поглавља:

$$(\lambda y.(\lambda x.xy))x \overset{\alpha}{=} \lambda y.(\lambda t.ty)[x/y] \overset{\beta}{=} (\lambda t.tx)$$

2.3.3 η-редукција

Постоји и трећи тип редукције, тзв. η-редукција. η-редукција користи екстензионалност, која у овом контексту значи да су две функције еквивалентне акко враћају исту вредност за све аргументе. Размотримо функцију:

$$\lambda x.fx$$

где се променљива x не јавља као слободна променљива у f. Важи:

$$(\forall y)\ (\lambda x.fx)y \equiv fy$$

тј. горе наведена функција за сваки унети аргумент враћа исту вредност као функција f, па на њу можемо применити η-редукцију и трансформисати је у f.

Понашање η-редукције можемо схватити као *„наслућивање"* β-редукције. Да смо функцију из претходног примера пропустили да η-редукујемо и заиста је применили на аргумент y, применом β-редукције би ипак дошли до еквиваленције са f:

$$\lambda x.fx[y/x] \stackrel{\beta}{=} f$$

Такође, при формалном доказивању еквиваленције одређених функција које се понашају исто за све аргументе, могућност позивања на принцип η-редукције умногоме олакшава сам доказ.

2.4 Аритметика и логика

Иако до сада нисмо користили ни аритметику ни логику у оквиру λ рачуна, очекивано је да постоји начин на који се представљају природни бројеви и логичке константе, као и да је могуће дефинисати функције за рад са њима. Ово решавају тзв. **Черчови нумерали** и **Черчове логичке вредности**.

Природни бројеви се дефинишу на следећи начин:

$$0 \equiv \lambda fx.x$$
$$1 \equiv \lambda fx.f(x)$$
$$2 \equiv \lambda fx.f(f(x))$$
$$\ldots$$

Број n је, дакле, представљен као $f^n(x)$.

Прва функција коју ћемо увести је функција **следбеник:**

$$S \equiv \lambda wgy.g(wgy)$$

Апликацијом функције S на Черчов нумерал који представља број 1 добијамо:

$$\begin{aligned} S(\lambda wgy.g(wgy))1 &= (\lambda wgy.g(wgy))(\lambda fx.f(x)) \\ &= \lambda gy.g((\lambda fx.f(x))gy) \\ &= \lambda gy.g(g(y)) \\ &= 2 \end{aligned}$$

Ако искористимо Черчов нумерал n као функцију која примењује унету функцију на аргумент n пута, можемо лако увести сабирање. Тачније, сабирање бројева x и y имплементирамо као $S^x(y)$.

$$+ \equiv \lambda xy.xSy$$

На сличан начин, коришћењем својства степеновања $(f^a)^b = f^{ab}$ уводимо множење као композицију Черчових нумерала:

$$* \equiv \lambda xyz.x(yz)$$

Сада ћемо се осврнути на логику. Дефинишемо следеће две функције које ћемо прогласити за константе $\top$ и $\bot$:

$$\top \equiv \lambda xy.x$$

$$\bot \equiv \lambda xy.y$$

Уз овакву дефиницију логичких константи као функција које враћају први, односно други од два унета аргумента, није тешко дефинисати и неке основне логичке функције:

$$\wedge \equiv \lambda xy.xy\bot$$

$$\vee \equiv \lambda xy.x\top y$$

$$\neg \equiv \lambda x.x\bot\top$$

Лако се показује да су овакве дефиниције валидне и конзистенте са истинитосним таблицама датих функција. Такође, постоји начин да се уведу и предикати, што ћемо видети кроз пример функције Z, која враћа $\top$ када је унети аргумент једнак нули, а $\bot$ у супротном.

$$Z \equiv \lambda x.x\bot\neg\bot$$

Ако се поново осврнемо на дефиницију Черчових нумерала јасно је да је $x\bot\neg = \bot^x(\neg)$. Како за сваку функцију важи $0fa \equiv a$ и како знамо да ће наведена функција $\bot^x(\neg)$ да постане функција идентита за $x \geq 1$ следи да ће функција Z да врати $\bot$ за све бројеве веће или једнаке 1, а $\neg\bot = \top$ за 0.

Сада ћемо приказати једну сложенију функцију, функцију **претходник**. Приметимо да је, због природе нумерала, ово теже него тражење следбеника броја.

Очигледно, једини начин да нађемо претходника броја n је да кренемо од нуле , и на неки начин функцијом S(следбеник) дођемо до броја $n-1$. Мање је очигледно како то извести, јер нам је за генерисање функције S^{n-1} већ неопходан број $n-1$, па ћемо се послужити триком: Кренућемо од пара $(0,0)$ и након n итерација помоћне функције $\phi \equiv (a,b) \mapsto (a+1,a)$ доћи до пара $(n,n-1)$ чији је други члан управо број који нам треба. Пар представљамо као $\lambda f.fab$ и јасно је да можемо користити $\bot$ и $\top$ како би издвојили појединачне елементе једног пара. Сада погледајмо поменуту помоћну функцију

$$\phi \equiv (\lambda pf.f(S(p\top))(p\top))$$

која креира нови пар од следбеника првог елемента улазног пара и првог елемента као таквог. Сада, као што смо већ рекли, имамо $\phi^n((0,0)) = (n,n-1)$ што нам омогућује налазак претходника:

$$P \equiv \lambda n.n\phi(\lambda z.z00)\bot$$

где $\bot$ на крају дефиниције функције издваја други члан из пара $(n,n-1)$.

Функције Z и P имају примену у многим сложенијим предикатима. Осврнимо се, на пример, на функцију

$$geq \equiv \lambda xy.Z(xPy)$$

која пореди бројеве x и y тако што број $y-x$, настао апликацијом функције претходник на аргумент y тачно x пута, пореди са нулом. Валидност ове функције осигурава чињеница да је $P(0) = 0$.

Сада када знамо како аритметика и логика функционишу остало је још да размотримо како се у оквиру λ рачуна имплементира рекурзија.

2.5 Рекурзија

Како функција у оквиру λ рачуна нема име, не може да реферише саму себе, тј. да би била у стању да направи рекурзивни позив морала да би да у свом телу садржи своју дефиницију, што није могуће. Овај проблем се решава функцијом Y**-комбинатор** коју дефинишемо на следећи начин:

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Применом Y-комбинатора на неку функцију F добијамо позив функције F са аргументом који представља рекурзивни позив YF, тј. $YF = F(YF)$. Све функције вишег реда K за које је задовољено својство $KF = F(KF)$ називамо **комбинатори фиксне тачке**, и помоћу њих је могуће симулирати рекурзију.[4]

$$\begin{aligned} YF &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F \\ &= (\lambda x.F(xx))(\lambda x.F(xx)) \\ &= F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &= F(YF) \end{aligned}$$

Демонстрирајмо ово на примеру функције R која рачуна суму свих бројева од 0 до унетог броја n. Користићемо се рекурентном везом $R(n) = R(n-1) + n$.

$$R \equiv \lambda rn.Zn0(nS(r(Pn)))$$

Број n је број за који желимо да израчунамо решење.

- Ако је $n = 0$, Zn ће вратити $\top$, а $\top$ по дефиницији Черчових логичких константи враћа први члан пара $(0, nS(r(Pn)))$. Дакле, $R(0) = 0$.
- Ако је $n > 0$, биће евалуиран други члан пара, тј. $nS(r(Pn))$: n пута позивамо функцију следбеник на **рекурзивни позив** са аргументом $n - 1$.

Тај рекурзивни позив симулирамо користећи поменуту функцију Y, тј. својство $YF = F(YF)$. Израчунавање $R(n)$ сада вршимо позивом YRn. Демонстрирајмо цео процес на примеру $n = 2$:

$$YR2 = R(YR)2 = Z20(2S(YR(P2))) = 2S(YR1)$$

[4] Y-комбинатор није једини комбинатор фиксне тачке. Један од великог броја постојећих (могуће је чак доказати да их има бесконачно много) је и тзв. **Тјурингов комбинатор фиксне тачке** θ, чији је творац поменути Алан Тјуринг.

Добијамо да је $YR2 = 2S(YR1)$. На сличан начин би дошли до $YR1 = 1S(YR0)$, а како је $YR0 = R(YR)0 = 0$, имамо:

$$YR2 = 2S(1S(0))) = 2 + 1 + 0 = 3$$

Успешно смо нашли вредност функције, симулирајући рекурзију коришћењем израза YR на месту аргумента r. Y-комбинатор је, као темељ рекурзије, једна од најважнијих функција λ рачуна.

Видели смо како је могуће увести аритметику, логику и рекурзију, како би демонстрирали моћ самог λ рачуна. Наравно, из саме Черч-Тјурингове тезе следи да је могуће дефинисати све функције које су израчунљиве, па је нпр. изводљива имплементација Тјурингове машине користећи λ рачун. Такве вежбе превазилазе обим овог рада, па ћемо се задржати на стварима дефинисаним до сад и у следећој глави прећи на **функционално програмирање**, програмску парадигму инспирисану λ рачуном. Ова глава представља солидан увод у функционално програмирање и илуструје неке његове концепте на сасвим фундаменталном нивоу где су и настали, у оквиру λ рачуна.

Глава 3

Функционално програмирање и програмски језик Haskell

„Programs must be written for people to read, and only incidentally for machines to execute.” - Хал Абелсон

Функционално програмирање (ФП) је програмска парадигма која третира израчунавање као евалуацију математичких функција. Функционални језици спадају у групу **декларативних језика**, језика чији програми не садрже наредбе, нити редослед извршавања. Просто речено, декларативни језици приступају решавању проблема тако што дефинишу израз који описује решење, насупрот императивним језицима чији програм је скуп наредби које се извршавају у одређеном редоследу. У декларативне језике спадају и нпр. **логички језици** (Prolog) који у свом израчунавању користе релације, за разлику од функционалних који проблем решавају коришћењем искључиво израза (дефиниције и апликације функција). Главна одлика функција у ФП је да оне не изазивају **споредне ефекте**, и такве функције зовемо **чисте функције**. У споредне ефекте се убрајају: промена вредности глобалних променљивих, читање и писање података, позивање других функција које имају споредне ефекте... Једина сврха чистих функција је да врате вредност, а њихов резултат зависи само од улаза, а не од тренутка у ком функцију позивамо или стања у меморији. Са друге стране, императивно програмирање такође садржи функције, али не у виду математичких пресликавања већ у виду подрутина, скупа команди које се позивом функције изврше. Такве функције могу да имају споредне ефекте и њихово извршавање често зависи од стања меморије, и мења га. Неке функције се чак и ослањају на своје споредне ефекте, а не на своју повратну вредност, а као пример издвајамо функције `scanf` и `printf` програмског језика C.

Филозофија чистих функција има многе предности, о чему смо већ говорили. Модуларност (независно функционисање делова кода) и концизност кодова утиче на повећану продуктивност програмера уз мање дебаговања, а тачност кодова се лакше доказује. Такође је битно напоменути да се функционални кодови због своје модуларности доста лако паралелизују, што представља велику предност. Све ово утиче на повећану популарност функционалног програмирања, за шта су примери већ изложени у уводу овог рада.

У наставку ове главе ће бити објашњено неколико важних принципа функционалног програмирања. Многи принципи воде порекло директно од математичког темеља ФП-а, λ рачуна, па ће често моћи да се успостави одговарајућа аналогија са принципима објашњеним у претходној глави. Како је неопходно видети те принципе у пракси, ФП ће бити уведен кроз већ поменути језик Haskell, у коме је у највећој мери рађен и имплементацијски део овог рада. Haskell одлично илуструје већину принципа ФП-а, па ће и поред неких идеја специфичних за тај језик свакако бити покривено све што ФП чини посебним и вредним помена.[1] Ипак, детаљно увођење свега што Haskell садржи и неких напреднијих концепата (функтори, монаде...) је ван обима овог рада. Главна препорука за даље изучавање овог језика је књига Мирана Липоваче „Learn You a Haskell for Great Good" [3], а валидан извор су и многе друге књиге које се баве истом темом.

3.1 Прве функције

Као што је већ речено, како у функционалном програмирању не постоје наредбе нити редослед извршавања, основу чине функције. Погледајмо основне примере који илуструју декларисање функција у Haskell-у.

```
dupliraj x = x * 2
saberiDva x y = x + y
pomnozi_dva x y = x * y
pi' = 3.14
vece5 x = 5 + if x>5 then 1 else 0
```

а При декларисању функције, са леве стране знака једнакости налазе се редом аргументи функције, а са десне стране повратна вредност. У овом примеру `dupliraj` је функција која прима један аргумент, `saberiDva/pomnozi_dva` при-

[1] Треба напоменути да се функционални језици деле на две групе у зависности од тога колико су *„чисти"* тј. колико су одани филозофији чистих функција. Haskell је чист функционалан језик, и остатак рада се фокусира управо на такве језике.

мају два, док је `pi'` функција без аргумената или тзв. **дефиниција**. Функција `vece5` садржи **if** конструкцију, која се у Haskell-у понаша као израз, тј. увек има вредност, па је зато неопходно да она увек садржи и then и else део.[2]

Имена функција у Haskell-у почињу малим словом и могу да садрже слова, бројеве, и два специјална карактера: _ и ' . Карактер ' се често користи када декларишемо алтернативну верзију већ постојеће функције, као у малопређашњем примеру. Функција `pi` већ постоји у Haskell-у, па смо на име наше функције додали апостроф.

Размотримо сада пример једног C++ кода:

```
int A;

int saberi(int B)
{ return A + B; }

void sabiraj()
{
        A = 1;
        printf("%d ", saberi(1));
        A = 5;
        printf("%d ", saberi(1));
}
```

Позив функције `void sabiraj()` ће исписати `2 6`, иако је оба пута направљен позив `saberi(1)`. То значи да овај код има споредне ефекте, јер резултат функције зависи од глобалне променљиве A. Споредни ефекти су потпуно валидни у императивним језицима, па имају смисла функције које не враћају вредност (void функције). Са друге стране, споредни ефекти не постоје у ФП, па је неопходно да функција увек врати вредност, и то увек исту вредност. Дакле, никада не може да се деси $f(x) \neq f(x)$, јер резултат сваке функције зависи само од аргумената које јој проследимо. За изразе кажемо да су **референцијално транспарентни** ако свако њихово појављивање у коду можемо да заменимо њиховом вредношћу. И заиста, функције у ФП имају ово својство.

[2]Имајући ово у виду, аналогија овој конструкцији у језику C, као карактеристичном императивном језику би пре била тернарни оператор (uslov?vrednost:alternativa), који је такође израз и увек враћа вредност, него if-else конструкција овог језика.

3.2 Систем типова

Haskell је **строго типизиран** и имплементира **статичку проверу типова**. То значи да постоји аутоматска провера типова тј. да је свакој вредности која се појављује у програму додељен тип већ при компилацији, анализом самог изворног кода. Статичка провера типова, за разлику од динамичке која проверава да ли су типови у програму валидни у току извршавања (и тиме донекле успорава извршавање), омогућава да већина грешака у вези са типовима буде ухваћена при самој компилацији. Ако се вредност коју прослеђујемо функцији не уклапа у тип који функција очекује, програм ће одбити да се покрене како би указао на грешку, што значајно скраћује процес дебаговања. Такође, овакав систем типова ослобађа програмера потребе да сам декларише типове у оквиру кода, јер компајлер сам закључује о њима тј. даје најопштија ограничења за типове која осигуравају да ће програм функционисати. Ово се зове **инференција типова**, а Haskell користи тзв. **Хиндли-Милнеров алгоритам** за типовну инференцију.

Размотримо функцију `saberiDva` из претходног поглавља:

```
saberiDva x y = x + y
```

Приметимо да оваква декларација функције не ограничава `x` и `y` на целобројни тип података, али да ипак постоје неки услови које ове променљиве морају да задовоље, тј. морају бити у класи бројева (јер је оператор + дефинисан не само за целе већ и за нпр. реалне бројеве). Овакве функције, чији типови аргумената нису унапред познати, па се могу применити на аргументе више различитих типова називамо **полиморфне функције**. Поставља се питање како тачно гласи ограничење које алгоритам инференције поставља на аргументе ове функције тако да дозволи све бројевне типове, не само целе бројеве, а да опет довољно ограничи коришћење функције како би осигурао правилно коришћење. У ту сврху се уводе **класе типова**. Свака класа типова представља нешто попут интерфејса за тип: дефинише скуп константи и функција које сваки тип које жели да буде представник те класе мора да имплементира. Дакле, алгоритам инференције решава поменути проблем постављањем ограничења `Num` на променљиве `x` и `y`. То ограничење говори да функција ради са типовима који су из `Num` класе, чији су чланови сви бројевни типови, и да прихвата само такве аргументе. Преглед неких основних типова и класа типова ће бити дат у 3.2.2.

Иако смо рекли да због строге типизираности и статичке провере типова није неопходно декларисати типове аргумената при декларацији функције, у Haskell-у је то могуће тзв. **типовним декларацијама**. Типовне декларације представљају кратак опис класних и типовних ограничења неке функције и пишу се изнад декларације саме функције. Иако нису неопходне, типовне декларације се препоручују јер чине код јаснијим. Такође, ако се ограничења добијена алгоритмом инференције не поклапају са ограничењима која смо задали типовном декларацијом, и која очекујемо да функција задовољава, компајлер ће нас упозорити при компилацији, што опет помаже у отклањању евентуалних грешака. Погледајмо пример функција са типовним декларацијама:

```
saberiInt :: Int -> Int -> Int
saberiInt x y = x + y

saberiDva :: (Num a) => a -> a -> a
saberiDva x y = x + y
```

Типовна декларација функције `saberiInt` је `Int -> Int -> Int`, што се тумачи као *„функција која узима два Int аргумента и враћа Int вредност”*. Тиме смо ограничили ову функцију на рад са строго целобројним вредностима[3]. Полиморфна варијанта ове функције са којом смо се већ сусрели, `saberiDva`, у својој типовној декларацији користи тзв. **типовне променљиве**. Типовне променљиве служе да се вредностима наметну ограничења у виду класа типова. У овом конкретном примеру, `(Num a) => a -> a -> a` значи да је `saberiDva` *„функција која узима два аргумента типа a и враћа вредност типа a*”, где је `a` из класе `Num`, чији су представници сви бројевни типови.

У наредном делу ће бити дат преглед основних типова и типовних класа. Иако типови делују стандардно (постојаће све што постоји и у императивним језицима: целобројни бројеви, реални бројеви, стрингови...) класе типова су новитет и карактеристичне су за језике са оваквим типовним системом. Пре тога, уведимо n**-торке** и **листе**, које се као основни елементи типовне структуре већине програмских језика, појављују и у функционалној парадигми.

[3]Бољи увид у начин на који се тумачи типовна декларација и објашњење чињенице да у самој декларацији не постоји ништа што на прави начин одваја типове аргумената од типа повратне вредности функције биће дати у 3.5.

3.2.1 Листе и n-торке

Листе су врло важна компонента језика Haskell, а може се рећи и целог функционалног програмирања. Како нема глобалних промењљивих нити споредних ефеката, листе су врло корисне као алат у процесу трансформације података, и налазе широку примену. Листе се дефинишу на стандардан начин:

Листа је или празна листа ([]), или се састоји од главе (првог елемента) и репа (осталих елемената), при чему је реп листа.

Листе у Haskell-у су **хомогене,** што значи да једна листа садржи податке само једног типа. Погледајмо најосновнију декларацију листе и неке основне функције за рад са листама:

```
brojevi = [1,2,3]
brojevi' = 0 : brojevi
josBrojeva = [4,5,6]
sviBrojevi = brojevi ++ josBrojeva
broj = brojevi !! 1
```

Када говоримо о листама, кључни оператор је `:`, оператор конструкције који додаје појединачне елементе на почетак листе. Начин на који смо декларисали листу brojevi је само лепши начин да декларишемо листу `1:2:3:[]`. Остали битни оператори су ++, оператор конкатенације две листе, и !! који извлачи елемент на задатом индексу (напоменимо да су листе индексиране од 0). Поред ових функција, како је Haskell језик прилично високог нивоа, и садржи у себи многе већ готове функције, богат је и функцијама за рад са листама попут `head, tail, length, reverse, sum, take, cycle`... Нећемо улазити детаљно у опис сваке од њих, већ ће оне које буду коришћене у наставку бити објашњене на одговарајућем месту.

Комбинација листа и **n-торки** (енг. n-tuple) је основни начин манипулације подацима у Haskell-у. Предност n-торки је чињеница да не морају бити хомогене. Једна n-торка има унапред задат број елемената па су корисне у ситуацијама када знамо са колико података којих типова радимо. Погледајмо пример:

```
ntorka = ("Alonzo", "Church", 1903)
```

Ова n-торка садржи два стринга (име и презиме) и једну целобројну вредност (година рођења).

Пре преласка на преглед типова и класа типова кратко се осврнимо на једну занимљиву синтаксну конструкцију, **сажет запис листе** (енг. list comprehension). Сажет запис листе омогућава ефикасно и концизно генерисање жељене листе. У оквиру ове конструкције дефинишемо **излазну функцију, улазне скупове,** и **предикате**. Ове три компоненте јасно дефинишу листу коју желимо да генеришемо. Погледајмо како сажет запис функционише:

```
parovi = [(x, y) | x <- [1,2,3], y <- [1..10], x+y == 10]
```

Овај запис генерише листу парова (2-торки) у којима су бројеви из задатих скупова такви да им је сума 10. Овде смо се ради лакшег записа листе послужили скраћеним записом, па је листу `[1,2,3,4,5,6,7,8,9,10]` могуће дефинисати као `[1..10]`. У неком од наредних поглавља, кад будемо говорили о лењој евалуацији, ће овај начин записа бити прилично користан.

3.2.2 Преглед типова и класа типова

Сада када знамо по којим принципима функционише систем типова уведимо неке основне типове:

- `Int` - цели бројеви фиксног распона (од -2^{31} до $2^{31}-1$)
- `Integer` - цели бројеви произвољног распона, без меморијског ограничења
- `Bool` - логичке вредности
- `Char` - појединачни знакови
- `String` - низови знакова, синоним за `[Char]`
- `Float/Double` - реални бројеви

Приметимо да листе и n-торке саме по себи нису типови већ постају типови тек када су познати типови вредности које ће садржати. Примери валидних типова би били `[Int]`, `[Bool]`, `(String, String, Int)`...

Основне класе типова уз функције које дефинишу су:

- `Eq (==, /=)` - типови који подржавају тестирање једнакости
- `Ord (<, >, <=, >=)` - типови који имају уређење
- `Show (show)` - типови који имају репрезентацију у виду String-ова, функција show претвара вредност таквог типа у String

- `Read (read)` - супротна класа, функција read узима String и враћа вредност типа који је члан класе Read
- `Num (+, *, ...)` - сви типови који могу да се понашају као бројеви

Ту су још и `Integral` и `Floating` као подскупови класе `Num`, као и класе `Enum` и `Bounded` које дефинишу типове који могу бити нумерисани, као и типове који имају неко ограничење. Иако свака од ових класа има примену, и показује се врло корисном, за сад ћемо се при целокупној причи о типовима и класама типова овде зауставити.

3.3 Лења евалуација

Различити програмски језици примењују различите стратегије при евалуацији израза. Типични представници две главне класе евалуационих стратегија су:

- **стратегија позива по вредности** (енг. call-by-value)
  - стриктна стратегија - аргументи функције се комплетно евалуирају пре примене саме функције
  - ефикасно али некад евалуира непотребне аргументе
  - користе је језици као што су C, Scheme, ML...
- **стратегија позива по имену** (енг. call-by-name)
  - нестриктна стратегија, аргументи функције се не евалуирају осим ако се не користе при евалуацији тела функције
  - иако не евалуира непотребне аргументе, копира и непотребно ре-евалуира оне који су потребни
  - Haskell користи оптимизовану варијанту ове стратегије која проблем ре-евалуације решава мемоизацијом, тзв. **стратегија позива по потреби** (енг. call-by-need)

Стратегија позива по потреби, позната је и као **лења евалуација**. Лења евалуација је кључан део Haskell-а и омогућава рад са бесконачним структурама и оптимизује програм евалуирањем само онда када је то заиста неопходно. Размотримо следећу функцију:

```
f a b = a*2
```

и позив `f 2 (1/0)`. Да користимо нпр. стратегију позива по вредности овај позив не би успешно вратио резултат јер евалуација једног од подизраза `(1/0)` не може да успе. Прво би биле израчунате вредности свих подизраза па тек онда вредност главног израза. Са друге стране, лења евалуација одлаже евалуацију подизраза све док саме вредности нису потребне, па ће бити евалуиран само први аргумент. Покажимо сада један пример рада са бесконачним структурама:

```
g = take 6 [1..]
```

Скраћеним записом листе споменутим при причи о листама функцији `take` прослеђујемо бесконачну листу природних бројева и аргумент `6`. Функција `take n xs` враћа листу првих `n` елемената листе `xs`, па ће резултат функције `g` бити `[1,2,3,4,5,6]`. Управо због лење евалуације ни у једном тренутку није заиста израчуната бесконачна листа, већ само њених првих `6` чланова, јер је толико довољно да комплетна евалуација успе.

3.4 Рекурзија и тражење узорака

Како је итерација сушта супротност филозофије ФП-а, ретко је могућа, а и кад јесте не препоручује се њено коришћење. Уместо тога, главни механизам израчунавања у ФП је **рекурзија.** Као први пример рекурзивне функције стандардно се наводи функција која рачуна n-ти фибоначијев број користећи просту рекурентну везу:

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Уз рекурзију често иду и **узорци и гардови.** Узорци омогућавају дефинисање више од једног тела функције за различите шаблоне, и уклапају улазне аргументе у шаблон, док гардови врше проверу неког својства улазних аргумената и на основу тога одређују шта ће бити враћено, налик на if конструкцију. Нећемо улазити у детаље тих поступака али ће основна синтакса бити приказана кроз пример функције `maximum'` која одређује максималан елемент листе:

```
maximum' [] = error "Prazna lista"
maximum' [x] = x
maximum' (x:xs)
    | x > maxRep = x
    | otherwise = maxRep
    where maxRep = maximum' xs
```

За ову функцију су дефинисана три различита тела. Прво пријављује грешку (функција `error`) ако је унета празна листа, друго ради са листом која има један елемент, а треће разлаже унету листу на главу и реп, и прави рекурзивни позив. Провера да ли је тренутна глава већа од досадашњег максимума се врши помоћу гардова (|), док су помоћне променљиве дефинисане у where делу који се пише на крају сваког тела функције. Провера да ли се аргументи уклапају у шаблон неког тела се дешава редом до првог поклапања, и неопходно је ,,ухватити" све могуће облике улазних података (грешке при овом поступку се не толеришу при компилацији). Ако улазни подаци одговарају шаблону задатом у неком телу функције, проверава се да ли задовољавају предикате задате гардовима (где је otherwise предикат који је увек тачан) и ако не задовољавају ни један од њих, наставља се провера по следећем телу функције. Ове методе су нарочито корисне при раду са листама, како због тога што ефикасно враћају различите вредности за различите шаблоне улазних података, тако и због чињенице да се у сваком телу функције улазна листа може разложити на делове, и ти делови везати за имена.

3.5 Каријеве функције и функције вишег реда

У поглављу 2.1 смо већ говорили о појму **вредности прве класе** и **Каријевим функцијама.** Функционално програмирање преузима те концепте из λ рачуна, па можемо да кажемо да за функцију у ФП важи да је вредност прве класе, тј. да је равноправна са свим другим типовима података. То својство омогућава да функције буду аргументи и повратне вредности других функција. Функције вишег реда (функције које као аргумент узимају друге функције) су веома заступљене у математици (извод, интеграл..), па ће и у самом функционалном програмирању њихово постојање бити од велике користи.

Пре него што објаснимо како функције вишег реда функционишу и зашто су корисне морамо се дотаћи појма **Каријевих функција.** Каријеве функције су такође споменуте у поглављу 2.1 и ФП их комплетно преузима из λ рачуна, тј. у ФП функције више аргумената технички не постоје већ их посматрамо као Каријеве. Ово наговештава типовна декларација `saberiInt :: Int -> Int -> Int` дата у 3.2, из које видимо да не постоји суштинска разлика између начина на који записујемо типове аргумената функције и тип њене повратне вредности. Ако посматрамо ову функцију као Каријеву добијамо `saberiInt :: Int -> (Int -> Int)` тј. ова функција узима аргумент типа `Int` и враћа функцију која слика

`Int` у `Int`. Као пример функција вишег реда навешћемо стандардну функцију `map` и њој сродну функцију `filter`, као битне компоненте начина размишљања заступљеног у ФП.

3.5.1 map и filter

Функција `map` се дефинише на следећи начин:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Дакле, `map` је функција вишег реда која као аргументе прима функцију `f` која слика вредности типа `a` у вредности типа `b` и листу елемената типа `a`. Применом функције `f` на појединачне елементе улазне листе добија се листа чији су елементи типа `b`. Та листа бива враћена као резултат примене функције `map`. Дакле, као што јој и само име каже, ова функција *„мапира"* функцију на листу. Друга функција коју смо споменули, `filter`, ради на сличан начин: као први аргумент прима предикат (функцију која слика вредност типа `a` у `True` или `False`) и враћа листу само оних елемената који задовољавају предикат. Напишимо функцију која из листе избацује све елементе мање или једнаке 3:

```
p x = if x>3 then True else False
samoVeci xs = filter p xs
```

У овом примеру смо као предикат дефинисали помоћну функцију `p`. Ова функција ће вратити `True` за сваку вредност промењиве `x` већу од три, а False у супротном. Како је функција `p` дефинисана само да би била искоришћена у главној функцији, непотребно смо заузели `p` као име функције. У дужим програмима то може представљати проблем, па се овај пример може написати друкчије коришћењем **анонимних функција** (ламбди). Ламбде, очигледно инспирисане λ рачуном, представљају функције које немају име и дефинишу се на истом месту где се и (једино) користе, чиме се ефикасно чува простор за имена других функција. Применом ламбди на наш пример добијамо еквивалентну функцију:

```
samoVeci xs = filter (\x -> if x>3 then True else False) xs
```

Овде је сам предикат дефинисан као анонимна функција, у оквиру саме главне функције. Знак \, одабран јер подсећа на слово λ означава почетак анонимне функције која у овом случају слика `x` у логичку вредност. Постоји још концизнији начин да се запише функција `samoVeci` али пре тога морамо увести појам

парцијалне примене, који каже да је резултат прослеђивања m аргумената функцији која захтева n аргумената (где је $n > m$) функција која захтева још $n - m$ аргумената. Ово је логично ако посматрамо функције као Каријеве. Још једном ћемо као пример искористити функцију `saberiInt`:

```
saberiInt :: Int -> Int -> Int
saberiInt x y = x + y
```

Рекли смо да се њена типовна декларација посматра као `Int -> (Int -> Int)` па када бисмо јој проследили само један, уместо два аргумента, добили бисмо функцију чији је тип `Int -> Int`, тј. функцију која *„чека"* још један аргумент. Ово нам омогућава да нпр. дефинишемо функцију која унети број сабира са бројем 3:

```
saberiTri :: Int -> Int
saberiTri x = saberiInt 3 x
```

Приметимо да `x` у декларацији функције није неопходно и запишимо функцију у облику:

```
saberiTri = saberiInt 3
```

Дакле, функција `saberiTri` представља парцијалну примену функције `saberiInt` на аргумент `3`. Парцијална апликација може да се примени и на **инфиксне** функције, функције чији се аргументи при апликацији налаже са различитих страна имена функције (xfy а не fxy). Таква функција је нпр. функција `<` па коначно долазимо до најконцизнијег, и са становишта ФП и његове филозофије, најбољег решења за горепоменути проблем:

```
samoVeci xs = filter (>3) xs
```

`(>3)` је, дакле, функција настала парцијалном применом функције `>` на аргумент `3`. Ова функција пореди унети број са `3`, а то је управо оно што нама треба као предикат.

У комбинацији са функцијама `map` и `filter` често се користе и стандардне функције `takeWhile` и `dropWhile`. Функција `takeWhile` узима предикат и листу и задржава елементе листе докле год је предикат задовољен. Листа се прекида на првом месту где предикат не врати `True`. Сродна функција `dropWhile` одбацује елементе док је предикат задовољен:

```
tw3 = takeWhile (>3) [5,4,1,2,3,6]
dw3 = dropWhile (>3) [5,4,1,2,3,6]
```

У овом примеру, `tw3 = [5,4]`, а `dw3 = [1,2,3,6]`.

На самом крају, погледајмо један сложенији пример који користи `map, filter` и `takeWhile` функције, функцију која рачуна суму свих непарних квадрата мањих од 1000:

```
suma = sum ( takeWhile (<1000) (filter odd (map (^2) [1..])))
```

Дакле, полазимо од бесконачне листе природних бројева. На њу мапирамо функцију која враћа квадрат броја, затим из добијеног скупа узимамо само непарне вредности и одбацујемо елементе почевши од првог који није мањи од хиљаду. На крају на добијени резултат применимо функцију која враћа суму елемената листе, и добили смо резултат, а код је кратак и јасан. Овакво конципирање решења представља темељ начина размишљања које карактерише функционално програмирање. Крећемо од широких скупова решења, па мапирањем и филтрирањем одбацујемо делове који нам нису потребни, и на крају добијамо тражени резултат.

Наравно, поред свега уведеног у овој глави постоје још многи концепти који заиста превазилазе обим овог рада, и не могу бити уведени на овом месту. Haskell је, као што је већ речено, висок програмски језик, и пописивање свих функција и идеја би претворило овај рад у уџбеник за Haskell, што није циљ. У уводном делу ове главе је већ поменута литература која такве ствари садржи, али само као илустрација ће бити приказана још ,,лепша" верзија кода којим смо се малопре бавили. Овај код користи операторе композиције и решава се непотребних заграда:

```
suma' = sum . takeWhile (<1000) . filter odd . map (^2) $ [1..]
```

Ова функција, овако написана, заиста на прави начин илуструје блискост кода програмеру која карактерише ФП. Код је разумљив, интуитиван, и прати начин размишљања који стоји у позадини.

Овим поглављем се завршава трећа глава. Након приче о λ рачуну у оквиру друге главе, и увода у функционално програмирање и језик Haskell прелазимо на имплементациони део овог рада. У четвртој глави ће уз делове кода бити објашњен процес израде компајлера, од дефиниције основних појмова који су део неопходне материје до финалног програма.

Глава 4

Имплементација компајлера у Haskell-у

„Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law." - Даглас Хофштадер

Пођимо од дефиниције компајлера. **Компајлер** је компјутерски програм који преводи изворни код из једног програмског језика који називамо **изворним језиком** (енг. source language) у други програмски језик који називамо **циљним језиком** (енг. target language). Најчешћи циљ компилације је креирање извршног фајла, фајла који може да се директно покрене и изврши на одређеној машини. Због тога је изворни језик у највећем броју случајева програмски језик вишег нивоа, језик са високим нивоом апстракције, а циљни језик је ниског нивоа, често и сам assembly. Програмски језици вишег нивоа олакшавају процес кодирања и читљиви су за људе, али не и за машину на којој би требало да се изврше, па је сам компајлер неопходан како би се код неког високог језика извршио.
Три главна процеса која компајлер врши су, редом:

- **Лексичка анализа**, која изворни код изворног језика претвара у низ значајних стрингова (тзв. токена);
- **Синтаксна анализа (парсирање)**, која токене у складу са граматиком којом је дефинисан улазни језик претвара у тзв. **апстрактно синтаксно дрво** (AST);
- **Генерисање кода**, где се на основу AST-а генерише код за циљни језик, пратећи његову синтаксу.

Одговарајуће компоненте компајлера које се баве овим процесима зовемо **лексером, парсером** и **генератором кода**. Наравно, компликованији језици за-

хтевају компликованији процес компилације, па је прича о оптимизацијским компајлерима и различитим техникама које се користе при компилацији јако битна, али и дуга, па се тиме овде нећемо бавити.

Тема имплементационог дела овог рада је израда компајлера у језику Haskell. Компајлери представљају одличан пример како функционални програмски језици (нарочито чисти попут Haskell-а) могу бити од користи. Сама имплементација на прави начин допуњава речено у претходне две главе и демонстрира филозофију о којој је причано. Конструкције и идеје које постоје у ФП, попут тражења узорака или екстензивног коришћења рекурзије, налазе одличну примену у изради самог компајлера, па се показује да то раде боље и од императивних језика. Поред тога, као што је већ поменуто у претходној глави, лакше је доказати тачност функционалних програма, а могућност доказа тачности програма је нарочито корисна код израде компајлера.

4.1 Структура пројекта

Пре него што уђемо у детаљну причу о компонентама самог пројекта и прикажемо релевантне делове кода, обратимо пажњу на целокупну структуру самог пројекта. Као улазни језик, пример високог језика, дефинисан је језик PV, који се може схватити као упрошћена верзија познатих императивних језика. Са друге стране, низак ниво представља **виртуелна стек машина** (VSM). Стек машина је машина која користи стек уместо регистара при евалуацији израза. Инструкције за VSM прате форму обрнуте пољске нотације (RPN).

У оквиру пројекта успешно су имплементирани лексер, парсер и генератор кода, као и сама стек машина. Све компоненте компајлера су писане у Haskell-у, пратећи суштину имплементационог дела. Имплементација саме стек машине није кључни део овог пројекта и не служи да илуструје моћ функционалног програмирања па је она писана у језику C++. Обраћења је пажња и на тестирање, па су у оквиру пројекта приложена и два теста, а при процесу компилације се приказују сви међурезултати добијени појединачним процесима у оквиру целе компилације. Како приказивање AST-а у текстуалном облику није најпријатније, коришћењем open source пакета GraphViz [16] генерисана је визуализација самог дрвета.

Како се формат који GraphViz прима разликује од формата добијеног парсирањем, било је неопходно написати кратак програм који врши конверзију. На самом крају, написане су и две batch скрипте које компајлирају све фајлове и покрећу их.

Главни директоријум пројекта садржи:

- директоријум **src** - садржи све изворне кодове;
  - `PV.hs` - представљање синтаксе улазног језика неопходно за даљи процес компилације;
  - `lexer.hs` - вршење лексичке анализе програма;
  - `parser.hs` - синтаксна анализа програма (парсирање), генерисање AST;
  - `codegen.hs` - генерисање кода за VSM;
  - `compiler.hs` - сам компајлер, повезује три претходно набројане компоненте;
  - `VSM.cpp` - имплементација VSM-а;
  - `ast_to_dot.cpp` - конверзија AST-а из формата који парсер даје као излаз у dot формат који GraphViz користи при генерисању графичке репрезентације самог стабла.
- директоријум **in** - садржи улазни фајл, програм у језику PV (program.pv);
- директоријум **out** - садржи све излазне фајлове настале процесом компилације;
  - `tokens.txt` - низ токена насталих лексичком анализом;
  - `ast.txt` - текстуална репрезентација AST-а коју генерише парсер;
  - `ast.dot` - текстуална репрезентација AST-а у формату погодном за визуализацију;
  - `ast.png` - визуелна репрезентација AST-а;
  - `program.vsm` - резултат компилације, скуп команди за VSM;
  - `vsm_out.txt` - резултат извршења програма на VSM-у, испис стања меморије на крају извршења.
- директоријум **tests** - садржи улаз и излаз за два поменута теста;
- директоријум **int** - садржи међурезултате компилације;

- скрипту `compile.bat` - која компајлира све сорс кодове и креира фајлове `compiler.exe` и `VSM.exe`;
- скрипту `run.bat` - која покреће све кодове.

У наставку ћемо говорити о појединачним компонентама целог процеса и приказати неке делове кода. Цео процес ћемо пратити кроз пример који је у директоријуму `tests` означен као `test0`.

4.2 Улазни језик PV

У овој секцији кратко представљамо синтаксу језика PV. Као што је већ речено, овај језик је осмишљен као упрошћење типичног императивног језика и служи као језик високог нивоа у овом пројекту. Његова синтакса се описује у виду контекстно слободне (енг. context-free) граматике (начин на који смо увели и ламбда рачун). Поново ћемо искористити ту нотацију да би увели наш језик, а у неком од наредних поглавља, кад термини из теорије формалних језика буду неопходни, ће потребне дефиницији бити изложене. Језик PV садржи:

- `x, y ∈ Var` (променљиве, алфанумерички стрингови који почињу словом)
- `n ∈ Num` (бројеви, целобројне вредности)
- `opa ∈ OPa` (аритметичке операције)
  `opa ::= + | *`
- `opb ∈ OPb` (логичке операције)
  `opb ::= and | or`
- `opr ∈ OPr` (релације)
  `opr ::= > | < | =`
- `a ∈ Aexp` (аритметички изрази)
  `a ::= n | x | a opa a | (a)`
- `b ∈ Bexp` (логички изрази)
  `b ::= true | false | !b | a opr a | b opb b | (b)`
- `S ∈ Stmt` (наредбе)
  `S ::= skip | x := a | S;S | if b then S else S | while b do S`

Семантика овог језика не захтева додатно објашњење јер су све конструкције познате. Само треба приметити да су наредбе одвојене знаком ;, и да после последње наредбе тај знак није потребан, нити дозвољен. Сви оператори су асоцијативни са леве стране, а приоритет операција/релација је следећи:

```
(*) > (+) > (=, >, <) > (and) > (or)
```

Такође, PV подржава и коментаре у једној или више линија. Коментари се ограничавају карактером /.

Типови података неопходни за процес компилације дефинисани су у фајлу `PV.hs`. Цело синтаксно стабло ће бити садржано у једној променљивој типа `Program`. Сва остала правила која налаже горенаведени опис језика су директно имплементирана, а овде наводимо само исечак кода, као и пример програма језика PV, из теста `test0`, који смо већ спомињали. Погледајмо део кода који се бави наредбама:

```
data Stmt = IfElse Bexp1 StmtBlock StmtBlock
          | WhileDo Bexp1 StmtBlock
          | Skip
          | Ass Id_String Aexp1
           deriving (Show)
```

`Stmt` представља наредбу, `StmtBlock` је блок наредби, а `Ass` је наредба доделе. Значење ознака `Bexp1` и `Aexp1` ће бити објашњено у поглављу које говори о парсирању. За сада је важно знати да оне означавају логичке и аритметичке изразе. Видимо да овај код веома личи на саму декларацију синтаксе дату изнад, из чега се већ може видети предност језика одабраног за имплементацију. Погледајмо програм дат примером `test0`:

```
b := 1;
if (b > 3)
then { a := 0 }
else { a := b + 1 }
```

Иако је други тест пример експресивнији, и демонстрира већи спектар наредби овог језика, у наставку рада ћемо се држати овог примера јер је ипак довољан да на основном нивоу илуструје све компоненте пројекта. Такође, приказивање већег тест примера би било незграпно и заузело превише простора. Из истог разлога, увек ће уместо целих кодова бити давани исечци који илуструју идеју иза тог дела имплементације.

4.3 Лексичка анализа

Први корак у процесу компилације је, као што је претходно поменуто, **лексичка анализа**. Пре него што објаснимо сам процес и његову имплементацију у оквиру овог пројекта морамо дефинисати пар појмова у вези са **регуларним изразима** и **коначним аутоматима**.

4.3.1 Регуларни изрази

Алфабет је коначан скуп Σ, чије елементе зовемо **симболима**. На пример, скуп малих слова енглеске абецеде $\Sigma = \{a, b, c, \ldots, z\}$ је валидан пример алфабета док скуп природних бројева $\mathbb{N} = \{1, 2, 3, \ldots\}$ не може бити алфабет, јер није коначан.

Стринг (ненулте) дужине n над алфабетом Σ је уређена n-торка елемената из Σ без знакова интерпункције. За $n = 0$ постоји јединствени стринг, тзв **празан стринг** који се означава са ε. Са Σ^* означавамо скуп свих стрингова над скупом Σ.

Регуларни изрази се дефинишу на следећи начин:

- $\emptyset$ је регуларан израз;
- ε је регуларан израз;
- Сваки симбол a из скупа Σ је регуларан израз.

Сви регуларни изрази се граде индуктивно, коначном применом следећих правила:

- **Конкатенација** - уколико су R и S регуларни израз тада је и RS регуларни израз;
- **Унија** - уколико су R и S регуларни израз тада је и $(R|S)$ регуларни израз;
- **Клинијева звезда** - уколико је R регуларни израз тада је и R^* регуларни израз.

Напоменимо још да Клинијева звезда има већу предност од уније.

Регуларни изрази представљају језик за представљање шаблона стрингова. Погледајмо правила за упаривање стрингова из Σ^* са регуларним изразима:

- ниједан стринг не одговара $\emptyset$;
- u одговара ε уколико $u = \varepsilon$;
- u одговара $a \in \Sigma$ уколико $u = a$;
- u одговара $R|S$ уколико u одговара R или u одговара S;
- u одговара RS уколико се може изразити као конкатенација два стринга vw где v одговара R и w одговара S;
- u одговара R^* уколико је $u = \varepsilon$ или се u може изразити као конкатенација два или више стрингова од којих сваки одговара R.

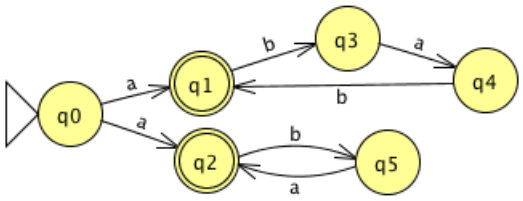
Користећи ова правила дефинишемо **регуларни језик** који је одређен регуларним изразом R над Σ:

$$L(r) \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid u \text{ одговара } R\}$$

Када имамо регуларни израз којим описујемо језик, користећи модел израчунавања који зовемо **детерминистичким коначним аутоматом** можемо у $O(n)$[1] одредити да ли унети стринг дужине n одговара том регуларном изразу, тј. да ли припада језику који он описује. Овај поступак представља основу лексичке анализе, и како би нам био у потпуности јасан, уведимо **коначне аутомате**.

4.3.2 Коначни аутомати

Коначни аутомат је математички модел израчунавања и састоји се од: **скупа стања, улазних симбола, прелаза, почетног стања** и **прихватајућих стања**. Објаснимо ово кроз пример једног коначног аутомата:



Слика 1: Пример коначног аутомата
(`http://www.cs.wcupa.edu/rkline/assets/img/FCS/nfa2.png`)

[1] Ово не укључује сложеност саме изградње аутомата која може бити експоненцијална у величини алфабета.

За овај коначни аутомат важи:

- Стања: $q_0, q_1, q_2, q_3, q_4, q_5$
- Улазни симболи: a, b
- Прелази: $q_0 \xrightarrow{a} q_1$, $q_0 \xrightarrow{a} q_2$, $q_1 \xrightarrow{b} q_3$ ...
- Почетно стање: q_0
- Прихватајућа стања: q_1, q_2

За стринг $u = a_1 a_2 \dots a_n$ кажемо да га **коначни аутомат прихвата** ако се састоји од алфабета улазних симбола тог аутомата, и постоје нека (не обавезно различита) стања $q_0, q_1, q_2, \dots, q_n$, таква да је q_0 почетно стање, q_n прихватајуће стање, и постоје прелази облика: $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_n} q_n$.

За аутомат из претходног примера кажемо да је **недетерминистички** јер из једног стања постоји више прелаза са истим симболом. Аутомати за које то није случај зовемо **детерминистичким**, и за њих важи да је прелаз одређен само тренутним стањем и симболом. Као што смо наговестили, детерминистички аутомати представљају темељ лексичке анализе, и сада можемо прећи на дефиницију лексера.

Лексер је компонента компајлера која претвара улазни програм у низ **токена**, стрингова који дају значење деловима кода. За сваки токен се дефинише регуларни израз, и сваки токен има приоритет. Након што улазни програм раздвојимо на делове на основу сепаратора (размак, нови ред, ...) и у истом пролазу уклонимо коментаре, на основу поменутих проритета тражимо ком токену, тј. ком регуларном изразу сваки од тих делова припада. Као што смо рекли у 4.3.1, то радимо користећи детерминистичке коначне аутомате, тако што сваки регуларни израз везан за токен придружујемо једном аутомату. Када се, на основу приротета, пронађе први регуларни израз ком тренутни стринг одговара, пронашли смо нови токен. Погледајмо сада имплементацију лексера у оквиру овог пројекта.

4.3.3 Имплементација лексера

У оквиру овог пројекта лексер је имплементиран у скрипти `lexer.hs` чија главна функција `lexx` узима стринг који представља програм у језику PV и враћа низ структуре `Token`. У поменутом фајлу `PV.hs` се налази опис те структуре:

```
data Token = NUM Int -- 1, 2, -3, ...
           | ID Id_String -- a, b, cC1, ...
           | BOOL Bool -- True/False
           | SEMICOLON  -- ;
           | PAREN_LEFT -- (
           | PAREN_RIGHT -- )
           | BRKT_LEFT -- {
           | BRKT_RIGHT -- }
           | KWRD_AND -- and
           | KWRD_OR -- or
           | KWRD_IF -- if
           | KWRD_THEN -- then
           | KWRD_ELSE -- else
           | KWRD_WHILE -- while
           | KWRD_DO -- do
           | KWRD_SKIP -- skip
           | OP_NOT -- !
           | OP_ASS -- :=
           | OP_PLUS -- +
           | OP_MUL -- *
           | OP_ORD Ordering -- >, <, =
           | EOF -- zavrsni token
           deriving (Eq, Show)
```

Ови токени комплетно одговарају синтакси језика и њихово значење је објашњено у коментарима. Поменута фунцкија `lexx` позива функцију `nextToken` након што уклони сепараторе и коментаре, и прослеђује јој стринг који почиње делом који треба претворити у токен. Функција `nextToken` механизмом поменутим у 4.3.2 враћа токен и остатак стринга који треба поново да прође кроз одстрањивање сепаратора и коментара. Погледајмо функцију `nextToken`:

```
nextToken :: String -> (Token, String)
nextToken s @ (x:xs)
        | isAlpha x = readKwrdIdBool s -- 1. pocinju slovom
        | isDigit x = readNum s '+'    -- 2. pocinju brojem
        | (x == ':') = readAss s       -- 3. operacija dodele (2 znaka)
        | (x == '-') = readNum s '-'   -- 4. negativni brojevi
        | otherwise = (readSign x, xs) -- 5. svi preostali tokeni (1 znak)
```

Из ове функције се виде приоритети токена о којима смо говорили. Свака од појединачних функција коју `nextToken` позива се бави једним типом токена, а као пример дајемо функцију `readNum` која ради са токенима који представљају бројеве:

```
readNum  :: String -> Char -> (Token, String)
readNum s '+' = (NUM (read number), rest)
    where (number, rest) = span isDigit s
readNum (x:xs) '-' = (NUM (negate $ read number), rest)
    where (number, rest) = span isDigit xs
```

Након што цео улазни програм буде преведен у токене додаје се завршни токен `EOF` и лексер завршава са радом. За пример `test0` резултат је следећи (`tokens.txt`):

```
[ID "b",OP_ASS,NUM 1,SEMICOLON,KWRD_IF,PAREN_LEFT,ID "b",OP_ORD GT,
NUM 3,PAREN_RIGHT,KWRD_THEN,BRKT_LEFT,ID "a",OP_ASS,NUM 0,BRKT_RIGHT,
KWRD_ELSE,BRKT_LEFT,ID "a",OP_ASS,ID "b",OP_PLUS,NUM 1,BRKT_RIGHT,EOF]
```

Уз овакву листу токена сада прелазимо на следећи део компилације, **синтаксну анализу**.

4.4 Синтаксна анализа

Синтаксна анализа (парсирање) представља процес претварања токена који су резултат лексичке анализе у структуру коју смо већ поменули у уводу ове главе, **апстрактно синтаксно дрво** (AST). Парсирање ради са контекстно слободном граматиком налик оној која је дата у 4.2 и помоћу типа аутомата који до сада нисмо спомињали, **недетерминистичког pushdown аутомата**, генерише резултат. Овај аутомат, дакле, није детерминистички, и користи помоћни **стек**, па прелаз сада зависи од тренутног стања, симбола и елемента на врху стека. Такође, при сваком прелазу је могуће променити стање врха стека (уз то да не можемо додатно празнити стек у тренутку када је већ празан).

Као што је наговештено у 4.2, сада ћемо увести неке појмове формалне теорије језика који су нам неопходни за даљи рад: **Формални језик** представља скуп стрингова симбола (алфабет) за које важе нека правила. Та правила задаје **формална граматика** у виду нечега што називамо **продукционим правилима**. Продукционо правило је правило трансформације стрингова, које у овом контексту трансформише симболе алфабета једне у друге. Симболе нижег нивоа који не подлежу трансформацијама зовемо **терминалним симболима**, док су симболи вишег нивоа који се морају трансформисати продукционим правилима тзв. **нетерминални симболи**. Дефиниције ламбда рачуна и језика PV дате у претходном делу овог рада су се састојале од низа продукционих правила за формалне језике, и биле су, као што је већ речено, део класе **контекстно слободних**

граматика. Најкраће речено, контекстно слободне граматике су граматике у којима лева страна сваког правила продукције садржи само један нетерминални симбол. У овом раду се бавимо граматикама само из те класе.

Сам процес парсирања није лак, и компликованији језици користе компликованије методе синтаксне анализе. Наш језик, PV, је довољно прост да уз мале модификације може да буде успешно парсиран методом **рекурзивног спуста**. Рекурзивни спуст се темељи на међусобно рекурзивним процедурама, где обично свака од њих имплементира једно правило продукције из граматике језика. Синтаксно стабло добијено парсирањем на тај начин подсећа на граматику коју парсер препознаје. Одлика ове методе је да не захтева бектрек, тј. могуће је парсирање урадити у једном пролазу, ако граматика језика задовољава одређена својства, тј. ако припада класи $LL(k)$, класи језика које је могуће парсирати ако у сваком тренутку имамо приступ тачно наредних k токена. Језик PV, на начин на који је дефинисан у 4.2 не припада овој класи па морамо направити одређене модификације.

Главни разлог због ког наш језик тренутно не задовољава услове за рекурзивни спуст је присуство **леве рекурзије**. За формалну граматику кажемо да је лево рекурзивна ако се најлевљи симбол у неком од правила продукције нетерминала r поново (после једне или више примена правила продукције) слика у r. Јасно је да кад би простим рекурзивним спустом покушали парсирати граматику која је лево рекурзивна, ушли бисмо у бесконачан циклус. Леву рекурзију језика PV илуструје продукционо правило за аритметичке изразе:
`a ::= n | x | a opa a | (a)`
Видимо да је продукција `a` $\rightarrow$ `a opa a` лево рекурзивна. Овај проблем регулишемо модификацијом сва три спорна правила продукције која изазивају леву рекурзију:

```
a ::= n | x | a opa a | (a)
b ::= true | false | !b | a opr a | b opb b | (b)
S ::= skip | x := a | S;S | if b then S else S | while b do S
```

Правила за аритметичке изразе, логичке изразе и наредбе трансформишемо поделом сваке од ових класа на више нивоа. Хијерархија нивоа је у складу са редоследом рачунских операција.

Аритметички изрази:

```
a3 ::= (a1) | n | x
a2 ::= a3 * a2 | a3
a1 ::= a2 + a1 | a2
```

Логички изрази:

```
b3 ::= true | false | (b1) | !b1 | a1 opr a1
b2 ::= b3 and b2 | b3
b1 ::= b2 or b1 | b2
```

Наредбе:

```
s2 ::= if b1 then s1 else s1 | while b1 do s1 | skip | x := a1
s1 ::= s2;s1 | s2
```

Како после ове модификације граматика не садржи леву рекурзију она постаје граматика из класе $LL(1)$, тј. могуће је парсирати је рекурзивним спустом без бектрека ако у сваком тренутку имамо приступ само првом наредном токену. Иако се у овој имплементацији лексер и парсер позивају потпуно независно, ово својство омогућава да они раде заједно, тако што ће парсер по потреби од лексера затражити парсирање следећег токена.

4.4.1 Имплементација парсера

У оквиру овог пројекта парсер је имплементиран у виду скрипте `parser.hs` чија главна функција `parse` узима низ токена настао радом лексера и враћа променљиву типа `Program` која представља синтаксно дрво програма. Имплементиран је рекурзивни спуст и за сваки симбол граматике постоји функција кога га парсира, и која је рекурзивно увезана са осталим функцијама. Свака функција враћа резултат парсирања и низ преосталих, непарсираних токена. Као пример погледајмо функцију `parseStmtBlock` која парсира блок наредби.

```
parseStmtBlock :: [Token] -> (StmtBlock, [Token])
parseStmtBlock ts
        | (head rest /= SEMICOLON) = (SingleStmt stmt, rest)
        | otherwise = let (block, blockRest) = parseStmtBlock . tail $ rest
                      in (StmtBlock stmt block, blockRest)
        where (stmt, rest) = parseStmt ts
```

Видимо да ова функција рекурзивно позива саму себе и успут позива функцију `parseStmt` чије резултате комбинује у променљиву типа `stmtBlock` . Погледајмо

сад и саму функцију parseStmt која врши парсирање једне наредбе:

```
parseStmt :: [Token] -> (Stmt, [Token])

parseStmt (KWRD_SKIP : ts) = (Skip, ts)

parseStmt (KWRD_IF : ts) = (IfElse b1 stmtThen stmtElse, rest)
              where (b1, restB) = parseB1 ts
                    (stmtThen, restThen) = parseStmtBlock (drop 2 restB)
                    (stmtElse, restElse) = parseStmtBlock (drop 3 restThen)
                    rest = tail restElse

parseStmt (KWRD_WHILE : ts) = (WhileDo b1 stmtDo, rest)
              where (b1, restB) = parseB1 ts
                    (stmtDo, restDo) = parseStmtBlock (drop 2 restB)
                    rest = tail restDo

parseStmt (ID name : OP_ASS : ts) = (Ass name a1, rest)
              where (a1, rest) = parseA1 ts

parseStmt ts = error "Parser error"
```

Ова функција узима низ токена, на основу првог токена позивима помоћних функција парсира све токене потребне да наредба буде комплетна, и враћа свој резултат (парсирану наредбу) уз низ токена који су остали неискоришћени. На сличан начин раде и остале функције за парсирање аритметичких и логичких израза.

Присетимо се, још једном, нашег тест примера test0 и погледајмо шта је резултат парсирања горенаведеног низа токена (ast.txt):

```
StmtBlock (Ass "b" (Aexp1' (Aexp2' (Number 1))))(SingleStmt (IfElse (Bexp1'
(Bexp2' (ParensB(Bexp1' (Bexp2' (Relation (Aexp1' (Aexp2' (Id "b"))) GT
(Aexp1' (Aexp2' (Number 3)))))))))(SingleStmt (Ass "a" (Aexp1' (Aexp2'
(Number 0))))))(SingleStmt (Ass "a" (Add (Aexp2' (Id "b")) (Aexp1' (Aexp2'
(Number 1)))))))
```

Како овакав облик AST-а није леп призор, покретањем помоћног програма ast_to_dot конвертујемо овај облик у dot формат који је погодан за визуализацију помоћу пакета GraphViz. Код самог програма који врши конверзију је сувишан у овом раду, али прикажимо део AST-а у формату који се добија као резултат (ast.dot):

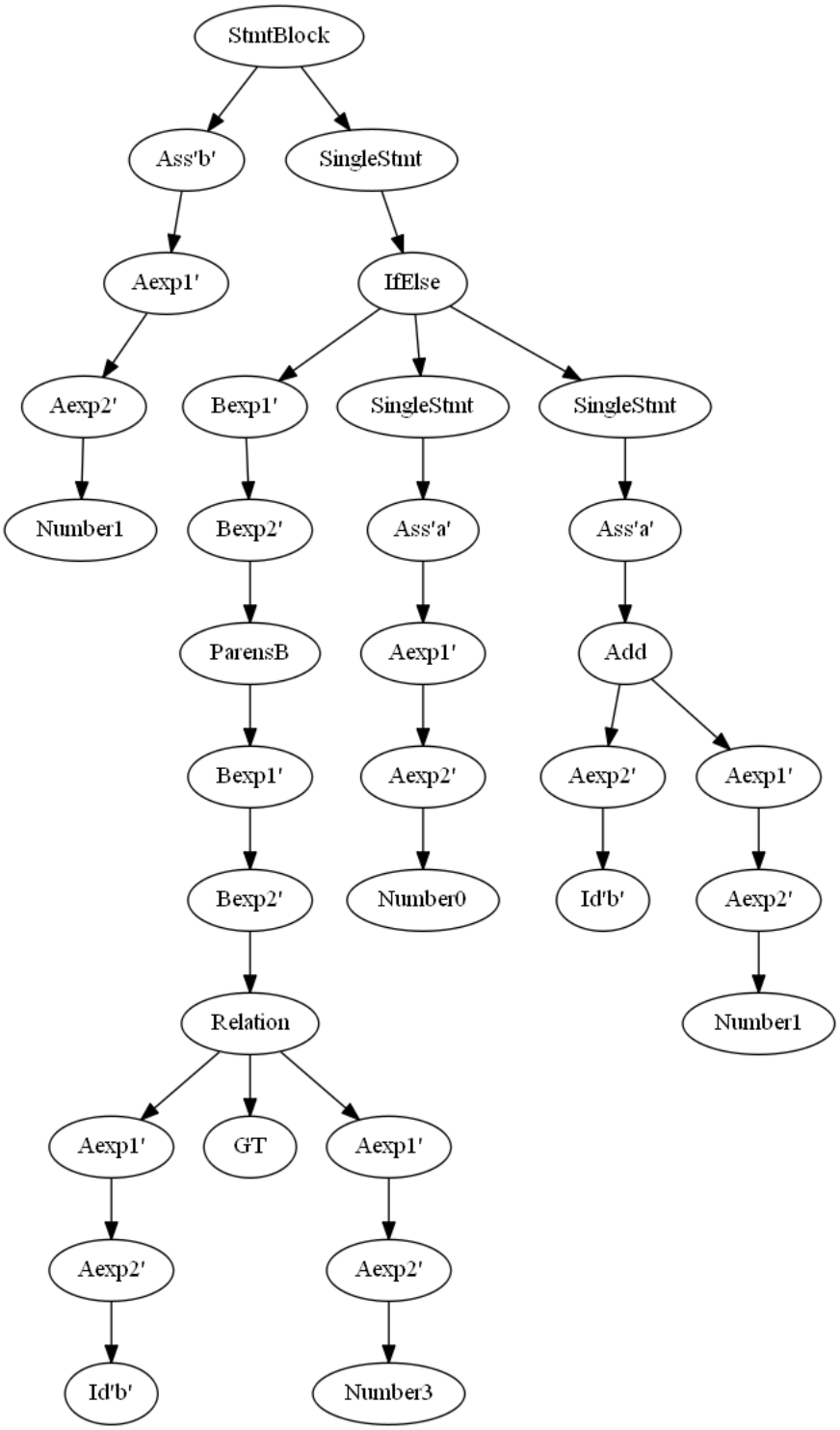
```
digraph{
0[label="StmtBlock"];
1[label="Ass'b'"];
2[label="Aexp1'"];
. . .
27 -> 30
30 -> 31
31 -> 32
}
```

Овај формат, сада погодан за визуализацију, прослеђујемо GraphViz-у, и добијамо жељену визуализацију дату на следећој страни (`ast.png`).

Сада, када је процес парсирања завршен, остало је још само да се на основу AST-а генерише циљни код. Пре тога, осврнимо се на саму структуру стек машине, као циљне машине, и код њене имплементације.

4.5 Виртуелна стек машина

Као што је већ речено, језик ниског нивоа као циљни језик компилације је у оквиру овог рада у облику инструкција за виртуелну стек машину. По дефиницији из 4.1, стек машина је машина која користи стек при евалуацији израза, а инструкције за њу прате форму обрнуте пољске нотације. Сама машина је имплементирана у језику C++ у коду `VSM.cpp`. Циљ те имплементације је да резултат компилације, скуп инструкција, изврши и исписом стања на крају извршавања потврди тачност компилације. За почетак, погледајмо синтаксу команди за VSM. Присутно је 16 инструкција, и свака инструкција има свој операциони код, као када је реч о процесорским инструкцијама. Операциони кодови заузимају по један бајт. Погледајмо део кода дат на почетку кода за VSM који представља преглед команди у виду дефиниције макроа за операционе кодове:



Слика 2: Визуализација синтаксног стабла за `test0`

```
// notacija:
// n : celobrojna promenljiva
// s : string ime promenljive
// S1 : vrednost na vrhu steka
// S2 : vrednost prvog elementa ispod vrha
// push/pop standardne stack instrukcije

#define NOP 0x00   // NOP : prazna instrukcija
#define PUSH 0x01  // PUSH n : push n
#define POP 0x02   // POP : pop
#define LOAD 0x03  // LOAD s : push vrednost s
#define STORE 0x04 // STORE s : postavljanje vrednosti s na S1 i pop
#define ADD 0x05   // ADD : push S2 + S1, pop S1 i S2
#define SUB 0x06   // SUB : push S2 - S1, pop S1 i S2
#define MUL 0x07   // MUL : push S2 * S1, pop S1 i S2
#define OR 0x08    //  OR  : push S2 | S1, pop S1 i S2
#define AND 0x09   // AND : push S2 & S1, pop S1 i S2
#define NOT 0x0A   // NOT : push !S1, pop S1
#define JMP 0x0B   // JMP n : skok na instrukciju n
#define JZ 0x0C    // JZ n : skok na inst. n ako je S1=0 (Jump Zero) i pop
#define JP 0x0D    // JP n : skok na inst. n ako je S1>0 (Jump Plus) i pop
#define JM 0x0E    // JM n : skok na inst. n ako je S1<0 (Jump Minus) i pop
#define HALT 0x0F  // HALT : zaustavljanje programa
```

У делу за дефиниције макроа налазе се још и ограничења за максималан број инструкција (65535), максималну величину стека (1024), и максималан број променљивих (1024). Сама стек машина се налази у оквиру структуре `vsm_struct`. Стек је имплементиран као низ коначне дужине уз логичку вредност која проверава прекорачење. Како машине овог типа заиста стек смештају као део меморије, постоји могућност доласка до прекорачења, па је ова имплементација остала томе верна. Поред стека у оквиру структуре се налазе:

- Стандардни program counter `PC` и показивач на врх стека `SP`;
- Структура `instruction` која представља инструкцију са својим операционим кодом и опционим аргументом;
- Структура `variable` која представља променљиву и садржи име и вредност променљиве;
- Низ инструкција `program` у који се учитава програм и променљива `progSz` која чува величину програма;

- `unordered_map<string, int> idxs`, хеш табела која свакој променљивој додељује јединствени индекс;
- Низ променљивих `vars` у коме чувамо имена и вредности, уз променљиву `varsSz` која чува број променљивих које су се појавиле до сада.

Извршавање програма почиње инстанцирањем стек машине и позивом њене функције `LoadProgram`, која учитава програм у меморију, и претвара изворни код у низ вредности типа `instruction`. Та функција позива помоћне функције `LoadStrArg, LoadIntArg, LoadNoArg` у зависности од типа аргумента који инструкција која се у датом тренутку учитава носи. Након учитавања програма у меморију почиње извршење позивом функције `ExecuteProgram`.

Та функција препознаје инструкцију на коју показује PC и позива одговарајуће функције које извршавају појединачну инструкцију, притом водећи рачуна о прекорачењу стека. Као пример ћемо приказати једну функцију за извршење појединачне инструкције `DoJZ` која извршава инструкцију JZ, условни скок који је користан при симулацији циклуса и `if` наредби.

```
void DoJz(int arg)
    {
        if(s[SP--] == 0)
            PC = arg;
        else
            PC++;
        return;
    }
```

Прикажимо сада и комплетан код функције `ExecuteProgram`.

```
void ExecuteProgram()
    {
        while(1)
        {
            // izvrsavanje instrukcije na koju pokazuje PC

            // zavisno od op koda poziva se odgovarajuca
            // funkcija koja izvrsava instrukciju

            switch(program[PC].op)
            {
                case(0x00) : DoNop(); break;
                case(0x01) : DoPush(program[PC].arg); break;
                case(0x02) : DoPop(); break;
                case(0x03) : DoLoad(program[PC].arg); break;
                case(0x04) : DoStore(program[PC].arg); break;
                case(0x05) : DoAdd(); break;
                case(0x06) : DoSub(); break;
                case(0x07) : DoMul(); break;
                case(0x08) : DoOr(); break;
                case(0x09) : DoAnd(); break;
                case(0x0A) : DoNot(); break;
                case(0x0B) : DoJmp(program[PC].arg); break;
                case(0x0C) : DoJz(program[PC].arg); break;
                case(0x0D) : DoJp(program[PC].arg); break;
                case(0x0E) : DoJm(program[PC].arg); break;
                case(0x0F) : DoHalt(); return;
            }

            // prekoracenje steka
            if(stackOverflow)
            {
                cout<<"Doslo je do prekoracenja steka na instrukciji br. "
                << PC << "." <<endl;
                return;
            }
        }
    }
```

На самом крају извршења програма позива се функција `PrintMemoryStatus()` која исписује стање меморије у тренутку позива.

Сада када смо упознати са структуром и синтаксом VSM-а можемо да објаснимо и последњи део процеса компилације, генерисање кода.

4.6 Генерисање кода

Генерисање кода представља процес конверзије синтаксног стабла у скуп инструкција за VSM. Програм који генерише код се налази у оквиру скрипте `codegen.hs` и претвара променљиву типа `Program` у стринг који представља тражени скуп инструкција. Свака функција прима део који треба да обради и показивач на линију (почевши од 1) у завршном коду од које треба да почне да исписује инструкције, а враћа низ инструкција добијен обрадом и показивач на последњу линију у завршном коду коју ће заузети. Параметри који означавају линије кода су битни због инструкција које се баве скоковима. Главна функција је `codegen`:

```
codegen :: Program -> String
codegen program = unlines $ (fst $ doStmtBlock program 1) ++ ["HALT"]
```

Функција codegen позива функцију `doStmtBlock` која обрађује блок наредби, и на повратни резултат те функције додаје завршну команду `HALT`. Поред функције `doStmtBlock` присутна је и функција `doSingleStmt` која обрађује појединачну команду, а ту су и функције које обрађују аритметичке и логичке изразе различитих нивоа. Као илустрацију, прикажимо последњи део функције `doSingleStmt` који се бави командом while:

```
doSingleStmt :: Stmt -> Int -> ([String], Int)

. . .

doSingleStmt (WhileDo bExp1 doBlock) ptr = (allInsts, doPtr + 1)
   where (bExpInsts, bExpPtr) = doBExp1 bExp1 ptr
         (doInsts, doPtr) = doStmtBlock doBlock (bExpPtr + 2)
             skipDo = ["JZ " ++ show (doPtr + 2)]
             newIter = ["JMP " ++ show ptr]
             allInsts = bExpInsts ++ skipDo ++ doInsts ++ newIter
```

Остали делове ове функције као и остале функције функционишу на сличан начин.

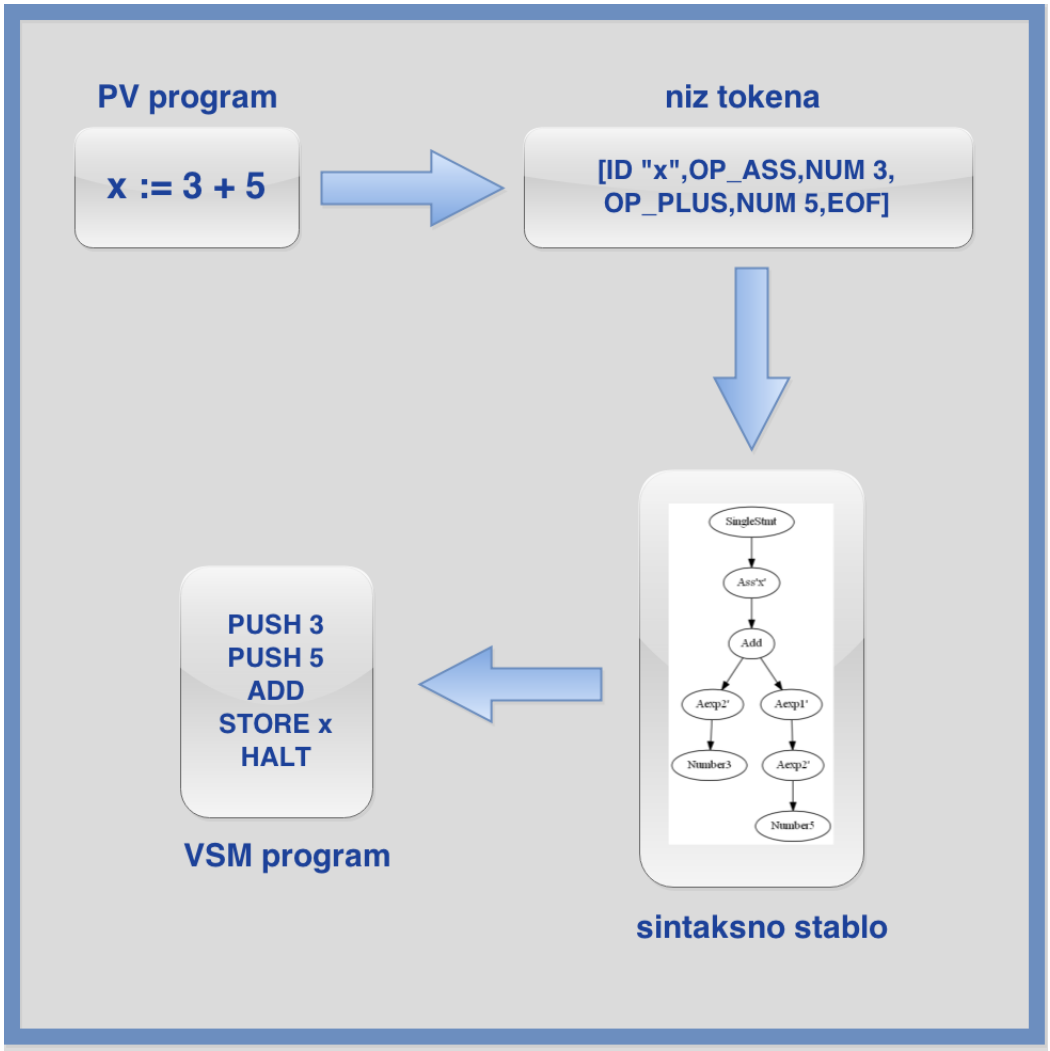
Сада се последњи пут осврнимо на `test0` и погледајмо како изгледа скуп инструкција за VSM добијен применом генератора кода на AST (`program.vsm`).

```
PUSH 1
STORE b
LOAD b
PUSH 3
SUB
JP 9
PUSH 0
JMP 10
PUSH 1
JZ 14
PUSH 0
STORE a
JMP 18
LOAD b
PUSH 1
ADD
STORE a
HALT
```

Коначно, процес компилације је завршен. Добијени код може да се изврши на стек машини, чиме добијамо следећи приказ стања меморије на крају извршења (`vsm_out.txt`):

```
Program se zaustavio na instrukciji br. 18.
-----------------
Stanje memorije:
Velicina programa = 18
Broj promenljivih = 2
Program counter = 18
Pokazivac na vrh steka = 0
b = 1
a = 2
-----------------
```

Ако се присетимо кода у језику PV који је дат на почетку овог поглавља у оквиру теста `test0`, видимо да је компилација успела и да је програм извршен на очекиван начин, што значи да је процес компилације успео. На самом крају ове главе, пре него што пређемо на закључак у коме ће бити сумирани утисци у вези са процесом израде теоријског и имплементационог дела овог рада, подсетимо се целог процеса компилације описаног у овој глави кроз дијаграм.



Слика 3: Дијаграм процеса компилације
(Online алат `www.draw.io`)

Глава 5

Закључак

Главни циљ овог матурског рада је био да пружи увод у начине размишљања и идеје које стоје у позадини ламбда рачуна и функционалног програмирања и на конкретном примеру имплементације компајлера демонстрира моћ функционалних програмских језика. Поменути концепти су недовољно заступљени а имају шта да пруже, јер доносе нов начин размишљања који не личи на уобичајено програмирање са којим се већина среће. Пратећи пројекат сматрам успешно изведеним: добијен је компајлер који успешно преводи програме из језика PV у језик за виртуелну стек машину, а успешно је извршена и имеплементација стек машине која добијене програме извршава.

Кроз рад сам покушао да увод у поменуте ствари пружим што концизније и јасније, а како има јако мало материјала који су у вези са овом облашћу на српском језику, надам се да ће мој рад бити од користи заинтересованима који желе да се упусте у свет ламбда рачуна и функционалног програмирања. Сама функционална парадигма, која ми је била потпуно страна до пре почетка израде овог рада, ме заиста одушевљава, и у току израде сам научио много тога, а мислим да ћу захваљујући овом раду наставити да се бавим повезаним темама и у будућности. Такође, верујем да нисам једини, и да би овакав приступ, да имају прилике да се сусретну са њим, многим ђацима који се баве програмирањем био занимљив и примамљив по доласку из императивних језика који неминовно представљају почетак свачијег упознавања са светом програмирања.

На самом крају, желео бих да изразим захвалност:

- **Ивану Чукићу** - ментору овог рада и менторском професору анализе са алгебром у прва два разреда средње школе, на изузетној посвећености успеху овог рада и значајном времену које је уложио у давање врло конструктивних коментара и примедби;
- **Милану Чабаркапи** - професору програмирања у прва три разреда основне школе, на неизмерној подршци у многим аспектима;
- **Петру Величковићу** - уваженом студенту универзитета у Кембрицу, на преданости давању подршке млађим ђацима, свеопштој присутности у тренутку када је потребна помоћ или мотивација, и безграничном стрпљењу, како у току израде овог рада тако и пре ње;
- **Свим професорима** - како у школи тако и ван ње, који су допринели мом образовању у току школовања. Такође, свим људима који су написали бар једну образовну књигу или tutorial, и труде се да пренесу своје знање, а нарочито **Мирану Липовачи**, на изузетном уводу у Haskell, који је успео да ме привуче у свет функционалног програмирања;
- **Свим осталима** - који нису наведени изнад, а допринели су стварању овог рада.

У Београду, мај 2015. Никола Јовановић

Литература

[1] Alonzo Church, A note on the Entscheidungsproblem, J. Symb. Log. 1 1, 1936.

[2] Stephen A. Edwards, Functional Programming and the Lambda Calculus, Columbia University, 2008,
www.cs.columbia.edu/~sedwards/classes/2010/w4115-summer/functional.pdf

[3] Miran Lipovača, Learn You a Haskell for Great Good!, 978-1-59327-283-8, 2011,
www.learnyouahaskell.com
Последњи приступ сајту: 27/05/2015

[4] Ненад Митић, Функционално програмирање - предности и недостаци, Математички факултет Универзитета у Београду, 2009,
http://www.dms.rs/DMS/data/seminari/seminar2009/N.Mitic.pdf

[5] Raul Rojas, A Tutorial Introduction to the Lambda Calculus, FU Berlin, WS-97/98,
http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

[6] Jan Šnajder, Funkcijsko programiranje i programski jezik Haskell, sveučilište u Zagrebu, Fakultet elektronike i računarstva, 2012,
http://www.fer.unizg.hr/_download/repository/PPIJ_Funkcijsko_programiranje_i_Haskell%5B3%5D.pdf

[7] Alan Mathison Turing, On computable numbers, with an application to the Entscheidungsproblem, Journal of Math 58 345-363, 1936,
http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[8] Wikipedia, Lambda calculus,
http://en.wikipedia.org/wiki/Lambda_calculus
Последњи приступ сајту: 27/05/2015

[9] Lambda tutorial,
https://files.nyu.edu/cb125/public/Lambda/
Последњи приступ сајту: 27/05/2015

[10] Wikipedia, Functional programming,
http://en.wikipedia.org/wiki/Functional_programming
Последњи приступ сајту: 27/05/2015

[11] Функционално програмирање,
http://perun.pmf.uns.ac.rs/documents/cc/fp.pdf

[12] HaskellWiki, Haskell in industry,
wiki.haskell.org/Haskell_in_industry
Последњи приступ сајту: 27/05/2015

[13] Companies using OCaml,
https://ocaml.org/learn/companies.html
Последњи приступ сајту: 27/05/2015

[14] Jane Street, technology,
https://www.janestreet.com/technology/
Последњи приступ сајту: 27/05/2015

[15] HackerRank, Functional programming - compilers,
https://www.hackerrank.com/domains/fp/compilers
Последњи приступ сајту: 27/05/2015

[16] Graphviz - Graph Visualization Software,
http://www.graphviz.org
Последњи приступ сајту: 27/05/2015