Nikola Jovanović

Primena funkcionalnog programiranja na problem izrade kompajlera

Rad se bavi primenom funkcionalnog programiranja na problem konstrukcije kompajlera i poređenjem takvog pristupa sa korišćenjem alata za automatsko generisanje leksičkih i sintaksnih analizatora. Za potrebe istraživanja definisan je jednostavan edukativni jezik PV koji poseduje karakteristične konstrukcije imperativnih programskih jezika, a služi kao ulazni jezik za proces kompajliranja. Takođe, definisan je i ciljni jezik u vidu seta instrukcija za virtuelnu stek mašinu. Objašnjen je proces izrade kompajlera u Haskell-u, kao i kompajlera generisanog uz pomoć alata Flex/Bison, uz uvid u ključne metode i probleme. Izvršeno je poređenje dva pristupa i zaključeno da funkcionalni jezici poseduju karakteristike koje ih čine pogodnim za problem konstrukcije kompajlera za jednostavne, domenski specifične jezike. Iako je funkcionalni pristup po pitanju ekspresivnosti koda i vremena potrebnog za testiranje i verifikaciju pokazao odlične rezultate, zaključeno je da su pristupi bazirani na generatorima programskog koda ipak bolji izbor na ovom nivou složenosti.

Uvod

Funkcionalno programiranje (FP) je programska paradigma u kojoj se sva izračunavanja vrše kroz evaluaciju matematičkih funkcija. Programi pisani funkcionalnim jezicima ne sadrže naredbe, niti redosled izvršavanja, već prosto definišu funkcije. Glavno svojstvo tih funkcija je da njihov izlaz zavisi samo od ulaza, a ne od trenutka u kom su pozvane ili stanja u memoriji. Takođe, ove funkcije ne izazivaju sporedne efekte, tj. ne interaguju sa globalnim promenljivama ili standardnim ulazom i izlazom. Ovakav, koncept čistih funkcija, predstavlja osnovu funkcionalnog načina razmišljanja i ima značajne prednosti. Prirodno se nameće konciznost i modularnost koda, pa je samim tim program manje podložan greškama i skraćeno je vreme testiranja. Vrlo bitna odlika funkcionalnih jezika, naročito u današnje vreme, je njihova podložnost paralelizaciji. Zbog svega ovoga funkcionalno programiranje biva sve popularnije u industriji (Web 1), ali i sa teorijske strane, kao kontrast imperativnim jezicima i objektno orijentisanoj paradigmi koja je danas najzastupljenija. Kao posledica toga nastali su mnogi hibridni jezici, a već postojeći imperativni jezici (Java, C++) u svojim najnovijim standardima implementiraju neke ideje funkcionalnog programiranja – lambda izrazi, izvođenje tipova itd. (Web 2; Web 3; Web 4).

Cilj rada je da na konkretnom i realnom primeru izrade kompajlera prikaže prednosti i mane primene funkcionalnog stila programiranja u toj oblasti, te da pomenuti pristup uporedi sa standardnim tehnikama zasnovanim na upotrebi generatora programskog koda.

Ulazni jezik PV i virtuelna stek mašina

Za potrebe ovog istraživanja definisan je jezik PV, i osmišljen set instrukcija nalik asemblerskim koje se izvršavaju na virtuelnoj stek mašini (VSM), mašini koja pri evaluaciji izraza koristi stek. Sledi opis sintakse jezika, kao i objašnjenje virtuelne stek mašine i njenih instrukcija.

Jezik PV

Programski jezik PV je jednostavan, domenski specifičan jezik nastao za potrebe ovog rada, osmišljen kao uprošćena verzija poznatih imperativnih jezika. Njegova sintaksa se opisuje u vidu kontekstno slobodne gramatike:

Nikola Jovanović (1996), Beograd, Tadeuša Košćuška 74, učenik 4. razreda Matematičke gimnazije u Beogradu

MENTOR: Miloš Savić, Prirodnomatematički fakultet Univerziteta u Novom Sadu

x, y ∈ **Var** (promenljive, alfanumerički stringovi koji počinju slovom)
n ∈ **Num** (brojevi, celobrojne vrednosti)
opa ∈ **OPa** (aritmetičke operacije)
opa ::= + | *
opb ∈ **OPb** (logičke operacije)
opb ::= and | or
opr ∈ **OPr** (relacije)
opr ::= | | =
a ∈ **Aexp** (aritmetički izrazi)
a ::= n | x | a opa a | (a)
b ∈ **Bexp** (logički izrazi)
b ::= true | false | !b | a opr a | b opb b | (b)
s ∈ **Stmt** (naredbe)
S ::= skip | x := a | S;S | if b then {S} else {S} | while b do {S}

PV koristi konstrukcije poznate iz imperativnih jezika i standardna pravila asocijativnosti i prioriteta operatora, pa nije neophodno dalje objašnjenje njegove semantike. Takođe, PV ima podršku za komentare u jednoj ili više linija, koji se ograničavaju karakterom '/'. Pogledajmo primer jednog programa napisanog u programskom jeziku PV:

```
b := 1;
a := 1;
while (b 4)
do
{
    a := a + b;
    b := b + 1 / uvecavamo brojac /
}
```

Primetimo da se komande u okviru jednog bloka komandi odvajaju znakom ';', koji se ne nalazi nakon poslednje komande u bloku.

Virtuelna stek mašina

Virtuelna stek mašina (VSM) je mašina koja pri evaluaciji izraza koristi stek i podržava evaluaciju u formi obrnute poljske notacije. U okviru ovog istraživanja VSM je implementirana u programskom jeziku C++, u cilju testiranja procesa kompajliranja i samog kompajlera. VSM podržava 16 instrukcija:

Instrukcije za rad sa stekom i memorijom:

- PUSH n – dodaje celobrojnu vrednost n na vrh steka
- POP – izbacuje element sa vrha steka (S1)
- LOAD s – dodaje vrednost promenljive s na vrh steka
- STORE s – postavlja vrednost promenljive s na S1, i izbacuje S1 sa steka

Aritmetičke i logičke instrukcije:

- ADD – uvećava drugi element na steku (S2) za S1 i izbacuje S1
- SUB – umanjuje S2 za S1 i izbacuje S1
- MUL– množi S2 sa S1 i izbacuje S1
- OR – postavlja S2 na logičku disjunkciju S2 i S1 i izbacuje S1
- AND – postavlja S2 na logičku konjukciju S2 i S1 i izbacuje S1
- NOT – vrši logičku negaciju nad S1

Skokovi:

- JMP n – bezuslovni skok na instrukciju n
- JZ n – skok na instrukciju n ako je S1 jednak nuli
- JP n – skok na instrukciju n ako je S1 veći od nule
- JM n – skok na instrukciju n ako je S1 manji od nule

NOP – prazna instrukcija
HALT – zaustavlja izvršenje programa

VSM u memoriji pored steka čuva program counter (PC), pokazivač na vrh steka (SP), program kao niz instrukcija, i niz indeksa promenljivih koje se pojavljuju u programu. VSM učitava niz instrukcija i izvršava ih, u svakom momentu izvršavajući instrukciju na koju pokazuje PC. Kao primer programa za VSM prikazujemo rezultat kompajliranja PV programa datog u prethodnoj sekciji:

```
PUSH 1          PUSH 1
STORE b         STORE a
LOAD b          LOAD b
PUSH 4          MUL
SUB             STORE a
JM 11           LOAD b
PUSH 0          PUSH 1
JMP 12          ADD
PUSH 1          STORE b
JZ 22           JMP 5
LOAD a          HALT
```

Kompajliranje

Kompajliranje je proces prevođenja izvornog koda iz jednog programskog jezika (izvorni jezik, često jezik sa visokim nivoom apstrakcije) u drugi programski jezik (ciljni jezik, uglavnom sa niskim nivoom apstrakcije, neretko i sam asembler). Cilj kompajliranja je u velikom broju slučajeva kreiranje izvršnog fajla koji može da se pokrene na ciljnoj mašini. U okviru ovog rada izvorni jezik je PV, dok skup instrukcija za VSM predstavlja ciljni jezik.

Kompajliranje se najčešće deli na tri glavna procesa:

1. leksička analiza
2. sintaksna analiza (parsiranje)
3. generisanje koda

U narednim sekcijama biće kratko objašnjeni ključni procesi kompajliranja kao i komponente kompajlera koje ih realizuju (leksički analizator, parser, generator koda) i detalji njihove implementacije u okviru ovog istraživanja. Sam kompajler je pisan u programskom jeziku Haskell, kao karakterističnom predstavniku funkcionalnih jezika. Pomoćni programi realizovani za potrebe rada poput implementacije virtuelne stek mašine ili programa koji se koristi u procesu vizualizacije sintaksnog stabla su pisani u jeziku C++.

Leksička analiza

Leksička analiza je prvi korak u procesu kompajliranja i pretvara ulazni program, koji je trenutno samo niz karaktera, u niz *tokena*, značajnih stringova koji predstavljaju osnovne jedinice nekog jezika (npr. ključne reči ili imena promenljivih). Osnovu leksičkog analizatora predstavljaju regularni izrazi koji opisuju tokene. Iz tih regularnih izraza izvodimo konačne automate od kojih je leksički analizator izgrađen. Upravo zbog toga je u Haskellu moguća vrlo efektna realizacija leksičkog analizatora korišćenjem ugrađenih konstrukcija jezika koje omogućavaju uparivanje uzoraka (eng. pattern matching) i uslovne izraze (eng. guards).

Nakon uklanjanja komentara i razdvajanja programa na delove na osnovu separatora, pokušavamo da trenutni niz karaktera uparimo sa regularnim izrazima koji odgovaraju tokenima, uz određeni prioritet. Ako je uneti program u skladu sa sintaksom ulaznog jezika, uparivanje će uspeti i proces se nastavlja na ostatku programa sve dok se ceo program ne tokenizuje. Ilustrujmo ovo delom Haskell koda iz implementacionog dela ovog rada, funkcijom **nextToken** koja predstavlja prvi deo leksičkog analizatora i razdvaja osnovne grupe tokena:

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

Izlaz koji daje leksički analizator kada se primeni na kod dat kao primer programa u jeziku PV je sledeći niz tokena:

```
[ID "b", OP_ASS, NUM1, SEMICOLON,
ID"a",OP_ASS,NUM1,SEMICOLON,
KWRD_WHILE,PAREN_LEFT,ID "b",
OP_ORDLT, NUM4, PAREN_RIGHT,
 KWRD_DO,BRKT_LEFT,ID "a",OP_ASS,
ID "a",OP_MUL,ID "b",SEMICOLON,
ID "b",OP_ASS,ID "b",OP_PLUS,
NUM 1,BRKT_RIGHT,EOF]
```

Niz tokena dobijen leksičkom analizom potom treba da prođe kroz sledeći korak kompajliranja, sintaksnu analizu.

Sintaksna analiza

Sintaksna analiza (parsiranje) je proces kojim se od niza tokena nastalog leksičkom analizom dobija *apstraktno sintaksno stablo* (AST). Parser je nedeterministički konačan automat sa stekom (engl. pushdown automata) koji se može konstruisati na osnovu kontekstno slobodne gramatike kojom se opisuje ulazni jezik. Koristeći pravila produkcije date gramatike, parser pokušava da da smisao nizu tokena i odredi sintaksnu strukturu programa. U okviru rada je korišćen *rekurzivni spust*, metoda parsiranja koja se sastoji od uzajamno rekurzivnih koraka gde se pri svakom koraku primenjuje jedno pravilo gra-

matike. Tehnika rekurzivnog spusta zahteva da gramatika jezika bude u LL(k) obliku objašnjenom u Allan (2009), što gramatika jezika PV u originalnom slučaju nije, kao što smo videli u sekciji gde je ovaj jezik definisan. Pre samog parsiranja, bilo je neophodno transformisati samu gramatiku tako da ne sadrži levu rekurziju, tj. preformulisati sva pravila oblika **A** → **AB**, gde je **A** neterminalni simbol, a **B** sekvenca terminalnih i/ili neterminalnih simbola gramatike. Prethodna pravila se još nazivaju i levo-rekurzivnim pravilima.

Prethodni problem je rešen tako što je u svim levo-rekurzivnim pravilima uvedena hijerarhija neterminala, pa su npr. aritmetički izrazi razdvojeni na aritmetičke izraze prvog, drugog i trećeg reda, što je omogućilo da se pravila definišu izbegavajući levu rekurziju. Ovu transformaciju ilustruju aritmetički izrazi koji su u originalnoj definiciji jezika PV definisani na sledeći način:

```
a ::= n | x | a opa a | (a)
```

Primećujemo da su pravila **a → a opa a i a → (a)** levo-rekurzivna, pa modifikujemo definiciju aritmetičkih izraza, tako da dobijemo skup pravila bez leve rekurzije:

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

Kada na isti način transformišemo logičke izraze i naredbe dobijamo gramatiku koja pripada LL(1) klasi i na nju se može primeniti rekurzivni spust. Ponovo se nagoveštava primena uparivanja uzoraka i ispostavlja se da je to, uz par postojećih funkcija za rad sa listama, dovoljno da se u Haskellu implementira ceo parser.

Svaka funkcija uzima niz tokena i na osnovu prvog tokena u nizu pozivima ostalih funkcija parsira sve tokene potrebne da se produkcija kompletira. Nakon toga, funkcija vraća svoj rezultat (parsiran neterminal gramatike) uz niz tokena koji su ostali neiskorišćeni. Kao primer pogledajmo funkciju parseStmt koja parsira naredbu realizujući rekurzivni spust po pravilima gramatike (primer na dnu ove strane).

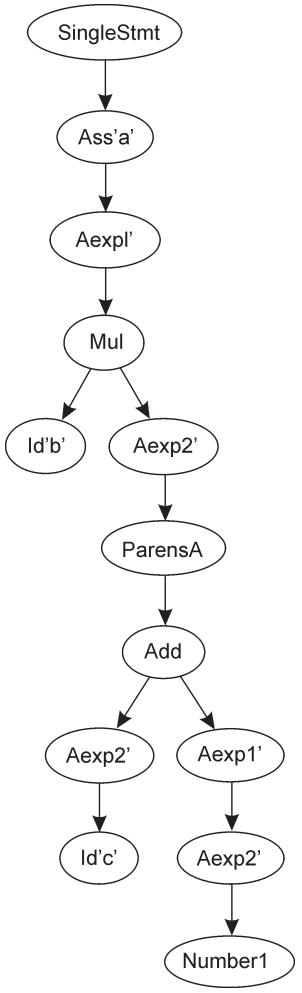
Parser kao izlaz daje apstraktno sintaksno stablo u tekstualnom obliku. Pogledajmo primer AST-a dobijenog u procesu kompajliranja prostog PV programa koji sadrži samo jednu naredbu dodele **a := b * (c + 1)**:

```
SingleStmt (Ass "a" (Aexp1' (Mul (Id
"b") (Aexp2' (ParensA (Add (Aexp2'
(Id "c")) (Aexp1' (Aexp2' (Number
1)))))))))
```

Kako apstraktno sintaksno stablo u tekstualnom formatu nije razumljivo, za vizualizaciju stabla korišćen je open-source paket GraphViz (Web 5) i napisan je pomoćni program koji pretvara tekstualno AST u .dot format koji je pogodan za vizualizaciju pomoću GraphViz-a. Na slici 1 je vizualizacija malopređašnjeg sintaksnog stabla za prostu naredbu dodele.

Primer 1. funkcija parseStmt parsira naredbu realizujući rekurzivni spust po pravilima gramatike:

```
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 "
```



Slika 1. Apstraktno sintaksno stablo za izraz
a := b * (c + 1)

Figure 1. Abstract syntax tree for the expression
a := b * (c + 1)

Generisanje koda

Poslednji korak samog kompajliranja je generisanje koda na osnovu sintaksnog stabla. Generator koda je podeljen na funkcije, gde svaka obrađuje jedan element jezika. Svaka funkcija kao argumente prima deo stabla na osnovu kog treba da generiše kod, kao i pokazivač na liniju u završnom kodu od koje treba da počne da ispisuje instrukcije. Nakon što obradi odgovarajući deo stabla, funkcija vraća niz instrukcija dobijen obradom i broj poslednje linije u završnom kodu koju taj deo koda zauzima. Praćenje broja instrukcija i linije u koju se trenutno upisuje kod je važno zbog if/else i while konstrukcija koje se u programskim jezicima sa izuzetno niskim nivoom apstrakcije realizuju korišćenjem uslovnih skokova. Na samom kraju programa nalazi se instrukcija HALT, i tu se proces generisanja koda završava. Deo funkcije doSingleStmt koji generiše kod za if/else konstrukciju dat je na dnu ove strane.

Finalna skripta compiler.hs čita program u jeziku PV, uspostavlja kontrolu toka podataka između komponenti kompajlera (leksički analizator, parser, generator koda) i na izlazu daje niz instrukcija, program koji se može izvršiti na VSM. Nakon što se prevedeni program završi, VSM ispisuje stanje memorije, program counter i sve korišćene promenljive čime se proverava validnost izlaznog koda pa samim tim i tačnost kompajliranja.

Flex/Bison pristup realizaciji kompajlera

Drugi deo ovog rada sastoji se od implementacije pomenutog kompajlera korišćenjem Flex/Bison alata (Web 6). Flex i Bison su alati za

Deo funkcije doSingleStmt koji generiše kod za if/else konstrukciju:

```
doSingleStmt :: Stmt - Int - ([String], Int)
doSingleStmt (IfElse bExp1 ifBlock elseBlock) ptr = (allInsts, elsePtr)
     where (bExpInsts, bExpPtr) = doBExp1 bExp1 ptr
                (ifInsts, ifPtr) = doStmtBlock ifBlock (bExpPtr + 2)
                (elseInsts, elsePtr) = doStmtBlock elseBlock (ifPtr + 2)
                 skipIf = ["JZ " ++ show (ifPtr + 2)]
                 skipElse = ["JMP " ++ show (elsePtr + 1)]
                 allInsts = bExpInsts ++ skipIf ++ ifInsts ++ skipElse ++
                                                              elseInsts
```

generisanje leksičkih i sintaksnih analizatora nastali kao nadogradnja unix alata lexx i yacc i često se koriste u procesu konstrukcije kompajlera. Oba alata su bazirana na jeziku C, i generišu leksički analizator i parser na osnovu specifikacionih fajlova kojima korisnik daje informacije o ulaznom i izlaznom jeziku, pravilima tokenizacije i (kontekstno slobodnoj) gramatici. Konkretno, Flex se bavi leksičkom analizom pa Flex specifikacioni fajl sadrži opis regularnih izraza i tokena koji su im pridruženi. Sa druge strane, Bison radi samu sintaksnu analizu koristeći tokene koje Flex proizvede i njegov specifikacioni fajl sadrži opis gramatike. U okviru oba specifikaciona fajla korisnik ima prostor za manuelnu interakciju sa procesom kompajliranja ubacivanjem delova C-koda. U našem slučaju, to je omogućilo laku nadogradnju generatora koda, pa je na osnovu opisa tokena jezika PV, opisa njegove gramatike i nekoliko linija umetnutog C-koda uspešno generisan kompajler koji prevodi program jezika PV u instrukcije za VSM. Flex i Bison koriste shift-reduce tehniku parsiranja, koja se značajno razlikuje od rekurzivnog spusta, te nema potrebe za eliminacijom leve rekurzije kojom se povećava broj pravila gramatike, što je čini manje razumljivom.

Diskusija

Nakon uspešne izrade kompajlera korišćenjem oba metoda moguće je napraviti međusobno poređenje, kao i diskutovati o prednostima i manama funkcionalnih jezika na problemu konstrukcije kompajlera. Prilikom implementacije primećena je konciznost, ekspresivnost i razumljivost Haskell funkcija, kao i znatno smanjeno vreme testiranja i verifikacije. Ideja rekurzivnog spusta kao i koncepti koji se nalaze u osnovi ključnih komponenata kompajlera (automati, regularni izrazi i gramatike) su vrlo bliski funkcionalnom pristupu budući da se on oslanja na transformacije podataka i rekurziju, kojom se nadoknađuje nedostatak iteracije. Takođe, na primetnu ekspresivnost utiče i postojanje Haskell-ovih sintaksnih konstrukcija poput uparivanja uzoraka i uslovnih izraza koje olakšavaju sam proces implementacije. Možemo zaključiti da su funkcionalni jezici veoma primenljivi na problem realizacije kompajlera za male domenski specifične jezike tehnikom rekurzivnog spusta, budući da su pomenute ideje vrlo prirodne za FP.

S druge strane, iako pomenuti alati (Flex/Bison) generišu kod dužine više hiljada linija, rad sa njima na ovom nivou i način na koji se alatima daju instrukcije u okviru specifikacionih fajlova su vrlo jednostavni. Dizajn parsera se svodi na specifikaciju gramatike korišćenjem formata nalik na standardnu BNF notaciju, dok je za generisanje leksičkog analizatora dovoljno samo navesti regularne izraze. Iako je Haskell blizak programeru i jasan, u prvoj implementaciji je ipak bilo neophodno voditi računa o detaljima procesa kompajliranja, na šta se gubi određeno vreme. Takođe, stvari poput regulisanja prednosti računskih operacija i asocijativnosti se u ovim alatima rešavaju prosto postavljanjem odgovarajućih flegova, a uklanjanje leve rekurzije nije neophodno, što je značajna prednost u odnosu na funkcionalni pristup i manuelno pisanje rekurzivnog spusta uopšte.

Po pitanju fleksibilnosti, na nivou jezika kojima se ovaj rad bavi oba pristupa su vrlo fleksibilna i podložna modifikacijama što je naročito izraženo kod rada sa Bison-om, gde se automatizovanost procesa i manjak kontrole nadoknađuje mogućnošću dodavanja korisničkog C koda koji se kao takav kopira u finalni kod parsera. Ipak, programeri iza kompajlera „velikih” jezika kao što su Java ili C++ su upravo zbog nedostatka kontrole nad kompajlerom odustali od alata za automatsko generisanje i prešli na ručno implementiran rekurzivni spust (Web 7). Na višem nivou razvoja kompajlera fokus je na komponentama poput optimizacije koda i lociranju i prijavljivanju grešaka u sintaksi ulaznih programa, gde se fleksibilnost ovih alata gubi i efikasnije je preći na ručnu implementaciju kompajlera. Takođe, veći jezici sadrže elemente kontekstno zavisnih gramatika koje Bison ne podržava.

Zaključak

U okviru rada je implementiran kompajler u programskom jeziku Haskell, kao i kompajler generisan pomoću Flex/Bison alata i poređena je ekspresivnost, fleksibilnost i pristupačnost ovih metoda. Primećeno je da su funkcionalni jezici dobar izbor za problem realizacije kompajlera za

proste domenski specifične jezike, i da u tom slučaju ideje iza funkcionalne paradigme i konstrukcije prisutne u funkcionalnim jezicima omogućavaju efikasnu implementaciju i nastajanje konciznog, razumljivog koda, uz minimalno vreme potrebno za testiranje. Sa druge strane, pokazana je prednost uobičajenih metoda (Flex/Bison alati) nad FP pristupom (i manuelnim pisanjem kompajlera uopšte) na nivou složenosti koji je prisutan u ovom radu.

Literatura

Allan S. J. 2009. *LL parsing*. Utah State University.

Edwards S. A. 2008. *Functional Programming and the Lambda Calculus*. Columbia University.

Lipovača M. 2011. Learn You a Haskell for Great Good! www.learnyouahaskell.com

Mitić N. 2009. *Funkcionalno programiranje – prednosti i nedostaci*. Beograd: Matematički fakultet Univerziteta u Beogradu.

Web 1. Companies using OCaml. https://ocaml.org/learn/companies.html

Web 2. cppreference, auto. http://en.cppreference.com/w/cpp/language/auto

Web 3. cppreference, lambda. http://en.cppreference.com/w/cpp/language/lambda

Web 4. The Java Tutorials, lambda expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Web 5. Graphviz – Graph Visualization Software, http://www.graphviz.org

Web 6. What are Flex and Bison. http://aquamentus.com/flex_bison.html

Web 7. GCC 3.4 Changelog. https://gcc.gnu.org/gcc-3.4/changes.html

---

Nikola Jovanović

Use of Functional Programming in Compiler Construction

This paper deals with the application of functional programming in the field of compiler construction. Functional programming is a paradigm that is characterized by brevity, reduced implementation and testing times, and overall code conciseness. These points lead to an assumption that the functional style of programming could be efficiently used in compiler construction. In this paper we describe the implementation of a compiler in the Haskell programming language. The source language for the compilation is PV, a simple domain-specific language that resembles a subset of well-known imperative languages, while the target machine is a virtual stack machine with a set of 16 standard instructions. We define both languages and focus on some key moments in compiler development. Another compiler for PV is developed using standard lexer/parser generation tools Flex and Bison in order to compare the functional to the standard, widely used approach to the problem. We conclude that functional languages exhibit certain characteristics related to code clarity and ease of testing that make them suitable for compiler construction, while the standard code generation tools still remain the best choice on this level of complexity.