Glavni koraci dizajna i ugradnje jednostavnog programskog prevoditelja

Benčević, Marin

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni preddiplomski studij računarstva

GLAVNI KORACI DIZAJNA I UGRADNJE JEDNOSTAVNOG PROGRAMSKOG PREVODITELJA

Završni rad

Marin Benčević

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 15.09.2018.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada

Ime i prezime studenta:	Marin Benčević
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3747, 26.09.2017.
OIB studenta:	21788317681
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	Dr.sc. Bruno Zorić
Sumentor iz tvrtke:	
Naslov završnog rada:	Glavni koraci dizajna i ugradnje jednostavnog programskog prevoditelja
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	15.09.2018.
Datum potvrde ocjene Odbora:	26.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 02.10.2018.

Ime i prezime studenta:	Marin Benčević
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3747, 26.09.2017.
Ephorus podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Glavni koraci dizajna i ugradnje jednostavnog programskog prevoditelja**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora Dr.sc. Bruno Zorić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ:

1. UVOD 1
1.1 Zadatak završnog rada 1
2. PREVODITELJI 2
2.1. Povijest prevoditelja 2
2.2. Proces prevođenja 3
3. FAZE PREVOĐENJA 5
3.1 Leksička analiza 5
3.2 Leksički analizator 6
3.3 Sintaksna analiza 8
3.4 Generiranje koda 11
4. IMPLEMENTACIJA PROGRAMSKOG PREVODITELJA KLANG 13
4.1 Kaleidoscope 13
4.2 Swift 14
4.3 LLVM Projekt 14
4.4 Detalji implementacije 15
4.5. Implementacija leksičkog analizatora 16
4.6 Implementacija sintaksnog analizatora 20
4.7 Generiranje koda uz pomoć LLVM-a 25
5. ZAKLJUČAK 33
LITERATURA 34
SAŽETAK 35
ABSTRACT 36
ŽIVOTOPIS 37
PRILOZI 38

1. UVOD

U povijesti se programiralo tako što se strojni kod ukucavao u memoriju računala. Kako su računala postala naprednija i kako su rasli zahtjevi za sve kompleksnijom programskom podrškom, tako se javila potreba za bržim načinom pisanja koda. Stoga su ljudi krenuli koristiti programske jezike, odnosno definirana je sintaksa i struktura koja omogućuje zapisivanje kompleksnih programa u manje teksta nego u strojnom kodu. Računalo izvršava strojni kod, pa je potrebno prevesti programske jezike u strojni kod. Prevoditelji su programi koji jedan programski jezik prevode u drugi, ili neki programski jezik prevode u strojni kod. Prevoditelj dakle kao ulaz uzima programski kod u jednom jeziku, a kao izlaz ima program u drugom programskom jeziku koji obavlja isti zadatak. Osim samog prevođenja, većina prevoditelja obavlja i analizu koda, provjeru pogrešaka i optimizaciju. Većina modernih prevoditelja razlikuje četiri glavne faze prevođenja programskog koda: leksičku analizu, sintaksnu analizu, semantičku analizu, optimizaciju i na kraju generiranje koda.

U drugom je poglavlju ovog rada dan pregled procesa prevođenja, kao i povijest prevoditelja od njihovog nastanka do danas. U trećem poglavlju prikazana je teorijska osnova procesa prevođenja po fazama prevođenja. Prikazane su faze leksičke i sintaktične analize te generacija koda. U četvrtom poglavlju prikazan je primjer implementacije programskog prevoditelja za jezik Kaleidoscope, napisan u Swift programskom jeziku.

1.1 Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati pojam programskog prevoditelja i njegovu ulogu. Detaljno valja prikazati i opisati korake prevođenja programa poput leksičke i analize sintakse, provjere tipova itd. U praktičnom dijelu rada potrebno je ugraditi jednostavni programski prevoditelj.

2. PREVODITELJI

Općenito, prevoditelj je računalni program koji neki programski jezik (većinom jezik više razine) prevodi u neki drugi jezik (većinom jezik niže razine), ili izravno u strojni kod. Prema [1], većina modernih prevoditelja ne prevodi jezik izravno u strojni kod, već prevodi u neki drugi jezik (C, C++) ili u neki prijelazni oblik između strojnog koda i izvornog jezika.

Na kraju svojeg rada, prevoditelji kreiraju izvršnu datoteku, koja se može pokrenuti i vidjeti rezultat izvođenja. Kao što stoji u [1], to razlikuje prevoditelje od interpretera, koji programski kod prevode za vrijeme izvođenja samog programa. Zbog toga su interpreteri većinom sporiji od prevoditelja.

2.1. Povijest prevoditelja

Prevoditelji su se javili 1950-ih, jer je programiranje postalo sve više i više zastupljeno i zahtjevi za programima su postajali sve kompleksniji. Do tada su programeri programirali strojnim kodom, nulama i jedinicama. Pisanje strojnog koda spor je i zahtjevan proces uslijed kojeg nastaje puno grešaka, stoga su prema [2] programeri krenuli razvijati programske jezike više razine, koji su znatno ubrzali i pojednostavili proces programiranja.

Jedan od prvih takvih jezika je FORTRAN, razvijen 1950-ih. Obzirom da se koristi jezik višeg reda, postajala je potreba taj jezik prevesti na strojni kod na automatski način. Zato je FORTRAN sa sobom donio prvi prevoditelj sa svim fazama koje imaju i moderni prevoditelji. FORTRAN je potaknuo razvoj teorije i dizajna prevoditelja i mnogi koncepti koji postoje u FORTRAN prevoditelju postoje i u današnjim prevoditeljima [3]. U kasnim pedesetima razvija se jezik COBOL, čiji se prevoditelj zasniva na FORTRAN-ovom, ali je prvi prevoditelj koji ima mogućnost prevođenja za više različitih platformi.

U početku, prevoditelji su se zasnivali na procesu leksičke i sintaktičke analize te je taj dio prevoditelja bio najkompleksniji. Uslijed razvoja alata za automatsku leksičku i sitnaksnu analizu i razvoja novih tehnika analize, ovaj dio postao je mnogo jednostavniji. Danas je najkompleksniji dio prevođenja optimizacija i generiranje koda, te se područje znanosti o prevoditeljima najviše bavi tim dijelovima prevođenja.

Važan dio povijesti prevođenja je razvoj standarda programskog prevođenja koji su neovisni o jeziku i razvoj virtualnih strojeva i alata za kreiranje prevoditelja. Takvi alati i standardi

omogućuju puno brži razvoj prevoditelja za više platformi i interoperabilnost između različitih jezika. U tablici 2.1 dan je pregled najpopularnijih današnjih biblioteka, platformi i alata za prevođenje.

Tablica 2.1 *Popularni alati i platforme za prevođenje.*

Platforma	Jezici	Namjena
JVM (Java Virtual Machine)	Java, Scala, Clojure, Kotlin, itd.	Virtualni stroj, upravljanje memorijom, Java bytecode prevoditelj, JIT, specifikacija, itd.
.NET Framework	C#, VB.NET, F#, itd.	Virtualni stroj, upravljanje memorijom, interoperabilnost jezika, itd.
LLVM	Swift, Objective-C, C#, Haskell, Python, Rust, itd.	Intermediate representation, generacija koda, sustav tipova, JIT, upravljanje memorijom, itd.
JavaScript/EcmaScript	JavaScript, Dart, TypeScript, F#, ReasonML, Elm, itd.	Odredišni jezik mnogo prevoditelja, interpreter, specifikacija

2.2. Proces prevođenja

Proces prevođenja jednog jezika u drugi zahtijeva nekoliko faza. Prvo, prevoditelj mora razumjeti značenje izvornog koda. Nakon toga, mora izvorni kod prevesti u apstraktan oblik koji ne ovisi o sintaksi ni jednog jezika. Tek nakon što ima originalan kod u takvom obliku, može iz apstraktnog prikaza rekonstruirati kod u odredišnom jeziku koji će imati isto značenje. Razumijevanje originalnog koda se ostvaruje leksičkom analizom tog koda. Nakon leksičke analize nastupa sintaksna analiza, gdje se gleda značenje koda i gradi apstraktni prikaz tog koda. Nakon što je napravljen apstraktni prikaz, na njemu se može raditi semantička analiza, odnosno analizira značenja koda i provjera postoje li pogreške. Također se može optimizirati kod. Naposlijetku se na osnovu apstraktnog prikaza gradi kod u odredišnom jeziku u fazi koja se naziva generiranje koda.

Ako je odredišni jezik strojni jezik, generirani se kod sprema u izvršnu datoteku i veže se s korištenim bibliotekama i bibliotekama unutar samog sustava. Različite procesorske arhitekture imaju različite instrukcije strojnog koda, tako da se jezici često prevode u posredni jezik, nalik na strojni kod, ali neovisan o arhitekturi. Tada se posredni jezik jednostavnim postupcima prevodi u strojni jezik za specifičnu arhitekturu na kojoj se nalazi.

Za razliku od prevoditelja, interpreteri ne proizvode odredišnu datoteku i proces prevođenja odvija se tzv. "*on-line*" načinom, odnosno kao dio izvršavanja programa. Interpreteri čitaju, prevode i izvršavaju kod tijekom rada programa, te zbog toga imaju nekoliko nedostataka. Zbog nemogućnosti višestrukog prolaska kroz izvorni kod, interpreteri imaju puno manje prilika za optimizaciju koda, te često izvođenje koda može biti sporije od prevoditelja.

Međutim prevoditelji imaju i prednosti. Nemaju vrijeme potrebno za prevođenje, tako da se kod može izvršiti gotovo odmah nakon pisanja. Uklanjanje pogreški je lakše jer se izravno izvršava kod, pa je lagano odrediti gdje je greška.

3. FAZE PREVOĐENJA

U ovom će poglavlju biti prikazan proces prevođenja i tri osnovne faze prevođenja: leksička analiza, sintaksna analiza i generiranje koda. Osim ove tri faze, prevoditelji često uključuju i semantičku analizu te optimizaciju.

3.1 Leksička analiza

Prva je faza prevođenja leksička analiza. Leksička analiza je zapravo raščlamba izvornog koda na fundamentalne jedinice koje se nazivaju leksičke jedinke (engl. *token*). Leksička je jedinka uređeni par leksema (riječi) te klase leksema u koju spada taj leksem. (npr. *(“3.14”, decimalniBroj)*) U programskim jezicima su to ključne riječi, zagrade, imena varijabli, vrijednosti i sl. Faza leksičke analize za ulaz ima kod u izvornom programskom jeziku (kao niz znakova), a na izlazu ima niz leksičkih jedinki.

Npr., ako je na ulazu sljedeći kod:

```
int value = 100;
```

taj kod se može podijeliti na sljedeće jedinke:

```
int (klj. riječ), value (ime varijable), = (operator), 100 (konstanta) ; (završetak)
```

Podjela klasa leksema ovisi o specifikaciji programskog jezika. Jedna klasa leksema predstavlja konačan skup svih leksema koji pripadaju toj klasi. Npr. riječi “ime” i “prezime” spadaju u klasu “ime varijable”, dok riječi “3.14” i “2.15” spadaju u klasu “decimalni broj”. Broj riječi i slova u nekoj klasi ovisi o specifikaciji same klase, tako da mogu postojati klase sa samo jednim leksemom, npr. “=“ koji čini cijelu klasu “jednako” ili “if” koji čini klasu “ako”.

Svi simboli koji se pojavljuju u nekom jeziku čine abecedu tog jezika. Kod programskih jezika to može biti mala abeceda od samo “0” i “1”, ali može biti i abeceda koja uključuje sve Unicode simbole. Leksemi su nizovi jednog ili više simbola iz te abecede, dok su leksičke klase skup više leksema. Sve leksičke klase zajedno čine leksičku strukturu nekoga jezika.

Leksička struktura jezika definira se regularnim jezikom, a jedna klasa leksema definira se regularnim izrazom (engl. *regex*). Regularni izraz je slijed znakova koji definira predložak koji se traži u nekom tekstualnom ulazu. Formalno, služe za opisivanje regularnih jezika, a sastoje se od konstanti (sljedova znakova) i operatora nad nizovima znakova.

U regularnom jeziku, definirana su tri skupa [2]:

1. Prazan skup (∅ = ∅)
2. Epsilon skup, skup koji sadržava prazan niz znakova (ε = { “ ” })
3. Skup slova, koji sadržava jedno slovo (npr. a = { “a” })

Također su definirane tri operacije koje se mogu vršiti na regularnim izrazima:

1. Ulančavanje, odnosno kartezijev produkt (AB), koji sadrži sve nizove gdje se prvo pojavljuje element iz A, a zatim iz B.
2. Unija (A + B), odnosno skup koji sadrži sve nizove gdje se pojavljuje ili element iz A ili iz B.
3. Kleenov operator (A*), koji predstavlja sve nizove sastavljene nizanjem nula ili više znakova iz A.

Ovim skupovima i operatorima može se definirati svaka klasa leksema. Npr. klasa leksema za sve prirodne brojeve može se definirati ovako:

```
ZNAM = “0” + “1” + “2” + “3” + “4” + “5” + “6” + “7” + “8” + “9”
BROJ = ZNAM ZNAM*
```

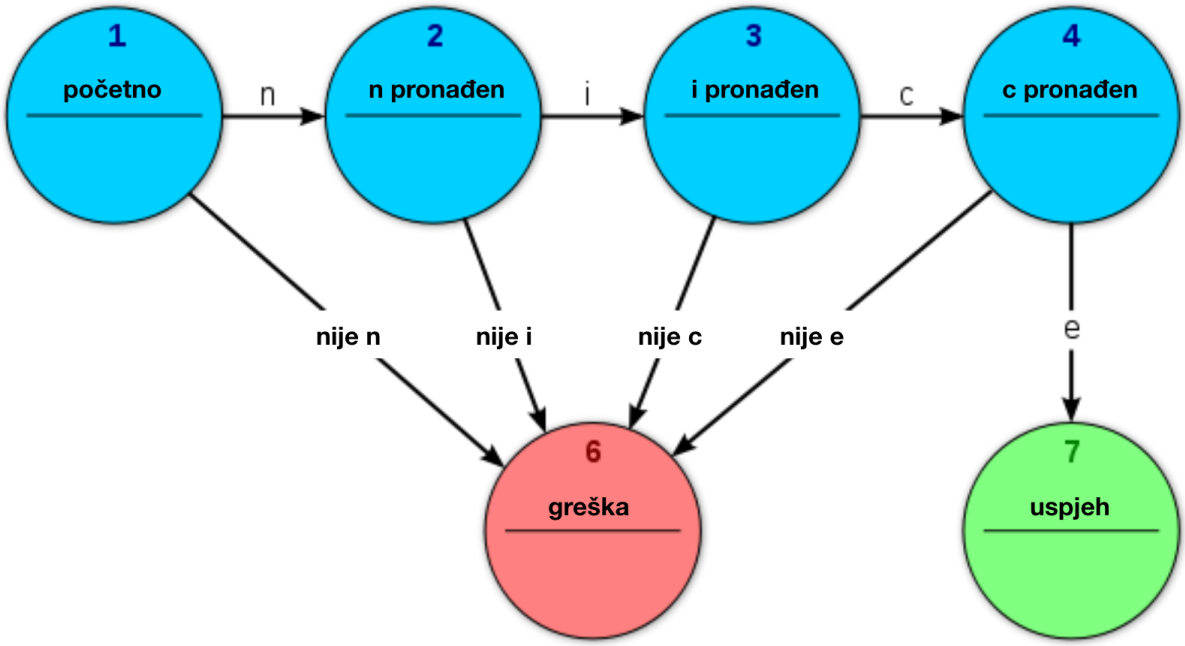
Skup ZNAM definira se kao unija jednoslovnih skupova za svaku znameknu. Nakon toga, definira se skup BROJ koji predstavlja skup svih nizova znakova koji počinju nekim elementom iz ZNAM i nakon toga mogu sadržavati nula ili više elemenata iz ZNAM.

3.2 Leksički analizator

Regularni se izrazi često implementiraju kao konačni automati. Konačni automat je matematički model koji se sastoji od konačnog broja različitih stanja i prijelaza između tih stanja. Svaki regularni izraz se može prikazati pomoću konačnog automata, tj. vrste konačnih automata koja se zove prihvatitelj ili prepoznavatelj.

Kod prihvatitelja postoje dvije posebne vrste stanja. Uvijek postoji jedno početno stanje na početku prepoznavanja. Postoji jedno ili više prihvatljivih stanja, odnosno stanja u kojima se prepoznalo ono što se pokušava prepoznati. Za svako stanje definiraju se mogući prijelazi u druga stanja, ovisno o tome koji je sljedeći ulaz. Ti se prijelazi mogu zapisati u tablicu prijelaza ili na dijagramu stanja. Prihvatitelj čita jedan po jedan znak iz niza znakova i ovisno o definiranim prijelazima prebacuje stanja. Ako se na kraju niza znakova automat nalazi u prihvatljivom stanju, tada je niz znakova prihvaćen. Inače je odbijen.

Na slici 3.1 prikazan je primjer dijagrama stanja konačnog automata koji prepoznaje riječ “nice”. Počinje se u početnom stanju (1) i ako se primi slovo “n”, prelazi se u stanje 2. Ako se primi bilo koje drugo slovo, prelazi se u stanje 6 koje je neprihvatljivo stanje. Sličan proces odvija se za sljedeća stanja dok se ne dođe do stanja 4. Ako je automat u stanju 4 i primi znak “e”, prelazi u stanje 7, koje je prihvatljivo stanje. U tom stanju se zna da se na ulazu nalazi “nice”.



Slika 3.1 *Primjer prihvatitelja za riječ “nice.”*

Moderni prevoditelji većinom ne razlikuju fazu leksičke specifikacije od sintaksne analize i ne specificiraju jezik na ovaj način. Suvremeni jezici često imaju kompleksne sintakse koje zahtijevaju više slobode u kodu koji dijeli ulaz na leksičke jedinke, pa ne koriste regularne izraze nego kod koji direktno djeluje na ulaz. Nedostatak tog pristupa je taj što je sintaksa jezika (ključne riječi, operatori itd.) čvrsto povezana sa semantikom jezika (značenje ključnih riječi), tako da mijenjanje jednog zahtjeva promjenu drugog [4].

Neki skup R naziva se specifikacija jezika ako vrijedi da je R unija svih regularnih izraza u nekom jeziku poredana po prioritetu leksičkih jedinki. Npr. ključnu riječ “if” stavit će se prije klase imena varijabli, da se onemogući varijabla imena “if”. Na ulazu analizatora se nalazi niz znakova x1…xn. Postoji prazan niz leksičkih jedinki A.

Algoritam leksičke analize je sljedeći:

1. Za svaki i od 0 do n, provjeri se je li prefiks ulaznog niza x1…xi element skupa R.
2. Ako se nalazi u R, prefiks x1…xi izbriše se s početka ulaza i jedinka se sprema u A.
3. Ako se više od jednog prefiksa nalazi u R (x0…xi i x0…xi + n), odabire se duži prefiks. (npr. ako je na ulazu "==", a u jeziku postoje i "=" i "==", odabire se "==")
4. Ako ni jedan prefiks nije element R, dio ulaza se ne uklapa u specifikaciju jezika, pa se izbacuje greška.
5. Vraća se na 1 dok ulaz nije prazan.

Tim algoritmom na kraju leksičke analize dobije se niz leksičkih jedinki nad kojim se može raditi sintaksna analiza. Dio programskog koda koji implementira takav algoritam i zadužen je za raščlambu ulaza na leksičke jedinke naziva se leksički analizator (engl. *lexer*).

3.3 Sintaksna analiza

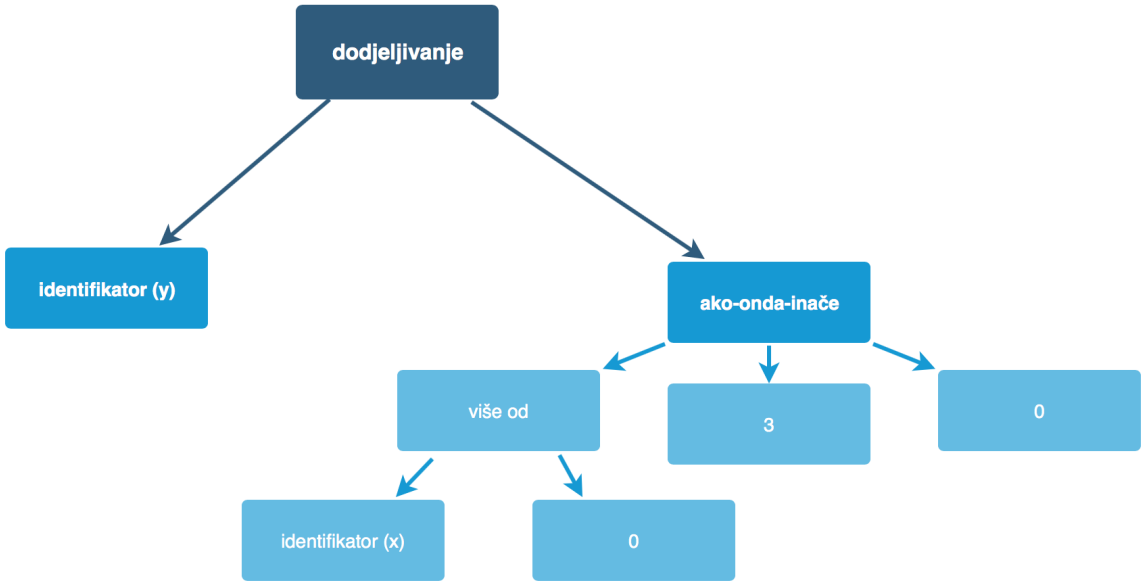
Druga faza prevođenja programskog jezika je sintaksna analiza. U toj se fazi niz tokena iz leksičkog analizatora pretvara u apstraktno sintaksno stablo (ASS). ASS je struktura podataka koja se koristi kao prikaz izvornog koda [2]. Svaki čvor ASS-a sadrži jedan izraz jezika (npr. *if-then-else*), dok njegovi potomci sadrže podizraze. Takva je struktura pogodna za prikaz programskih jezika, jer većinom imaju hijerarhijsku strukturu izraza, gdje se prvo moraju izvršiti podizrazi a zatim sam izraz [2], naprimjer, ako se gleda 'ako-onda-inače' strukturu koja se nalazi u puno programskih jezika.

```
y = ako x > 0 onda 3 inače 0
```

Ovakav se kod tijekom leksičke analize može raščlaniti na 10 različitih tokena, uključujući dva identifikatora, dva operatora, tri ključne riječi i tri broja. Takva struktura ne govori ništa o samom značenju koda, pa se prevodi u sintaksno stablo koje je prikazano na slici 3.2. Sam kod je zapravo izraz dodjeljivanja, koji ima svoju lijevu i desnu stranu. S lijeve strane nalazi se ime varijable kojoj se dodjeljuje vrijednost, a sa desne strane izraz čiji je rezultat vrijednost koju treba dodijeliti. Taj izraz je ako-onda-inače izraz koji ima 3 podizraza: uvjet, ako i inače. Uvjet se sastoji od operatora više od i 2 operanda, dok su ostali izrazi samo brojevi.

Ovakva struktura se naziva apstraktnom jer ne ovisi o samoj sintaksi jezika. Ključne riječi, operatori, raspored zagrada i ostali sintaksni detalji mogu se mijenjati bez da se promijeni izgled

sintaksnog stabla. Zbog toga su sintaksa stabla pogodan način prikazivanja izvornog koda za prevođenje u drugi jezik.



Slika 3.2 *Sintaksno stablo za grananje*.

Za sintaksnu analizu potrebno je prvo definirati gramatiku jezika. Općenito, prema [5] gramatika se sastoji od 3 dijela: skupa terminalnih znakova (N), skupa neterminalnih znakova (T) i skupa produkcija (P). Terminalni znakovi su osnovni elementi jezika i još se nazivaju abeceda. Neterminalni znakovi nisu dio samog jezika već se koriste za definiranje dijelova jezika. Produkcije su gramatička pravila definirana kao skup zamjena jednog niza terminala i neterminala u drugi niz.

Naprimjer, ako se zamisli jezik za aritmetiku koji uključuje zbrajanje i množenje, terminali tog jezika su “+”, “*”, “(“, “)” i brojevi od 0 do 9. Neterminali su bilo koji izraz E, izraz “(E)”, “E + E”, i “id”. Moguće produkcije takve gramatike bile bi sljedeće:

```
E -> E + E
E -> E * E
E -> (E)
E -> znamenka
```

Gramatika služi za generiranje rečenica jezika, tako da se jednu gramatički točnu rečenicu ovoga jezika može generirati tako što se odabere proizvoljni rezultat produkcije od početnog simbola “E” i na njemu vrše zamjene s rezultatima produkcije sve dok se ne dođe do niza znakova koji

sadrži samo neterminalne znakove. Zamjena neterminala s rezultatom produkcije naziva se derivacija neterminala. Svaka rečenica jezika može se dobiti nizom derivacija od početnog neterminala. Tako da su sljedeće rečenice gramatički točne za gore navedenu gramatiku:

```
3 + 2
3 + 2 * 5
(3 + 2) * 5
((3) + 2)) * 5
```

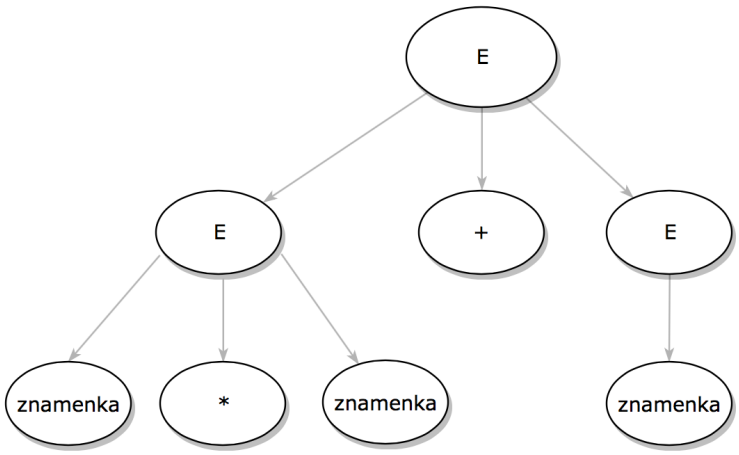
Ali sljedeća rečenica nije točna:

```
3 (+ 2) * 5
```

Ne postoji ni jedan niz produkcija od početnog znaka koji bi dao takvu rečenicu.

Gramatike su alat kojim se definiraju programski jezici. Sintaksna analiza bazira se na provjeri je li neki ulazni niz simbola rečenica unutar zadane gramatike. Jedan od najjednostvnijih algoritama za sintaksnu analizu je reukurzivna silazna sintaksna analiza [5]. Ona se temelji na izradi stabla izvođenja počevši od vrhovnog čvora stabla do završnih listova. Nakon sintaksne analize, čitanjem listova stabla s lijeva na desno može se dobiti ulazni niz znakova.

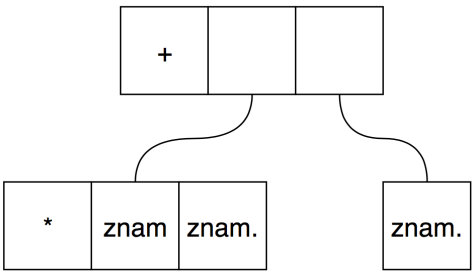
Izvršno stablo započinje početnim simbolom gramatike, a svaka grana predstavlja jednu produkciju od roditeljskog čvora do potomka. Kao što je prije rečeno, svaka rečenica gramatike nastaje nizom produkcija od početnog simbola. Izvršno stablo pokazuje te produkcije u obliku stabla. Npr. za rečenicu “3 * 2 + 5” može se konstruirati ovakvo izvršno stablo koje je prikazano na slici 3.3.



Slika 3.3 *Izvršno stablo za niz znakova“3 * 2 + 5”.*

Rekurzivna silazna sintaksna analiza konstruira ovakvo stablo tako što redom pokušava izvršiti sve produkcije gramatike od početnog znaka. Nakon svakog koraka se provjeri odgovara li produkcija ulaznom nizu. Ako ne, vraća se korak unazad i pokušava druga produkcija. Ako se izvrše sve moguće produkcije a i dalje se nije dobio ulazni simbol, završava se algoritam sa greškom. Ako algoritam završi uspješno, na kraju se dobije konstruirano izvršno stablo za ulazni niz.

Kreiranje apstraktnog sintaksnog stabla od izvršnog stabla je trivijalno. Apstraktno sintaksno stablo ima sličnu strukturu kao izvršno, ali suvišne grane se mogu izbaciti. Npr. nije potrebno imati granu od E do E, pa opet od E do znamenke, nego postoji samo grana od E do znamenke, i sl. Za gore stablo prikazano na slici 3.3, apstraktno sintaksno stablo bi izgledalo kao što je prikazano na slici 3.4. Dio programa zadužen za sintaksnu analizu naziva se sintaksni analizator [6].



Slika 3.4 *Apstraktno sintaksno stablo za niz znakova"3 * 2 + 5".*

3.4 Generiranje koda

Posljednji proces kojeg prevoditelj obavlja je generiranje koda. To je proces kreiranja koda u odredišnom jeziku iz apstraktnog sintaksnog stabla dobivenog kao rezultat prethodnih koraka prevođenja. Rezultat nakon ovog koraka je programski kod u drugom programskom jeziku koji je semantički ekvivalentan izvornom kodu [6].

Generator koda ima visoke zahtjeve koje mora zadovoljiti. Generirani kod mora biti sematički jednak izvornom kodu, a da pri tome izvršavanje prevedenog koda bude brzo i učinkovito, te da sam proces generiranja koda isto tako bude brz.

Krajnji oblik svakog programskog jezika je strojni kod, odnosno procesorska instrukcija u binarnom obliku koju procesor izravno izvršava. Problem kod generiranja strojnog koda je što strojni kod ovisi o procesorskoj arhitekturi na kojoj se izvršava, različiti procesori koriste

različiti strojni jezik. Stoga većina modernih prevoditelja ne prevodi kod izravno u strojni jezik, nego u posredni jezik, kojeg onda prevodi u strojni kod procesom koji se zove odabir instrukcija. Posredni jezici većinom nalikuju na strojni jezik, ali ne ovise o karakteristikama procesora na kojemu se nalaze. Primjeri posrednih jezika su Java bytecode, LLVM Intremediate Representation, CIL (.NET), WebAssembly i sl.

Postoje različiti alati za pretvaranje posrednih jezika u strojni kod. Oni interpretiraju posredni jezik ili ga pretvaraju u strojni kod za arhitekturu na kojoj se trenutno nalaze. Među popularnijim takvim alatima u LLVM i JVM. U ovom se radu koristi LLVM i izlaz iz prevoditelja je LLVM Intermediate Representation (IR).

Tijekom generiranja koda obavlja se nekoliko procesa. Prvi od njih je odabir instrukcija. Nakon što se uzme neki oblik posrednog prikaza programa (ili sintaksno stablo ili posredni jezik), potrebno je za svaku instrukciju prikaza odabrati instrukcije odredišnog strojnog koda koje odgovaraju izvornom prikazu. Ovaj se proces većinom radi zamjenom predložaka pronađenih u posrednom prikazu sa nizom instrukcija. Nakon toga se još jednom prođe kroz generirani kod kako bi se pronašle dodatne prilike za optimizaciju.

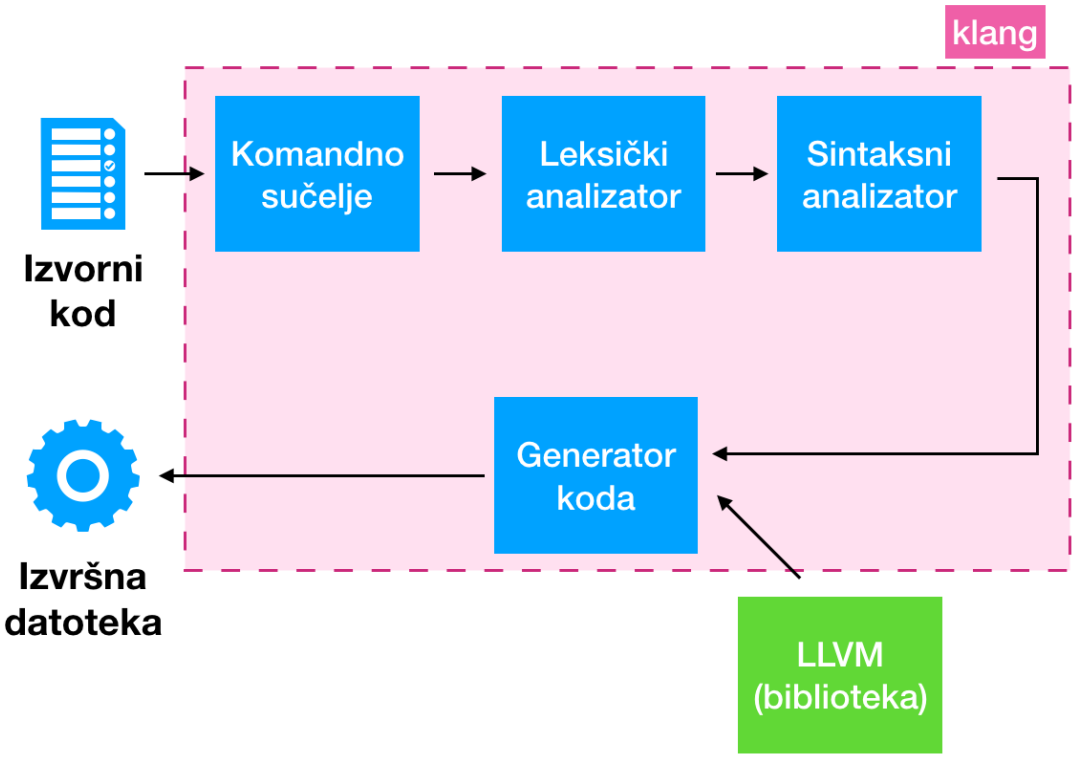
Nakon odabira instrukcija, odvija se raspoređivanje instrukcija. Neke instrukcije se odvijaju brže od drugih, ulaz nekih ovisi o izlazu drugih, a neke uzrokuju zastoje zbog ograničenja procesa i memorije. Zadatak raspoređivanja instrukcija je odrediti koji je optimalan redoslijed za izvršavanje instrukcija da bi se izbjegli problemi sa zastojima i ovisnostima između instrukcija.

Pošto se rasporede instrukcije, alociraju se registri. Registri procesora koriste se za spremanje koda i varijabli. Ovisno o tome koliko je registara dostupno te koliko se koriste varijable i koliko zauzimaju memorije alocira se određen broj registara za korištenje tijekom izvršavanja programa.

Ovisno o okruženju u kojemu se program izvršava, nekad su potrebni dodatni podaci o programskom kodu (imena varijabli, njihova lokacija, imena funkcije itd.) u svrhu pronalaska greški u programu. Stoga se nekad tijekom generiranja koda generiraju i pomoćni dodatni podaci o kodu, koji pomažu programerima tijekom rada.

4. IMPLEMENTACIJA PROGRAMSKOG PREVODITELJA KLANG

U ovom je poglavlju prikazana implementacija prevoditelja "klang". Prevoditelj ima zadatak prevođenja programskog jezika Kaleidoscope. Najteži dio implementacije bio je izabrati pogodne strukture podataka za prikaz različitih dijelova sintakse jezika. Taj se problem najviše očituje kod operatora (aritmetičkih i operatora za usporedbu). Za binarne operatore moraju se spremiti dvije vrijednosti, ona s lijeva i s desna. To znači da se mora gledati ulazni niz jedan znak unaprijed, kako bi se mogla spremiti vrijednost s lijeve strane operatora. Srećom, Swift ima posebnu strukturu podataka enumeracije s pridodanom vrijednosti, koja omogućuje spremanje dodatnih podataka uz samu kategoriju operatora, tako da je taj problem elegantno riješen. Dijagramski prikaz prevoditelja prikazan je na slici 4.1



Slika 4.1 *Shematski prikaz prevoditelja.*

4.1 Kaleidoscope

Kaleidoscope je vrlo jednostavan proceduralni jezik opisan u [7], koji podržava isključivo brojčane varijable i to 64-bitne brojeve s pomičnim zarezom (*double* u C-u). Jezik ima jednostavnu sintaksu nalik na Python. Sve funkcije moraju završavati izrazom koji vraća

brojčanu vrijednost. Jezik podržava i pozivanje vanjskih funkcija definiranih ključnom riječi *extern*. Slika 4.2 prikazuje neke primjere Kaleidoscope koda.

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

fib(40)
```

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(0.4), cos(42))
```

Slika 4.2 *Primjeri Kaleidoscope koda.*

Kaleidoscope također podržava standardne aritmetičke operatore koji se nalaze u većini programskih jezika (zbrajanje i oduzimanje, množenje i dijeljenje, modulo itd.). Podržava kontrolu toka sa ako-onda-inače izrazima. Ključnom rječju "def" definiraju se nove funkcije koje se onda mogu pozivati u najvišoj razini koda, dakle ne postoji glavna funkcija kao u nekim jezicima. Izrazi koji se izvršavaju jedan nakon drugog na istoj razini moraju se odvojiti točkom sa zarezom (";").

4.2 Swift

Ovaj prevoditelj implementiran je u programskom jeziku Swift. Swift je suvremeni programski jezik koji nudi značajke pogodne za izradu prevoditelja: brz je, siguran, koncizan i ima implementaciju *Stringa* prema UTF-8 standardu.

Podržava više programskih paradigmi (funckijsko, proceduralno i objektno orijentirano programiranje) i tehnike iz svake paradigme koriste se za implementaciju prevoditelja. Swift je pogodan za programiranje sustava i komponenti niže razine jer lagano radi s funkcijama koje su napisane u programskom jeziku C, a može postići slične performanse kao C.

4.3 LLVM Projekt

Kao pozadinsko sučelje prevoditelja koristi se LLVM Projekt (engl. *LLVM Project)*. LLVM je biblioteka i skup alata za generiranje i optimiziranje koda za različite procesorske arhitekture. LVVM u sebi uključuje i C prevoditelj (*clang*), JIT prevoditelj, sakupljanje smeća i sl. LLVM funkcionira tako što primi programski jezik LLVM IR (*Intermediate Representation*) i njega prevodi u jezik pogodan za procesorsku arhitekturu na kojoj se nalazi [8]. LVVM je jedan od najpopularnijih alata za izradu programskih jezika i prevoditelja. Koriste ga Swift, Objective-C, Rust, Emscripten, Clang, Kotlin Native, PlayStation, OpenCL i dr. [9].

LLVM omogućuje optimiziran rad programa na svim poznatim procesorskim arhitekturama, bez potrebe za izravnim generiranjem strojnog koda za svaku pojedinu arhitekturu. Zadatak je Swift dijela prevoditelja generirati LLVM IR kod iz Kaleidoscope koda, predati taj IR LLVM-u i na kraju dobiti objektni kod. Taj objektni kod sadrži funkciju *main*, koju izvršava LLVM interpreter.

4.4 Detalji implementacije

Sam predovitelj implementiran je kao komandno sučelje za macOS i Linux, gdje korisnik odredi koju datoteku želi prevesti i kao rezultat dobije rezultat izvođenja programa koji se izvrši preko LLVM interpretera. Na primjer:

```
$ ./CompileKal "someprogram.kal"
$ result: 32.0000
```

U glavnom dijelu programa se prvo učita sadržaj datoteke sa podatkovnog sustava, kao što je prikazano na slici 4.3.

```
func getCode()-> String? {
  let args = CommandLine.arguments

  guard args.count > 1 && args[1] != "-h" else {
    print(helpText)
    return nil
  }

  let filePath = args[1]
  do {
    return try String(contentsOfFile: filePath)
  } catch {
    print("The provided file could not be read:")
    print(error)
    print(helpText)
    return nil
  }
}

func run() {
  guard let code = getCode() else {
    return
  }
```

Slika 4.3 *Učitavanje koda.*

Nakon što je dobavljen sadržaj datoteke, može se vršiti leksička analiza, čiji se rezultat prosljeđuje sintaksnom analizatoru, dok se njegov rezultat dalje prosljeđuje generatoru koda.

Generator koda će ispisati LLVM IR kod u datoteku na podatkovnom sustavu. Nakon toga se poziva LLVM-ov interpreter s tom datotekom, koji će izvesti program i rezultat izvedbe ispisati na konzolu. To je prikazano na slici 4.4.

```
  let tokens = Lexer.lex(input: code, definition: Syntax.definition)
  let parser = Parser(tokens: tokens)

  do {
    let fileData = try parser.parseFile()
    let codeGen = CodeGen(fileData)
    try codeGen.generate(toPath: codeGenPath)
    Interpreter.interpret(fromFilePath: codeGenPath)
  } catch {
    print(error)
  }
}
```

Slika 4.4 *Proces prevođenja.*

4.5. Implementacija leksičkog analizatora

Leksički analizator za zadatak ima raščlaniti ulazni niz znakova (u Swiftu *String*) na niz (odnosno *Array)* leksičkih jedinki. Različite leksičke jedinke prikazuju se kao Swift *enum* vrijednosti, kao što je prikazano na slici 4.5.

```
enum Token {
  enum Operator {
    case plus
    case minus
    case divide
    case times
    case modulo
    case greater
    case less
    case greaterEqual
    case lessEqual
    case equal
    case notEqual
    case not
  }

  case whitespace

  // Values

  case identifier(String)
  case value(Double)
  case comment(String)

  // Characters

  case openParen
  case closeParen
  case comma
  case eof

  // Keywords

  case `if`, then, `else`, extern, `for`, `in`, def

  // Operators

  case op(Operator)
}
```

Slika 4.5 *Opis leksičkih jedinki.*

Moraju se definirati svi operatori koji se koriste (za usporedbu i za aritmetiku), sve ključne riječi, svi simboli u jeziku poput zareza, zagrada i sl., te razmaci i kraj datoteke.

Swift *enum* vrijednosti mogu imati pridodane vrijednosti, tako npr. slučaj "value", koji predstavlja broj, može imati pridodanu vrijednost tog broja. Pridodane su vrijednosti slične poljima na klasi, ali vežu se uz jedan slučaj enuma. Slučaj "value" može imati jednu pridodanu vrijednost tipa "Int", dok slučaj "identifier" može imati pridodanu vrijednost tipa "String" koja predstavlja ime te varijable. Te vrijednosti su potrebne u kasnijim dijelovima izrade prevoditelja.

Nakon definiranja leksičkih jedinica koje postoje u jeziku, pravi se specifikacija samog jezika, odnosno definiraju se regularni izrazi kojima se opisuju leksičke jedinice. Kod implementacije leksičkog analizatora, bitno je znati duljinu svake pronađene jedinke, kako bi se taj broj znakova izbacio iz ulaza kod vršenja raščlambe. Zato, za svaku se leksičku jedinku mora znati njezina dužina.

U Swiftu se takva struktura može prikazati kao funkcija koja za ulaz ima *String*, a izlaz iz te funkcije je ili kreirana leksička jedinica, ili prazna vrijednost, odnosno *nil*. Specifikacija jezika se u tom slučaju prikazuje kao niz takvih funkcija, po jedna za svaku leksičku jedinicu. Ti su tipovi prikazani na slici 4.6.

```
typealias TokenDefinition = (String)-> (Token, length: Int)?
typealias LanguageSpec = [TokenDefinition]
```

Slika 4.6 *Tipovi definicije jezika.*

Kreirane su i dvije pomoćne funkcije koje će olakšati kreiranje funkcija za definciju leksičkih jedinki. Potrebna je jedna funkcija za definiranje leksičkih jedinki koje sadržavaju samo jedan mogući leksem, kao što su ključne riječi i operatori. Također je potrebna jedna funkcija za definiranje jedinki koje mogu biti više različitih riječi. Obje funkcije će kao parametar primiti regularni izraz kao *String*, a kao izlaz će vratiti funkciju za definiranje tokena. Funkcije su prikazane na slici 4.7.

```
/// If found a regex match, return the token, otherwise return nil.
func const(_ regex: String, _ token: Token)-> TokenDefinition {
  return { input in
    guard input.match(regex: regex) != nil else {
      return nil
    }

    return (token, regex.filter { $0 != "\\" }.count)
  }
}

/// If found a regex match, create a token based on that match. Otherwise return nil.
func match(_ regex: String, _ makeToken: @escaping (String)-> Token)-> TokenDefinition {
  return { input in
    guard let match = input.match(regex: regex) else {
      return nil
    }

    return (makeToken(match), match.count)
  }
}
```

Slika 4.7 *Pomoćne funkcije za definiranje leksičkih jedinki.*

Tim se funkcijama može definirati specifikacija jezika prikazana na slici 4.8.

```
class Syntax {

  /// Language specification.
  static let definition:
LanguageSpec = [

    // Whitespace

    match("[ \t\n]", { _
in .whitespace}),

    // Symbols

    const("\\(", .openParen),
    const("\\)", .closeParen),
    const("\\,", .comma),

    // Keywords

    const("def", .def),
    const("if", .if),
    const("then", .then),
    const("else", .else),
    const("extern", .extern),

    // Operators

    const("\\*", .op(.times)),
    const("\\+", .op(.plus)),
    const("\\-", .op(.minus)),
    const("\\/", .op(.divide)),
    const("\\%", .op(.modulo)),
    const("==", .op(.equal)),

const(">=", .op(.greaterEqual)),
    const("<=", .op(.lessEqual)),
    const(">", .op(.greater)),
    const("<", .op(.less)),
    const("!=", .op(.notEqual)),
    const("!", .op(.not)),

    // Values

    match("[a-zA-Z][a-zA-Z0-9]*",
          { .identifier($0) }),
    match("[0-9](\\.[0-9]*)?",

{ .value(Double($0)!) })
  ]
}
```

Slika 4.8 *Definicija jezika.*

Za svaku leksičku jedinku definira se funkcija koja će analizirati ulazni niz znakova, i ako ulazni niz zadovolji uvjet, kreira se leksička jedinka. Ako ulazni niz ne zadovoljava uvjet regularnog izraza, funkcija će vratiti *nil*.

Sam leksički analizator prikazuje se kao klasa s nazivom *Lexer*, koja će imati jednu javnu statičku metodu *lex*, koja prima niz ulaznih znakova i vraća niz leksičkih jedinki. Implementacija te funkcije slična je općenitom algoritmu leksičkog analizatora koji je prije spomenut, a prikazana je na slici 4.9.

```
class Lexer {
  static func lex(input: String, definition: LanguageSpec) -> [Token] {

    var tokens = [Token]()
    var content = input

    while content.count > 0 {
      guard let (token, length) = getToken(content, definition) else {
        let index = content.index(after: content.startIndex)
        content = String(content[index...])
        continue
      }

      let index = content.index(content.startIndex, offsetBy: length)
      content = String(content[index...])

      if case .whitespace = token {
        continue
      }

      tokens.append(token)
    }

    return tokens
  }
}
```

Slika 4.9 *Klasa Lexer.*

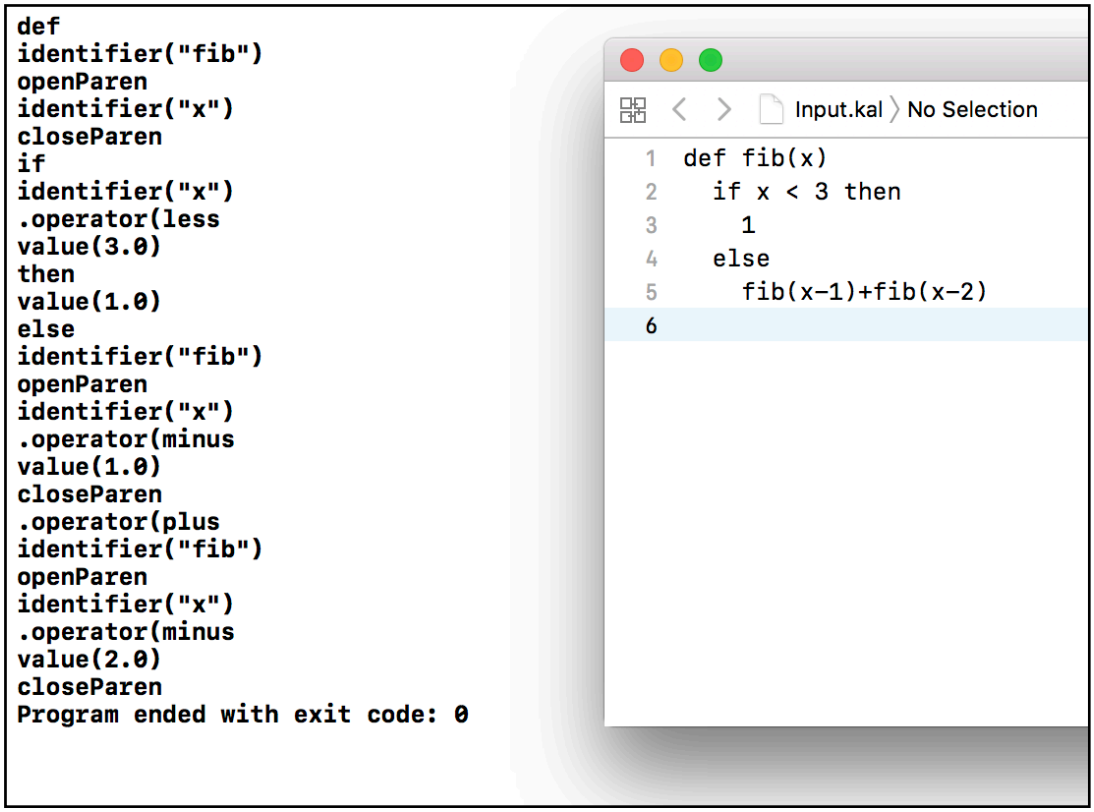
Dok ulazni niz nije prazan, pokušava se pronaći leksička jedinka u tekstu. Ako nije pronađena jedinka, pomiče se jedan znak unaprijed i pokušava opet. Ako je jedinka pronađena, ubacuje se u niz, osim ako je *whitespace*, odnosno razmak.

Preostalo je još definitati pomoćnu funkciju *getToken* koja će na ulaznom nizu pokušati, redom, pretražiti definicije leksičkih jedinki dok ne pronađe prvu, i nju će vratiti. Ako ne pronađe ni jednu, vratiti će *nil*, kao što je prikazano na slici 4.10.

```
private static func getToken(_ input: String, _ definition: LanguageSpec)-> (Token,
Int)? {
  for test in definition {
    if let token = test(input) {
      return token
    }
  }

  return nil
}
```

Slika 4.10 *Pomoćna funkcija za testiranje leksičkih jedinki u ulaznom nizu.*

Sada se sa sadržajem ulazne datoteke može pozvati funkciju *lex*, i vidjeti tokene iz ulaznog niza. Na slici 4.11 prikazan je primjer ulazne datoteke, i ispisa izlaza iz funkcije *lex* na konzolu.



Slika 4.11 *Primjer izlaza iz leksičkog analizatora.*

4.6 Implementacija sintaksnog analizatora

Za Kaleidoscope je dovoljan sintaksni analizator koji koristi rekurzivni silazni algoritam sintaksne analize. Kao izlaz iz sintaksnog analizatora koristi se struktura podataka koja predstavlja jednu datoteku programskog koda. Jedna datoteka sastoji se od deklaracija vanjskih funkcija, definicija funkcija i niza vrhovnih naredbi, odnosno izraza. Struktura je prikazana na slici 4.12.

```
struct File: CustomStringConvertible {
  var externalFunctions: [FunctionDeclaration] = []
  var functions: [Function] = []
  var expressions: [Expression] = []
  var declarations: [String: FunctionDeclaration] = [:]
}
```

Slika 4.12 *Klasa "File".*

Za izraze programskog jezika koristi se enum koji je indirektan, odnosno vrijednosti enuma mogu sadržavati same sebe. Ovo je korisno jer su izrazi u programskim jezicima većinom rekurzivni, odnosno jedan izraz može sadržavati neki drugi. Enum je prikazan na slici 4.13.

```
indirect enum Expression {
  case number(Double)
  case identifier(Identifier)
  case `operator`(Token.Operator, lhs: Expression, rhs: Expression)
  case functionCall(name: Identifier, args: [Expression])
  case ifThenElse(condition: Expression, `if`: Expression, `else`: Expression)
}
```

Slika 4.13 *Enum "Expression".*

Osim izraza, u Kaleidoscopeu postoje i deklaracije funkcija. Funkcije mogu biti definirane izvana (iz neke druge datoteke ili iz C-a) ili mogu biti definirane u samoj datoteci. Stoga se koriste dvije strukture, jedna za deklaraciju funkcija i druga za definiciju funkcija. Obzirom da u Kaleidoscopeu svaka funkcija mora vratiti jednu vrijednost, tijelo funkcije se može prikazati kao jedan izraz. To je prikazano na slici 4.14.

```
struct FunctionDeclaration {
  let name: Identifier
  let params: [Identifier]
}

struct Function {
  let declaration: FunctionDeclaration
  let body: Expression
}
```

Slika 4.14 *Strukture za funkciju i deklaraciju funkcije.*

Sam sintaksni analizator prikazuje se kao jedna Swift klasa. Sintaksni analizator kreiran je s nizom tokena koji su izlaz iz leksičkog analizatora. Sintaksni analizator potom iterira kroz sve tokene i na osnovu heuristika kod određenih tokena pokušava analizirati nadolazeći niz tokena. Npr. na pojavu ključne riječi "def" može se pretpostaviti da nadolazi definicija funkcije, tako da se može pokušati pronaći definicija funkcije. Ako nadolazeći niz tokena ne odgovara definiciji funkcije, izbacuje se greška. Analiza takvih slučaja prikazana je na slici 4.15.

Ovdje se očituje mana rekurzivnog silaznog leksičkog analiziranja. Ako je gramatika jezika takva da je za "def" moguće imati više čvorova sintaksnog stabla, algoritam nam postaje puno kompliciraniji. Srećom, gramatika Kaleidoscopea je dovoljno jednostavna da se može koristiti ovakav algoritam.

Leksička analiza pojedinih čvorova sintaksnog stabla prikaza je na slici 4.16 i radi se postepenim preskakanjem tokena koji odgovaraju određenom uvjetu. Npr. za vanjsku funkciju u nizu tokena moraju se nalaziti, jedno iza drugog, ključna riječ "extern", deklaracija funkcije i točka-zarez. Stoga se koristi funkcija "*consume*" iz slike 4.15, koja će pokušati pronaći primljen token na trenutnom indeksu i ako ga pronađe, inkrementirat će trenutni indeks na kojem se analizator nalazi.

```
class Parser {

  private let tokens: [Token]

  init(tokens: [Token]) {
    self.tokens = tokens
  }

  // State

  private var index = 0

  private var currentToken: Token? {
    return index < tokens.count ? tokens[index] : nil
  }

  private func consumeToken(n: Int = 1) {
    index += n
  }

  private func consume(_ token: Token) throws {
    guard let tok = currentToken else {
      throw "Unexpected end of file!"
    }
    guard token == tok else {
      throw "Unexpected token \(token)"
    }
    consumeToken()
  }

  func parseFile() throws -> File {
    var file = File()
    while let tok = currentToken {
      switch tok {
      case .extern:
        file.externalFunctions.append(try parseExtern())
      case .def:
        file.functions.append(try parseFunction())
      default:
        let expr = try parseExpression()
        try consume(.semicolon)
        file.expressions.append(expr)
      }
    }
    return file
  }
```

Slika 4.15 *Klasa "Parser".*

Analiza izraza funkcionira na sličan način kao analiza cijele datoteke. Obzirom na trenutni token zaključuje se o kojem je izrazu riječ i pokušava se pronaći taj izraz. Svaki taj zaključak ekvivalentan je jednoj produkciji gramatike, kao što je prikazano na slici 4.17

```
  private func parseExtern() throws -> FunctionDeclaration {
    try consume(.extern)
    let declaration = try parseDeclaration()
    try consume(.semicolon)
    return declaration
  }

  private func parseFunction() throws -> Function {
    try consume(.def)
    let declaration = try parseDeclaration()
    let expr = try parseExpression()
    let function = Function(declaration: declaration, body: expr)
    try consume(.semicolon)
    return function
  }

  private func parseDeclaration() throws -> FunctionDeclaration {
    let name = try parseIdentifier()
    let params = try parseList(parseIdentifier)
    return FunctionDeclaration(name: name, params: params)
  }
```

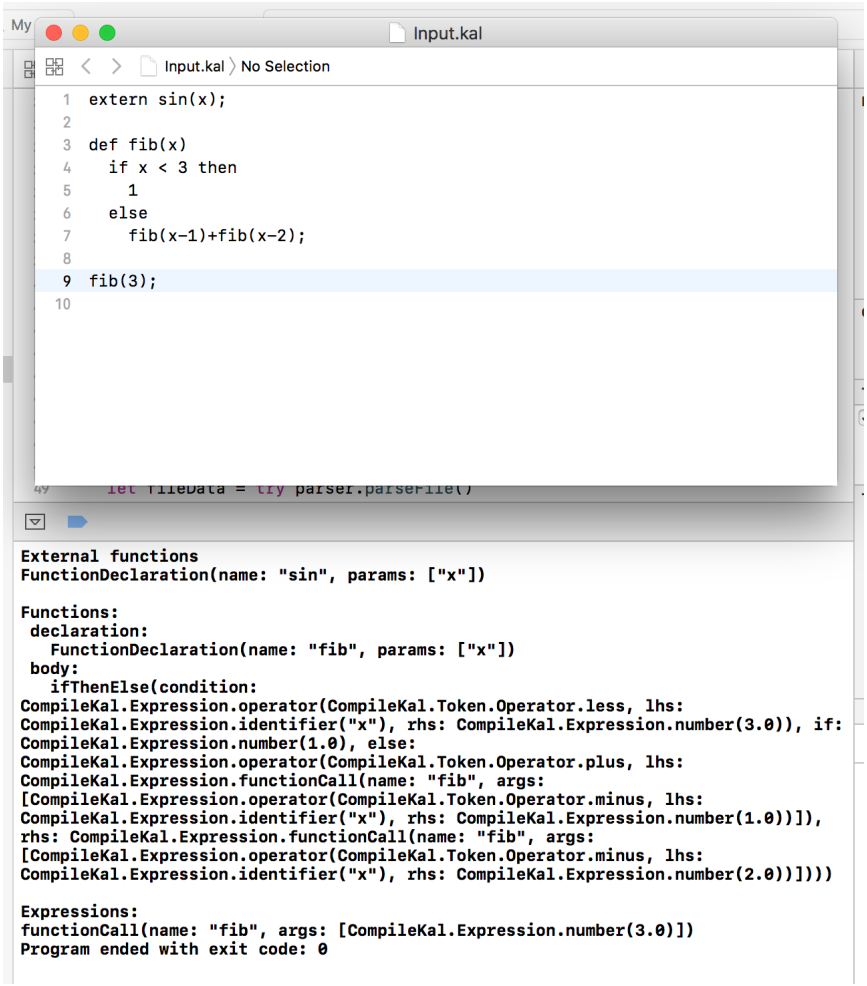
Slika 4.16 *Funkcije za analizu pojedinih čvorova stabla.*

Javno sučelje klase "Parser" je samo funkcija "parseFile", tako da se ona koristi za sintaksnu analizu niza tokena iz analizatora. Na kraju procesa sintaksne analize dobije se izlaz u obliku strukture "File", kao što je prikazano na slici 4.18.

```
private func parseExpression() throws -> Expression {
    guard let token = currentToken else {
      throw "Unexpected end of file!"
    }

    var expr: Expression

    switch token {
    case .openParen: // ( <expr> )
      consumeToken()
      expr = try parseExpression()
      try consume(.closeParen)
    case .value(let value):
      consumeToken()
      expr = .number(value)
    case .identifier(let value):
      consumeToken()
      if case .openParen? = currentToken {
        let args = try parseList(parseExpression)
        expr = .functionCall(name: value, args: args)
      } else {
        expr = .identifier(value)
      }
```

```
    case .if: // if <expr> then <expr> else <expr>
      consumeToken()
      let cond = try parseExpression()
      try consume(.then)
      let thenVal = try parseExpression()
      try consume(.else)
      let elseVal = try parseExpression()
      expr = .ifThenElse(condition: cond, if: thenVal, else: elseVal)
    default:
      throw "Unexpected token \(token)"
    }

    if case .op(let op)? = currentToken {
      consumeToken()
      let rhs = try parseExpression()
      expr = .operator(op, lhs: expr, rhs: rhs)
    }

    return expr
  }
```

Slika 4.17 *Sintaksna analiza izraza.*



Slika 4.18 *Primjer izlaza iz sintaksnog analizatora.*

4.7 Generiranje koda uz pomoć LLVM-a

Prevoditelj koristi LVVM kao pozadinsko sučelje za generiranje koda. Pozadinsko sučelje LLVM očekuje programski kod u posrednom obliku, odnosno u jeziku *LLVM Intermediate Representation*. To je programski jezik niske razine nalik na strojni kod. Za razliku od strojnog koda, LLVM IR ima strogo određivanje tipova što omogućuje dodatne optimizacije koda. LLVM taj posredni prikaz koda kasnije pretvara u strojni kod za arhitekturu na kojoj se nalazi.

LLVM podržava mogućnosti viših programskih jezika kao što su definicije funkcija, vraćanje podataka određenog tipa iz funkcija, pozive funkcija, deklaracije, blokove koda i sl. LLVM IR je također čitljiv ljudskim okom, tako da je lagan za provjeru rezultata prevođenja. Primjer LLVM IR koda prikazan je na slici 4.19.

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

Slika 4.19 *Primjer LLVM IR koda.*

LLVM nudi beskonačno privremenih registara koji se mogu koristiti za dodjeljivanje varijabli. Svaka varijabla se smije dodijeliti samo jednom. Promjene na varijabli rade se tako da se varijabla spremi u novi registar s promijenjenom vrijednosti. Također, sve varijable moraju biti definirane prije nego što se koriste.

Zbog toga generiranje koda od sintaksnog stabla mora započeti od listova stabla. Npr. ako se generira kod za zbrajanje dva izraza, mora se prvo generirati kod za izraz s lijeva i s desna, i spremiti ih u registre. Vrijednost tih registara se može koristiti pri generiranju koda za zbrajanje.

LLVM je skoro u cijelosti napisan u C++-u, ali za svrhe ovog rada koristi se biblioteka LLVMSwift, koja je Swift sučelje za LLVM funkcije u C-u. LVVMSwift, prema [10], nudi funkcije za kreiranje modula koda, novih funkcija, blokova, poziva funkcija i drugih konstukra iz LLVM IR. Na kraju nudi opciju pisanja generiranog koda u datoteku po izboru.

Definirana je klasa za generiranje koda “CodeGen”, prikazana na slici 4.19. Ona će od podataka dobivenih iz sintaktsnog analizatora generirati LLVM IR modul kojeg će onda uz pomoć LLVMSwift biblioteke ispisati u datoteku.

U jednoj datoteci nalaze se deklaracije vanjskih funkcija, definicije funkcija i vrhovne naredbe koje program izvodi, tako da se generiranje koda cijele datoteke svodi na generiranje tih dijelova, kao što je prikazano u slici 4.20.

Za pokretanje samog programa potrebna je “main” funkcija, prikazana na slici 4.21. U main funckiji izvode se sve vrhovne naredbe programa (one koje nisu u funkciji nego u globalnom opsegu) i njihov rezultat ispisuje se u konzolu preko funkcije “printf” iz C-a.

```
class CodeGen {

  private let module: Module
  private let builder: IRBuilder
  private let file: File

  // State

  init(_ file: File) {
    self.file = file
    module = Module(name: "main")
    builder = IRBuilder(module: module)
  }

  func generate(toPath path: String) throws {
    try emitFile()
    try module.verify()
    try module.emitBitCode(to: path)
  }
```

Slika 4.20 *Klasa “CodeGen”.*

Da bi se generirala neka funkcija, mora se znati tip svakog argumenta kao i tip vrijednosti koju funkcija vraća. Nakon toga se kreira kod za deklaraciju funkcije. Zatim se kreira blok koda (ekvivalent bloku strojnog koda sa labelom) na kojeg će program skočiti kad se pozove funkcija. U taj blok generira se kod tijela funkcije. Na kraju bloka generira se kod za izlaz iz funkcije i vraćanje vrijednosti.

```
  private func emitFile() throws {
    for extern in file.externalFunctions {
      _ = emitDeclaration(extern)
    }
    for function in file.functions {
      _ = try emitFunction(function)
    }
    try emitMain()
  }
```

Slika 4.21 *Generiranje koda jedne datoteke.*

Deklaracije vanjskih funkcija rade se na sličan način kao gornji primjer za “printf” funkciju, kao što je prikazano na slici 4.22.

Za generiranje funkcije koristi se funkcija za generiranje deklaracije iz slike 4.23, ali je potrebno generirati još i tijelo funkcije. Tijelo funkcije je specifično jer se u njemu mogu koristiti parametri koje funkcija primi. Zbog toga generator koda mora znati kontekst u kojem se nalazi, odnosno koji parametri su dostupni unutar tijela funkcije. Generiranje funkcije prikazano je na slici 4.24.

```
private func emitPrint()-> LLVM.Function {
  if let function = module.function(named: "printf") {
    return function
  }

  // int printf(char *, ...)
  let argTypes = [PointerType(pointee: IntType.int8)]
  let type = FunctionType(argTypes: argTypes,
    returnType: IntType.int32, isVarArg: true)

  return builder.addFunction("printf", type: type)
}

private func emitMain() throws {
  let type = FunctionType(argTypes: [], returnType: VoidType())
  let function = builder.addFunction("main", type: type)
  let entry = function.appendBasicBlock(named: "entry")
  builder.positionAtEnd(of: entry)

  let format = builder.buildGlobalStringPtr("%f\n")
  let print = emitPrint()

  for expression in file.expressions {
    let value = try emitExpression(expression)
    _ = builder.buildCall(print, args: [format, value])
  }

  builder.buildRetVoid()
}
```

Slika 4.22 *Generiranje “main” funkcije.*

```
private func emitDeclaration(_ declaration: FunctionDeclaration) -> LLVM.Function {
    if let function = module.function(named: declaration.name) {
      return function
    }

    let argTypes = Array(repeating: FloatType.double,
      count: declaration.params.count)
    let funcType = FunctionType(argTypes: argTypes, returnType: FloatType.double)

    let function = builder.addFunction(declaration.name, type: funcType)

    for (var param, name) in zip(function.parameters, declaration.params) {
      param.name = name
    }

    return function
  }
```

Slika 4.23 *Generiranje deklaracije funkcije.*

```
  private func emitFunction(_ function: Function) throws -> IRValue {
    let declaration = emitDeclaration(function.declaration)
    let originalParams = function.declaration.params
    let parameterValues: [String: IRValue] = originalParams
      .enumerated()
      .reduce([:]) { (acc, next) in
        let (index, name) = next
        var acc = acc
        acc[name] = declaration.parameter(at: index)
        return acc
      }

    let entry = declaration.appendBasicBlock(named: "entry")
    builder.positionAtEnd(of: entry)
    let expression = try emitExpression(function.body, parameters: parameterValues)
    builder.buildRet(expression)
    return declaration
  }
```

Slika 4.24 *Generiranje funkcije.*

Preostaje još generiranje samih izraza programa. Generiranje brojčanih vrijednosti je trivijalno i svodi se na generiranje decimalne konstante. Generiranje operatora je jednostavno jer LLVM IR ima ugrađene instrukcije za osnovne aritmetičke operacije i uspoređivanje. Generiranje aritmetičkih operacija i brojeva prikazano je na slici 4.25.

```
private func emitExpression(_ expression: Expression,
  parameters: [String: IRValue] = [:]) throws -> IRValue {

  switch expression {
  case .number(let value):
    return FloatType.double.constant(value)
  case let .operator(op, lhs, rhs):
    let lhs = try emitExpression(lhs, parameters: parameters)
    let rhs = try emitExpression(rhs, parameters: parameters)
    switch op {
    case .plus: return builder.buildAdd(lhs, rhs)
    case .minus: return builder.buildSub(lhs, rhs)
    case .divide: return builder.buildDiv(lhs, rhs)
    case .times: return builder.buildMul(lhs, rhs)
    case .modulo: return builder.buildRem(lhs, rhs)
    case .equal:
      let comparison = builder.buildFCmp(lhs, rhs, .orderedEqual)
      return builder.buildIntToFP(comparison,
        type: FloatType.double, signed: false)
    }
```

Slika 4.25 *Generiranje brojčanih vrijednosti i operatora.*

Za poziv funkcije potrebno je prvo emitirati kod za svaki argument i onda generirati poziv funkcije s tim argumentima. U svrhu osiguravanja od pogreški u kodu, prilikom generiranja funkcije provjerava se da je ta funkcija prethodno definirana, te da je ispravan broj argumenata funkcije.

Za generiranje varijable koristi se kontekst parametara koji je predan funkciji “emitExpression”. Ako varijabla s tim imenom ne postoji u kontekstu, izbacuje se greška, kao što je prikazano na slici 4.26.

```
  case let .functionCall(name, args):
    guard
      let declaration = file.declarations[name],
      declaration.params.count == args.count
    else {
      throw "Undefined function \(name) with \(args.count) arguments."
    }

    let function = emitDeclaration(declaration)
    let args = try args.map { try emitExpression($0, parameters: parameters) }
    return builder.buildCall(function, args: args)
```

Slika 4.26 *Generiranje poziva funkcije.*

Na kraju ostaje još definirati grananje programa. Slično kao u strojnom kodu, potrebno je definitati blokove za “onda” i “inače” dijelove programa. Ovisno o uvjetu, skače se ili u “onda” ili u “inače”. Cijeli ako-onda-inače izraz mora rezultirati jednim decimalnim brojem. Taj decimalni broj prikazuje se preko LLVM-ovog “phi” čvora. Phi čvor poprima određenu vrijednost ovisno o tome iz kojeg bloka se skočilo, tako da će poprimiti vrijednost “onda” izraza ako se skočilo iz “onda” bloka, te analogno za “inače”. Taj phi čvor stavlja se u zajednički

završni blok koda u kojeg skačemo iz "onda" i "inače" blokova. Grananje je prikazano na slici 4.27.

```
    case .identifier(let name):
      guard let value = parameters[name] else {
        throw "Undefined variable \(name)"
      }
      return value
```

Slika 4.27 *Generiranje varijabli.*

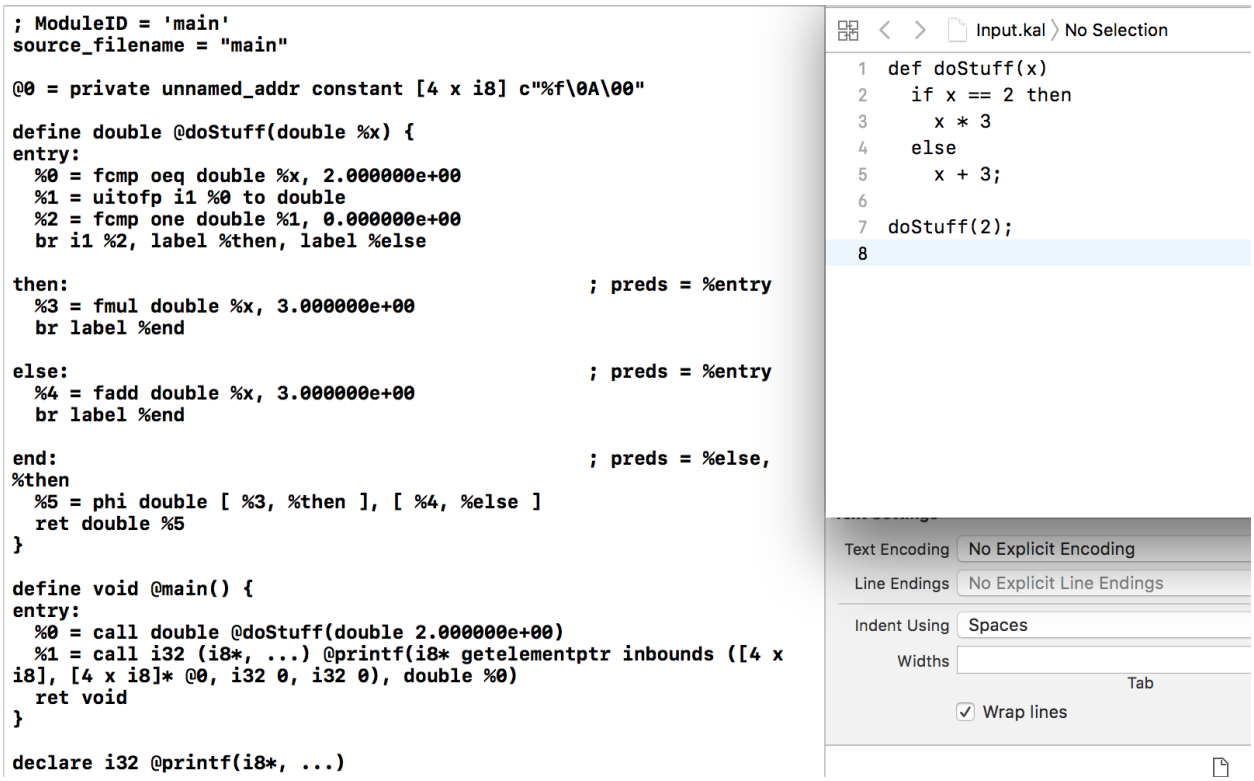
CodeGen ima jednu javnu metodu "generate(toPath:)" koja generira kod za cjelokupnu Kaleidoscope datoteku u jednu datoteku u podatkovnom sustavu. CodeGen generira datoteku u binarnom formatu, ali pozivom "dump" metode na modulu može se prikazati oblik strojnog koda čitljiv ljudima. Primjer takvog izlaza prikazan je na slici 4.28.

```
    case let .ifThenElse(condition, `if`, `else`):
      let check = builder.buildFCmp(try emitExpression(condition, parameters:
parameters), FloatType.double.constant(0), .orderedNotEqual)
      let thenBlock = builder.currentFunction!.appendBasicBlock(named: "then")
      let elseBlock = builder.currentFunction!.appendBasicBlock(named: "else")
      let endBlock = builder.currentFunction!.appendBasicBlock(named: "end")

      builder.buildCondBr(condition: check, then: thenBlock, else: elseBlock)

      builder.positionAtEnd(of: thenBlock)
      let thenValue = try emitExpression(`if`, parameters: parameters)
      builder.buildBr(endBlock)

      builder.positionAtEnd(of: elseBlock)
      let elseVal = try emitExpression(`else`, parameters: parameters)
      builder.buildBr(endBlock)

      builder.positionAtEnd(of: endBlock)
      let phi = builder.buildPhi(FloatType.double)
      phi.addIncoming([(thenValue, thenBlock), (elseVal, elseBlock)])
      return phi
    }
  }
```

Slika 4.28 *Generiranje grananja.*

Nakon što je kod izgeneriran i ispisan u datoteku, ta datoteka predaje se LLVM-ovom interpreteru koji je izvrši i rezultat ispisuje na konzolu, kao što je prikazano na slici 4.29.

U budućnosti se planira dodati petlje (*for, while*), strukture, promjenjive varijable i mogućnost objektno orijentiranog programiranja.



Slika 4.29 *Primjer izlaza iz generatora koda.*

U tablici 4.1 prikazani su primjeri Kaleidoscope koda, te pored njih rezultat izvođenja koda klang prevoditeljom.

Tablica 4.1 *Primjeri Kaleidoscope koda.*

Kod	Rezultat Izvođenja

`def max(a, b)`
`  if b >= a`
`  a`
`  else`
`  b`
 	
`max(3, 2);`
	3

`def fact(x)`
`  if x == 0`
`  1`
`  else`
`  n * fact(x - 1)`
 	
`fact(5);`
	120

```
def sumDigits(x)
  if x == 0
    0
  else
    x % 10 + sumDigits(x / 10)

sumDigits(25);
```

7

5. ZAKLJUČAK

Zadatak rada bio je prikazati teorijsku osnovu i implementaciju osnovnog programskog prevoditelja. Prevodio se programski jezik Kaleidoscope u LLVM IR, koji se onda izvršava pomoću LLVM interpretera. Sam prevoditelj napisan je kao macOS sučelje za konzolu koje prima Kaleidoscope datoteku i izvrši je. Obrađene su osnovne faze prevođenja: leksička analiza (raščlamba koda na osnovne elemente), sintaksna analiza (kreiranje sintaksnog stabla) i generiranje koda (kreiranje LLVM IR koda iz sintaksnog stabla).

Zbog jednostavnosti Kaleidoscope jezika bilo je moguće obaviti cijeli proces prevođenja u te tri faze i jednim prolaskom kroz kod. Kompleksniji jezici zahtijevaju dodatne faze semantičke analize (provjere točnosti programa) i optimizacije.

U procesu kreiranja prevoditelja implentiran je leksički analizator koji je neovisan o samom jeziku, odnosno može se koristiti i u drugim projektima za leksičku analizu. Sam Kaleidoscope jezik moguće je dodatno proširiti dodavanjem petlji, promjenjivih varijabli, operatorom za dodjeljivanje vrijednosti, te lokalnim varijablama. Unatoč mogućim proširenjima, sadašnja implementacija je dobar temelj za dodatno razvijanje prevoditelja za Kaleidoscope, kao i za druge jezike.

LITERATURA

[1] PCMAG, Definition of compiler, https://www.pcmag.com/encyclopedia/term/40105/compiler, pristupljeno: srpanj 2018.

[2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Pearson Education, Boston MA, 2006.

[3] D. Padua, The FORTAN I Compiler, Computing in Science and Engineering, br. 1, Vol. 2, pp. 70-75, siječanj 2000., dostupno na: http://polaris.cs.uiuc.edu/publications/c1070.pdf, pristupljeno: srpanj 2018.

[4] E. Visser, Scannerless generalized-lr parsing, Technical Report P9707, Programming Research Group, University of Amsterdam, 1997., dostupno na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.7828&rep=rep1&type=pdf, pristupljeno: srpanj 2018.

[5] Z. Dovedan, Formalni jezici i prevodioci, Filozofski fakultet, Sveučilište u Zagrebu, Zagreb, 2002., dostupno na: http://dzs.ffzg.unizg.hr/text/FJP/predgovor.htm, pristupljeno: srpanj, 2018.

[6] S. Srbljić: Prevođenje programskih jezika, Element, Zagreb, 2008.

[7] LLVM Project, The Basic Language, dostupno na: https://llvm.org/docs/tutorial/LangImpl01.html#the-basic-language, pristupljeno: srpanj 2018.

[8] LLVM Project, LLVM Features, dostupno na: http://llvm.org/Features.html, pristupljeno: srpanj 2018.

[9] LLM Project, LLVM Users, dostupno na: http://llvm.org/Users.html, pristupljeno: srpanj 2018.

[10] H. Haskins, R. Widmann, LLVMSwift, GitHub, https://github.com/llvm-swift/LLVMSwift, pristupljeno: srpanj, 2018.

SAŽETAK

Ovaj završni rad prikazuje korake i teorijsku pozadinu implementacije jednostavnog programskog prevoditelja za probni jezik Kaleidoscope. Rad opisuje tri osnovne faze prevođenja programa: leksičku analiza, sintaksnu analizu i generiranje koda. Također prikazuje implementaciju jednostavnog prevoditelja napisanu u jeziku Swiftu, koja koristi LLVM set alata za generiranje i izvođenje koda. Rezultat rada je programski prevoditelj koji se može koristiti preko komandnog sučelja za izvršavanje programskog koda.

Ključne riječi:
leksička analiza, LLVM, prevoditelj, sintaksna analiza, Swift

ABSTRACT

MAIN STEPS IN DESIGNING AND IMPLEMENTING A SIMPLE COMPILER

This bachelor thesis shows the steps and theoretical background of implementing a simple compiler for a toy programming language called Kaleidoscope. The thesis describes the three main steps of compiling a program: lexical analysis, parsing and code generation. It also shows an implementation of a simple compiler written in Swift, using the LLVM toolset for code generation and interpretation. The result of this thesis is a working compiler that can be used as a command line tool to execute code.

Keywords:

lexical analysis, LLVM, compiler, parsing, Swift

ŽIVOTOPIS

Marin Benčević rođen je 4. 11. 1996. godine u Vinkovcima. Završio je osnovnu školu "Zrinski" u Nuštru i Gimnaziju M. A. Reljkovića u Vinkovcima. Trenutno studira na „Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek" u sklopu Sveučilišta Josipa Jurja Strossmayera u Osijeku gdje pohađa preddiplomski studij računarstva. Tri godine radi kao iOS developer i godinu dana kao Unity developer.

PRILOZI

1. “Glavni koraci dizajna i ugradnje jednostavnog programskog prevoditelja” u .docx formatu

2. “Glavni koraci dizajna i ugradnje jednostavnog programskog prevoditelja” u .pdf formatu

3. Programski kod