Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Samo Jelovšek

Razširitev prevajalnika za Kaleidoscope na osnovi ogrodja LLVM

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2013

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00410/2013
Datum: 05.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SAMO JELOVŠEK**

Naslov: **RAZŠIRITEV PREVAJALNIKA ZA KALEIDOSCOPE NA OSNOVI OGRODJA LLVM**

EXTENDING LLVM-BASED KALEIDOSCOPE COMPILER

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Ogrodje LLVM predstavlja zadnji del prevajalnika in s tem omogoča hitrejšo izdelavo prevajalnikov. Preučite obstoječi prevajalnik za programski jezik Kaleidoscope, ki temelji na ogrodju LLVM, in nadomestite leksikalni in sintaksni analizator z novima analizatorjema, ki bosta izdelana z orodjema Flex in Bison. V obstoječi prevajalnik dodajte dve novi zanki: while-do in repeat-until. Za nov prevajalnik pripravite ustrezne testne primere.

Mentor:
pred. dr. Boštjan Slivnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Samo Jelovšek, z vpisno številko **63070033**, sem avtor diplomskega dela z naslovom:

Razširitev prevajalnika za Kaleidoscope na osnovi ogrodja LLVM

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 23. septembra 2013 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku za pomoč, nasvete ter izkazano potrpežljivost pri izdelavi diplomske naloge. Zahvala gre tudi družini in prijateljem, ki so me tekom študija podpirali in mi stali ob strani ter pripomogli k uspešnem zaključku študija.

Pričujoče diplomsko delo posvečam vsem, ki jih področje prevajalnikov zanima in bodo delo v celoti prebrali.

Kazalo

Povzetek

Diplomsko delo najprej opiše pojem prevajalnika, kaj to je ter kako je sestavljen tipični prevajalnik. Ta osnovna opredelitev je bila uporabljena za razvoj izboljšanega prevajalnika za programski jezik Kaleidoscope. Prevajalnik je bil razvit s pomočjo ogrodja LLVM. Razvit je bil prednji del prevajalnika, za sestavo zadnjega dela prevajalnika pa so bile uporabljene knjižnice projekta LLVM. Diplomsko delo predstavi projekt LLVM, poda podrobni opis vmesne kode LLVM, ki je eden izmed glavnih delov razvitega prevajalnika. Opisane so faze razvoja prevajalnika: leksikalna analiza (s pomočjo orodja Flex), sintaksna analiza (s pomočjo orodja Bison), generiranje vmesne kode, kjer je predstavljen del knjižnice LLVM, ki je bila uporabljena. Opisano je še kako tako generirano vmesno kodo izvajamo s pretvorbo v objektno datoteko ELF. Na koncu je še opisano dodajanje dveh novih zank: while-do ter repeat-until.

Ključne besede: prevajalnik, LLVM, vmesna koda LLVM, izvršljiva datoteka, Kaleidoscope.

Abstract

This thesis first describes the concept of a compiler, what it is and what typical compiler consists of. This basic definition has been used to develop a improved compiler for Kaleidoscope programming language. Compiler was developed using LLVM framework. Compiler uses its own front end, back end was developed with use of the LLVM library. The thesis presents the LLVM project, with main focus on the LLVM intermediate representation, which is one of the main most important parts of the developed compiler. It presentes the stages of the developed compiler: lexical analysis (using Flex tool), syntax analysis (using Bison tool), LLVM intermediate representation code generation, where it describes the part of the LLVM code library, which was used in the development. It is described how the generated LLVM intermediate representation code is executed by conversion to an ELF object file. In the last part it is described addition of new while-do and repeat-until loops.

Keywords: compiler, LLVM, LLVM intermediate representation, executable file, Kaleidoscope.

Poglavje 1

Uvod

1.1 Opis problema

Prevajalnik je računalniški program, ki prevede oz. pretvori vhodno izvorno kodo programa napisanega v programskem jeziku A v kodo napisano v programskem jeziku B. Običajno prevajamo višje programske jezike (*high-level programming languages*) v nižje programske jezike (*low-level programming languages*) kot npr. zbirni jezik. Kodo v zbirnem jeziku nato zbirnik pretvori v strojno kodo (*machine code*), ki jo vključi v objektno datoteko (*object file*). Objektna datoteka omogoča izvajanje programa na nekem računalniku. Na splošno objektnim datotekam pravimo tudi izvršljive datoteke (*executable file*). Najbolj znani vrsti objektih datotek, kot tudi najbolj pogosto uporabljeni, sta ELF (*Executable and Linkable Format*) ter PE (*Portable Executable*). Slednje se uporabljajo na operacijskih sistemih Windows, objektne datoteke ELF pa na Unixu podobnih operacijskih sistemih. Za končni rezultat prevajalnika bi lahko rekli, da je to izvršljiva datoteka, kar pa ni vedno končni cilj prevajalnikov. Obstajajo tudi prevajalniki, pri katerih se koda izvaja preko tolmača (*interpreter*). Izvajanje programov na tolmačih je običajno počasnejše od izvajanja izvršljivih datotek, saj koda ni prilagojena procesorju točno določene vrste.

Obe vrsti prevajalnikov sta si po strukturi v določeni delih podobni, ven-

dar se to delo osredotoči na prevajalnike prve vrste, ki nas pripeljejo do izvršljive datoteke. Ti prevajalniki se po zgradbi v osnovi delijo na dva dela: prednji (*front end*) ter zadnji del (*back end*).

Prednji del prevajalnika je odvisen od programskega jezika, zadnji del pa od tipa procesorja, na katerem želimo izvajati program. Za lažje obvladovanje problema prevajanja, prevajalnik običajno razdelimo v več različnih faz. Prednji del prevajalnika običajno vsebuje naslednje faze: leksikalna analiza, sintaksna analiza ali razčlenjevalnik kode (*parser*), semantična analiza ter generiranje vmesne kode. Na tem mestu velja omeniti, da je realizacija prednjega dela prevajalnika močno odvisna od programskega jezika, za katerega želimo razviti prevajalnik.

Leksikalna analiza, kot prva faza prevajanja, na vhod prejme izvorno kodo, napisano v določenem programskem jeziku. Končni cilj leksikalne analize je, da to vhodno kodo pretvori v zaporedje osnovnih simbolov (*tokens*). Osnovni simboli predstavljajo ključne besede programskega jezika (npr. while), operatorje (npr. +, -, *, /), imena (npr. spremenljivk, funkcij) itd. V tej fazi tudi poskrbimo, da se odstranijo nepotrebni znaki in odseki kode (npr. presledki in komentarji). Leksikalno analizo ponavadi realiziramo s pomočjo generatorjev leksikalnih analizatorjev. Znan odprtokoden projekt na tem področju je Flex [1].

Naslednja faza sintaksna analiza na vhod prejema osnovne simbole in jim preverja pravilnost zaporedja in s tem je preverjena pravilnost sintakse izvorne kode programa. Sintaksa programskega jezika je navadno določena s kontekstno neodvisno gramatiko (*context-free grammar*). Pri realizaciji sintaksne analize si lahko pomagamo s odprtokodnim projektom Bison [2], ki omogoča generiranje razčlenjevalnikov glede na podano kontekstno neodvisno gramatiko. Izhod sintaksne analize je abstraktno sintaksno drevo (*abstract syntax tree*).

Sledi faza imenovana semantična analiza, ki je namenjena predvsem določanju ter preverjanju tipov posameznih izrazov. Pri uporabi imen (npr. spremenljivk) se v tej fazi še preveri, če so ti ustrezno deklarirani. Preverjena

je tudi ustreznost določenega tipa pri določeni operaciji, recimo prevajalnik ne dovoli seštevanja, kjer je prvi operand niz in drugi celo število. Preverjanje tipov je seveda odvisno od tipiziranja programskega jezika [8]. Ločimo dve osnovni tipizaciji: statično ter dinamično. Pri statični tipizaciji morajo biti vsi tipi strogo določeni, pri dinamični pa prevajalnik skuša sam ugotoviti, za kateri tip gre. Informacije o tipih se navadno dodajo posameznim vozliščem abstraktnega sintaksnega drevesa. Rezultat sintaksne analize je še vedno abstraktno sintaksno drevo, a z nekaterimi dodatnimi informacijami o tipih.

V zadnji fazi prednjega dela prevajalnika se abstraktno sintaksno drevo pretvori v vmesno kodo. Vmesna koda predstavlja abstraktno strojno kodo, ki ni vezana na točno določeno strojno opremo. Generiranje vmesne kode sicer ni nujno potrebna faza, vendar nam prinaša veliko prednosti kot npr. visoko prenosljivost [7] ter modularnost. Tega principa se poslužuje tudi prevajalnik GCC (*GNU Compiler Collection*) [3], ki za vmesno kodo uporablja jezike imenovane GENERIC, GIMPLE ter RTL (*Register Transfer Language*), ki se uporabljajo v različne optimizacijske namene.

Zadnji del prevajalnika običajno vsebuje naslednje faze: analiza vmesne kode, optimizacija vmesne kode ter generiranje kode v zbirnem jeziku. Vse faze v zadnjem delu prevajalnika so močno odvisne od tipa procesorja, na katerem bomo izvajali vhodni program. Analiza vmesne kode služi kot osnovi vir informacij za optimizacijo kode. V zadnji fazi se izvede samo še preslikava iz vmesne kode v kodo zbirnega jezika.

1.2 Cilj diplomske naloge

V prejšnjem poglavju 1.1 smo opisali osnovno zgradbo tipičnega prevajalnika. Iz tega je moč narediti sklep, da je prednji del prevajalnika veliko lažje razviti kot zadnji del prevajalnika. To kaže že samo dejstvo, da imamo trenutno veliko število različnih procesorjev, ki delujejo povsem drugače, recimo procesorji tipa x86 ter ARM. Poleg obsežnosti problema se pojavi še problem, da bi od proizvajalcev procesorjev težko ali sploh nebi mogli pri-

dobiti podrobnejšega opisa delovanja njihovega procesorja. Posledica tega bi bilo počasnejše in neučinkovito izvajanje prevedenega programa, saj bi zaradi pomanjkanja podatkov o delovanju procesorja težko ocenili, kaj je najboljši način za izvajanje določene operacije na določenem procesorju. Omenjenim problemom se lahko izognemo z uporabo odprtokodnega projekta LLVM, ki lahko med drugim služi tudi kot zadnji del prevajalnika.

Cilj diplomske naloge je preučiti trenutni prevajalnik za mini programski jezik Kaleidoscope in na novo razviti prednji del prevajalnika, s pomočjo orodij Flex in Bison. Pri tem, da zadnji del prevajalnika še vedno ostane realiziran s pomočjo knjižnic projekta LLVM. V tem diplomskem delu prevajalniku dodamo nekaj novih razširitev, kot npr. prevajanje programov v objektne datoteke ELF. Programskemu jeziku Kaleidoscope pa dodamo tudi dve novi zanki, while-do in repeat-until.

Poglavje 2

LLVM

2.1 Opis projekta

LLVM je odprtokoden projekt, ki se osredotoča na področja povezana s prevajalniki. V osnovi predstavlja projekt LLVM knjižnico oz. ogrodje, ki nam omogoča lažji in predvsem hitrejši razvoj prevajalnikov, saj lahko uporabimo veliko že narejenih delov prevajalnika. V našem primeru je to zadnji del prevajalnika. V začetkih projekta je šlo za raziskovalni projekt Univerze Illinois, kasneje pa se je uveljavil kot uspešen odprtokoden projekt tudi na komercialnem področju. Začetnika projekta sta Vikram Adve in Chris Lattner. Projekt je izdan pod licenco University of Illinois Open Source License, gre za licenco tipa BSD. Kratica LLVM je v začetkih projekta pomenila Low Level Virtual Machine. Zaradi razširitve na druga področja, ki presegajo kratico, pa se je kasneje kratica opustila in trenutno LLVM ni kratica za nič in predstavlja polno ime projekta.

Knjižnica, ki jo predstavlja projekt LLVM, je napisana v programskem jeziku C++ in je zasnovana tako, da jo lahko ponovno uporabimo. Jedro projekta LLVM predstavlja vmesna koda LLVM (*LLVM intermediate representation*), ki smo jo podrobno opisali v pogljavju 2.4. Vmesno kodo LLVM lahko na številne načine optimiziramo ter izvajamo (preko tolmača, prevajlnika JIT ali izvršljive datoteke) in to s pomočjo metod, ki so del knjižnice.

Slika 2.1: Logo projekta LLVM.

Kot zanimivost omenimo tudi možnost, da prevajalnik s pomočjo knjižnice LLVM prevede izvorno kodo v programski jezik C kot vmesno točko, kar kasneje prevedejo s poljubnim prevajalnikom za programski jezik C (npr. Intelov prevajalnik) v izvršljivo datoteko.

Praktično uporabo knjižnice LLVM predstavljajo številni podprojekti. Najbolj znani in pomembni podprojekti so: Clang, ki predstavlja resno alternativo prevajalniku GCC; Dragonegg, vtičnik za prevajalnik GCC, ki uporablja prednji del prevajalnika GCC ter zadnji del prevajalnika realiziranega preko knjižnice LLVM ter LLDB, ki je razhroščevalnik za programske jezike C, C++ ter Objective C. To diplomsko delo se osredotoči LLVM različice 3.3., ki je bila izdana 17. junija 2013. Na sliki 2.1 lahko vidimo logo projekta LLVM.

2.2 Clang

Pomemben del projekta LLVM je podprojekt, ki predstavlja prevajalnik Clang. Clang je prevajalnik za programske jezik C, C++, Objective C ter

Objective C++. Clang je prednji del prevajalnika in za zadnji del uporablja knjižnico LLVM. Glavni pobudnik projekta je podjetje Apple, Inc., saj večino njihovih razvojnih orodji uporablja programski jezik Objective C. Za prehod na prevajalnik Clang se je podjetje Apple, Inc. odločilo zaradi nizke prioritete razvoja in dopolnjevanje jezika Objecitve C pri prevajalniku GCC. Podjetje Apple, Inc. trenutno zaposluje enega izmed začetnikov projekta LLVM, Chrisa Lattnerja.

Cilji projekta iz vidika končnega uporabnika so: hitro prevajanje programov in nizka poraba pomnilnika, bolj bogati opisi napak pri prevajanje, kompatibilnost s prevajalnikom GCC in boljša združljivost z razvojnimi orodji. Prevajalnik Clang se uporablja kot primarni prevajalnik Applovega razvojnega orodja Xcode od različice 3.2 naprej. Mac OS X je od različice 10.7 naprej preveden v celoti s prevajalnikom Clang.

Njegova uporaba ima pomemben vpliv na projekt LLVM kot celoto, saj projektu omogoča konstanten razvoj, hitro odpravo hroščev ter veliko lepo urejene dokumentacije, ki je v pomoč tako začetnikom kot tudi naprednim uporabnikom oz. razvijalcem projekta.

2.3 Orodja v ukazni vrstici

Ob namestitvi LLVM-ja dobimo poleg knjižnic številna orodja v ukazni vrstici, ki so nam v pomoč pri delu s knjižnico LLVM in pri izvajanju vmesne kode LLVM. Sledi kratek opis osnovnih orodij.

llvm-as

Orodje *llvm-as* je zbirnik LLVM. Prevede ročno napisano vmesno kodo LLVM v bitno kodo LLVM, ki jo nato lahko izvajamo.

llvm-dis

Nasprotno vlogo prejšnjemu orodju ima orodje *llvm-dis*. Bitno kodo LLVM prevede v človeku berljivo vmesno kodo LLVM, ki jo lahko uporabljamo v razhroščevalne namene.

opt

S pomočjo orodja *opt* lahko izvajamo različne optimizacije in analize na vmesni kodi LLVM. Izvedemo lahko optimizacijo, ki spremeni vse dinamično dodeljene spremenljivke na skladu v registre. Orodje omogoča tudi različne vizualizacije vmesne kode LLVM, npr. skokov po vmesni kodi LLVM. Vhodna datoteka je lahko vmesna koda LLVM oz. bitna koda LLVM.

llc

Orodje *llc* omogoča prevajanje vmesne kode LLVM in bitne kode LLVM v zbirni jezik trenutnega procesorja. Podprt je precej širok nabor procesorjev, omenimo najbolj popularne: x86, x86_64, ARM, MIPS ...

lli

Orodje *lli* omogoča izvajanje bitne kode LLVM preko prevajalnika JIT, če je le-ta na voljo za trenutni procesor, v nasprotnem primeru pa kodo izvede preko tolmača.

llvm-link

Povezovalnik za bitno kodo LLVM v več različnih datotekah predstavlja orodje *llvm-link*.

llvm-ar

Orodje *llvm-ar* omogoča pripravo dinamičnih knjižnic namenjenih za povezovanje v LLVM programe.

llvm-nm

Orodje *llvm-nm* nam izpiše simbolno tabelo bitne kode LLVM, npr. imen funkcij in globalnih spremenljivk.

llvm-config

Za programerje je zelo pomembno orodje *llvm-config*. S pomočjo tega orodja prevajalniku podamo nabor knjižnic LLVM, ki jih potrebujemo pri prevajanju in povezovanju našega programa, napisanega s pomočjo knjižnice LLVM.

llvm-diff

Orodje *llvm-diff* omogoča primerjavo dveh datotek napisanih v vmesni kodi LLVM oz. bitni kodi LLVM. Osredotoča se predvsem na definicije posameznih funkcij s primerjavo njihovih blokov kode.

llvm-stress

V namene testiranja se uporablja orodje *llvm-stress*. Orodje proizvede naključno sestavljene datoteke z vmesno kodo LLVM, ki so namenjene testiranju različnih razvitih funkcionalnosti.

llvm-symbolizer

Orodje *llvm-symbolizer* prebere datoteko s podanimi pomnilniškimi lokacijami določenih objektnih datotek in jim nato določi pripadajoče vrstice, kjer se pojavijo v izvorni kodi.

2.4 Vmesna koda LLVM

2.4.1 Splošna razlaga

Najpomembnejši pogled na zasnovo projekta LLVM predstavlja vmesna koda LLVM (*LLVM Intermediate Representation, kratko LLVM IR*). Uporablja se v treh oblikah, ki so si po uporabnosti enakovredne. Te oblike so: v pomnilniku prevajalnika kot vmesna koda, shranjena v obliki bitne kode na disku (primerna za hitro izvajanje preko prevajalnika JIT) ter v ljudem berljivi obliki (običajno shranjena v datoteki s končnico `ll`). V tem poglavju opišemo slednjo.

Vmesna koda LLVM je zasnovana s ciljem, da je preprosta in nizko nivojska, hkrati pa ohranja možnost izvedb visoko nivojskih idej, je tipizirana ter razširljiva. Na prvi pogled je vmesna koda LLVM zelo podobna zbirnemu jeziku. Vmesna koda LLVM je nabor navideznih RISC-u podobnih nizko nivojskih ukazov. Ukazi so tri operandni, kar pomeni, da se rezultat operacije shrani v neki tretji register. Glavna razlika med pravim zbirnim jezikom je tudi ta, da imamo tukaj neskončno število registrov, več o tem kasneje. Teži

tudi k temu, da bi bila čim bolj univerzalna vmesna koda za prevajalnike, z namenom čim lažje implementacije programskih jezikov. Uporablja tudi princip statičnega enkratnega dodeljevanja (*static single assignment, kratko SSA*). Na izpisu 2.1 lahko vidite primer funkcij v programskem jeziku C, ki izvede zmnožek dveh števil ter izračuna fakulteto števila.

```
1  unsigned mul(unsigned a, unsigned b) {
2    return a*b;
3  }
4
5  unsigned factorial(unsigned n) {
6    if (n == 1)
7      return 1;
8    else
9      return n*factorial(n-1);
10 }
```

Izpis 2.1: Primer kode v programskem jeziku C.

Zgornji primer kode na izpisu 2.1 uporabimo za pretvorbo v vmesno kodo LLVM, rezultat je viden na izpisu 2.2.

```
1  define i32 @mul(i32 %a, i32 %b) {
2  entry:
3    %tmp1 = mul i32 %a, %b
4    ret i32 %tmp1
5  }
6
7  define i32 @factorial(double %n) {
8  entry:
9    %tmp1 = icmp eq i32 %n, 1
10   br i1 %tmp1, label %ifcont, label %else
11
12 else:
13   %subtmp =  sub i32 %n, 1
14   %calltmp = call i32 @factorial(i32 %subtmp)
15   %multmp = mul i32 %calltmp, %n
16   ret i32 %multmp
17
18 ifcont:
19   ret i32 %n
20 }
```

Izpis 2.2: Pretvorjena vmesna koda iz programskega jezika C.

2.4.2 Statično enkratno dodeljevanje

V prejšnjem poglavju 2.4.2 smo omenili, da vmesna koda LLVM uporablja princip statičnega enkratnega dodeljevanja, v nadaljevanju SED. Gre za princip, kjer vsaki spremenljivki dodelimo vrednost samo enkrat. Ena spremenljivke ima zato več različic, kjer z vsako različico pripisujemo novo želeno vrednost. Ta princip se uporablja izključno v optimizacijski fazi, saj na ta način proizvedemo veliko količino spremenljivk, kar pa nam bo pri generiranju strojne kode močno škoduje zaradi zelo omejenega števila registrov na procesorjih.

```
1 y := 1
2 y := 2
3 x := y
```

Izpis 2.3: Spreminjanje vrednosti spremenljivkam [4].

V izpisu 2.4 imamo prikazano običajno spreminjanje vrednosti spremenljivk zapisano v psevdo kodi. Če kodo pretvorimo po prencipu SED, dobimo kodo prikazano v 2.4. Na ta način optimizacijski algoritmi lažje in predvsem hitreje ugotovijo, da je dodelitev v prvi vrstici nepotrebna, saj se ime `y1` pojavi samo enkrat in nikoli več kasneje v kodi. Zategadelj lahko ukaz preprosto odstranimo in pridobimo na hitrosti izvajanja programa.

```
1 y1 := 1
2 y2 := 2
3 x1 := y2
```

Izpis 2.4: Spreminjanje vrednosti spremenljivkam po prencipu SED [4].

2.4.3 Osnovni sestavni deli

Vsak program napisan v vmesni kodi LLVM se hrani v strukturi imenovani modul (*module*). Vsak modul je zbirka globalnih spremenljivk, funkcij ter zapisov v simbolni tabeli. Module lahko med seboj povezujemo s povezovalnikom LLVM (*LLVM linker*), ki združi globalne spremenljivke, funkcije, razreši vnaprejšnje deklaracije ter ustrezno združi zapise v simbolni tabeli.

Način poteka združitve je določen z določili povezovanja (*linkage types*) ter dostopa (*visibility styles*), s čemer se v tem diplomskem delu ne bomo ukvarjali. Vsi prestavljeni primeri bodo delovali samostojno, brez povezovanja z drugimi moduli in ne bodo uporabljali globalnih spremenljivk.

Imena v vmesni kodi LLVM ločimo v dve skupini: globalna in lokalna. Globalna imena (funkcij, globalnih spremenljivk) se začno z znakom `@`. Lokalna imena (registrov, tipov) se začno z znakom `%`. Ločimo dve obliki:

Imenovane vrednosti. Predstavljene so z nizom in ustrezno predpono. Primer: `%lokalna`, `@Globalna.spremenljivka`. Dovoljena je uporaba vseh znakov, ki spadajo v množico določeno z naslednjim regularnim izrazom: `[%@][a-zA-Z$._][a-zA-Z$._0-9]*`. Uporabimo lahko tudi druge znake, če ime zapišemo med dvojne narekovaje. Posebne znake lahko zapišemo v obliki \`xx`, kjer `xx` nadomestimo z ASCII kodo zapisano v šestnajstiški obliki.

Neimenovane vrednosti. Predstavljene so kot nepredznačena števila. Primer: `%11`, `@4`, `%42`.

Imena imajo predpono iz dveh razlogov. Ne more priti do zamenjave med imenom in ključno besedo ter jeziku je možno dodati nove množice ključnih besed brez konfliktov in skrbi, da bi prišlo do prekrivanja imen. Neimenovanim vrednostim pa lahko prevajalnik na hiter način določi ime začasnega imena brez možnosti za konflikte v simbolni tabeli.

Ključne besede v vmesni kodi LLVM so podobne kot v drugih jezikih. Ločimo med ključnimi besedami za določene operacije (`add`, `br`, `ret`...), imena primitivnih tipov (`void`, `i32`, `double`...). Zaradi predpone pri imenih jih ne moremo zamenjati z imeni.

Komentarji se začno z znakom `;` ter končajo na koncu vrstice. Primer lahko vidimo na izpisu 2.5.

```
1 %0 = add i32 %X, %X           ; yields {i32}:%0
2 %1 = add i32 %0, %0           ; yields {i32}:%1
3 %result = add i32 %1, %1
```

Izpis 2.5: Komentarji v vmesni kodi LLVM.

Klasifikacija	Ključna beseda
celo število	i1, i2, i3, ... i8, ... i16, ... i32, ..., i64, ...
število v plavajoči vejici	half, float, double, x86_fp80, fp128, ppc_fp128

Tabela 2.1: Tipi v vmesni kodi LLVM, povzeto po [5].

Funkcijo definiramo s ključno besedo `define`, nato sledi ime tipa, ki ga funkcija vrača, sledi ime funkcije (gre za globalno ime, ki mora imeti predpono `@`), v oklepajih se nahaja seznam tipiziranih parametrov, sledijo oglati oklepaji, med katerimi se nahaja jedro funkcije. Opisano je malo poenostavljena definicija funkcije, ki za naše potrebe zadostuje. Na izpisu 2.6 si lahko ogledamo celotno sintakso za definicijo funkcije. Imena zapisana v oglatih oklepajih niso obvezna, imena zapisana med matematičnima simboloma za manjše in večje pa so obvezna in jih mora imeti vsaka definicija funkcije. Naprednejšemu bralcu, ki ga zanima pomen ostalih neobveznih oznak, priporočamo branje vira [5]. Ta predstavlja celotno referenco vmesne kode LLVM. Prvo definicijo funkcije smo pokazali že na izpisu 2.1.

```
1 define [linkage] [visibility]
2        [cconv] [ret attrs]
3        <ResultType> @<FunctionName> ([argument list])
4        [fn Attrs] [section "name"] [align N]
5        [gc] { ... }
```

Izpis 2.6: Definicija funkcije v vmesni kodi LLVM.

Funkcijo lahko tudi samo deklariramo. Deklaracija funkcije je sintaktično enaka definiciji funkcije le, da ključno besedo `define` nadomesti ključna beseda `declere` ter jedro funkcije ostane nedefinirano, očitno.

Jedro funkcije sestavlja seznam osnovnih blokov kode (*basic blocks*). Vsak blok kode se lahko opcijsko začne z imenom (`label`). Ime privzetega bloka je `entry` in ima to posebnost, da vanj kasneje ne moremo več skočiti nazaj. Zategadelj se tudi ne sme uporabljati ukaza `phi`, o katerem nekoliko kasneje.

V tabeli 2.1 smo prikazali tipe za delo s celimi števili. Poleg zgoraj naštetih imamo na voljo še tipe kot so: nični tip (*void*), tabela, funkcija,

kazalec, struktura, vektor ipd. Za naše potrebe zadostuje, da poznamo tip za števila v plavajoči vejici `double` ter N-bitna cela števila, ki jih zapišemo kot `iN`, kjer N predstavlja število bitov, recimo 32-bitno celo število zapišemo kot tip `i32`.

2.4.4 Nabor ukazov

V tem poglavju opišemo ukaze, ki jih bomo potrebovali pri našem generiranju vmesne kode. Poleg opisanih ukazov ima jezik vmesne kode LLVM še precej drugih ukazov, ki jih naš prevajalnik ne bo potreboval. Vsi ukazi so podrobno opisani v viru [5]. Podobno kot v poglavju 2.4.3 tudi v tem poglavju matematična simbola za manjše in večje pomenita obvezno uporabo parametra, oglati oklepaji pa označujejo neobvezno uporabo.

Seznam ukazov, ki jih bo potreboval naš prevajalnik pri generiranju vmesne kode LLVM:

add

```
1  <result> = add <ty> <op1>, <op2>
```

Izpis 2.7: Sintaksa ukaza add.

Ukaz `add` vrne vsoto operandov `op1` ter `op2` (prikazana na izpisu 2.7), ki sta oba identičnega tipa `ty`. Tip `ty` mora biti celo število ali pa vektor celih števil. Primer uporabe:

```
1  %result = add i32 %i, 1
```

Izpis 2.8: Primer uporabe ukaza add.

Izpis 2.8 prikazuje prištevanje števila 1 lokalni spremenljivki `i` in to vsoto shrani v lokalno v lokalno spremenljivko `result`. Isto sintakso in pravila glede tipov imajo tudi ukazi, vendar le za druge operacije: `sub` (odštevanje), `mul` (množenje), `udiv` (deljenje). Za operacije s števili v plavajoči vejici uporabljamo ukaze `fadd`, `fsub`, `fmul` ter `fdiv`.

ret

```
1  ret <type> <value>
2  ret void
```

Izpis 2.9: Sintaksa ukaza ret.

Ukaz `ret` uporabimo na koncu bloka kode, ko se želimo vrniti nazaj v blok kode, iz katerega je bila funkcija klicana. Sintakso v prvi vrstici na izpisu 2.9 uporabimo, kadar naša funkcija vrača neko vrednost, `type` se mora ujemati s tipom podanim v deklaraciji funkcije in `value` predstavlja vrednost, ki jo želimo vrniti. Sintakso iz druge vrstice 2.9 uporabimo, kadar smo v deklaraciji funkcije podali tip `void`. Primer uporabe:

```
1  ret double 2.100000e+01
```

Izpis 2.10: Primer uporabe ukaza ret.

V izpisu 2.10 vrnemo vrednost `21,0` tipa `double`.

icmp

```
1  <result> = icmp <cond> <ty> <op1>, <op2>
```

Izpis 2.11: Sintaksa ukaza icmp.

Ukaz `icmp` s sintakso predstavljeno v izpisu 2.11 primerja vrednosti tipa `ty`. Tip `ty` je lahko eden izmed naštetih: celo število, vektor celih števil, kazalec ali vektor kazalcev. Kot operanda `op1` ter `op2` podamo vrednosti, ki ju želimo primerjati. Ukaz `icmp` vrne vrednost `0` ali `1` tipa `i1`. Kakšno primerjavo bomo izvajali določimo v operandu `cond`, ki mora biti ena izmed naslednjih ključnih besed:

1. `eq` preverja, ali sta operanda enaka,
2. `ne` preverja, ali sta operanda različna,

3. `ugt` preverja, ali je prvi operand strogo večji od drugega operanda (obe števili morata biti nepredznačeni),
4. `uge` preverja, ali je prvi operand večji ali enak od drugega operanda (obe števili morata biti nepredznačeni),
5. `ult` preverja, ali je prvi operand manjši od drugega operanda (obe števili morata biti nepredznačeni),
6. `ule` preverja, ali je prvi operand manjši ali enak od drugega operanda (obe števili morata biti nepredznačeni),
7. `sgt` preverja, ali je prvi operand strogo večji od drugega operanda (obe števili morata biti predznačeni),
8. `sge` preverja, ali je prvi operand večji ali enak od drugega operanda (obe števili morata biti predznačeni),
9. `slt` preverja, ali je prvi operand manjši od drugega operanda (obe števili morata biti predznačeni),
10. `sle` preverja, ali je prvi operand manjši ali enak od drugega operanda (obe števili morata biti predznačeni).

```
1   %result = icmp eq i32 4, 5
```

Izpis 2.12: Primer uporabe ukaza icmp.

Izpis 2.12 prikazuje primerjavo ali je štiri enako pet in rezultat shrani v lokalno spremenljivko `result`.

fcmp

```
1   <result> = fcmp <cond> <ty> <op1>, <op2>
```

Izpis 2.13: Sintaksa ukaza fcmp.

Ukaz `fcmp` s sintakso prikazano v izpisu 2.13 primerja vrednosti tipa `ty`. Tip `ty` mora število v plavajoči vejici. V operanda `op1` ter `op2` podamo vrednosti, ki ju želimo primerjati. Ukaz `fcmp` vrne vrednost `0`

ali `1` tipa `i1`. Kakšno primerjavo bomo izvajali določimo v operandu `cond`, ki mora biti ena izmed naslednjih ključnih besed:

1. `false` vedno vrne `0`,
2. `oeq` preverja, ali sta operanda enaka (oba operanda pravi števili (*orderd*)),
3. `ogt` preverja, ali je prvi operand strogo večji od drugega operanda (oba operanda pravi števili),
4. `oge` preverja, ali je prvi operand večji ali enak od drugega operanda (oba operanda pravi števili),
5. `olt` preverja, ali je prvi operand manjši od drugega operanda (oba operanda pravi števili),
6. `ole` preverja, ali je prvi operand manjši ali enak od drugega operanda (oba operanda pravi števili),
7. `one` preverja, ali sta operanda različna (oba operanda pravi števili),
8. `ord` preverja, ali sta oba operanda pravi števili,
9. `ueq` preverja, ali sta operanda enaka (možnost neštevila (*unorderd*)),
10. `ugt` preverja, ali je prvi operand strogo večji od drugega operanda (možnost neštevila),
11. `uge` preverja, ali je prvi operand večji ali enak od drugega operanda (možnost neštevila),
12. `ult` preverja, ali je prvi operand manjši od drugega operanda (možnost neštevila),
13. `ule` preverja, ali je prvi operand manjši ali enak od drugega operanda (možnost neštevila),
14. `une` preverja, ali sta operanda različna (možnost neštevila),
15. `uno` preverja, ali sta oba operanda neštevili,

16. `true`, vedno vrne `1`.

Pri primerjavah tipa oba operanda pravi števili (*orderd*) ukaz vrne `1` v primeru, če sta oba operanda pravi števili (nista neštevili) in hkrati, ko drži posamezna primerjava. Primer: pri uporabi ključne besede `ogt` ukaz vrne `1` v primeru, če sta oba operanda pravi števili in hkrati je prvi operand strogo večji od drugega operanda.

Pri primerjavah tipa možnost neštevila (*unorderd*) ukaz vrne `1` v primeru, če je kateri od operandov neštevilo (*NaN - not a number*) ali pa v primeru resničnosti primerjave. Primer: pri uporabi ključne besede `ueq` ukaz vrne `1` v primeru, če je kateri od operandov neštevilo ali, če sta operanda enaka.

```
1   %result = fcmp oeq float 4.0, 5.0
```

Izpis 2.14: Primer uporabe ukaza fcmp.

Izpis 2.14 prikazuje primerjavo, ali je štiri enako pet, in rezultat shrani v lokalno spremenljivko `result`.

br

```
1   br i1 <cond>, label <iftrue>, label <iffalse>
2   br label <dest>
```

Izpis 2.15: Sintaksa ukaza br.

Ukaz `br` se uporablja za kontrolo pretoka in omogoča pogojni skok na dva različna bloka kode trenutne funkcije. Sintaksa ukaza prve vrstice na izpisu 2.15 mora kot operand imenovan `cond` prejeti vrednost tipa `i1` (eno bitno celo število) in v primeru, da je ta vrednost `1`, skočimo na blok kode imenovan v operandu `iftrue`, sicer pa skočimo na blok kode imenovan v operandu `iffalse`. V drugi vrstici izpisa 2.15 je prikazana sintaksa, ki izvede brezpogojni skok na blok kode imenovan v operandu `dest`. Na tem mestu velja še omeniti, da se mora vsak blok kode zaključiti z ukazom `ret` ali `br`. Primer uporabe:

```
1  define i32 @eq(i32 %x, i32 %y) {
2   first:
3    br label %sec
4   second:
5    %cond = icmp eq i32 %x, %y
6    br i1 %cond, label %if, label %else
7   if:
8    ret i32 1
9   else:
10   ret i32 0
11 }
```

Izpis 2.16: Primer uporabe ukaza br.

Izpis 2.16 prikazuje funkcijo `eq`, ki v prvem bloku imenovanem `first` naredi brezpogojni skok na blok imenovan `second`. V bloku `second` najprej preverja enakost števil podanih v argumentih funkcije. V primeru enakosti skočimo na blok `if` in vrnemo `1`, sicer vrnemo `0`.

select

```
1  <result> = select selty <cond>, <ty> <val1>, <ty> <
       val2>
```

Izpis 2.17: Sintaksa ukaza select.

Ukaz `select` izbere vrednost glede na pogoj, ki je na izpisu 2.17 zapisan kot operand imenovan `cond`. Besedo `selty` moramo nadomestiti s ključno besedo `i1` ali {`<N x i1>`}. Če je vrednost operanda `cond` enaka `1`, ukaz vrne vrednost `val1` tipa `ty`, sicer pa vrne vrednost `val2` tipa `ty`. Tipa obeh vrednosti morata biti seveda enaka.

```
1  %X = select i1 true, i8 17, i8 42
```

Izpis 2.18: Primer uporabe ukaza select.

V izpisu 2.18 je prikazan primer uporabe ukaza `select`, kjer se izbere vrednost `17`, ker je pogoj vedno enak `1`.

call

```
1   <result> = [tail] call [cconv] [ret attrs] <ty> [<fnty
        >*] <fnptrval>(<function args>) [fn attrs]
```

Izpis 2.19: Sintaksa ukaza call.

Izpis 2.19 prikazuje celotno sintakso ukaza `call`. Za naše potrebe so pomembni samo obvezni operandi. Torej kot operand `ty` navedemo tip, ki ga funkcija vrača, operand predstavlja `fnptrval` ime funkcije, ki jo želimo poklicati, operand `function args` pa predstavlja seznam argumentov, ki je lahko tudi prazen.

```
1   %retval = call i32 @test(i32 %argc)
```

Izpis 2.20: Primer uporabe ukaza call.

Izpis 2.20 prikazuje primer klica funkcije `test`, ki vrača 32-bitno celo število, kjer kot argument podamo lokalno spremenljivko `argc`, ki je 32-bitno celo število. Vrnjena vrednost se shrani v lokalno spremenljivko `retval`.

fptoui

```
1   <result> = fptoui <ty> <value> to <ty2>
```

Izpis 2.21: Sintaksa ukaza fptoui.

Ukaz `fptoui` pretvori število v plavajoči vejici v celo število. Na izpisu 2.21 operand `ty` predstavlja tip, v katerem je trenutno operand `value`, recimo `double`, operand `ty2` pa predstavlja tip, v katerega želimo pretvoriti število v plavajoči vejici, recimo `i32`.

```
1   %tmp1 = fptoui double 1.00e+00 i1
```

Izpis 2.22: Primer uporabe ukaza fptoui.

Izpis 2.22 prikazuje pretvorbo števila `1,00` v eno bitno celo število, v tem primeru `1`.

alloca

```
1   <result> = alloca <type>[, <ty> <NumElements>][, align
        <alignment>]
```

Izpis 2.23: Sintaksa ukaza alloca.

V izpisu 2.23 je za nas pomemben samo operand `type`, ki predstavlja tip, za katerega želimo rezervirati prostor. Ukaz `alloca` rezervira prostor v velikosti podanega tipa na skladu. Ukaz vrača kazalec na prostor, ki je bil rezerviran.

```
1   %ptr = alloca i32
```

Izpis 2.24: Primer uporabe ukaza alloca.

Izpis 2.24 rezervira prostor za 32-bitno celo število in vrne kazalec tipa `i32*`.

load

```
1   <result> = load [volatile] <ty>* <pointer>[, align <
        alignment>][, !nontemporal !<index>][, !invariant.
        load !<index>]
```

Izpis 2.25: Sintaksa ukaza load.

Za naše potrebe sta pomembna samo obvezna operanda `ty` ter `pointer`, ki sta prikazana na izpisu 2.25. Operand `ty` predstavlja tip spremenljivke, ki ga želimo naložiti iz pomnilnika v operand `result`. Operand `pointer` predstavlja ime kazalca.

```
1   %p = alloca i32
2   store i32 42, i32* %p
3   %val = load i32* %p
```

Izpis 2.26: Primer uporabe ukaza load.

Izpis 2.26 prikazuje uporabo ukaza `load`, kjer najprej rezerviramo prostor (1. vrstica), nato naložimo na ta prostor vrednost `42` (2. vrstica) in

končno to vrednost preberemo iz pomnilnika ter jo shranimo v lokalno spremenljivko `val` (3.vrstica).

store

```
1   store [volatile] <ty> <value>, <ty>* <pointer>[, align
        <alignment>][, !nontemporal !<index>]
```

Izpis 2.27: Sintaksa ukaza store.

Za naše potrebe zadostuje uporaba obveznih operandov, ki jih prikazuje izpis 2.27. Operand `ty` predstavlja tip vrednosti, ki jo želimo shraniti v pomnilnik, operand `value` pa vrednosti, ki jo želimo shraniti v pomnilnik. Primer uporabe ukaza `store` je prikazan v izpisu 2.26.

phi

```
1   <result> = phi <ty> [ <val0>, <label0>], ...
```

Izpis 2.28: Sintaksa ukaza phi.

Ukaz `phi` potrebujemo v primerih, kadar imamo več blokov kode, po katerih skačemo, in si želimo imeti končno vrednost neke spremenljivke iz določenega bloka. Posledica uporaba principa SED nam onemogoča rabo ene same spremenljivke, ki bi lahko hranila vrednost ne glede, iz katerega bloka kode bi skočili. V izpisu 2.28 operand `ty` predstavlja tip, ki ga bomo vrnili. Operand `val0` predstavlja vrednost, ki jo želimo vrniti, če smo prejšnji ukaz izvršili v bloku kode imenovanemu v operandu `label0`. Teh parov operandov lahko navedemo poljubno mnogo.

```
1   define i32 @f(i32 %x, i32 %y) {
2    entry:
3     %cmptmp = icmp eq i32 %x, %y
4     br i1 %cmptmp, label %loop, label %end
5
6    loop:
```

```
7       %addtmp = add i32 %x, 1
8       br label %end
9
10     end:
11      %bodyret.0 = phi i32 [ %addtmp, %loop ], [ %x, %
            entry ]
12      ret i32 %bodyret.0
```

Izpis 2.29: Primer uporabe ukaza phi.

Izpis 2.29 prikazuje funkcijo `f`, ki vrne vrednost `x+1`, če sta števili `x` in `y` enaki, sicer pa vrne vrednost argumenta `x`. To nam omogoča ukaz `phi` v 11. vrstici izpisa 2.29, ki vrne vrednost spremenljivke `addtmp`, če je bil prejšni ukaz izveden v bloku kode imenovanem `loop`, sicer pa vrne vrednost spremenljivke `x`.

Pozoren bralec se morda sprašuje, zakaj v takih primerih preprosto ne uporabimo ukazov `load`/`store`. To je sicer rešitev, vendar je optimizacijsko slaba in prevede navadno dinamično dodeljene spremenljivke na skladu v ukaze tipa `phi`, ki omogoča hitrejše izvajanje.

Poglavje 3

Programski jezik Kaleidoscope

3.1 Opis jezika

V tem diplomskem delu bomo razširili in izboljšali prevajalnik za programski jezik Kaleidoscope. Kaleidoscope smo izbrali zato, ker je razvoj osnovnega prevajalnika za ta programski jezik predstavljen v začetniškem vodiču LLVM [6], v nadaljevanju vodič.

Gre za sila preprost postopkovni (*procedural*) programski jezik, ki ima funkcije in nima globalnih spremenljivk ali izrazov. Vsak globalen izraz (tj. izraz, ki ni v jedru neke funkcije) se pretvori v neimenovano funkcijo, ki v jedru funkcije vsebuje taisti izraz. To pomeni, da teh funkcij kasneje ne moremo več klicati, saj je nemogoče vedeti kakšno ime jim je bilo dodeljeno. Tak način pretvarjanja izrazov nam koristi le v primeru uporabe interaktivne konzole, več o tem nekoliko kasneje.

Programski jezik Kaleidoscope ima samo en tip in to so 64-bitna števila v plavajoči vejici (v programskem jeziku C znana kot števila tipa double), kar nam nekoliko olajša gradnjo prednjega dela prevajalnika, saj nam ni potrebno določati in preverjati tipov, kar navadno predstavlja velik del semantične analize. Poleg funkcij in njenih neobveznih parametrov ima programski jezik Kaleidoscope tudi lokalne spremenljivke, ki jim lahko spreminjamo vrednost. Od stavkov, ki omogočajo kontrolo pretoka, ima jezik pogojni stavek if ter

zanko for.

Podobno kot v programskem jeziku C++ lahko v programskem jeziku Kaleidoscope uporabniško določimo nove operatorje (binarne ali unarne) in jim določimo prioriteto glede na ostale operatorje.

Jezik poleg definicij funkcij omogoča tudi deklaracijo funkcij. To nam omogoča uporabo funkcij iz knjižnic programskega jezika C (npr. funkcije za izračun sinusa - sin) ter lastnih funkcij napisanih v programskem jeziku C. V izvedbi prevajalnika po vodiču sta lastni funkciji le dve, in sicer `putchard`, ki izpiše znak, zapisan v ACSII kodi, ter `printd`, ki izpiše število.

Na izpisu 3.1 vidimo primer kode, kjer so uporabljene vse ključne funkcionalnosti programskega jezika Kaleidoscope. Razlaga posameznih delov je zapisana v komentarjih, tj. v vrsticah, ki se začno z znakom `#`.

```
1  # definicija unarnega operatorja '-', negativna stevila
2  def unary-(v) 0-v;
3
4  # definicija binarnega operatorja vecji z prioriteto 10
5  def binary> 10 (l r) r<l;
6
7  # funkcija, ki izracuna n-ti clen Fibonaccijevega zaporedja
8  def fib(x)
9   if x < 3 then
10   1
11  else
12   fib(x-1)+fib(x-2);
13
14 # izracuna 10. clen Fibonaccijevega zaporedja
15 var x=9 in fib(x+1);
16
17 # deklaracija zunanje funkcije
18 extern putchard(x);
19
20 # funkcija, ki izpise n znakov '*'
21 def star(n)
22  for i=1, i<n, 1.0 in
23    putchard(42);
24
25 # klic funkcije star, ki izpise 16 znakov '*'
26 star(16);
```

Izpis 3.1: Primer programa v programskem jeziku Kaleidoscope.

V izvedbi prevajalnika po vodiču poteka izvajanje kode le preko vmesnika interaktivne konzole, kjer vpišemo izraz in se le-ta izvede. Vmesnik je podoben Pythonovem tolmaču. Izvajanje poteka preko prevajalnika JIT, o katerem nekoliko več v poglavju 3.5.

Na izpisu 3.2 lahko vidimo primer izvajanja funkcije `fib` v interaktivni konzoli. V interaktivni konzoli nam prevajalnik sproti izpisuje generirano vmesno kodo LLVM za deklaracije oz. definicije funkcij. Slednja možnost nam je prišla zelo v pomoč pri razvoju prevajalnika v fazi, kjer generiramo vmesno kodo. Naš vnos se nahaja v 1. in 21. vrstici.

```
1  ready> def fib(x) if x < 3 then 1 else fib(x-1)+fib(x-2);
2  ready> Read function definition:
3  define double @fib(double %x) {
4  entry:
5    %cmptmp = fcmp ult double %x, 3.000000e+00
6    br i1 %cmptmp, label %ifcont, label %else
7
8  else:                                             ; preds = %
       entry
9    %subtmp = fadd double -1.000000e+00, %x
10   %calltmp = call double @fib(double %subtmp)
11   %subtmp5 = fadd double -2.000000e+00, %x
12   %calltmp6 = call double @fib(double %subtmp5)
13   %addtmp = fadd double %calltmp, %calltmp6
14   br label %ifcont
15
16 ifcont:                                           ; preds = %
       entry, %else
17   %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %
         entry ]
18   ret double %iftmp
19 }
20
21 ready> fib(10);
22 ready> Evaluated to 55.000000
23 ready>
```

Izpis 3.2: Primer izvajanja v interaktivni konzoli.

Jezik ima naslednje ključne besede: `def`, `extern`, `if`, `then`, `else`, `for`, `in`, `var`, `binary`, `unary`. Imena so vsi nizi, ki niso enaki kateri izmed

ključnih besed, ter spadajo v množico, ki je določena z regularnim izrazom `[a-zA-Z][a-zA-Z0-9]*`. Kaleidoscope ima enovrstične komentarje, ki se začno z znakom `#` in končajo z koncem vrstice oz. znakom za novo vrstico. Podprte so tudi osnovne matematične binarne operacije: seštevanje (`+`), odštevanje (`-`), množenje (`*`), operator manjše (`<`).

V naslednjih poglavjih opišemo delovanje in izvedbo prevajalnika realiziranega po vodiču. Prevajalnik je napisan v programskem jeziku C++, saj je tako najlažje uporabljati LLVM-jeve knjižnice.

3.2 Leksikalna analiza

Leksikalna analiza je realizirana ročno. Različni tipi prej omenjenih osnovnih simbolov so definirani v enumeracijskem tipu imenovanem `Token`. Enumeratorji predstavljajo različne tipe osnovnih simbolov (npr. konec datoteke, ime, število, ključno besedo `def` ...). Celotna leksikalna analiza je realizirana v precej enostavni funkciji imenovani `getttok`, ki iz standardnega vhoda, v nadaljevanju vhod, bere znak za znakom.

Funkcija najprej preskoči vse znake, ki predstavljajo bel prostor (npr. presledek). Če iz vhoda prebere črko predpostavi, da gre za ime ali pa eno izmed ključnih besed. Če iz vhoda prebere cifro predpostavi, da gre za število v plavajoči vejici in zato bere znake iz vhoda toliko časa, dokler ne prebere katerega koli drugega znaka, ki ni število ali pika. Vrednost števila funkcija shrani v globalno spremenljivko tipa double, ki je dostopna vsem kasneje uporabljenim funkcijam v sintaksni analizi. Podobno se zgodi pri imenih, kjer je uporabljena globalna spremenljivka tipa string iz standardne knjižnice C++.

Če iz vhoda prebere znak `#`, bere znake toliko časa, dokler ne prebere znaka za novo vrstico. Če nismo prišli do konca datoteke funkcija vrne vrednost, ki jo vrne rekurzivni klic funkcije, v nasprotnem primeru vrne osnovni simbol za konec datoteke. Če iz vhoda prebere znak, ki ni črka ali številka in hkrati tudi ni znak za beli prostor, njegovo ASCII vrednost preprosto vrne.

Funkcija vrača celo število, ki je ena izmed vrednosti enumeratorjev ali pa predstavlja ASCII vrednost posebnega znaka (npr. +, -, *...).

3.3 Sintaksna analiza

Glavna naloga sintaksne analize je, da zaporedje osnovnih simbolov, ki ga prejema od leksikalne analize, pretvori v abstraktno sintaksno drevo. Abstraktno sintaksno drevo je sestavljeno iz posameznih vozlišč, ki predstavljajo določene dele izvorne kode. V ta namen so v izvedbi prevajalnika po vodiču pripravljeni naslednji razredi vozlišč. Glavni razred `ExprAST` je nadrazred vsem razredom, ki predstavljajo izraze (števila, binarne operatcije, stavek if ...). Kot smo omenili že v poglavju 3.1, programski jezik Kaleidoscope ne podpira globalnih spremenljivk oz. izrazov. Zategadelj se mora izraz nahajati v neki funkciji, ki jo predstavlja razred `FunctionAST`. Na sliki 3.1 lahko vidimo hierarhijo razredov abstraktnega sintaksnega drevesa. Sledi opis vseh razredov in njihovih atributov.

BinaryExprAST

Razred predstavlja binarni izraz (npr. seštevanje, odštevanje ...). Deklaracije atributov so predstavljene v izpisu 3.3.

```
1 char op;
2 ExprAST *lhs, *rhs;
```

Izpis 3.3: Atributi razreda BinaryExprAST.

Atribut `op` predstavlja znak operatorja binarne operacije, `lhs` predstavlja levi operand in `rhs` predstavlja desni operand.

CallExprAST

Razred predstavlja klic funkcije. Deklaracije atributov so predstavljene v izpisu 3.4.

```
1 std::string callee;
2 std::vector<ExprAST*> args;
```

Izpis 3.4: Atributi razreda CallExprAST.

Atribut `callee` predstavlja ime klicane funkcije, atribut `args` pa hrani seznam vseh argumentov klicane funkcije.

ForExprAST

Razred predstavlja izraz zanke for. Deklaracije atributov so predstavljene v izpisu 3.5.

```
1 std::string varName;
2 ExprAST *start, *end, *step, *body;
```

Izpis 3.5: Atributi razreda ForExprAST.

Atribut `start` predstavlja začetno vrednost, ki se dodeli spremenljivki z imenom, ki se hrani v atributu `varName`. V atributu `end` hranimo izraz, ki predstavlja pogoj zanke for. Izraz, ki se izvede v telesu zanke for, predstavlja atribut `body`. V atributu `step` hranimo izraz, ki se ob koncu izvajanja jedra zanke prišteje spremenljivki z imenom, ki je določen v atributu `varName`.

IfExprAST

Razred predstavlja stavek if. Deklaracije atributov so predstavljene v izpisu 3.6.

```
1 ExprAST *cond, *then, *elseExpr;
```

Izpis 3.6: Atributi razreda IfExprAST.

V atributu `then` hranimo izraz, ki se izvede ob resničnosti izraza v atributu `cond`, sicer se izvede izraz iz atributa `elseExpr`.

NumberExprAST

Razred predstavlja število v plavajoči vejici. Deklaracije atributov so predstavljene v izpisu 3.7.

```
1 double val;
```

Izpis 3.7: Atributi razreda NumberExprAS.T

Atribut `val` hrani vrednost števila v plavajoči vejici.

UnaryExprAST

Razred predstavlja unarne operacije. Deklaracije atributov so predstavljene v izpisu 3.8.

```
1 char opcode;
2 ExprAST *operand;
```

Izpis 3.8: Atributi razreda UnaryExprAST.

Atribut `opcode` predstavlja operator, ki izvede operacijo nad operandom v atributu `operand`.

VarExprAST

Razred predstavlja izraz var. Deklaracije atributov so predstavljene v Izpisu 3.9.

```
1 std::vector<std::pair<std::string, ExprAST*> > varNames;
2 ExprAST *body;
```

Izpis 3.9: Atributi razreda VarExprAST.

V atributu `varNames` hranimo imena spremenljivk in njihove vrednosti ob inicializaciji. Atribut `body` vsebuje izraz, ki se izvede po inicializaciji spremenljivk.

VariableExprAST

Razred predstavlja parametre funkcij. Deklaracija atributa je predstavljena v izpisu 3.10.

```
1 std::string name;
```

Izpis 3.10: Atributi razreda VariableExprAST.

Atribut `name` predstavlja ime parametra.

PrototypeExprAST

Razred predstavlja deklaracijo funkcije. Deklaracije atributov so predstavljene v izpisu 3.11.

```
1 std::string name;
2 std::vector<std::string> args;
3 bool isOperator;
4 unsigned precedence;
```

Izpis 3.11: Atributi razreda PrototypeExprAST.

Atribut `name` predstavlja ime funkcije, atribut `args` je seznam parametrov, atribut `isOperator` določa ali je funkcija uporabniško določen operator, če gre za uporabniško določen operator je določena še prednost v atributu `precedence`.

FunctionExprAST

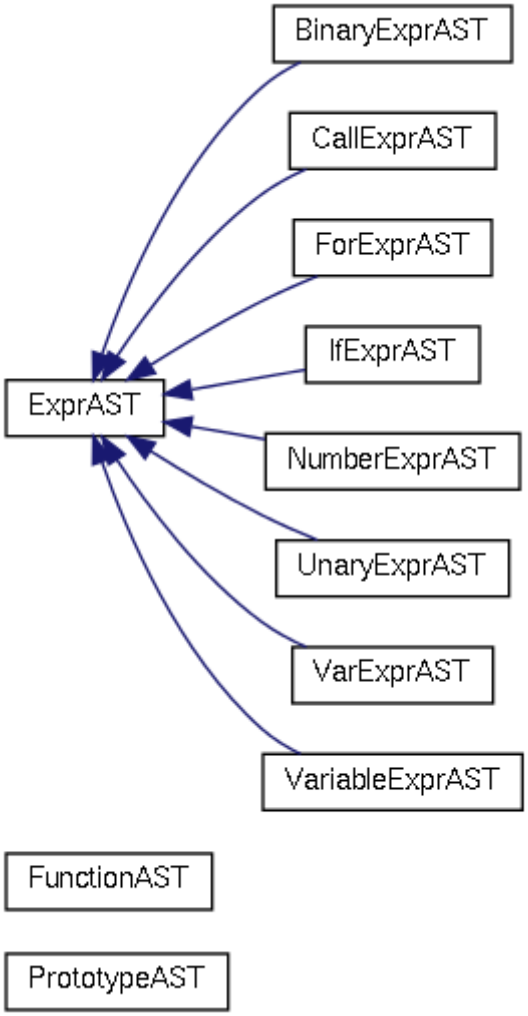
Razred predstavlja definicijo funkcije. Deklaracije atributov so predstavljene v izpisu 3.12.

```
1 PrototypeAST *proto;
2 ExprAST *body;
```

Izpis 3.12: Atributi razreda FunctionExprAST.

Atribut `proto` hrani deklaracijo funkcije, v atributu `body` pa hranimo izraz, ki je v telesu funkcije.

V izvedbi prevajalnika po vodiču poteka razčlenjevanje kode z vrha navzdol, z gramatiko, ki pripada razredu LL(1). Za vsako produkcijo gramatike je napisana funkcija, ki razčleni določeno produkcijo. Prednost posameznih operatorjev je določena z metodo razčlenjevanja prioritete operatorjev (*operator-precedence parsing*), ki uporablja slovar, v katerem določimo vrednost od 1 do 100 vsakemu od operatorjev, pri čemer vrednost 1 pomeni najmanjšo prioriteto, 100 pa najvišjo možno prioriteto. To nam omogoči preprosto rabo uporabniško določenih operatorjev, saj je problem prioritete rešen že v sintaksni analizi.



Slika 3.1: Hierarhija razredov, ki predstavljajo abstraktno sintaksno drevo.

3.4 Generiranje vmesne kode

Vsa koda do te faze ni uporabljala nobenega izmed razredov iz knjižnice LLVM. Prejšnji fazi sta splošni in bi jih lahko uporabili tudi pri kakšni drugi realizaciji prevajalnika, ki bi generirala drugačno vmesno kodo, recimo RTL (vmesna koda prevajalnika GCC). V tej fazi pretvorimo abstraktno sintaksno drevo v vmesno kodo LLVM. V tej fazi so prvič uporabljeni razredi iz knjižnice LLVM. Naštejmo nekaj glavnih razredov knjižnice LLVM, ki jih bomo uporabljali pri generiranju vmesne kode LLVM:

Module

Predstavlja modul LLVM, kot smo ga opisali v poglavju 2.4.1. Razred je vsebovalnik za vso vmesno kodo, ki jo generiramo. Preko tega razreda lahko preverimo, če je določena funkcija definirana (metoda `getFunction`). Razred nam omogoča tudi izpisovanje generirane vmesne kode (metoda `dump`).

LLVMContext

Razred hrani stanje prevajalnika. Zasnova razreda pride do izraza predvsem, ko želimo izvesti sočasno prevajanje v več neodvisnih nitih. To diplomsko delo se s sočasnim prevajanem ne ukvarja in zategadelj uporabljamo samo eno globalno stanje. Globalno stanje vrača funkcija `getGlobalContext`, ki je dostopna v področju imen (*namespace*) `llvm`.

IRBuilder

Razred omogoča generiranje in vstavljanje ukazov vmesne kode LLVM. Privzeto se generiran ukaz vstavi na konec trenutnega bloka kode. Vsebuje metode za generiranje vseh ukazov vmesne kode LLVM (npr. `CreateFAdd`, `CreateFSub` ...). Pri ukazih kot je npr. seštevanje skuša narediti optimizacijo in opraviti seštevanje, vendar to se lahko zgodi samo v primeru, če sta oba operanda konstanti.

ConstantFP

Razred predstavlja konstante, ki vsebujejo vrednosti števil v plavajoči

vejici. V našem primeru uporabljamo predvsem statično metodo `get`, ko pretvarjamo konstante (števila v plavajoči vejici) v predstavitev števila v vmesni kodi LLVM.

Function

Razred predstavlja definicijo funkcije v vmesni kodi LLVM. Vsebuje deklaracijo ter jedro funkcije, ki je sestavljeno iz blokov kode. S pomočjo tega razreda ustvarjamo nove funkcije (statična metoda `Create`), dodajamo bloke kode v funkcijo (preko metode `getBasicBlockList`), ugotovimo število parametrov (metoda `arg_size`), funkcijo pobrišemo iz trenutnega modula (metoda `eraseFromParent`) itd.

BasicBlock

Razred predstavlja osnovni blok kode. Blok kode je vsebovalnik ukazov, ki se zaporedno izvajajo. Nov blok kode ustvarimo preko statične metode `Create`, zelo uporabna metoda pa je tudi metoda `getParent`, ki nam vrne funkcijo, v kateri se blok nahaja.

Value

Razred je generalizacija za vse, kar ima vrednost ali pa jo program lahko izračuna. Slednjo vrednost lahko uporabimo kot operand za izračun drugih vrednosti. Razred `Value` je nadrazred razredom, ki predstavljajo konstante, ukaze ter funkcije. Vsaka instanca tega razreda mora imeti tudi določen tip (razred `Type`).

Type

Osnovni razred, ki predstavlja tipe. V našem primeru bomo pogosto uporabljali statično metodo `getDoubleTy`, ki vrača LLVM tip double.

Vsi zgoraj našteti razredi se nahajajo v področju imen (*namespace*) `llvm`.

Vmesno kodo generiramo z metodo `codegen`, ki je abstraktno deklarirana v razredu `ExprAST`, zato mora biti metoda v vseh razredih, ki dedujejo od razreda `ExprAST`, ponovna definirana. Metoda vrača kazalec na objekte tipa

`Value` zato, da lahko vrednost najprej uporabljamo v drugih izrazih. Razreda `PrototypeAST` ter `FunctionAST`, ki nista podrazreda razreda `ExprAST`, tudi vsebujeta metodo `codegen` le, da ta vrača objekte tipa `Function`. Po koncu sintaksne analize vedno dobimo nek objekt tipa `FunctionAST`, saj se tudi globalni izrazi prevedejo v anonimne funkcije, in na tem objektu nato izvedemo metodo `codegen`, ki nam poleg zgenerirane kode v modulu, vrne kazalec na objekt tipa `Function`. To nam omogoča izvajanje funkcij preko prevajalnika JIT, o katerem v poglavju 3.5.

Za generiranje vmesne kode so potrebne tri globalne spremenljivke, ki so predstavljene na izpisu 3.13. Gre za instanco razreda `Module`, kamor se shranjuje vsa generirana vmesna koda LLVM in je dokončno definirana v funkciji `main`. Nepogrešljiva je tudi instanca razreda `IRBuilder`, ki jo potrebujemo čisto v vsaki metodi `codegen` za generiranje vmesne kode. Zadnja zelo pomembna globalna spremenljivka `namedValues` pa predstavlja simbolno tabelo, v kateri se hrani imena spremenljivk in njihov naslov, kje na skladu se nahaja, saj s tem omogočimo branje oz. pisanje po tistem naslovu.

```
1 static Module *theModule;
2 static IRBuilder<> builder(getGlobalContext());
3 static std::map<std::string, AllocaInst*> namedValues;
```

Izpis 3.13: Globalne spremenljivke potrebne za generiranje vmesne kode.

V naslednjih poglavjih predstavimo generiranje vmesne kode za binarne izraze ter stavke if, ki predstavljajo ključne koncepte, ki so uporabljeni tudi pri generiranju drugih delov kode.

3.4.1 Generiranje vmesne kode za binarne izraze

Pri generiranju vmesne kode vgrajenih binarnih operacij je potrebno najprej pridobiti vrednosti levega in desnega operanda, kar je prikazano v 4. in 5. vrstici izpisa 3.14. Gre za klic metode ustreznega razreda, ki je lahko načeloma tudi rekurziven, če gre za daljši izraz. Operanda sta lahko katerega koli tipa (klic funkcije, ime spremenljivke ali število ...), to nas na tem mestu niti ne zanima. V naslednjem koraku preverimo za katero vrsto operacijo gre

in preko objekta `builder` vrnemo ustrezen ukaz ali pa kar izračunano vrednost (v primeru dveh konstant), ki se doda v trenutni blok kode. Funkciji za generiranje ukaza podamo vrednost levega in desnega operanda ter ime spremenljivke kamor se vrednost shrani. To ime spremenljivke se uporablja kot osnova za izbiro imena, navadno sledi številka, če je podano ime že uporabljeno, recimo `addtmp4`. Posledica tega je velika boljša preglednost, ko kodo ročno pregledujemo. Pri operaciji manjše moramo vrednost, ki jo vrne ukaz `fcmp`, še pretvoriti v število v plavajoči vejici, saj ukaz `fcmp` vrača celo število 0 ali 1. Iz imena klica metode za generiranje ukaza `fcmp` lahko tudi razberemo, da gre za primerjavo tipa `ult`, kar pomeni, da preverja ali je prvi operand manjši od drugega.

```
1  Value *BinaryExprAST::codegen() {
2    . . .
3
4    Value *l = lhs->codegen();
5    Value *r = rhs->codegen();
6
7    switch (op) {
8      case '+':
9        return builder.CreateFAdd(l, r, "addtmp");
10     . . .
11     case '/':
12       return builder.CreateFDiv(l,r, "divtmp");
13     case '<':
14       l = builder.CreateFCmpULT(l, r, "cmptmp");
15       return builder.CreateUIToFP(l, Type::getDoubleTy(
             getGlobalContext()),
16                                   "booltmp");
17     . . .
18   }
19   . . .
20 }
```

Izpis 3.14: Del funkcije, ki generira vmesno kodo binarnih operacij.

Za vhod `1+1` bi zgornja metoda zgenerirala vmesno kodo LLVM, ki bi predstavljala konstanto v plavajoči vejici zapisano kot `2.000000e+00`. Ker gre za dve konstanti, bi metoda `CreateFAdd` že kar sama naredila vsoto števil.

Oglejmo si bolj zanimiv primer na izpisu 3.15, kjer seštevamo spremenljivko `y` ter vrednost `1`. Na spodnjem primeru je tudi lepo predstavljeno delovanje spremenljivk zaradi principa SED (statična enkratna dodelitev). Najprej rezerviramo prostor na skladu (4. vrstica), na katerega shranimo vrednost `1.0` (5. vrstica), to je prevod vhodne kode `y=1`. Sledi pridobitev vrednosti iz naslova, ki smo ga rezervirali za spremenljivko `y` (6. vrstica). V 7. vrstici imamo ukaz, ki ga je zgenerirala zgornja metoda. Ukaz lahko prepoznamo po tem, da je končna vrednost ukaza shranjena v spremenljivko imenovano `addtmp`. To je prevod vhodne kode `y+1`. V spodnjem primeru spremenljivki `y1` prištevamo vrednost `1`, saj se imena spremenljivk prevedejo v ukaze `load`. Ker izraz var vrača vrednost v njegovem jedru, na koncu še vrnemo vrednost spremenljivke `addtmp`. Na spodnjem primeru lahko tudi vidimo, da se globalni izraz prevede v neimenovano funkcijo.

```
1 ready> var y=1 in y+1;
2 define double @0() {
3 entry:
4   %y = alloca double
5   store double 1.000000e+00, double* %y
6   %y1 = load double* %y
7   %addtmp = fadd double %y1, 1.000000e+00
8   ret double %addtmp
9 }
```

Izpis 3.15: Seštevanje v vmesni kodi LLVM.

V zgornji metodi `codegen` za binarne izraze je tudi realizirano generiranje vmesne kode za uporabniško določene operatorje. Ob definiciji uporabniško določenega operatorja, se zgenerira vmesna koda za funkcijo imenovano `binaryOPER`, kjer `OPER` predstavlja znak uporabniško določenega operatorja, npr. za operator `>` bi bila funkcija imenovana `binary>`. Če zgornja `codegen` funkcija ne prepozna vgrajenega operatorja, v modulu `theModule` poiščemo funkcijo z imenom, ki ga sestavimo iz niza `binary` ter znaka `op`, ki ga hrani razred `BinaryExprAST`. Nato preko objekta `builder` zgeneriramo vmesno kodo za klic funkcije, ki ji kot argumenta podamo levi in desni operand.

3.4.2 Generiranje kode za stavek if

Pri generiranju vmesne kode za stavek if si za lažjo predstavo najprej oglejmo vmesno kodo, ki bi jo radi zgenerirali.

```
1 extern println(x);
2 def f(x) if x then println(42.0) else println(43.1);
```

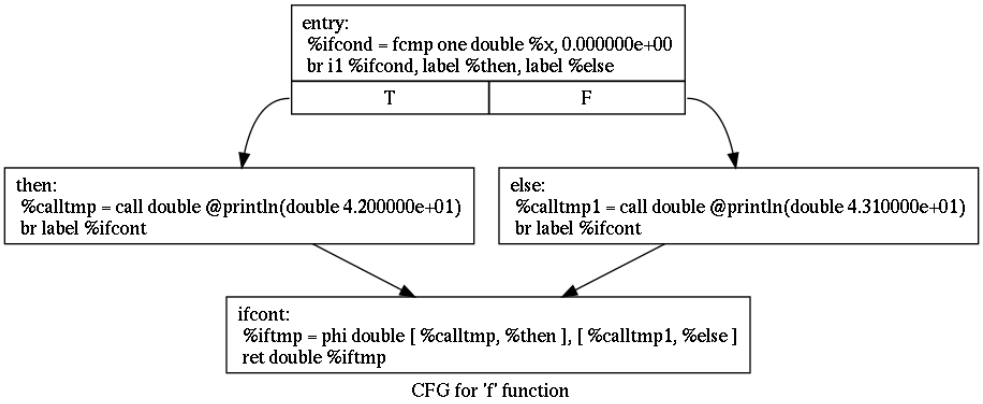
Izpis 3.16: Stavek if v programskem jeziku Kaleidoscope.

Za kodo na izpisu 3.16 želimo, da se prevede v vmesno kodo predstavljeno na izpisu 3.17. Tu je potrebna uporaba ukaza `phi`, saj želimo, da stavek if vrača vrednost ukaza, ki se izvede. Stavek if v programskem jeziku Kaleidoscope deluje podobno kot v programskem jeziku C. Če je vrednost pogoja `0.0`, je pogoj neresničen, v vseh ostalih primerih pa resničen. Skoke lahko nazorno vidimo na sliki 3.2.

```
1  declare double @println(double)
2
3  define double @f(double %x) {
4  entry:
5    %ifcond = fcmp one double %x, 0.000000e+00
6    br i1 %ifcond, label %then, label %else
7
8  then:           ; preds = %entry
9    %calltmp = call double @println(double 4.200000e+01)
10   br label %ifcont
11
12 else:           ; preds = %entry
13   %calltmp1 = call double @println(double 4.310000e+01)
14   br label %ifcont
15
16 ifcont:         ; preds = %else, %then
17   %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else
         ]
18   ret double %iftmp
19 }
```

Izpis 3.17: Željena generirana koda za stavek if.

V funkciji `codegen` za generiranje vmesne kode stavka if najprej pridobimo vrednost izraza, ki predstavlja pogoj. Z ukazom `fcmp` preverimo, če je vrednost različna od `0.0`, in si to vrednost zapomnimo. Nato moramo ustvariti



Slika 3.2: Graf, ki predstavlja vmesno kodo stavka if.

tri nove bloke kode: blok kode `then`, na katerega skočimo v primeru resničnosti izraza; blok kode `else`, na katerega skočimo v primeru neresničnosti izraza; blok kode, na katerega brezpogojno skočimo iz blokov `then` ali `else`, imenovanega `ifcont`. Blok kode `then` dodamo na konec trenutne funkcije. Preostala bloka dodamo kasneje. Ustvarimo ukaz `br` in določimo skoke. Opisano je prikazano v izpisu 3.18.

```
1  Value *IfExprAST::codegen() {
2    Value *condV = cond->codegen();
3    if (condV == 0)
4      return 0;
5
6    condV = builder.CreateFCmpONE(condV, ConstantFP::get(
       getGlobalContext(), APFloat(0.0)), "ifcond");
7
8    Function *theFunction = builder.GetInsertBlock()->getParent
       ();
9
10   BasicBlock *thenBB = BasicBlock::Create(getGlobalContext(),
       "then", theFunction);
11   BasicBlock *elseBB = BasicBlock::Create(getGlobalContext(),
       "else");
12   BasicBlock *mergeBB = BasicBlock::Create(getGlobalContext()
       , "ifcont");
13
14   builder.CreateCondBr(condV, thenBB, elseBB);
15 . . .
```

Izpis 3.18: Generiranje kode za stavek if.

Nadaljujemo z vstavljanjem kode v blok `then`. Pridobimo vrednost izraza, ki se izvede, če je pogoj resničen, in dodamo ukaz za brezpogojni skok na končni blok kode `ifcont`. To je prikazano v izpisu 3.19.

```
1  . . .
2    builder.SetInsertPoint(thenBB);
3
4    Value *thenV = then->codegen();
5    if (thenV == 0)
6      return 0;
7
8    builder.CreateBr(mergeBB);
9    thenBB = builder.GetInsertBlock();
10 . . .
```

Izpis 3.19: Nadaljevanje kode za stavek if.

Na konec trenutne funkcije dodamo blok kode `else` ter podobno kot pri bloku `then` pridobimo vrednost izraza, ki se izvede, če je pogoj neresničen, in dodamo brezpogojni skok na konec. To je prikazano v izpisu 3.20.

```
1  . . .
2    theFunction->getBasicBlockList().push_back(elseBB);
3    builder.SetInsertPoint(elseBB);
4
5    Value *elseV = elseExpr->codegen();
6    if (elseV == 0)
7      return 0;
8
9    builder.CreateBr(mergeBB);
10   elseBB = builder.GetInsertBlock();
11 . . .
```

Izpis 3.20: Nadaljevanje kode za stavek if.

V trenutno funkcijo dodamo še blok kode `ifcont`, ki vsebuje ukaz `phi`, ta vsebuje vrednost zadnjega izvedenega ukaza. To vrednost nato le še vrnemo. Prikazano v izpisu 3.21.

```
1  . . .
```

```
2   theFunction->getBasicBlockList().push_back(mergeBB);
3   builder.SetInsertPoint(mergeBB);
4
5   PHINode *pn = builder.CreatePHI(Type::getDoubleTy(
        getGlobalContext()), 2, "iftmp");
6
7   pn->addIncoming(thenV, thenBB);
8   pn->addIncoming(elseV, elseBB);
9
10  return pn;
11 }
```

Izpis 3.21: Nadaljevanje kode za stavek if.

3.5 Izvajanje kode

V izvedbi prevajalnika po vodiču je izvajanje kode realizirano le preko prevajalnika JIT (*just-in-time*). Prevajalnik JIT nam med izvajanjem omogoča prevajanje vmesne kode LLVM v strojno kodo, ki jo nato izvede. Prevajalnik JIT je v času pisanja popolnoma podprt za procesorje x86, x86_64 ter MIPS. Na ostalih nepodprtih procesorjih je izvajanje možno le preko tolmača.

Izvajanje v prevajalniku je omogočeno s pomočjo abstraktnega razreda `ExecutionEngine`. Novo instanco objekta ustvarimo preko metode `create` razreda `EngineBuilder`, ki na podlagi podatkov o trenutnem procesorju, pripravi okolje za izvajanje preko prevajalnika JIT ali tolmača. Instanca objekta nam omogoča izvajanje funkcij vmesne kode LLVM. To je tudi eden izmed glavnih razlogov, zakaj se globalni izrazi prevedejo v neimenovane funkcije, saj to omogoča preprosto izvajanje globalnih izrazov. Izvajanje poteka preko metode `runFunction`. Klic metode dodamo v razčlenjevalno funkcijo `handleTopLevelExpression`, ki potem, ko zgenerira vmesno kodo LLVM, funkcijo tudi zažene, brez argumentov. Vrednost, ki jo funkcija vrača izpiše na standarni izhod.

Koda na izpisu 3.22 najprej razčleni globalni izraz, nato abstraktno sintaksno drevo pretvori v funkcijo v vmesni kodi LLVM in na koncu izvede to funkcijo. Spremenljivka `theExecutionEngine` je globalna spremenljivka, ki

je objekt tipa `ExecutionEngine`.

```
1  static void handleTopLevelExpression() {
2    if (FunctionAST *f = parseTopLevelExpr()) {
3      if (Function *lf = f->codegen()) {
4          std::vector<GenericValue> noargs;
5          GenericValue result = theExecutionEngine->runFunction
               (lf, noargs);
6          fprintf(stdout,"Evaluated␣to␣%f\n", result.DoubleVal)
               ;
7        }
8      }
9  . . .
10 }
```

Izpis 3.22: Izvajanje neimenovanih funkcij.

Na izpisu 3.23 vidimo primer izvajanja, kjer se izvede samo drugi izraz, saj za prvega niti ne bi bilo smiselno izvajanje, ker gre za definicijo funkcije.

```
1 ready> def f(x) 2*x;
2 ready> Read a function definition:
3 ready> f(16);
4 ready> Read a top-level expr:
5 Function returned 32.00.
6 ready>
```

Izpis 3.23: Primer izvajanja.

3.6 Optimizacija kode

Knjižnica LLVM omogoča več metod za optimizacijo zgenerirane vmesne kode. Za to uporablja obhode kode (*pass*). Obhode kode delimo v tri osnovne skupine. Tiste, ki vmesni kodi LLVM dodajo dodatne informacije (analizirajo vmesno kodo) in služijo v pomoč drugim obhodom. Obhode, ki spremenijo oz. transformirajo kodo, in pomožne obhode, ki ne sodijo ne v prvo in ne v drugo skupino. Slednji na primer vmesno kodo pretvorijo v strojno kodo in podobno. Prevajalnik po vodiču in tudi naš prevajalnik bo uporabljal obhode druge vrste, ki nam bodo precej optimizirali vmesno kodo. Knjižnica

LLVM ponuja številne obhode, v prevajalniku po vodiču pa je uporabljenih naslednjih pet obhodov.

mem2reg

Spremeni vse na skladu dodeljene spremenljivke (preko ukaza `alloca`) v SED spremenljivke in ukaze tipa `phi`. Obhod močno optimizira generirano kodo v našem primeru, saj se znebimo vseh parov ukazov `load`/`store`. Obhod deluje samo za spremenljivke, ki imajo rezerviran prostor na skladu ter imajo rezervacijo narejeno v `entry` bloku kode. Za naš primer vse te reči držijo.

instcombine

Obhod združuje algebraične ukaze in s tem ustvari manj ukazov bolj preproste ukaze. Recimo pri seštevanju, kjer spremenljivki prištejemo ena, nato pa dobljenemu rezultatu še ena, obhod spremeni ukaz v seštevanja, kjer prvotni spremenljivki prištejemo dve.

reassociate

Spremeni vrstni red pri komutativnih izrazih. Primer: izraz 1 + (x + 2) spremeni v x + (1 + 2).

gvn

Odstrani ukaze za nalaganje vrednosti, ki so že na voljo. Recimo, če imamo dva zaporedna ukaza `load`, ki dostopa do iste spremenljivke, je eden ukaz odveč in ga zato ta obhod odstrani.

simplifycfg

Odstrani mrtvo kodo ter združuje bloke kode. Recimo odstrani blok kode, ki vsebuje samo brezpogojni skok in podobno.

Na izpisu 3.24 lahko vidimo neoptimizirano vmesno kodo LLVM funkcije, ki izračuna n-ti Fibonaccijevega zaporedja, predstavljeno na izpisu 3.1 v poglavju 3.1. Na izpisu 3.25 pa lahko vidimo optimizirano kodo iste funkcije, ki uporablja vse zgoraj naštete transformacije.

```
define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %
    entry
  br label %ifcont

else:                                             ; preds = %
    entry
  %x3 = load double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:                                           ; preds = %
    else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %
      else ]
  ret double %iftmp
}
```

Izpis 3.24: Primer neoptimizirane vmesne kode.

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  br i1 %cmptmp, label %ifcont, label %else

else:                                             ; preds = %
    entry
  %subtmp = fadd double -1.000000e+00, %x
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fadd double -2.000000e+00, %x
  %calltmp6 = call double @fib(double %subtmp5)
```

```
11   %addtmp = fadd double %calltmp, %calltmp6
12   br label %ifcont
13
14 ifcont:                                          ; preds = %
       entry, %else
15   %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %
         entry ]
16   ret double %iftmp
17 }
```

Izpis 3.25: Primer optimizirane kode z zgoraj naštetimi transformacijami.

Poglavje 4

Razširitve

4.1 Leksikalna analiza

V izvedbi prevajalnika po vodiču je leksikalna analiza implementirana ročno. Za začetne predstavitvene potrebe prevajalnika taka implementacija sicer zadostuje, ko pa začnemo programski jezik in s tem prevajalnik razširjati, dodajati nove funkcionalnosti, pa taka implementacija ni več primerna. V tem podpoglavju zato opišemo implementacijo leksikalne analize s pomočjo orodja Flex. Flex omogoča bolj strukturirano in pregledno implementacijo leksikalne analize, predvsem pa olajša kasnejše dodajanje novih osnovnih simbolov v programski jezik. Poleg tega pa implementacija leksikalne analize v izvedbi prevajalnika po vodiču vsebuje hrošče. Recimo, vhodni niz `2.2.2` prepozna kot število `2.2`, preostali del niza (`.2`) pa odreže. Pravilno bi bilo, da bi v tem primeru prevajalnik javil napako.

4.1.1 Orodje Flex

Flex (*fast lexical analyzer generator*) [1] je generator leksikalnih analizatorjev. Gre za odprtokoden projekt, ki je izdan pod BSD licenco. V osnovi je Flex program, ki iz datoteke prebere opis leksikalnega analizatorja in generira kodo leksikalnega analizatorja v programskem jeziku C ali C++. Datoteka je sestavljena iz parov regularnih izrazov ter C kode, ki jih imenujemo pravila.

Izhod programa Flex je datoteka `lex.yy.c` ki vsebuje funkcijo imenovano `yylex`, ki ob klicu analizira vhodne znake in ob ujemanju z določenim regularnim izrazom, izvede pripadajočo akcijo, napisano v C kodi. Datoteka `lex.yy.c` predstavlja realizacijo determinističnega končnega avtomata z več deset stanji. Običajno se Flex uporablja v kombinaciji z orodjem Bison [2], kar bomo storili tudi v našem primeru. Naš sintaksni analizator bo uporabljal funkcijo `yylex` za pridobitev osnovnih simbolov iz vhoda. V tem diplomskem delu uporabljamo Flex različice 2.5.37.

4.1.2 Opis leksikalnega analizatorja

Datoteka, v kateri se nahaja opis leksikalnega analiazorja, ima običajno končnico `flex`. Struktura datoteke je prikazana na izpisu 4.1. V odseku definicije lahko določimo različna imena določenim regularnim izrazom, ki jih potem uporabljamo v odseku pravil. Recimo `CIFRA [0-9]` dodeli ime `CIFRA` regularnem izrazu `[0-9]`. Ime lahko nato prosto uporabljamo v odseku pravil, `{CIFRA}+"."{CIFRA}+` pomeni isto kot naslednji regularni izraz `([0-9])+"."([0-9])+`. V odsek definicij navadno podamo zaglavne datoteke in drugo C kodo, za katero želimo, da se doda pred funkcijo `yylex`. Zaglavne datoteke in C kodo moramo zapisati med niza `%{` ter `%}`. V odseku uporabniška koda lahko vključimo kakršno koli C kodo (ni je potrebno zapisati med prej omenjena niza), za katero želimo, da se doda na konec datoteke `lex.yy.c`. Na tem mestu je ponavadi definirana funkcija `main`. Odsek je lahko tudi prazen.

```
1 definicije
2 %%
3 pravila
4 %%
5 uporabniska koda
```

Izpis 4.1: Struktura datoteke flex.

V odseku pravil podamo pare regularnih izrazov in akcij. Če želimo napisati akcijo v več vrsticah, lahko le-te podamo med zavite oklepaje. Na izpisu 4.2

lahko vidimo primer, kjer imamo eno pravilo, ki ob pojavitvi niza `uporabnik` le-tega zamenja z uporabniškim imenom trenutno prijavljenega uporabnika. Privzeto pravilo za vse ostale nize, ki se ne ujamejo z našimi pravili, je izpis na standardni izhod. Na spodnjem primeru lahko v odseku uporabniške kode vidimo uporabo spremenljivke `yyin`, ki predstavlja vhod iz kje naj poteka branje znakov, privzeto bere znake iz standardnega vhoda.

```
1  %{
2          #include <unistd.h>
3  %}
4
5  %option noyywrap
6
7  %%
8  uporabnik         printf("%s", getlogin());
9  %%
10 int main(int argc, char **argv) {
11   if (argc == 2) {
12     yyin = fopen(argv[1], "r");
13     yylex();
14   }
15
16   return 0;
17 }
```

Izpis 4.2: Struktura datoteke flex.

V pomoč pri pisanju akcij so nam na voljo naslednje spremenljivke, ki nam močno olajšajo delo:

char **yytext

Predstavlja kazalec na tabelo, kjer se nahaja trenutni niz, ki se ujema z regularnim izrazom pravila.

int yyleng

Predstavlja dolžino niza, ki se ujema z regularnim izrazom pravila.

FILE *yyin

Predstavlja vhod, iz kjer funkcija `yylex` bere znake.

int yylineno

Dosegljiva le v primeru, če to določimo v odseku deklaracij (`%option yylineno`). Spremenljivka predstavlja trenutno vrstico, v kateri se nahajamo.

4.1.3 Priprava datoteke za prevajalnik

Pri pripravi datoteke flex za prevajalnik je potrebno najprej določiti vse tipe osnovnih simbolov. Te smo že našteli v poglavju 3.1. V grobem ima programski jezik Kaleidoscope naslednje različne tipe osnovnih simbolov: različne ključne besede, znake, ki predstavljajo operatorje; imena ter števila v plavajoči vejici. V naši izvedbi prevajalnika bolj omejimo množico znakov, ki jih lahko uporabimo za definicijo uporabniško določenih operatorjev. V izvedbi prevajalnika po vodiču lahko kot znak za uporabniško določen operator uporabimo vse ASCII znake, ki niso števila ali črke. V naši izvedbi te znake omejimo na !, ", $, %, &, ', ., >, ?, @, \, ^, _, |, {, }, ~ ter :.

```
1  %{
2          #include <cstdio>
3          #include <cstdlib>
4
5          int debug = 0;
6          #define DEBUG(t) if (debug) printf("%s\n", t);
7
8  %}
9
10 %option noyywrap
11 %option yylineno
12
13 %%
14 [ ]+                      { }
15 (\r)?\n                   { }
16 \t                       { }
17 #.*\n?                   { }
18 "def"                     { DEBUG("DEF"); }
19 . . .
20 "+"                      { DEBUG("ADD"); }
21 . . .
22
23 [0-9]+(\.[0-9]+)?         { DEBUG("DOUBLE"); }
```

```
24 . . .
25
26 .                          { DEBUG("N/A");
27                              printf("Invalid␣symbol.\n"); }
28 <<EOF>>                     { DEBUG("ENDF"); }
```

Izpis 4.3: Del opisa leksikalnega analizatorja za Kaleidoscope.

Na izpisu 4.3 je prikazan del datoteke flex za prevajalnik. V odseku definicij podamo knjižnice programskega jezika C++. Definiramo spremenljivko `debug`, ki nam bo služila v razhroščevalne namene. S pomočjo makroja `DEBUG`, ki izpiše vrsto osnovnega simbola, si v sintaksni fazi pomagamo, da vidimo točno kateri osnovni simbol je bil nazadnje prebran, ko gre kaj narobe. V odseku pravil lahko vidimo, da preskočimo vse komentarje in znake, ki predstavljajo bel prostor. Uporabljeno je tudi posebno stanje <<`EOF`>>, ki predstavlja konec datoteke. Za enkrat smo definirali samo pravila, akcije bomo dopolnili v naslednjem poglavju, ko bomo integrirali Flex z Bisonom. Pri akcijah nam manjka shranjevanje vrednosti osnovnih simbolov, manjkajo nam tudi lokacije, kje se pojavljajo določeni osnovni simboli. Vrednosti bomo shranjevali v spremenljivko `yylval`, gre za strukturo tipov osnovnih simbolov, ki jo definirano v sintaksni analizi.

4.2 Sintaksna analiza

Sintaksa analiza je v izvedbi prevajalnika po vodiču realizirana ročno. To pomeni, da imamo veliko število funkcij, v katerih je uporabljena tudi rekurzija. Posledično lahko taka koda prav hitro postane nepregledna in zato je težko dodajati nove funkcionalnosti v programski jezik. Na tem mestu nam priskoči na pomoč orodje Bison. Orodje Bison nam omogoča realizacijo sintaksne analize na strukturiran in pregleden način.

4.2.1 Orodje Bison

Orodje Bison je generator razčlenjevalnikov vrste LALR(1) [2]. Deluje na podoben princip kot orodje Flex, kot vhod pripravimo datoteko (običajno s končnico `y`), ki opisuje kontekstno neodvisno gramatiko. Kot izhod program `bison` zgenerira datoteko `parser.tab.c`, ki predstavlja razčlenjevalnik vrste LALR(1) napisan v programskem jeziku C, za podano kontekstno neodvisno gramatiko. Datoteka `parser.tab.c` vsebuje funkcijo `yyparse` preko katere poteka razčlenjevanje. Funkcija za svoje delovanje potrebuje funkcijo `yylex` preko katere prejema vhodne osnovne simbole. V našem primeru bomo uporabili funkcijo `yylex`, ki jo zgenerira orodje Flex. V tem diplomskem delu uporabljamo orodje Bison različice 2.4.3.

4.2.2 Opis kontekstno neodvisne gramatike

Struktura datoteke, v kateri opišemo kontekstno neodvisno gramatiko, je podobna strukturi datoteke, v kateri smo opisali leksikalni analizator pri orodju Flex. Osnovna struktura datoteka je predstavljena na izpisu 4.4.

V odsek prolog podamo kodo v programskem jeziku C, za katero želimo, da se doda nekje na začetku izhodne datoteke `parser.tab.c`. Običajno so v tem odseku podane zaglavne datoteke, deklaracije spremenljivk in funkcij.

V odsek deklaracije Bison podamo definicije različnih tipov osnovnih simbolov, osnovnim simbolom in posameznim produkcijam določimo tipe, določimo lahko tudi prednost in asociativnost določenih osnovnih simbolov.

V odseku pravil gramatike na podoben način kot pri orodju Flex podamo kontekstno neodvisno gramatiko v obliki, ki je podobna obliki BNF (*Backus-Naur Form*) in semantične akcije, ki naj se zgodijo ob ujemanju posamezne produkcije. V odseku epilog lahko opcijsko podamo kodo v programskem jeziku C, ki se bo dodala na konec datoteke `parser.tab.c`. Običajno na tem mestu najdemo definicije v prologu deklariranih funkcij.

```
1 %{
2 prolog
3 %}
```

```
4 deklaracije Bison
5 %%
6 pravila gramatike
7 %%
8 epilog
```

Izpis 4.4: Struktura vhodne datoteke Bison.

V odseku pravil gramatike produkcijam določamo semantične akcije. Kot smo zgoraj omenili lahko vsaki produkciji in osnovnemu simbolu določimo tip (npr. `int` - celo število). Primer, kjer nastopa samo en tip lahko vidimo na izpisu 4.5. Kadar uporabljamo samo en tip ga določimo v prologu (2. vrstica na izpisu 4.5), v primeru več tipov pa je potrebno uporabiti unijo, o kateri nekoliko več kasneje. Z deklaracijo `%token` določimo končne simbole, v primeru uporabe več tipov mu na tem mestu določimo še tip. Primer v 6. vrstici deklarira končni simbol `NUM`, ki je tipa celo število (`int`), saj je to edini tip. Z deklaracijo `%left` določimo levo asociativnost končnih simbolov `+` ter `-`. V 12. vrstici lahko vidimo primer uporabe semantične vrednosti nekončnega simbola `expr` produkcije `line`. Do semantičnih vrednosti trenutne produkcije dostopamo prek imena `$$`, do n-tega simbola trenutne produkcije pa prek imena `$n`. V 16. vrstici najprej seštejemo semantično vrednost prvega in tretjega simbola. Dobljena vrednost postane semantična vrednost celotne produkcije.

```
1  %{
2    #define YYSTYPE int
3    #include <stdio.h>
4  %}
5
6  %token NUM
7
8  %left '+' '-'
9
10 %%
11 line:
12   expr '\n' { printf("%d\n", $1); }
13 ;
14 expr:
15   NUM             { $$ = $1;       }
16   | expr '+' expr { $$ = $1 + $3; }
```

```
17    | expr '-' expr { $$ = $1 - $3; }
18  ;
19  %%
```

Izpis 4.5: Preprost primer.

Zgornjemu primeru manjka še ključna funkcija `yylex`. V našem primeru bomo za implementacijo te funkcije uporabili orodje Flex. Sledi seznam konstruktov, ki jih bomo potrebovali pri gradnji sintaksnega analizatorja za programski jezik Kaleidoscope:

yyparse

Ime glavne funkcije, kjer poteka razčlenjevanje osnovnih simbolov, ki jih prejme iz funkcije `yylex`. Funkcija vrne `0` v primeru uspešnega razčlenjevanja oz. `1` v primeru napake. Razčlenjevanje lahko ročno prekinemo z uporabo makroja `YYACCEPT`, v primeru uspeha, oz. `YYABORT`, v primeru neuspeha.

yyerror

Ime funkcije, ki jo kliče funkcija `yyparse` v primeru napake.

yylval

Ime spremenljivke, v katero v času leksikalne analize shranimo vrednost trenutnega osnovnega simbola. V primeru uporabe več tipov (deklaracija `%union`) je to unija, kjer so dostopni vsi deklarirani tipi.

yylloc

Ime spremenljivke, ki predstavlja strukturo, v katero shranjujemo lokacijo trenutnega leksema. V našem primeru bomo uporabljali kar privzeto strukturo, ki vsebuje naslednje člane: `first_line` (vrstica, kjer se leksem začne), `first_column` (stolpec, kjer se leksem začne), `last_line` (vrstica, kjer se leksem končna), `last_column` (stolpec, kjer se leksem konča).

4.2.3 Priprava kontekstno neodvisne gramatike

Preden smo začeli pripravljati datoteko, ki opisuje sintaksni analizator za programski jezik Kaleidoscope, smo morali pripraviti kontekstno neodvisno gramatiko, ki predstavlja celotno sintakso programskega jezika. Pri tem smo si pomagali s funkcijami ročne sintaksne analize v izvedbi prevajalnika po vodiču.

Po podrobni analizi smo ugotovili, da imamo težavo pri določanju prednosti operatorjev. Težava se pojavi, ko v izrazu nastopajo uporabniško določeni operatorji. Kako v tem primeru ugotoviti kateri operator ima najvišjo prioriteto in kateri najmanjšo prioriteto?

V izvedbi sintaksne analize po vodiču se ob razčlenjevanju definicije uporabniškega določenega operaterja doda v slovar tudi informacija kako močno določen operator veže. Za vgrajene operatorje pa to informacijo podamo v funkciji `main`, tako imamo v vsakem trenutku na voljo informacijo kako močno določen operator veže. S pomočjo slovarja in funkcije, ki razčlenjuje prioritete operatorjev (*operator-precedence parsing*), lahko vedno sestavimo pravilno abstraktno sintaksno drevo, ki upošteva posamezne prioritete operatorjev. V primeru generiranja razčlenjevalnika z orodjem Bison pa te informacije ne bomo imeli.

Problem lahko rešujemo na več načinov. Lahko bi še vedno shranjevali posamezne prioritete operatorjev. V kontekstno neodvisni gramatiki bi vsem operatorjem določili isto prioriteto. Po končanem razčlenjevanju pa bi naredili obhod po abstraktnem sintaksnem drevesu in drevo spremenili tako, da bi bile upoštevane prave prioritete. Pri tem bi morali ohraniti oklepaje v izrazih, saj te ključno vplivajo na prioriteto.

Zaradi precejšne kompleksnosti slednje rešitve smo se odločili, da določanje prednosti uporabniških operatorjev ukinemo. Tako imajo uporabniško določeni binarni operatorji najmanjšo prioriteto, uporabniško določeni unarni operatorji pa največjo. Če bi v izrazu želeli drugačno prioriteto lahko to rešimo z uporabo oklepajev.

Druga težava, ki se je pojavila pri načrtovanju kontekstno neodvisne gra-

matike, je zaključevanje stavčnih izrazov. Primer lahko vidimo na izpisu 4.6. Ali bo rezultat spodnjega izraza v 1. vrstici 1 ali 3? Na prevajalniku po vodiču bi bil rezultat spodnjega izraza 1. Rezultat 3 bi lahko dobili le z uporabo oklepajev pri prvem stavku if. Zaradi dvoumnosti oz. neberljivosti take sintaksne smo se odločili, da na koncu vsakega stavka if dodamo ključno besedo `fi` in s tem odpravimo konflikt pomik/prevedba (*shift/reduce conflict*) ter izboljšamo berljivost kode. Primer nove sintaksne je prikazan v 4. vrstici. Podobno smo dodali var stavku na konec ključno besedo `end` ter zanki for ključno besedo `done`.

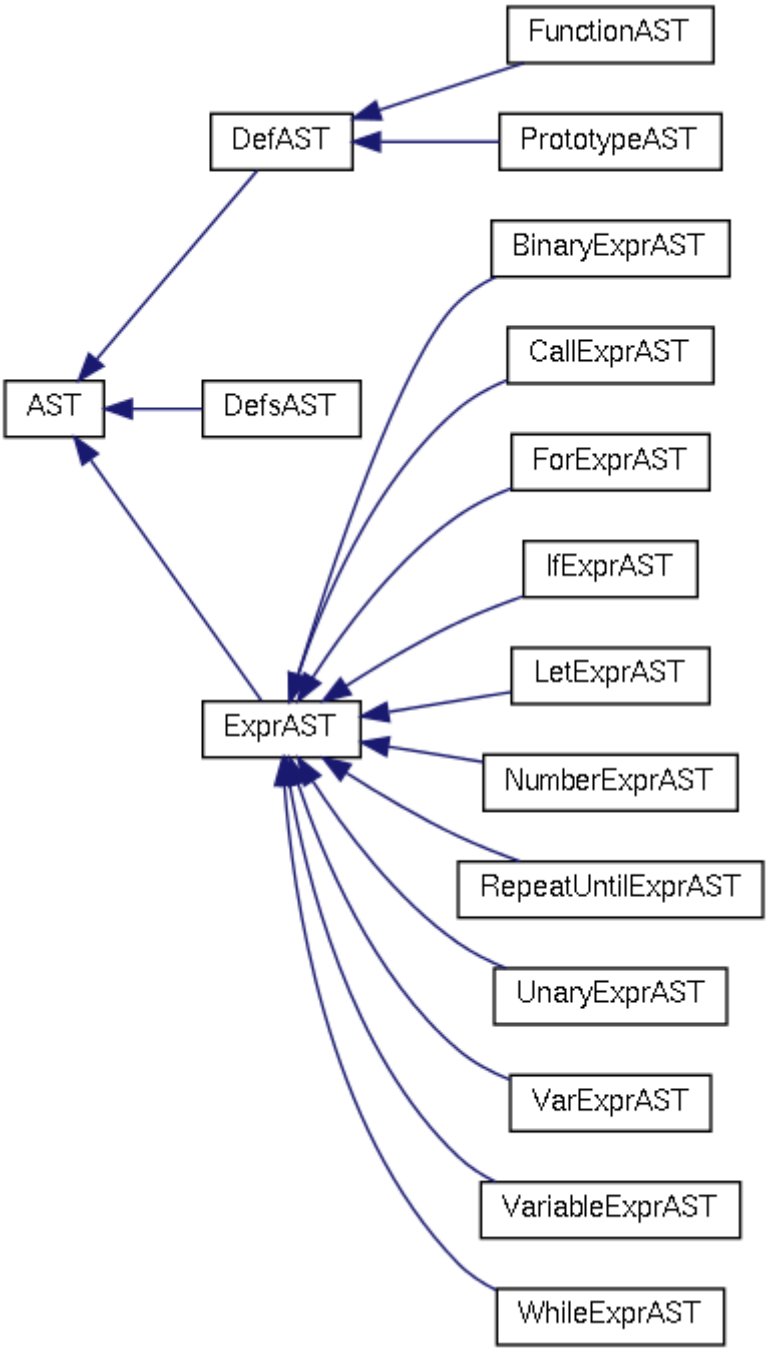
```
1 if 1 then 1 else 0+if 1 then 3 else 5
2
3 # dodana kljucna beseda 'fi'
4 if 1 then 1 else 0 fi+if 1 then 3 else 5 fi
```

Izpis 4.6: Primer dvoumnosti stavka if.

V kontekstno neodvisno gramatiko smo tudi dodali `,` (vejico), ki med seboj loči različne definicije oz. deklaracije funkcij. To je pomenilo, da smo morali dodati tudi novi vozlišči v naše sintaksno drevo, `DefAST` - predstavlja definicijo oz. deklaracijo funkcije, `DefsAST` - vsebuje eno ali več instanc razreda `DefAST`. Za boljše povezanost drevesa smo dodali še razred AST, ki je nadrazred vsem razredom, ki predstavljajo vozlišča abstraktnega sintaksnega drevesa. Na sliki 4.1 lahko vidite novo hierarhijo razredov z že dodanimi vozlišči za zanki while-do ter repeat-until in stavek let, ki jih opišemo v naslednjih poglavjih. Celotno kontekstno neodvisno gramatiko lahko vidite v dodatku A.

4.2.4 Priprava Bison datoteke

Pri načrtovanju Bison datoteko smo ustvarili zaglavno datoteko `node.h`, v katero smo podali vse deklaracije razredov, ki predstavljajo abstraktno sintaksno drevo. To zaglavno datoteko moramo vključiti v Bison datoteki, da bomo lahko ustvarjali vozlišča dreves. Definirali smo tudi unijo, v kateri zajamemo vse tipe, ki jih potrebujemo. Ključnim besedam določimo tip `int`,



Slika 4.1: Hierarhija razredov, ki predstavljajo novo abstraktno sintaksno drevo.

imenom razred `std::string` ter operatorjem tip `char`. Podobno pri produkciji določimo različne razrede, ki predstavljajo posamezno vozlišče abstraktnega sintaksnega drevesa, recimo za izraze uporabimo razred `ExprAST`. Posebnost so recimo parametri funkcije. Tam izberemo tip kot je uporabljen pri atributu razreda `PrototypeAST`, `std::vector<std::string>`. Identično naredimo pri klicih funkcije, ki vsebujejo argumente ter inicializaciji spremenljivk. Ko smo določili tipe posameznih produkcij, je bilo potrebno samo še prepisati prej načrtovano kontekstno neodvisno gramatiko.

```
1 postfix_expression:
2   IF expression THEN expression ELSE expression FI { $$ = new
        IfExprAST($2, $4, $6); }
3
4   | IF expression THEN expression ELSE expression error {
5      printf("missing␣'fi'␣keyword␣at␣the␣end␣of␣if␣statment:␣
          %d.%d-%d.%d\n",
6      @7.first_line, @7.first_column, @7.last_line, @7.
          last_column);
7      YYABORT; }
```

Izpis 4.7: Primer pravila gramatike.

Na izpisu 4.7 lahko vidimo primer pravila za razčlenjevanje stavka if. V 2. vrstici dodelimo semantično vrednost produkciji in sicer ustvarimo novo instanco razreda `IfExprAST`, kjer uporabimo semantične vrednosti nekončnih simbolov, ki predstavljajo posamezne dele stavka if. Orodje Bison ima vedno definirano produkcijo `error`, ki jo uporabimo v primeru napak. V času razvoja smo ugotovili, da je zelo pogosta napaka, da programer pozabi na koncu stavka if dodati ključno besedo `fi`. V primeru take napake želimo programerja obvestiti, da je pozabil dodati ključno besedo `fi` na koncu stavka if. To prikazuje 3. vrstica na izpisu 4.7. Produkcija na koncu namesto ključne besede, vsebuje simbol `error`. V takem primeru želimo natančno opisati kje je do napake prišlo in zaključiti z razčlenjevanjem. Preko imena `@n` lahko dostopamo do strukture, ki vsebuje informacije o lokacijo n-tega simbola. V zgornjem primeru izpišemo lokacijo kjer je do napake prišlo.

4.2.5 Prevajanje

Prevajanje prevajalnika za programski jezik Kaleidoscope poteka v več fazah. Najprej je potrebno zgenerirati datoteki, ki predstavljata leksikalni in sintaksni analizator. Nato sledi skupaj s ostalo kodo prevedemo prevajalnik. Na izpisu 4.8 lahko vidimo ukaze v konzoli, ki so potrebni za uspešen prevod. Stikalo d pri ukazu bison, nam zgenriria zaglavno datoteko `parser.tab.h`, ki jo vključimo v datoteki `kaleidoscope.lex`. Datoteka `node.cpp` vsebuje implementacijo metod abstraktnega sintaksenga drevesa, večinoma metod, ki generirajo vmesno kodo. V datoteki `k.cpp` se nahaja funkcija `main`, iz katere tudi kličemo funkcijo `yylex` ter nato iz dobljenega abstraktnega drevesa zgenerimo vmesno kodo LLVM.

```
1 $ bison -d parser.y
2 $ flex kaleidoscope.lex
3 $ g++ lex.yy.c parser.tab.c node.cpp k.cpp -rdynamic -Wl,--
    whole-archive liba.a -Wl,--no-whole-archive $(llvm-config
    --cxxflags --libs all) $(llvm-config --ldflags) -o k
```

Izpis 4.8: Prevajanje prevajalnika.

4.3 Prevajanje v objektno datoteko ELF

Ko enkrat imamo zgenerirano vmesno kodo LLVM, pot do objektne datoteke ELF ni več dolga. Naš prevajalnik nam bo zgeneriral kodo v zbirnem jeziku za procesor, na katerem trenutno teče prevajalnik. Nato bomo s pomočjo prevajalnika Clang (alternativno lahko uporabimo tudi prevajalnik GCC) ustvarili objektno datoteko ELF.

Preko razredov `Triple`, `Target` ter `TargetMachine` iz knjižnice LLVM določimo ciljno arhitekturo, za katero bomo zgenerirali kodo v zbirnem jeziku. Na koncu je potrebno preko objekta tipa `PassManager` narediti obhod po vmesni kodi LLVM, ki zgenerira kodo v zbirnem jeziku. Koda se zapiše v izhodno datoteko iz imenom vhodne datoteke in s končnico `s`. Poleg datoteke z kodo v zbirnem jeziku zgeneriramo tudi datoteko z vmesno kodo LLVM

(datotečna končnica `ll`) ter bitno kodo LLVM (datotečna končnica `bc`).

Pri izvajanju objektne datoteke ELF se tudi pokaže potreba po klicanju funkcij napisanih v programskem jeziku C, saj če ne moremo izpisovati števil oz. znakov so izvršljive datoteke praktično neuporabne. V ta namen smo razvili knjižnico liba. V knjižnici se nahajajo naslednje funkcije napisane v programskem jeziku C: `putchard` (izpiše ASCII znak), `getd` (prebere število iz standardnega vhoda), `println` (izpiše število na standardni izhod) ter `doNothing` (ne naredi nič). Za namene vključevanja v objektne datoteke ELF pripravimo statično knjižnico `liba`. Za statično knjižnico smo se odločili zato, ker je knjižnica liba razmeroma majhna in zato se bistveno ne poveča velikost izvršljive datoteke. Hkrati pa zagotovimo večjo prenosljivost, ker ko enkrat prevedemo program, ne potrebujemo več datoteke liba.a. Na izpisu 4.9 je prikazana priprava statične knjižnice.

```
1 $ g++ -c liba.cpp
2 $ ar rcs liba.a liba.o
```

Izpis 4.9: Priprava statične knjižnice.

Na izpisu 4.10 lahko vidimo primer prevajanja ter izvajanja objektne datoteke ELF. Kot lahko vidimo morajo izvorne datoteke imeti definirano funkcijo `main`, saj se podobno kot pri drugih jezikih, pri izvajanju kliče funkcija `main`. Zategadelj smo tudi v prevajalniku spremenili, da funkcija z imenom `main` vrača 32-bitna števila. Tako se izračunana vrednost funkcije `main` najprej pretvori v 32-bitno celo število in nato vrne. Podobno se vrednost vsakega klica funkcije `main` v kodi najprej pretvori v število v plavajoči vejici.

```
1 $ cat first.k
2 extern println(x),
3 def main()
4  println(2+2/2);
5 $ ./k -f first.k
6
7 $ clang first.s liba.a -o first
8 $ ./first
9 3.00
```

Izpis 4.10: Prevajanje izvorne kode v objektne datoteke ELF.

4.4 Popravljena zanka for

V prevajalniku po vodiču zanka for ne deluje pravilno. Na izpisu 4.11 lahko vidimo napačno delovanje. Zanka for se ne bi smela nikoli izvesti, vendar se kljub neresničnosti pogoja enkrat izvede. Pozorna analiza izpisa 4.11 pokaže, da je problem v brezpogojnem skoku v bloku kode `entry`.

```
1  ready> extern putchard(x);
2  ready> Read extern:
3  declare double @putchard(double)
4
5  ready> for i=10, i<5, 1 in putchard(42);
6  ready>
7  define double @0() {
8  entry:
9    br label %loop
10
11 loop:                                             ; preds = %
       loop, %entry
12   %i.0 = phi double [ 1.000000e+01, %entry ], [ %nextvar, %
         loop ]
13   %calltmp = call double @putchard(double 4.200000e+01)
14   %cmptmp = fcmp ult double %i.0, 5.000000e+00
15   %nextvar = fadd double 1.000000e+00, %i.0
16   br i1 %cmptmp, label %loop, label %afterloop
17
18 afterloop:                                        ; preds = %
       loop
19   ret double 0.000000e+00
20 }
21
22 Evaluated to 0.000000
23 ready> *
```

Izpis 4.11: Napačno delovanje zanke for.

Rešitev problema je popravljena metoda `codegen` razreda `ForExprAST`. Na začetku metode je bilo potrebno zgenerirati vmesno kodo, ki predstavlja pogoj - atribut `end`. Ko smo v `entry` bloku kode, naredimo primerjavo, ali je pogoj različen od števila 0. Na koncu samo še odstranimo brezpogojni skok in dodamo pogojni skok, ki v primeru resničnosti izraza skoči v blok kode `loop`, sicer pa v blok kode `afterloop`. V izpisu 4.12 lahko vidimo popravljen

del metode `codegen`.

```
1  Value *ForExprAST::codegen() {
2    . . .
3    Value *endCond = end->codegen();
4    if (endCond == 0)
5      return endCond;
6
7    . . .
8
9    builder.SetInsertPoint(preheaderBB);
10
11   Value *initCond = builder.CreateFCmpONE(endCond, ConstantFP
         ::get(getGlobalContext(), APFloat(0.0)), "initcond");
12
13   //builder.CreateBr(loopBB);
14   builder.CreateCondBr(initCond, loopBB, afterBB);
15   . . .
16 }
```

Izpis 4.12: Popravljen del codegen metode.

4.5 Realizacija zank while-do in repeat-until

Pri realizaciji zank while-do in repeat-until smo morali dodelati vse tri faze prednjega dela prevajalnika. Delo ni bilo posebno zahtevno, kar sta nam omogočili orodji Flex in Bison. Sintaksno obeh zank lahko vidimo na spodnjem izpisu 4.13. Zanko while-do zaključimo na podoben način kot zanko for s ključno besedo `done`. Pri zanki repeat-until smo problem dvoumnosti konca stavčnega izraza tako, da smo dodali pogoj v oklepaje in na ta način točno določili, kje se pogoj začne in kje konča.

```
1 while pogoj do izraz done
2 repeat izraz until (pogoj)
```

Izpis 4.13: Sintaksa zank while-do in repeat-until.

V leksikalni analizator je bilo potrebno dodati ključne besede `while`, `repeat` ter `until`. Ključni besedi `do` ter `done` pa že imamo, saj so na podoben način uporabljene pri zanki for. V datoteko z opisom leksikalnega analizatorja smo

tako dodali vrstice prikazane na izpisu 4.14. Pomen akcij posameznih vrstic na Izpisu 4.14 je podoben kot za ostale ključne besede: najprej nastavimo lokacijo vrednosti osnovnega simbola, v primeru razhroščevanja ključno besedo izpišemo na standarni izhod in na koncu funkcija vrne tip osnovnega simbola, ki predstavlja posamezno ključno besedo.

```
1 "while"  { SET; DEBUG("WHILE"); return WHILE;   }
2 "repeat" { SET; DEBUG("REPEAT"); return REPEAT; }
3 "until"  { SET; DEBUG("UNTIL"); return UNTIL;   }
```

Izpis 4.14: Dodane ključne besede v leksikalni analizator.

Potrebno je bilo ustvariti dva nova razreda, ki bosta predstavljala while-do ter repeat-until vozlišči v abstraktnem sintaksnem drevesu. Oba razreda bosta izpeljana iz razreda `ExprAST`, kar pomeni, da bosta morala implementirati metodo `codegen`, kjer bomo generirali vmesno kodo LLVM za stavčna izraza. Razredu `WhileExprAST` smo definirali dva atributa `cond` (predstavlja pogoj) ter `body` (predstavlja izraz zanke), oba tipa `ExprAST`. Podobna atributa smo definirali tudi pri razredu `RepeatUntilExprAST`, saj zanka podobno kot while-do zanka vsebuje konstrukta pogoj ter izraz.

V sintaksni analizator smo morali dodati štiri nove produkcije imenovane `postfix_expression`, po dve za vsako zanko. Prva predstavlja pravilno zapisano zanko, druga pa namesto zadnjega simbola (ključna beseda `done` ali zaklepaj) vsebuje simbol `error`, da lahko prevajalnik programerju poda boljši opis napake, če npr. pozabi pravilno zaključiti zanko. Na izpisu 4.15 lahko vidimo produkcije dodane v sintaksni analizator našega prevajalnika. V primeru pravilne sintaksne se ustvari nova instanca ustreznega razreda (`WhileExprAST` ali `RepeatUntilExpr`), ki jo nato dodelimo kot semantično vrednost produkcije. V primeru manjkajoče ključne besede `done` oz. zaklepaja pa izpišemo napako ter kje točno mora programer dodati ključno besedo oz. zaklepaj. Makro `YYABORT` določa, da v primeru napake zaključimo z razčlenjevnajem.

```
1 postfix_expression:
2 . . .
3   | WHILE expression DO expression DONE
```

```
4       { $$ = new WhileExprAST($2, $4);  }
5     | WHILE expression DO expression error
6       { printf("missing␣'done'␣keyword␣at␣the␣end␣of␣the␣while␣
            statment:␣%d.%d-%d.%d\n",
7         @5.first_line, @5.first_column, @5.last_line, @5.
              last_column);
8         YYABORT;
9       }
10    | REPEAT expression UNTIL LPARENT expression RPARENT
11      { $$ = new RepeatUntilExprAST($2, $5); }
12    | REPEAT expression UNTIL LPARENT expression error
13      { printf("missing␣')'␣at␣the␣end␣of␣the␣repeat␣statment:␣
            %d.%d-%d.%d\n",
14        @6.first_line, @6.first_column, @6.last_line, @6.
              last_column);
15        YYABORT;
16      }
17  . . .
```

Izpis 4.15: Dodane nove produkcije za zanki while-do ter repeat-until.

Potrebno je bilo le še razviti metodi `codegen` za razreda `WhileExprAST` ter `RepeatUntilExprAST`. Pri tem smo si pomagali tako, da smo najprej ročno načrtovali, kakšno vmesno kodo LLVM želimo za določen stavčni izraz zgenerirati.

Pri zanki while-do želimo, da stavčni izraz vrača vrednost izraza v jedru zanke, izračunanega ob zadnjem izvajanju. Če se jedro zanke nikoli ne izvede, pa želimo, da zanka while-do vrne vrednost `-1.0`. Opisno povedano moramo narediti naslednje.

V trenutnem bloku kode najprej zgeneriramo vmesno kodo pogoja. Nato na skladu rezerviramo prostor (ukaz `alloca`), poimenujemo ga `bodyret`, kjer bomo hranili vrednost izraza v jedru zanke. Naslednji ukaz shrani vrednost (ukaz `store`) `-1.0`, v primeru, da jedro zanke ni nikoli izvedeno. Potem je potrebno ustvariti dva nova bloka kode. Prvi blok kode, ki smo pa poimenovali `loop`, predstavlja jedro zanke while-do, drugi blok kode, ki smo ga poimenovali `end`, pa konec zanke. Nato prej dobljeno vrednost pogoja primerjamo (`fcmp one`), če je različna od nič. Na pogladi tega naredimo ukaz pogojni skok (`br`), v primeru resničnosti skočimo v blok kode `loop`, sicer pa

v blok kode `end`.

V bloku kode `loop` nato zgeneriramo in pridobimo vrednost izraza, ki ga shranimo na prej rezervirano mesto `bodyret`. Podobno kot prej še enkrat izvedemo primerjavo pogoja (`fcmp one`) z številom nič. Pridobljeno vrednost nato uporabimo v pogojnem skoku (`br`), ki v primeru resničnosti skoči nazaj na blok kode `loop`, sicer pa na blok kode `end`, ki zaključi izvajanje zanke. V bloku kode `end` samo preberemo vrednost (ukaz `load`) `bodyret` in si jo zapomnimo, saj predstavlja rezultat zanke while-do. Dobesedni prevod kode brez optimizacij lahko vidimo na izpisu 4.16.

```
1  > var x=1 in while x<5 do let x in x+1 end done end;
2  define double @2() {
3  entry:
4    %bodyret = alloca double
5    %x = alloca double
6    store double 1.000000e+00, double* %x
7    %x1 = load double* %x
8    %cmptmp = fcmp ult double %x1, 5.000000e+00
9    %booltmp = uitofp i1 %cmptmp to double
10   store double -1.000000e+00, double* %bodyret
11   %cond = fcmp one double %booltmp, 0.000000e+00
12   br i1 %cond, label %loop, label %end
13
14 loop:                                             ; preds = %
       loop, %entry
15   %x2 = load double* %x
16   %addtmp = fadd double %x2, 1.000000e+00
17   store double %addtmp, double* %x
18   store double %addtmp, double* %bodyret
19   %x3 = load double* %x
20   %cmptmp4 = fcmp ult double %x3, 5.000000e+00
21   %booltmp5 = uitofp i1 %cmptmp4 to double
22   %cond6 = fcmp one double %booltmp5, 0.000000e+00
23   br i1 %cond6, label %loop, label %end
24
25 end:                                              ; preds = %
       loop, %entry
26   %0 = load double* %bodyret
27   ret double %0
28 }
29
30 Function returned 5.00.
```

Izpis 4.16: Generiranje vmesne kode za zanko while-do.

Izpis 4.16 prikazuje tudi uporabo stavčnega izraza let, ki je namenjen spreminjanju oz. dodeljevanju vrednosti spremenljivk. Izvedba prevajalnika po vodiču tega izraza ne pozna. Zgoraj opisan postopek naredimo programsko v metodi `codegen` razreda `WhileExpr`. S tem je realizacija zanke while-do zaključena.

Generiranje vmesne kode LLVM za zanko repeat-until je nekoliko bolj preprosto. Za zanko repeat-until želimo, da vrača vrednost jedra zanke. Tukaj si te vrednosti ne potrebujemo vsakič shranjevati na posebej dodeljeno mesto, saj vemo, da se bo jedro zanke izvedlo vsaj enkrat. V trenutni funkciji najprej ustvarimo dva nova bloka kode, `body` (predstavlja jedro zanke) ter `end` (predstavlja konec zanke). V trenutnem bloku kode naredimo brezpogojni skok (ukaz `br`) v blok kode `body`. V bloku kode `body` zgeneriramo vmesno kodo jedra zanke ter kode, ki predstavlja pogoj. Nato naredimo primerjavo (ukaz `fcmp oeq`), ki primerja, ali je vrednost pogaja enaka številu nič. Vrednost primerjave uporabimo pri pogojnem skoku, v primeru resničnosti skočimo nazaj na začetek bloka kode `body`, sicer pa v blok kode `end`. V bloku kode `end` si le še zapomnimo vrednost jedra funkcije. Na izpisu 4.17 vidimo dobesedni prevod zgoraj opisane vmesne kode.

```
1  > var x=10 in repeat let x in x-1 end until (x<5) end;
2  define double @0() {
3  entry:
4    %x = alloca double
5    store double 1.000000e+01, double* %x
6    br label %body
7
8  body:                                             ; preds = %
       body, %entry
9    %x1 = load double* %x
10   %subtmp = fsub double %x1, 1.000000e+00
11   store double %subtmp, double* %x
12   %x2 = load double* %x
13   %cmptmp = fcmp ult double %x2, 5.000000e+00
14   %booltmp = uitofp i1 %cmptmp to double
15   %cond = fcmp oeq double %booltmp, 0.000000e+00
```

```
16   br i1 %cond, label %body, label %end
17
18 end:                                             ; preds = %
       body
19   ret double %subtmp
20 }
21
22 Function returned 4.00.
```

Izpis 4.17: Generiranje vmesne kode za zanko repeat-unitil.

S tem je realizacija zanke repeat-until zaključena.

4.6 Testni primeri

Delovanje naše izvedbe prevajalnika za programski jezik Kaleidoscope smo testirali z manjšimi primeri, ki uporabljajo različne konstrukte jezika. Na izpisu 4.18 imamo prilagojeno kodo za našo izvedbo prevajalnika, ki smo jo predstavili v poglavju 3.1.

```
1  # definicija unarnega operatorja '~', negativna stevila
2  def unary~(v)
3   0-v,
4
5  # definicija binarnega operatorja vecji
6  def binary> (l, r)
7   r<l,
8
9  # funkcija, ki izracuna n-ti clen Fibonaccijevega zaporedja
10 def fib(x)
11  if x < 3 then
12   1
13  else
14   fib(x-1)+fib(x-2)
15  fi,
16
17 # izracuna 10. clen Fibonaccijevega zaporedja
18 var x=9 in fib(x+1) end,
19
20 # deklaracija zunanje funkcije
21 extern putchard(x),
22
23 # funkcija, ki izpise n znakov '*'
```

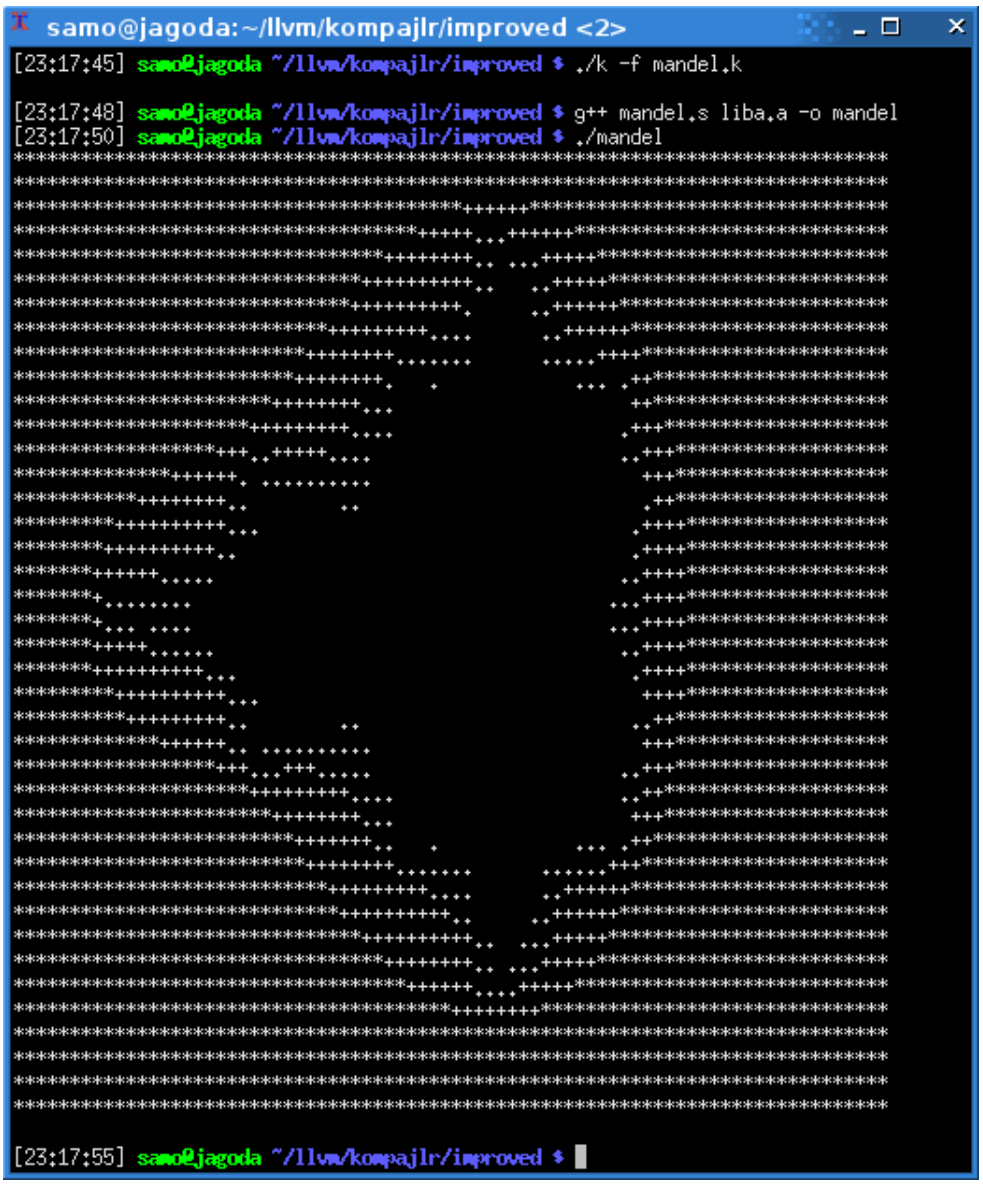
```
24 def star(n)
25  for i=1, i<n, 1.0 do
26    putchard(42)
27  done,
28
29 # klic funkcije star, ki izpise 16 znakov '*'
30 star(16)
```

Izpis 4.18: Popravljena koda v programskem jeziku Kaleidoscope.

Preko zgornje kode smo testirali vse ključne konstrukte programskega jezika Kaleidoscope. Zgornja koda preverjeno deluje na našem prevajalniku. Pravilnost delovanja smo preverili tudi na malo večjem primeru, ki smo ga dobili v vodiču.

Program računa števila Mandelbrotove množice [9] in jih izrisuje na standardni izhod. Izvorno kodo programa smo morali malo prilagoditi, da se prevede in pravilno izvaja na naši izvedbi prevajalnika, saj program v veliki meri uporablja uporabniško določene operatorje, zato smo morali z oklepaji določiti pravilno prioriteto operatorjem. V programu so uporabljena tudi negativna števila, ki jih naš prevajalnik in prevajalnik po vodiču privzeto ne podpirata. V prevajalniku po vodiču lahko za definicijo unarnega operatorja uporabimo tudi znak `-` (minus), v naši izvedbi to ni mogoče, zato smo za predstavitev negativnih števil uporabili znak `~`. Popraviti smo morali tudi vse stavke if in for, da so zaključeni z ustrezno ključno besedo. Zaradi spremenjenega delovanja zanke for se ob izvajanju programa prevedenega z našim prevajalnikov izpiše en stolpec in en vrstica manj kot pri izvajanju z prevajalnikom razvitem po vodiču.

Na sliki 4.2 lahko vidite prevod in izvajanje programa z našim prevajalnikom. Izvorna kodo datoteke `mandel.k` si lahko ogledate v dodatku B.



Slika 4.2: Prevajanje in izvajanje programa Mandelbrot.

Poglavje 5

Zaključek

V prejšnjih poglavjih smo prikazali konstrukcijo preprostega prevajalnika. Prikazano je, kako preprosto je narediti prevajalnik, ki prevede programe vse do objektnih datotek, ki so pripravljene na izvajanje. Pri tem je potrebno predvsem poznavanje poznavanje vmesne kode LLVM ter osnovno znanje o računalniški arhitekturi. Potrebno se je tudi spoznati s programsko knjižnico LLVM, ki je precej obsežna in lahko začetnika odbije uporabe. Pri realizaciji našega prevajalnika se nismo ozirali preveč na dobre programske prakse, saj na primer uporabljamo veliko globalnih spremenljivk, kar iz praske vemo, da ni najbolje. Na tem mestu je še precej prostora za izboljšave. Naštejmo jih samo nekaj: uporaba objektno orientiranega pristopa pri realizaciji leksikalne in sintaksne analize, uporaba načrtovalskega vzorca obiskovalca (*visitor*) ...

V času razvoja prevajalnika smo testirali izvajanje programov na procesorjih x86, x86_64 ter ARM. Na slednjih je podpora za prevajalnik JIT še bolj v začetnih fazah, saj je podprto samo izvajanje nad celoštevilskimi operandi, kar je v našem primeru popolnoma onemogočilo pravilno izvajanje preko prevajalnika JIT. Te težave bodo verjetno odpravljene v prihodnjih različicah projekta LLVM, saj je projekt LLVM, zaradi podpore ameriškega podjetja Apple, eden izmed boljše razvitih in organiziranih odprtokodnih projektov. Na procesorjih vrste x86 ter x86_64 pa je bilo delovanje zelo zanesljivo in zadovoljivo.

Literatura

[1] Flex, The Flex Project, 2013. Dostopno na:
http://flex.sourceforge.net/.

[2] Bison, Free Software Foundation, Inc., 2013. Dostopno na:
http://www.gnu.org/software/bison/.

[3] GCC, the GNU Compiler Collection, Free Software Foundation, Inc., 2013. Dostopno na:
http://gcc.gnu.org/.

[4] Static single assignment form, Wikimedia Foundation, Inc., 2013. Dostopno na:
http://en.wikipedia.org/wiki/Static_single_assignment_form.

[5] LLVM Assembly Language Reference Manual, The LLVM Compiler Infrastructure, 2013. Dostopno na:
http://llvm.org/docs/LangRef.html.

[6] LLVM Tutorial, The LLVM Compiler Infrastructure, 2013. Dostopno na:
http://llvm.org/docs/tutorial/index.html.

[7] B. Vilfan (2004) Prevajanje programskih jezikov. Dostopno na:
http://lalg.fri.uni-lj.si/~vilfan/Prev2004.pdf.

[8] I. Kononenko, "*Programski jeziki*", Radovljica: Didakta, 1992.

[9] Mandelbrot set, Wikimedia Foundation, Inc., 2013. Dostopno na: http://en.wikipedia.org/wiki/Mandelbrot_set.

Dodatek A

Kontekstno neodvisna gramatika programskega jezika Kaleidoscope

source → global
source → global ;
source → `ENDF`
global → global , top
global → top
top → definition
top → external
top → toplevelexpr
definition → `DEF` prototype expression
definition → `DEF` error
prototype → `IDENTIFIER (` def_parameters `)`
prototype → `IDENTIFIER (` error
prototype → `BINARY OPER ( IDENTIFIER, IDENTIFIER )`
prototype → `BINARY` error
prototype → `UNARY OPER ( IDENTIFIER )`
prototype → `UNARY` error

def_parameters → `IDENTIFIER`

def_parameters → def_parameters `,` `IDENTIFIER`

def_parameters → {}

external → `EXTERN` prototype

external → `EXTERN` error

toplevelexpr → expression

expression → custom_expression

expression → error

custom_expression → relational_expression

custom_expression → custom_expression `OPER` relational_expression

custom_expression → custom_expression `OPER` error

relational_expression → additive_expression

relational_expression → additive_expression `<` additive_expression

relational_expression → additive_expression `<` error

additive_expression → multiplicative_expression

additive_expression → additive_expression `+` multiplicative_expression

additive_expression → additive_expression `+` error

additive_expression → additive_expression `-` multiplicative_expression

additive_expression → additive_expression `-` error

multiplicative_expression → prefix_expression

multiplicative_expression → multiplicative_expression `*` prefix_expression

multiplicative_expression → multiplicative_expression `*` error

multiplicative_expression → multiplicative_expression `/` prefix_expression

multiplicative_expression → multiplicative_expression `/` error

prefix_expression → postfix_expression

prefix_expression → `OPER` prefix_expression

prefix_expression → `OPER` error

postfix_expression → `DOUBLE`

postfix_expression → `IDENTIFIER`

postfix_expression → `IDENTIFIER` `(` function_args `)`

postfix_expression → `(` expression `)`

postfix_expression → `(` expression error
postfix_expression → `IF` expression `THEN` expression `ELSE` expression `FI`
postfix_expression → `IF` expression `THEN` expression `ELSE` expression error
postfix_expression → `FOR IDENTIFIER =` expression `,` expression `,` expression `DO` expression `DONE`
postfix_expression → `FOR IDENTIFIER =` expression `,` expression `,` expression `DO` expression error
postfix_expression → `FOR IDENTIFIER =` expression `,` expression `DO` expression `DONE`
postfix_expression → `FOR IDENTIFIER =` expression `,` expression `DO` expression error
postfix_expression → `VAR` decls `IN` expression `END`
postfix_expression → `VAR` decls `IN` expression error
postfix_expression → `WHILE` expression `DO` expression `DONE`
postfix_expression → `WHILE` expression `DO` expression error
postfix_expression → `LET` expression `IN` expression `END`
postfix_expression → `LET` expression `IN` expression error
postfix_expression → `REPEAT` expression `UNTIL (` expression `)`
postfix_expression → `REPEAT` expression `UNTIL (` expression error
function_args → expression
function_args → function_args `,` expression
function_args → {}
decls → assign
decls → decls `,` assign
assign → `IDENTIFIER`
assign → `IDENTIFIER =` expression

Dodatek B

Izvorna koda programa, ki izračuna ter izriše števila Mandelbrotove množice

```
1  extern putchard(x),
2
3  # Logical unary not.
4  def unary!(v)
5    if v then
6      0
7    else
8      1
9    fi,
10
11 # Unary negate.
12 def unary~(v)
13   0-v,
14
15 # Define > with the same precedence as <.
16 def binary>  (LHS, RHS)
17   RHS < LHS,
18
19 # Binary logical or, which does not short circuit.
20 def binary| (LHS, RHS)
21   if LHS then
22     1
23   else if RHS then
```

```
24      1
25    else
26      0
27    fi fi,
28
29  # Binary logical and, which does not short circuit.
30  def binary& (LHS, RHS)
31    if !LHS then
32      0
33    else
34      !!RHS
35    fi,
36
37  # Define ':' for sequencing: as a low-precedence operator
       that ignores operands
38  # and just returns the RHS.
39  def binary : (x, y) y,
40
41  def printdensity(d)
42    if d > 8 then
43      putchard(32)  # '␣'
44    else if d > 4 then
45      putchard(46)  # '.'
46    else if d > 2 then
47      putchard(43)  # '+'
48    else
49      putchard(42) # '*'
50    fi fi fi,
51
52  # Determine whether the specific location diverges.
53  # Solve for z = z^2 + c in the complex plane.
54  def mandleconverger(real, imag, iters, creal, cimag)
55    if (iters > 255) | (real*real + imag*imag > 4) then
56      iters
57    else
58      mandleconverger(real*real - imag*imag + creal,
59                      2*real*imag + cimag,
60                      iters+1, creal, cimag)
61    fi,
62
63  # Return the number of iterations required for the iteration
       to escape
64  def mandleconverge(real, imag)
65    mandleconverger(real, imag, 0, real, imag),
66
```

```
67 # Compute and plot the mandlebrot set with the specified 2
       dimensional range
68 # info.
69 def mandelhelp(xmin, xmax, xstep,   ymin, ymax, ystep)
70   for y = ymin, y < ymax, ystep do (
71     (for x = xmin, x < xmax, xstep do
72        printdensity(mandleconverge(x,y)) done)
73     : putchard(10)) done,
74
75 # mandel - This is a convenient helper function for plotting
       the mandelbrot set
76 # from the specified position with the specified
       Magnification.
77 def mandel(realstart, imagstart, realmag, imagmag)
78   mandelhelp(realstart, realstart+realmag*78, realmag,
79              imagstart, imagstart+imagmag*40, imagmag),
80
81 def main()
82   mandel(~2.3, ~1.3, 0.05, 0.07) : putchard(10)
```

Izpis B.1: Izvorna koda datoteke mandel.k.