Programski jezik C0

Vinković, Vedran

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Vedran Vinković

PROGRAMSKI JEZIK C0

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, rujan 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Programski jezik C0 razvijen je na sveučilištu Carnegie Mellon. Sastoji se od sigurnog podskupa funkcionalnosti programskog jezika C te je stvoren s ciljem upoznavanja studenata s imperativnim programiranjem. Kad kažemo „siguran", mislimo da je po dizajnu jezika nemoguće izazvati nedefinirano ponašanje.

U ovom radu upoznat ćemo se s detaljima jezika C0, infrastrukturom sustava LLVM te postupkom kompajliranja kôda napisanog u jeziku C0.

Poglavlje 1

Programski jezik C0

1.1 Tipovi

U svakom programskom jeziku, pa tako i u C0, prije nego što počnemo programirati, moramo znati koje tipove podataka taj jezik podržava. Zato ćemo u nastavku navesti tipove podataka u jeziku C0, uz objašnjenje čemu svaki od njih služi i kako se koristi.

1.1.1 Cijeli brojevi — `int`

Jedini numerički tip u C0 je `int`, koji predstavlja cijele brojeve. Vrijednosti tipa `int` interno su prikazane kao 32-bitne riječi, zapisane metodom dvojnog komplementa. Na cijelim brojevima su podržane osnovne aritmetičke operacije: zbrajanje (+), oduzimanje (-), množenje (*), cjelobrojno dijeljenje (/) i modulo (%).

Budući da su cijeli brojevi veličine 32 bita, moguće vrijednosti cjelobrojnih konstanti su između -2^{32} i $2^{32} - 1$ te se navedene operacije uglavnom računaju modulo 2^{32}. Pogledajmo na primjerima kako se ponašaju dijeljenje i modulo:

```
19 / 6 == 3
25 % 6 == 1
-43 / 8 == -5
-37 % 4 == -1
-4294967296 / -1  diže iznimku overflow
-4294967296 % -1  diže iznimku overflow
2 / 0  diže iznimku overflow
9 % 0  diže iznimku overflow
```
(1.1)

Rezultati operatora usporedbe (<, >, <=, >=), operatora jednakosti (==) i operatora nejednakosti (!=) vraćaju logičke vrijednosti (v. točku 1.1.2).

Ako to želimo, podatke tipa `int` u jeziku C0 možemo obrađivati kao nizove bitova duljine 32, a podržane operacije za to su veznici i (`&`), ili (`|`), isključivo ili (`^`), bitovni komplement (`~`) te operacije lijevog (<<) i desnog (>>) pomaka. U tu svrhu možemo koristiti heksadekadski prikaz broja, koji uvijek počinje s `0x` ili `0X` i sadrži heksadekadske znamenke, koje mogu biti: `0`, . . . , `9`, `A`, . . . , `F`, `a`, . . . , `f`.

Operacije pomaka pomiču bitove broja ulijevo, odnosno udesno. Također, pri pomaku ulijevo, u nove najmanje značajne bitove se upisuje 0, a pri pomaku udesno se u nove najznačajnije bitove upisuje bit predznaka. Na primjer, vrijedi:

```
17 << 4 == 272
-29 >> 3 == -4
```
(1.2)

Desni argument pomaka mora biti iz $[0, 31]$, inače program mora prijaviti grešku.

Uočimo, pomak bitova ulijevo za b mjesta ekvivalentan je množenju s 2^b, a pomak udesno za b mjesta ekvivalentan je cjelobrojnom dijeljenju s 2^b.

Varijable svih tipova, pa tako i tipa `int`, se u jeziku C0 deklariraju kao u jeziku C. Ako ih eksplicitno ne inicijaliziramo, radi sigurnosti jezika im se pridružuje vrijednost 0.

1.1.2 Logičke vrijednosti — `bool`

Logičke vrijednosti su reprezentirane tipom `bool` te mogu biti `true` ili `false`. Podržane logičke operacije nad tipom `bool` su konjunkcija (`&&`), disjunkcija (`||`) i negacija (`!`).

Binarne logičke operacije podržavaju takozvanu *lijenu evaluaciju*. Na primjer, ako `a` ima vrijednost `false` u izrazu `a&&b`, tada se `b` uopće ne evaluira te cijeli izraz ima vrijednost `false`. Analogno, ako `a` ima vrijednost `true` u izrazu `a||b`, `b` se ne evaluira i cijeli izraz ima vrijednost `true`.

Tip `bool` podržava i ternarni uvjetni operator `?:`, koji se poziva izrazom oblika

uvjet `?` *grana_true* `:` *grana_false*, (1.3)

gdje je *uvjet* izraz tipa `bool`, a *grana_true* i *grana_false* bilo kakvi izrazi. Uvjetni operator također podržava lijenu evaluaciju: ako se *uvjet* evaluira u `true`, tada se evaluira izraz *grana_true* te izraz (1.3) ima njegovu vrijednost, a izraz *grana_false* se uopće ne evaluira. Analogno, ako se *uvjet* evaluira u `false`, nakon njega se evaluira samo *grana_false*.

Kao i podatke tipa `int`, podatke tipa `bool` možemo uspoređivati operatorima == i `!=`.

Također, ako varijabla tipa `bool` nije inicijalizirana pri deklaraciji, radi sigurnosti joj se automatski pridružuje vrijednost `false`.

1.1.3 Znakovi — `char`

Tip `char` predstavlja znakove, sastavnu jedinicu stringova. Vrijednosti tipa `char` se pišu u obliku `'`c`'`, gdje c može biti bilo koji ASCII-znak koji se može ispisati na ekran ili neki od posebnih nizova znakova kao što su: `\t` (predstavlja tabulator), `\n` (predstavlja novi red), `\'` (predstavlja jednostruki navodnik), `\0` (predstavlja nul-znak).

Ako deklariramo varijablu tipa `char` bez inicijalizacije, pridružuje joj se vrijednost `'\0'`.

1.1.4 Stringovi — `string`

Za razliku od jezika C, jezik C0 za stringove ima poseban tip podatka — `string`. Stringovi imaju oblik "$c_1c_2 \ldots c_n$", gdje svaki c_i, $i = 1, 2, \ldots, n$ može biti bilo koji ASCII-znak ili poseban niz znakova naveden u dijelu 1.1.3 osim `\0`. Ako želimo znak `'"'` koristiti u stringu, moramo ga zapisati kao `\"`.

Ako varijabla tipa `string` nije inicijalizirana, automatski joj se pridružuje vrijednost "" (prazan string).

1.1.5 Polja — T`[]`

Polja su nešto zanimljiviji objekti od onih s kojima smo se dosad susreli. Polje čiji su elementi tipa T u jeziku C0 označavamo T`[]`. Na primjer, polje cijelih brojeva označavamo s `int[]`, polje znakova s `char[]`, i tako dalje.

Polja se alociraju isključivo dinamički, i to pozivom `alloc_array`:

```
alloc_array(int, 6);
```
(1.4)

Poziv (1.4) alocira memoriju za polje koje se sastoji od 6 cijelih brojeva i vraća referencu na to polje. Općenito, ako je T tip i e izraz tipa `int`, `alloc_array(`T`,` e`)` će alocirati memoriju za polje od e elemenata tipa T i vratiti referencu na to polje. Na nama je samo još da je pridružimo varijabli:

```
string[] moje_polje = alloc_array(string, 14);.
```
(1.5)

Dobro je imati na umu da u C0 memoriju ne dealociramo, nego to za nas radi *garbage collector*. To je još jedan aspekt sigurnosti jezika; ne možemo nepažnjom izazvati curenje memorije.

Polja se indeksiraju počevši od nule, a elementima polja se pristupa operatorom `[]`. Kad pristupamo elementima polja, moramo paziti da pristupamo samo onim elementima za koje smo alocirali memoriju. Na primjer, ako inicijaliziramo `moje_polje` kao u (1.5), možemo pristupati samo elementima s indeksima $0, 1, \ldots, 13$. Za razliku od jezika C, u C0 pristupanje elementima polja izvan dosega ne izaziva nedefinirano ponašanje, nego program staje s izvršavanjem te podiže iznimku *out of range*.

Za svaki tip T, postoji istaknuto prazno polje polje tipa T`[]`, koje označava polje bez ijednog elementa. Prije nego što mu alociramo memorju, svako polje je jednako praznom polju odgovarajućeg tipa. Takva polja smijemo samo uspoređivati operatorima == i !=.

Važno je zapamtiti da je polje referenca na blok memorije te je uspoređivanje varijabli tipa T`[]` zapravo uspoređivanje referenci, a ne elemenata polja. Analogno, kad prosljeđujemo varijablu tipa T`[]` kao argument funkciji ili je pridružujemo drugoj varijabli, prosljeđuje odnosno pridružuje se referenca, a ne elementi polja.

Zbog povezanosti s jezikom C, C0 nema funkciju ni ključnu riječ za računanje ili dohvaćanje duljine polja. Ipak, *specifikacijski ugovori*, koje ćemo obraditi u točki 1.3, nekad moraju znati duljinu polja kako bi mogli osigurati da neće doći do greške u izvršavanju programa. Iz tog razloga postoji „funkcija" `\length` koja se smije koristiti isključivo u specifikacijskim ugovorima. U slučaju da se specifikacijski ugovori provjeravaju za vrijeme izvršavanja programa, kompajler vodi računa o tome da uz polje u memoriju sprema i njegovu duljinu.

1.1.6 Pokazivači — T*

Slično kao kod polja, pokazivač koji pokazuje na vrijednost tipa T ima tip T*. Memorija na koju pokazivač pokazuje se također alocira dinamički, koristeći poziv `alloc`, koji prima tip podatka na koji pokazivač treba pokazivati. Na primjer, `int* moj_pokazivac = alloc(int);` alocira memoriju za vrijednost tipa `int`. Da bismo pristupili vrijednosti na koju pokazivač pokazuje, dereferenciramo ga unarnim operatorom *. Na primjer, `*moj_pokazivac = 9;` pridružuje vrijednost 9 objektu na koji pokazuje `moj_pokazivac`. Ako ne inicijaliziramo objekt na koji pokazivač pokazuje, poprimit će podrazumijevanu vrijednost za svoj tip0.

Možemo dereferencirati svaki izraz pokazivačkog tipa i dalje raditi što god želimo s vrijednošću na koju pokazuje. To nam omogućuje činjenica da operator * vraća *lijevu vrijednost*, odnosno vrijednost koju možemo mijenjati; tu spadaju varijable, elementi polja, vrijednosti na koje pokazuju pokazivači te elementi struktura.

Kao kod polja, memoriju koju alociramo za pokazivače ne dealociramo sami,

nego to radi *garbage collector*. Još jedna sličnost s poljima je da za svaki pokazivački tip `T*` postoji poseban pokazivač `NULL` istog tipa. Svaka varijabla pokazivačkog tipa za koju ne alociramo memoriju jednaka je `NULL`.

Ako pokušamo dereferencirati `NULL`, dići će se izuzetak koji će zaustaviti izvršavanje programa. Ne samo da je zabranjeno dereferencirati `NULL`, nego je zabranjeno uopće napisati `*NULL`; taj kôd se ne može niti prevesti (ne samo izvršiti) jer se pokazivač `NULL` jednako zove za svaki pokazivački tip. To za posljedicu ima činjenicu da je izraz `*NULL` nepoznatog tipa.

Na prvi pogled se čini da je višeznačnost pokazivača `NULL` besmislena i da čini više štete nego koristi. Međutim, u svim ostalim slučajevima se njegov tip može jednoznačno odrediti kad god je to potrebno. Na primjer, ako imamo funkciju prototipa `f(int*)` i pozovemo je s `NULL`, u izrazu `f(NULL)` je jasno da je `NULL` tipa `int*`.

1.1.7 Strukture — `struct`

Strukture su jedini tip podatka koji može sadržavati vrijednosti različitih tipova. Kad želimo deklarirati strukturu i nazvati je `moja_struktura`, to radimo sa `struct moja_struktura;`, a ako je želimo i definirati, moramo navesti sve njene elemente i njihove tipove:

```
struct moja_struktura {
    T_1 var_1;
    T_2 var_2;
    ⋮    ⋮
    T_n var_n;
};
```
(1.6)

Istaknimo da strukture možemo deklarirati, odnosno definirati, isključivo u globalnom dosegu. Za strukturu (1.6) kažemo da sadrži elemente var_1, var_2, ..., var_n, s tipovima T_1, T_2, ..., T_n redom. Imena struktura i njihovih elemenata se nalaze u zasebnim prostorima imena, što znači da se mogu podudarati s imenima varijabli i funkcija. Nakon što je struktura definirana kao gore, svaki izraz `e` tipa `struct moja_struktura` se može koristiti za pristupanje elementima strukture pomoću operatora pristupa (`.`). Takvo pristupanje elementima se ponaša slično kao operator `[]` kod polja. Primjerice, *e*`.`*var* `= 4` postavlja vrijednost varijable *var* u strukturi *e* na 4.

Sami moramo alocirati memoriju za strukture, a postupak je nešto kompliciraniji nego kod polja i pokazivača. Naime, strukture su *veliki* tip podatka, što znači da je struktura zaista kolekcija svih svojih varijabli, a ne referenca na njih. Ne

možemo ih spremati u varijable ni prosljeđivati kao argumente funkcija, a to pak znači da programi s njima uglavnom manipuliraju preko pokazivača ili polja.

Iz tog razloga nemamo još jedan izraz koji služi samo za alociranje struktura; jednostavno deklariramo pokazivač na strukturu i alociramo memoriju za njega koristeći `alloc`, ili deklariramo polje struktura i alociramo memoriju za cijelo polje koristeći `alloc_array`. Budući da su pokazivači na strukture toliko česti, u C0 postoje dva dodatna jezična konstrukta koji uvelike olakšavaju korištenje pokazivača na strukture: operator `->`, i ključna riječ `typedef`.

Operator `->` je kompozicija operatora dereferenciranja i pristupa, odnosno izraz `(`*e*`)->`*var* je ekvivalentan izrazu `(*(`*e*`)).`*var*, gdje je *e* izraz nekog pokazivačkog tipa.

Ključna riječ `typedef` općenito služi za definiranje tipova: `typedef` *T* `ime;`, gdje je *T* tip, a `ime` njegovo novo ime. Prije nego opišemo kako nam `typedef` olakašava korištenje struktura, slijedi par napomena:

- `typedef` se smije nalaziti jedino izvan svih funkcija, u globalnom dosegu;
- nakon što tipu *T* dodijelimo ime `ime`, to se ime može koristiti u cijelom programu nakon točke dodjele;
- imena `ime` su u istom prostoru imena kao imena funkcija i varijabli pa moramo paziti da se nijedna varijabla ili funkcija ne zove `ime`.

Jedan od načina kako `typedef` možemo iskoristiti u ovom slučaju je `typedef struct s* s;`, nakon čega ime `s` predstavlja tip pokazivača na `struct s`.

Ako želimo, možemo deklarirati strukturu bez definicije:

```
struct moja_struktura2;.
```
(1.7)

Naravno, ne možemo pristupati elementima strukture koja je samo deklarirana zato što ne znamo koje elemente uopće ima, ali možemo koristiti pokazivače na `struct moja_struktura2`, jer su svi pokazivači iste veličine pa kompajler zna koliko memorije treba alocirati za njih.

1.1.8 Funkcije

Funkcije se mogu definirati samo u globalnom dosegu, na sljedeći način:

T `moja_funkcija(`T_1 `arg1, `T_2 `arg2, ..., `T_n `argn){`*tijelo*`},` (1.8)

gdje je T povratni tip funkcije, T_i tip i-tog argumenta funkcije, a *tijelo* sadrži deklaracije i naredbe (v. točku 1.2) koje se izvrše pri pozivu funkcije. Uočimo da nakon definicije funkcije ne pišemo `;`, za razliku od definicije strukture. Doseg

argumenata funkcije je njeno tijelo, a doseg funkcije je cijela datoteka nakon mjesta gdje je deklarirana.

Funkcije se, slično kao strukture, mogu deklarirati bez definicije:

$$T\ \texttt{moja\_funkcija(}T_1\ \texttt{arg1, }T_2\ \texttt{arg2, ..., }T_n\ \texttt{argn);} \tag{1.9}$$

Funkcija može biti deklarirana više puta u istoj datoteci, ali svaki put mora imati isti povratni tip i tipove argumenata; jedino što se smije promijeniti su imena argumenata.

Tipovi argumenata funkcije i njen povratni tip moraju biti *mali* tipovi, odnosno ne mogu biti strukture. Umjesto struktura, treba koristiti pokazivače ili polja struktura.

Funkcije pozivamo izrazom oblika `moja_funkcija(`e_1`, `e_2`, ..., `e_n`);`, gdje svaki e_i mora biti izraz tipa T_i. Pri pozivu funkcije se e_i evaluiraju s lijeva na desno, a njihove vrijednosti se prosljeđuju funkciji kao argumenti. To je velika razlika u odnosu na jezik C, gdje standardom nije propisan red evaluacije argumenata funkcije. Istaknimo da se vrijednosti e_i prenose *po vrijednosti*, odnosno njihove se vrijednosti kopiraju u varijable `arg1, arg2, ..., argn`. To znači da, ako pozovemo funkciju s izrazom koji je varijabla, i dalje ne možemo originalnim vrijednostima koje su joj proslijeđene kao argumenti, nego s njihovim kopijama u trenutku poziva.

1.2 Naredbe

Kad znamo koje tipove podataka C0 podržava i kako se koriste, htjeli bismo nešto raditi s njima. Za to koristimo naredbe. Naredbe ne možemo pisati u globalni doseg, već samo u tijela funkcija. Dosad smo se već susreli s nekim naredbama; sada ćemo ih detaljnije istražiti.

1.2.1 Izrazi

Izraze možemo izvršavati kao naredbe. Kad se izraz e izvršava, evaluira se, ali ne sprema svoju vrijednost, ako je ima. Među najčešće korištenim izrazima su pridruživanja, odnosno izrazi oblika `a = e;`, gdje je e neki izraz, a a lijeva vrijednost.

Da bi se pridruživanje moglo provesti, e mora biti istog tipa kao a. Uz to, strukture se nikad ne mogu pridruživati. Kad se pridruživanje izvršava, prvo se evaluira a, zatim e, nakon čega se pokuša vrijednosti a pridružiti e.

Osim operatora =, imamo i operatore op=, za op $\in$ {+, -, *, /, %, &, ^, |, <<, >>} te se izraz *a* op= *e* ponaša isto kao izraz *a* = *a* op *e*[1].

Uočimo da se *a* u drugom primjeru evaluira dvaput: jednom kad se evaluira lijeva strana i jednom kad se evaluira desna, dok se u prvom primjeru evaluira samo jednom.

Uz navedene binarne operatore pridruživanja, imamo i unarne operatore: ++ i -- te vrijedi[2]:

a++ je isto što i *a* += 1
a-- je isto što i *a* -= 1. (1.10)

1.2.2 Blokovi

Blok je slijed deklaracija i naredbi u vitičastim zagradama, oblika

```
{
  deklaracije
  naredbe
},
```
(1.11)

gdje je *deklaracije* (možda prazan) slijed deklaracija varijabli, a *naredbe* slijed naredbi. Dakle, blokove možemo stavljati unutar drugih blokova. Deklaracija može imati jedan od dva oblika:

- *T* `var;`
- *T* `var =` *e*`;`

U prvom obliku se deklarira varijabla `var` tipa *T* i inicijalizira se na podrazumijevanu vrijednost za taj tip. U drugom se evaluira izraz *e* te se varijabla `var` inicijalizira na tu vrijednost umjesto podrazumijevane. Uočimo da je `var` i dalje tipa *T* pa i *e* mora biti tog tipa. Na primjer, sljedeće deklaracije su legalne:

```
int a;
int b = 8+14;
char c = 'b';
```
(1.12)

[1]Ne ponaša se sasvim isto. Na primjer, neka je *a* polje s elementima 5, 10 i 18 i $i = 0$. Nakon izvršavanja *a*[*i*++]=*a*[*i*++]+*i*, *a* ima elemente 12, 10 i 18, a nakon izvršavanja *a*[*i*++]+=*i*, *a* ima elemente 6, 10 i 18. Iako se lako konstruiraju, ovakvi primjeri se iznimno rijetko pojavljuju u praksi i prosječan korisnik se ne treba zamarati s njima.

[2]C0 nema prefiksne operatore ++ i --, nego se postfiksni ponašaju kao što se prefiksni ponašaju u jeziku C.

Sljedeće deklaracije nisu legalne:

```
bool a = 6; — 6 je tipa int, a ne bool
char b = 7+35; — 7+35 je tipa int, a ne char
int a = 2==2; — 2==2 je tipa bool, a ne int
```
(1.13)

Za razliku od jezika C, varijable deklarirane u vanjskom dosegu, bilo kao argumenti funkcije, ili deklaracijama u vanjskom bloku, ne mogu se opet deklarirati u unutarnjem dosegu s istim imenom.

1.2.3 Grananja

Grananje je naredba oblika

`if (`e`)` n_1 `else` n_2, (1.14)

gdje je e izraz tipa `bool`, a n_1 i n_2 naredbe. Uočimo da nakon grananja ne pišemo `;`, ali naredba n_2 svejedno može završavati znakom `;`. Pri izvršavanju grananja, najprije se evaluira e, zatim n_1 ako e ima vrijednost `true`, odnosno n_2 ako ima vrijednost `false`. Ako izostavimo `else` i naredbu n_2, grananje će se izvršiti kao:

`if (`e`)` n_1 `else {}`. (1.15)

Ako imamo više `if`-ova, a samo jedan `else`, na primjer:

```
if (a >= 8) if(a < 16) ++a; else --a;,
```
(1.16)

tada `else` stoji uz zadnji `if` koji nema `else`, vodeći računa o tome da su u istom bloku, pa se naredba izvršava kao:

```
if (a >= 8){if(a < 16) ++a; else --a;}.
```
(1.17)

1.2.4 Petlje

U C0 imamo dvije vrste petlji:

`while (`e`)` n_3 i
`for (`n_1`;` e`;` n_2`)` n_3, (1.18)

gdje je e izraz tipa `bool`, a $n_{1,2,3}$ su naredbe. Petlja `while` prvo evaluira e te, ako je njegova vrijedost `true`, izvrši naredbu n_1. Zatim opet evaluira e i izvrši n_1 i ponavlja to sve dok vrijednost e ne postane `false`, nakon čega izvršavanje programa ide dalje.

Petlja `for` najprije izvrši naredbu n_1, koja mora biti izraz. Za razliku od jezika C[3], n_1 ne može biti deklaracija. Zatim evaluira e i, ako mu je vrijednost `true`, izvršava redom n_3 i n_2, nakon čega ponavlja evaluaciju e te izvršavanje n_3 i n_2 sve dok e ne postane `false`.

Uz to, izvršavanje petlji možemo kontrolirati pomoću ključnih riječi `break` i `continue`. Naredba `break;` odmah prekida izvršavanje petlje u kojoj se nalazi, a `continue;` evaluira e u slučaju petlje `while`, odnosno izvršava n_2 i evaluira e u slučaju petlje `for`, odnosno, prekida izvršavanje trenutnog prolaza kroz petlju i započinje sljedeći.

1.2.5 Povratak iz funkcije — `return`

Naredba `return` se može nalaziti bilo gdje u tijelu funkcije i odmah prekida njeno izvršavanje. Ako funkcija nema povratni tip `void`, naredbu `return` pišemo kao `return` e`;`, gdje je e izraz onog tipa koji je povratni tip funkcije. Ako funkcija ima povratni tip `void`, pišemo samo `return;`.

1.3 Specifikacijski ugovori

Specifikacijski ugovori (u nastavku: ugovori) sadrže tvrdnje o kôdu koji slijedi. Ugovori nikad nisu nužni za kompajliranje i izvršavanje kôda, ali je moguće dinamički provjeriti jesu li zadovoljeni.

Sintaksno, ugovori izgledaju kao posebni komentari te ih kompajleri koji ih ne podržavaju jednostavno ignoriraju.

Ugovori mogu počinjati s `//@` i protezati se duž cijelog retka, ili biti između `/*@` i `@*/` (u primjerima ćemo radi jednostavnosti prikazivati samo varijantu `//@`). Nikad ne bi trebali spremati ništa u memoriju i uvijek bi trebali stati s izvršavanjem, ali zasad kompajleri to ne mogu provjeriti. Dozvoljeno im je da dižu iznimke, pa tako i iznimku da ugovor nije zadovoljen.

1.3.1 Funkcijski ugovori

Ponekad funkcije moraju postaviti uvjete na svoje ulazne parametre da bi ispravno radile. Na primjer, funkcija koja računa logaritam mora primati pozitivan broj, a funkcija koja dodaje element u niz može očekivati da je niz sortiran. Takav uvjet postavljamo s `@requires`.

Također, funkcija može jamčiti da će njen izlaz imati poseban oblik; na primjer, funkcija koja dodaje element u sortirani niz ga može staviti na njegovo

[3] U jeziku C je to dozvoljeno počevši od standarda C99.

pripadajuće mjesto i vratiti sortirani niz. Istaknimo da se "izlaz" ne odnosi samo na povratnu vrijednost funkcije, nego i na njene ulazno-izlazne parametre. Dakle, ako funkcija prima pokazivač koji pokazuje na sortirani niz, funkcija može jamčiti da će na kraju njenog izvršavanja taj pokazivač i dalje pokazivati na sortirani niz. Takav uvjet postavljamo s `@ensures`.

Općenito, definicija funkcije s ugovorima ima oblik:

$$\begin{aligned} &T\ \texttt{moja\_funkcija(}T_1\ \texttt{arg1, }T_2\ \texttt{arg2, ..., }T_n\ \texttt{argn)} \\ &\textit{ugovori} \\ &\{\textit{naredbe}\}, \end{aligned} \tag{1.19}$$

gdje su *ugovori* slijed ugovora `//@requires` e`;` i `//@ensures` e`;`.

Izraz e mora biti tipa `bool`, a u njemu se smiju pojavljivati varijable `arg1`, `arg2`, ..., `argn`. Dodatno, u e se smije pojavljivati posebna „funkcija" `\length` te, u ugovoru, `@ensures` posebna varijabla `\result` i izraz `\old(`e`)`, koji vraća vrijednost izraza e u trenutku poziva funkcije.

Ugovori oblika `@requires` e`;` se izvršavaju neposredno prije izvršavanja tijela funkcije, a oni oblika `@ensures` e`;` se provjeravaju neposredno prije izvršavanja naredbe `return`, s time da je vrijednost varijable `\result` jednaka povratnoj vrijednosti funkcije. Izrazi `\old(`e`)` u ugovorima `@ensures` se izvršavaju nakon ugovora `@requires`, ali prije tijela funkcije.

1.3.2 Invarijante petlji

Invarijante petlji su ugovori oblika

$$\texttt{//@loop\_invariant } e\texttt{;}, \tag{1.20}$$

gdje je e izraz tipa `bool`. Petlje s invarijantama imaju oblik:

$$\begin{aligned} &\texttt{while (}e\texttt{) }\textit{invarijante naredba}\texttt{; i} \\ &\texttt{for (}n_1\texttt{; }e\texttt{; }n_2\texttt{) }\textit{invarijante naredba}\texttt{;}, \end{aligned} \tag{1.21}$$

gdje je *invarijante* slijed invarijanti. Invarijante se provjeravaju pri svakom prolazu kroz petlju, prije nego što se evaluira uvjet nastavka e.

1.3.3 Tvrdnje

U jeziku C0 se tvrdnje mogu koristiti kao naredbe i kao ugovori. Tvrdnja kao naredba ima oblik

$$\texttt{assert(}e_1\texttt{, }e_2\texttt{);}, \tag{1.22}$$

gdje je e_1 izraz tipa `bool` (ono što ispitujemo), a e_2 izraz tipa `string` (poruka o grešci koju ispisujemo ako e_1 ima vrijednost `false`). Tvrdnja kao ugovor ima oblik

```
//@assert e;,
```
(1.23)

gdje je e izraz tipa `bool`. Ugovor `@assert` se mora nalaziti prije naredbe na koju se odnosi; najčešće ta naredba uopće nema smisla ako ugovor nije zadovoljen.

Poglavlje 2

Implementacija kompajlera za C0

Kad smo se detaljno upoznali s C0, možemo iskoristiti to znanje da napišemo kompajler za njega. Kompajliranje se odvija u četiri velika koraka:

1. leksička analiza,
2. sintaksna analiza,
3. semantička analiza,
4. prevođenje u strojni kôd.

Za leksičku i sintaksnu analizu nećemo koristiti gotove alate, nego ćemo ispočetka pisati lekser i parser za jezik C0 slično kao što se radi na Interpretaciji programa. Za semantičku analizu i prevođenje u strojni kôd koristimo infrastrukturu LLVM koja, iako nije *čarobni štapić*, rješava značajan dio posla za nas. Jezik u kojem radimo je C++ te se izvorni kôd cijelog kompajlera može naći u repozitoriju [1]. Budući da je naglasak ovog diplomskog rada na korištenju infrastrukture LLVM, detaljno ćemo proći samo zadnja dva koraka.

2.1 Leksička analiza

U prvom koraku, lekser čita izvorni kôd znak po znak te pokušava prepoznati jedan od zadanih leksičkih *tokena*. Tablica svih korištenih tokena i njihovih značenja se nalazi u dodatku A. Tipovi tokena su implementirani kao enumeracija `TokenTip`, tokeni kao klasa `Token`, a lekser kao klasa `Lekser`.

```
class Token
{
  private:
    TokenTip m_tip;
    std::string m_sadrzaj;
    int m_redak;
    int m_stupac;

  public:
    Token(TokenTip tip = PRAZNO, std::string const& sadrzaj = "",
      int redak = -1, int stupac = -1) :
        m_tip(tip), m_sadrzaj(sadrzaj), m_redak(redak), m_stupac(stupac) {}

    TokenTip getTip() const { return m_tip; }
    int getRedak() const { return m_redak; }
    int getStupac() const { return m_stupac; }

    std::string const& getSadrzaj() const { return m_sadrzaj; }
    bool isOfType(TokenTip tip) const { return m_tip == tip; }
    bool isOfType(Token const& token) const
      { return m_tip == token.m_tip; }

    operator Leaf() const;

    friend std::ostream& operator<<(std::ostream& out, Token const&);
};
```

Isječak 2.1: Klasa Token

Sve metode klase Token su kratke i jasno je što rade, osim eventualno operatora konverzije u `Leaf`, na koji ćemo se osvrnuti u dijelu 2.4.

```
class Lekser
{
  public:
    Lekser(std::shared_ptr<std::ifstream> code);
    std::deque<std::shared_ptr<Token>> leksiraj();

  private:
    std::shared_ptr<std::ifstream> m_code;
    std::deque<std::shared_ptr<Token>> m_tokeni;
    std::string m_linija;
    std::string m_sadrzaj;
    bool m_vracanjeOk;
    int m_redak;
    int m_stupac;

    static std::set<char> escapeZnakovi;
    static std::set<char> escapeSekvence;
```

```
18      void carriageReturn() { m_stupac = -1; }
19
20    protected:
21      char citaj();
22      char procitaj(char znak);
23      bool probajProcitati(char znak);
24      int KleeneZvijezda(std::function<bool(char)> && uvjet);
25      void vrati();
26      void tokeniziraj(TokenTip tip);
27  };
```

Isječak 2.2: Klasa `Lekser`

Metode `citaj`, `procitaj` i `probajProcitati` čitaju sljedeći znak i spremaju ga u međuspremnik. Metoda `citaj` to radi bezuvjetno, `procitaj` to radi samo ako je sljedeći znak jednak onom koji je primila kao argument te u suprotnom prijavljuje grešku, a `probajProcitati` radi isto što i `procitaj`, osim što pri neuspjehu ne prijavljuje grešku, nego poziva `vrati`. Radi jednostavnosti, dozvoljavamo vraćanje unazad samo za jedan znak pa metoda `vrati`, osim što se vraća jedno mjesto unazad u izvornom tekstu i odbacuje zadnji pročitani znak, postavlja zastavicu `m_vracanjeOk` na vrijednost `false`. Ako je vrijednost zastavice `m_vracanjeOk` `false`, onda će program prijaviti grešku i stat će s izvršavanjem. Da bi zastavica `m_vracanjeOk` opet poprimila vrijednost `true`, potrebno je pročitati novi znak pomoću metode `citaj`, `probajProcitati`, ili `procitaj`.

2.2 Sintaksna analiza

U drugom koraku parser preuzima tokene od leksera, utvrđuje da zadovoljavaju gramatiku jezika te pritom od njih sastavlja apstraktna sintaksna stabla (engl. *abstract syntax tree*, u nastavku AST). Beskontekstna gramatika za jezik C0 se može naći u [2]. Slično kao lekser, parser je implementiran kao klasa.

```
1  class Parser
2  {
3    private:
4      std::deque<std::shared_ptr<Token>> m_tokeni;
5      std::shared_ptr<Token> m_zadnji;
6      std::shared_ptr<AST> m_korijen;
7      bool m_vracanjeOk;
8
9      std::deque<std::shared_ptr<Token>> parseUse();
10     std::shared_ptr<AST> parseGlobal();
11     std::shared_ptr<AST> parseType();
12     std::shared_ptr<AST> parseStatement();
```

```
13      std::shared_ptr<AST> parseSimple();
14      std::shared_ptr<AST> parseExpression();
15      std::shared_ptr<AST> parseLogical();
16      std::shared_ptr<AST> parseBitwise();
17      std::shared_ptr<AST> parseEquality();
18      std::shared_ptr<AST> parseComparison();
19      std::shared_ptr<AST> parseShifts();
20      std::shared_ptr<AST> parseAdd();
21      std::shared_ptr<AST> parseFactor();
22      std::shared_ptr<AST> parseAssign();
23      std::shared_ptr<AST> parseAllocate();
24      std::shared_ptr<AST> parseAllocArray();
25      std::shared_ptr<AST> parseArrow();
26      std::shared_ptr<AST> parseDot();
27      std::shared_ptr<AST> parseUnary();
28      std::shared_ptr<AST> parseBase();
29
30      friend std::deque<std::shared_ptr<Token>>
31      UseDirektiva::izvrsi(Parser&);
32
33    protected:
34      Token& citaj();
35      Token& procitaj(TokenTip tip);
36      bool sljedeci(TokenTip tip);
37      void vrati();
38
39    public:
40      Parser(std::deque<std::shared_ptr<Token>>&& tokeni,
41        std::shared_ptr<AST> korijen);
42
43      std::shared_ptr<AST> Korijen() const { return m_korijen; }
44      void parsiraj();
45  };
```

Isječak 2.3: Klasa `Parser`

Sva apstraktna sintaksna stabla proširuju apstraktnu klasu `AST`.

```
1   class AST : public std::enable_shared_from_this<AST>
2   {
3     private:
4       static std::shared_ptr<AST> korijen;
5       static std::list<std::shared_ptr<AST>> sirocad;
6       int m_redak;
7       int m_stupac;
8       AST* m_roditelj;
9       void ucitajDjecu(std::list<std::shared_ptr<AST>> const& djeca);
10      void pobrisiDjecu();
11
12    protected:
```

```
13      static LLVMCompatibility::Context context;
14      AST(int redak, int stupac);
15      AST(AST const& drugi);
16      AST(AST&& drugi);
17      AST(int redak, int stupac,
18      std::list<std::shared_ptr<AST>>&& djeca);
19
20      virtual void Ispisi(std::ostream& out) const = 0;
21      void IspisiDjecu(std::ostream& out) const;
22
23    public:
24      std::list<std::shared_ptr<AST>> m_djeca;
25
26      void dodajDijete(std::shared_ptr<AST> dijete);
27      void Roditelj(AST* roditelj);
28      static void setKorijen(std::shared_ptr<AST> novi_korijen)
29        { korijen = novi_korijen; }
30
31      AST* Roditelj() const { return m_roditelj; }
32      int Redak() const { return m_redak; }
33      int Stupac() const { return m_stupac; }
34      bool IsRoot() const { return m_roditelj == nullptr; }
35
36      AST& operator=(AST const& drugi);
37      AST&& operator=(AST&& drugi);
38
39      virtual llvm::Value* GenerirajKodIzraz() = 0;
40      virtual llvm::Function* GenerirajKodFunkcija() = 0;
41      virtual llvm::Type* GenerirajKodTip() = 0;
42
43      static LLVMCompatibility::Context& Context() { return context; }
44      static llvm::IRBuilder<>& Builder()
45        { return *context.Builder(); }
46      static std::unique_ptr<llvm::Module>& Module()
47        { return context.Module(); }
48      static std::shared_ptr<AST> Korijen() { return korijen; }
49
50      friend std::ostream& operator<<(std::ostream& f,
51        AST const& ast);
52
53      virtual ~AST();
54  };
```

Isječak 2.4: Klasa AST

Funkcije Context, Builder, Module, GenerirajKodIzraz, GenerirajKodTip, i GenerirajKodFunkcija služe za povezivanje s infrastrukturom LLVM, s kojom se najprije trebamo bolje upoznati.

2.3 Uvod u LLVM

LLVM su 2003. godine razvili Cris Lattner i Vikram Adve pod imenom *Low Level Virtual Machine*. Projekt je brzo narastao i prestao imati veze samo s virtualizacijom pa je staro ime odbačeno 2011. godine te je „LLVM" proglašeno punim imenom projekta.

LLVM je nastao s ciljem omogućavanja takozvane „cjeloživotne optimizacije kôda". Takva cjeloživotna optimizacija obuhvaća interproceduralne optimizacije[4] za vrijeme povezivanja, za vrijeme instalacije programa na korisničko računalo, dinamičke optimizacije za vrijeme izvršavanja programa, te optimizacije između pokretanja programa koristeći podatke dobivene profiliranjem korisničkog računala. Više o tome se može naći u [3]. Navedeni ciljevi se postižu u dva koraka:

1. posebna reprezentacija kôda,
2. dizajn kompajlera koji iskorištava prednosti te reprezentacije.

Reprezentacija kôda koju LLVM koristi se naziva *LLVM Intermediate Representation* (u nastavku LLVM IR), što je strogo tipiziran reduciran skup instrukcija (RISC). Glavne razlike između LLVM IR i uobičajenih reduciranih skupova instrukcija su to što, umjesto konačno mnogo registara, LLVM IR koristi beskonačan skup privremenih varijabli snazivima `%0`, `%1`, itd. te sadrži dodatne informacije o programu u svrhu bolje analize, kao što su informacije o tipovima i grafovi kontrole toka. Veliki dio semantičke analize će se svoditi upravo na prevođenje jezika C0 u LLVM IR. Više o LLVM IR se može naći u [4].

2.4 Semantička analiza

2.4.1 Prostor imena `LLVMCompatibility`

Sada znamo osnove o LLVM, ali to nam još uvijek nije dovoljno da znamo kako se koristi. Zato ćemo se sad upoznati s klasama koje spajaju AST i LLVM IR te ćemo posebnu pažnju obratiti na detalje o tome kako što spajamo. Počnimo s klasama iz prostora imena `LLVMCompatibility`: `Block` i `Context`.

```
1 class Block
2 {
3   private:
4     llvm::BasicBlock* m_sadrzaj;
```

[4]To su optimizacije koje poboljšavaju performanse onih programa koji sadrže mnogo poziva kratkih funkcija.

```
5     Block* m_roditelj;
6     std::map<std::string, llvm::AllocaInst*> m_varijable;
7     static Block* trenutniBlok;
8   public:
9     void Sadrzaj(llvm::BasicBlock* sadrzaj) { m_sadrzaj = sadrzaj; }
10    llvm::BasicBlock* Sadrzaj() { return m_sadrzaj; }
11    std::map<std::string, llvm::AllocaInst*>& Varijable()
12      { return m_varijable; }
13
14    Block* Roditelj() const { return m_roditelj; }
15    void Roditelj(Block* roditelj) { m_roditelj = roditelj; }
16
17    llvm::AllocaInst* Trazi(std::string const& imeVarijable);
18    std::string const& Trazi(llvm::Value* varijabla);
19    llvm::AllocaInst* Dodaj(llvm::AllocaInst* varijabla,
20      std::string const& ime);
21    static Block* TrenutniBlok() { return trenutniBlok; }
22    static void TrenutniBlok(Block* noviTrenutni)
23      { trenutniBlok = noviTrenutni; }
24 };
```

Isječak 2.5: Klasa `Block`

Klasa `Block` predstavlja jedan blok u izvornom kôdu programa, a implementirana je kao čvor u stablu blokova. Pogledamo li privatni dio klase, dvije stvari nam odmah upadaju u oči: `BasicBlock` i `AllocaInst`. `BasicBlock` je klasa koja LLVM-u služi za spremanje instrukcija LLVM IR, a `AllocaInst` predstavlja komad memorije na stogu (gdje se obično nalazi neka varijabla). Dakle, klasa `Block` sadrži pokazivač na popis instrukcija koje sadrži, pokazivač na roditelja, popis varijabli koje su deklarirane u njemu i pokazivač na zadnji otvoreni nezatvoreni blok. Jedine netrivijalne metode klase `Block` su metode imena `Trazi`.

```
1 llvm::AllocaInst* Block::Trazi(std::string const& imeVarijable)
2 {
3   if (m_varijable.find(imeVarijable) != m_varijable.end())
4     return m_varijable[imeVarijable];
5   else if (m_roditelj != nullptr)
6     return m_roditelj->Trazi(imeVarijable);
7   else
8     return nullptr;
9 }
```

Isječak 2.6: Metoda `Trazi(std::string const&)`

Prva metoda `Trazi` je jednostavna: prima ime varijable i vrati pokazivač na nju ako varijabla s danim imenom postoji u bloku. Ako ne, traži je u bloku roditelju i rekurzivno nastavlja dok ne dođe do korijena. Ako je ni tamo ne nađe, vraća `nullptr`.

```
1  std::string const& Block::Trazi(llvm::Value* varijabla)
2  {
3    for (std::map<std::string, llvm::AllocaInst*>::iterator
4      it = m_varijable.begin(); it != m_varijable.end(); ++it)
5    {
6      if (varijabla->getName().find(it->first) == 0)
7      {
8        for (llvm::Value::use_iterator jt = it->second->use_begin();
9          jt != it->second->use_end(); ++jt)
10       {
11         if (jt->getUser()->getName() == varijabla->getName())
12           return it->first;
13       }
14     }
15   }
16   if(m_roditelj != nullptr)
17     return m_roditelj->Trazi(varijabla);
18
19   return std::string();
20 }
```

Isječak 2.7: Metoda `Trazi(llvm::Value*)`

Druga metoda `Trazi` je zanimljivija: ona prima pokazivač na neku vrijednost i, ako je ta vrijednost varijabla, vraća njeno ime. Koristimo pokazivač na `Value` jer je `Value` polimorfna bazna klasa od `AllocaInst`.

Pogledamo li kôd, vidimo da klasa `Value` ima metodu `getName`. Razlog zašto ne možemo samo pozvati `getName` je taj što LLVM IR koristi *Static Single Assignment* (u nastavku SSA). SSA je tehnika koja omogućuje da svaka varijabla bude deklarirana prije korištenja te da se svakoj varijabli vrijednost pridružuje točno jednom. Pokušamo li postojećoj varijabli pridružiti novu vrijednost, LLVM će umjesto toga generirati novu varijablu čije ime će biti konkatenacija imena originalne varijable i rednog broja pridruživanja te će pridružiti vrijednost novoj varijabli. Takve varijable ćemo nazivati *verzijama*. Ovakva implementacija SSA dozvoljava da verzije imaju dvosmislena imena, na primjer `var11` — radi li se o verziji 11 varijable `var` ili o verziji 1 varijable `var1`? Odgovor ovisi o kontekstu. LLVM vodi računa o tome da ne reciklira imena verzija pa će `var11` biti verzija 11 varijable `var` ako je do tog trenutka varijabla `var` već imala 10 verzija, a `var1` još nije imala nijednu. Isto tako, `var11` će biti verzija 1 varijable `var1` ako do tog trenutka varijabla `var` nije imala 11 verzija i time zauzela to ime.

Budući da ime verzije sadrži ime originalne varijable, prvo gledamo postoji li u bloku varijabla čije ime je početni komad imena argumenta. Ako postoji, pogledamo postoji li u popisu njenih verzija jedna čije ime je isto kao ime argumenta. Ako da, našli smo varijablu koju smo tražili. Ako ne, tražimo dalje. Opet, ako nema odgovarajuće varijable u bloku, nastavljamo rekurzivno tražiti u

roditeljskim blokovima i, ako je ne nađemo, vraćamo prazan string.

Ovaj algoritam iskorištava gore opisana svojstva LLVM-ove implementacije SSA. Pogledamo li opet primjer `var11`, vidimo da je to isključivo ili ime verzije od `var` ili od `var1`. Dakle, iako su i `var` i `var1` početni komadi od `var11`, samo jedna od tih varijabli ima `var11` u svom popisu verzija pa će funkcija `Trazi` uvijek vratiti ispravno ime originalne varijable.

Promotrimo klasu `Context`.

```
1  class Context
2  {
3    private:
4      std::deque<std::shared_ptr<Blok>> m_blokovi;
5      std::unique_ptr<llvm::LLVMContext> m_context;
6      std::unique_ptr<llvm::Module> m_modul;
7      std::unique_ptr<llvm::IRBuilder<>> m_irBuilder;
8
9      std::map<std::string, std::map<std::string, int>> m_strukture;
10     std::map<std::string, llvm::Type*> m_poznatiTipovi;
11     std::map<llvm::Type*, std::string> m_imenaTipova;
12     std::map<std::string, std::string> m_aliasiTipova;
13     llvm::Function* m_funkcijaMain;
14
15   public:
16     Context();
17
18     void DodajBlok(llvm::BasicBlock* noviBlok);
19     void ZatvoriBlok()
20       { Blok::TrenutniBlok(Blok::TrenutniBlok()->Roditelj()); }
21     void GenerirajKod(AST& korijenStabla);
22     void IzbrisiBlok() { m_blokovi.pop_back(); }
23     void DodajStrukturu(std::string const& imeStrukture);
24     void DodajElementStrukture(std::string const& imeStrukture,
25       std::string const& imeElementa);
26     void DodajNoviTip(std::string const& imeTipa, llvm::Type* tip);
27     int DohvatiIndeksElementaStrukture(std::string const&
28       imeStrukture, std::string const& imeElementa)
29         { return m_strukture[imeStrukture][imeElementa]; }
30     std::map<std::string, llvm::AllocaInst*>& LokalneVarijable()
31       { return TrenutniBlok()->Varijable(); }
32     std::map<std::string, std::string>& AliasiTipova()
33       { return m_aliasiTipova; }
34     Blok* TrenutniBlok() { return Blok::TrenutniBlok(); }
35     std::unique_ptr<llvm::Module>& Module() { return m_modul; }
36     std::unique_ptr<llvm::IRBuilder<>>& Builder()
37       { return m_irBuilder; }
38     std::unique_ptr<llvm::LLVMContext>& LLVMContext()
39       { return m_context; }
40     llvm::Type* PrevediTip(std::string const& tip);
```

```
41      std::string const& PrevediTip(llvm::Type* tip)
42        { return m_imenaTipova[tip]; }
43
44      llvm::AllocaInst* StackAlokacija(llvm::Function* funkcija, llvm::Type*
            tipVarijable, std::string const& imeVarijable);
45  };
```

Isječak 2.8: Klasa `Context`

Klasa `Context` je spremnik za podatke koji se tiču cijelog programa koji kompajliramo, a sadrži popis svih blokova koji postoje, deklariranih struktura, tipova i njihovih aliasa (definiranih s `typedef`). Uz to, sadrži i pokazivače na `LLVMContext`, `Module` i `IRBuilder`.

`LLVMContext` je klasa koja ima ulogu vlasnika svih objekata koje LLVM kreira. Ponaša se kao crna kutija pa ne znamo što točno radi i kako to radi, osim da upravlja svom memorijom koju LLVM koristi.

`Module` predstavlja jednu smislenu cjelinu kôda koji se kompajlira te sadrži sve globalne varijable, deklaracije funkcija i njihove implementacije.

`IRBuilder` nam daje sučelje za lakše spremanje instrukcija u `BasicBlock`. `IRBuilder` nije klasa, nego predložak klase koji prima dva parametra: klasu koju koristi za kreiranje konstanti i klasu koju koristi kao alat za dodavanje instrukcija. Podrazumijevana vrijednost za prvi argument pruža minimalno predizračunavanje konstanti[5], što je za nas dovoljno dobro. Korištenjem drugog parametra možemo izvršiti proizvoljni kôd prilikom svakog dodavanja instrukcije, ali time se nećemo baviti u ovom projektu pa i tu koristimo podrazumijevanu vrijednost.

U dijelu 2.4.4 ćemo se osvrnuti na ostale članske varijable. Što se tiče metoda klase `Context`, sve su vrlo jednostavne pa ćemo bolje proučiti samo metodu `GenerirajKod`.

```
1  void Context::GenerirajKod(AST& korijenStabla)
2  {
3    std::vector<llvm::Type*> tipoviArgumenata;
4    llvm::FunctionType* tipMaina = llvm::FunctionType::get(
5      PrevediTip("void"), tipoviArgumenata, false);
6
7    m_funkcijaMain = llvm::Function::Create(tipMaina,
8      llvm::GlobalValue::ExternalLinkage, "main", m_modul.get());
9
10   korijenStabla.GenerirajKodIzraz();
11 }
```

Isječak 2.9: Metoda `GenerirajKod` klase `Context`

[5]To je računanje vrijednosti aritmetičkih izraza s konstantama za vrijeme kompajliranja umjesto za vrijeme izvođenja.

`GenerirajKod` deklarira funkciju `main` i pokreće kompajliranje cijelog programa. Prvo deklariramo prazan `vector` koji predstavlja njene argumente (jer ne prima nijedan), a zatim zapamtimo njen tip: `void()`. Na kraju deklariramo funkciju, dodijelimo joj blok za spremanje instrukcija te započnemo generiranje LLVM IR iz AST-ova tako da pozovemo `GenerirajKod` na korijenu apstraktnog sintaksnog stabla programa.

2.4.2 Bazne klase

Sad znamo dovoljno da se možemo vratiti na funkcije koje su nam preostale iz dijela 2.2. Pogledamo li opet isječak 2.4, bit će jasno da `Context`, `Builder` i `Module` samo dohvaćaju `Context` i njegov `IRBuilder`, odnosno `Module` pa nam nisu zanimljive. Metode `GenerirajKodTip`, `GenerirajKodFunkcija` i `GenerirajKodIzraz` služe svakom izvedenom AST-u da od sebe i svoje djece napravi strukturu koju LLVM zna prevesti u LLVM IR. Za to nam nije dovoljna jedna univerzalna funkcija jer LLVM ima posebne strukture za funkcije, tipove i vrijednosti. U ostatku odjeljka ćemo vidjeti kako se od svakog AST-a napravi struktura s kojom LLVM može raditi.

Počnimo s klasama koje predstavljaju funkcije, tipove i izraze.

```
1  class FunkcijaAST : public AST
2  {
3    public:
4      FunkcijaAST(int redak, int stupac) : AST(redak, stupac) {}
5      FunkcijaAST(int redak, int stupac,
6        std::list<std::shared_ptr<AST>>&& djeca) : AST(redak, stupac,
7          std::move(djeca)) {}
8      FunkcijaAST(FunkcijaAST const& drugi) : AST(drugi) {}
9      llvm::Function* GenerirajKod() { return GenerirajKodFunkcija(); }
10     virtual llvm::Type* GenerirajKodTip() override
11       { return nullptr; }
12     virtual llvm::Value* GenerirajKodIzraz() override
13       { return nullptr; }
14     virtual ~FunkcijaAST() {}
15 };
```

Isječak 2.10: Klasa `FunkcijaAST`

```
1  class TipAST : public AST
2  {
3    public:
4      TipAST(int redak, int stupac) : AST(redak, stupac) {}
5      TipAST(int redak, int stupac,
6        std::list <std::shared_ptr<AST>>&& djeca) : AST(redak, stupac,
7          std::move(djeca)) {}
```

```
8     TipAST(TipAST const& drugi) : AST(drugi) {}
9     llvm::Type* GenerirajKod() { return GenerirajKodTip(); }
10    virtual llvm::Function* GenerirajKodFunkcija() override
11      { return nullptr; }
12    virtual llvm::Value* GenerirajKodIzraz() override
13      { return nullptr; }
14    virtual ~TipAST() {}
15 };
```

Isječak 2.11: Klasa `TipAST`

```
1  class IzrazAST : public AST
2  {
3    public:
4      IzrazAST(int redak, int stupac) : AST(redak, stupac) {}
5      IzrazAST(int redak, int stupac,
6        std::list<std::shared_ptr<AST>>&& djeca) : AST(redak, stupac,
7          std::move(djeca)) {}
8      IzrazAST(IzrazAST const& drugi) : AST(drugi) {}
9      llvm::Value* GenerirajKod() { return GenerirajKodIzraz(); }
10     virtual llvm::Function* GenerirajKodFunkcija() override
11       { return nullptr; }
12     virtual llvm::Type* GenerirajKodTip() override
13       { return nullptr; }
14     virtual ~IzrazAST() {}
15 };
```

Isječak 2.12: klasa `IzrazAST`

Ove tri klase su i dalje apstraktne jer ne definiraju sve funkcije `GenerirajKod` klase `AST`. Ako pogledamo, vidjet ćemo da svaka gore navedena klasa trivijalno implementira sve funkcije `GenerirajKod` osim „svoje". Odnosno, `IzrazAST` ne implementira `GenerirajKodIzraz`, `FunkcijaAST` ne implementira `GenerirajKodFunkcija`, a `TipAST` ne implementira `GenerirajKodTip`. To smo tako ostavili jer želimo da konkretne klase za AST nasljeđuju od ovih, malo manje apstraktnih klasa od klase `AST`, te da svaka sebi definira ispravnu funkciju za generiranje kôda.

Sada ćemo pogledati dvije nestandardne AST klase — `Program` i `UseDirektiva`. Kažemo da su nestandardne jer ne nasljeđuju niti od jedne gore navedene klase pa im nešto drugačije pristupamo pri kompajliranju.

```
1  class Program : public AST
2  {
3    public:
4      Program();
5      Program(int redak, int stupac,
6      std::list <std::shared_ptr<AST>>&& djeca) = delete;
```

```
7     Program(TipAST const& drugi) : AST(drugi) {}
8     virtual void Ispisi(std::ostream& out) const override
9       { IspisiDjecu(out); }
10    virtual llvm::Function* GenerirajKodFunkcija() override
11      { return nullptr; }
12    virtual llvm::Value* GenerirajKodIzraz() override
13      { return nullptr; }
14    virtual llvm::Type* GenerirajKodTip() override
15      { return nullptr; }
16    void GenerirajKod();
17 };
```

Isječak 2.13: Klasa `Program`

`Program` je korijen AST-a cijelog programa i nije funkcija, izraz ni tip pa trivijalno implementira sve tri funkcije za generiranje kôda. Umjesto njih, `Program` koristi svoju funkciju `GenerirajKod`.

```
1  void Program::GenerirajKod()
2  {
3    for (std::list<std::shared_ptr<AST>>::iterator it = m_djeca.begin();
4      it != m_djeca.end(); ++it)
5    {
6      (*it)->GenerirajKodTip();
7      (*it)->GenerirajKodFunkcija();
8    }
9  }
```

Isječak 2.14: Funkcija `GenerirajKod`

Program kompajliramo tako da kompajliramo ono što piše u njemu. Svako dijete pokušamo kompajlirati i kao tip i kao funkciju jer je jedan od tih poziva uvijek trivijalan, a drugi zapravo nešto radi. Mogli smo u petlji pozvati i `GenerirajKodIzraz`, ali taj poziv nikad ništa ne bi napravio. Prisjetimo se, jezik C0 u globalnom dosegu može imati samo direktive `#use`, deklaracije i definicije struktura te deklaracije i definicije funkcija.

```
1  class UseDirektiva : public AST
2  {
3    private:
4      std::string path;
5    protected:
6      virtual void Ispisi(std::ostream& out) const override {}
7    public:
8      UseDirektiva(int redak, int stupac,
9        std::list<std::shared_ptr<AST>>&& djeca);
10     UseDirektiva(UseDirektiva const& drugi);
11     std::deque<std::shared_ptr<Token>> izvrsi(Parser& parser);
12     virtual llvm::Value* GenerirajKodIzraz() override
```

```
13      { return nullptr; }
14    virtual llvm::Value* GenerirajKodFunkcija() override
15      { return nullptr; }
16    virtual llvm::Value* GenerirajKodTip() override
17      { return nullptr; }
18    virtual ~UseDirektiva(){}
19 };
```

Isječak 2.15: Klasa `UseDirektiva`

`UseDirektiva` nema veze s prevođenjem u LLVM IR, nego se ponaša kao makro naredba koja sebe zamijeni cijelim sadržajem datoteke koja joj je argument. Ovdje se pojavljuje problem jer tek u fazi sintaksne analize prepoznamo da imamo direktivu `#use` u programu — polazna datoteka je već leksirana i spremna je za parsiranje, a ona koja je proslijeđena direktivi `#use` tek treba biti leksirana.

Taj problem rješavamo pomoću poziva funkcije `izvrsi` za vrijeme parsiranja.

```
1  std::deque<std::shared_ptr<Token>> UseDirektiva::izvrsi(Parser& parser)
2  {
3    std::shared_ptr<std::ifstream> datoteka =
4      std::make_shared<std::ifstream>(path);
5    if (!datoteka->is_open())
6      throw Greska("Greška", Redak(), Stupac(),
7        "Ne mogu otvoriti datoteku " + path);
8
9    std::deque<std::shared_ptr<Token>> drugiTokeni;
10   Lekser drugiLex(datoteka);
11   std::ofstream drugiLexOut(path + "LekserOut.txt");
12   try
13   {
14     drugiTokeni = drugiLex.leksiraj();
15     for(std::deque<std::shared_ptr<C0Compiler::Token>>::iterator it
16       = drugiTokeni.begin(); it != drugiTokeni.end(); ++it)
17         drugiLexOut << **it << std::endl;
18   }
19   catch (LeksickaGreska const& e)
20   {
21     std::cout << e.what();
22   }
23   return drugiTokeni;
24 }
```

Isječak 2.16: Funkcija `Izvrsi`

Čim vidimo da polazna datoteka sadrži direktivu `#use` (i ustanovimo da nakon nje piše staza do datoteke), na njoj pozovemo metodu `izvrsi` i spremimo tokene koje ta metoda vrati u međuspremnik.

```
1  std::deque<std::shared_ptr<Token>> resultTokeni;
2  procitaj(POCETAK);
3
4  while (sljedeci(USE))
5  {
6    std::deque<std::shared_ptr<Token>> tempTokeni =
7      std::move(parseUse());
8
9    for (std::deque<std::shared_ptr<Token>>::iterator it =
10     tempTokeni.begin(); it != tempTokeni.end(); ++it)
11       resultTokeni.push_back(*it);
12 }
13
14 for (std::deque<std::shared_ptr<Token>>::reverse_iterator rit =
15   resultTokeni.rbegin(); rit != resultTokeni.rend(); ++rit)
16     m_tokeni.push_front(*rit);
17
18 while(!citaj().OfType(KRAJ))
19   m_korijen->dodajDijete(parseGlobal());
```

Isječak 2.17: Funkcija `Parsiraj`

Nakon što isto napravimo za svaku direktivu `#use`, dodamo sadržaj međuspremnika na početak liste svih tokena iz polazne datoteke i nastavimo parsiranje.

U nastavku slijede standardne klase za `AST`, u smislu da sve predstavljaju izraz, funkciju ili tip te nasljeđuju od odgovarajuće apstraktne bazne klase. Počet ćemo od onih koje je najjednostavnije prevesti u LLVM IR i postupno ćemo se penjati do najsloženijih.

Budući da su sve klase za `AST` međusobno jako slične, u većini slučajeva nećemo prikazivati kako izgleda pojedina klasa, nego ćemo se usredotočiti samo na implementaciju funkcija `GenerirajKodTip`, `GenerirajKodIzraz`, odnosno `GenerirajKodFunkcija`.

2.4.3 ASTList

Klasa `ASTList` je najjednostavnija klasa za `AST` jer ne radi ništa i ne prevodi se u LLVM IR. Koristimo je na mjestima gdje ne znamo unaprijed koliko će djece neki čvor imati, na primjer, kao spremnik za argumente funkcije, članove strukture, naredbe u funkciji itd.

```
1  llvm::Value* ASTList::GenerirajKodIzraz()
2  {
3    for (iterator it = begin(); it != end(); ++it)
4    {
5      (*it)->GenerirajKodIzraz();
```

```
6     (*it)->GenerirajKodTip();
7   }
8   return llvm::Constant::getNullValue(
9     llvm::Type::getInt32Ty(*Context().LLVMContext()));
10 }
```

Isječak 2.18: Metoda GenerirajKodIzraz

Listu čvorova kompajliramo tako da kompajliramo svaki element liste. Svaki njen element pokušamo prevesti kao izraz i kao tip jer znamo da je točno jedan od tih poziva netrivijalan, samo ne znamo koji. Ne pozivamo GenerirajKodFunkcija jer su sve klase koje implementiraju tu metodu uvijek u globalnom dosegu pa se nikad neće naći pod ASTList-om — više o njima u odjeljku 2.4.15. Na kraju vraćamo LLVM IR konstantu 0 samo zato što nešto moramo vratiti, ali nigdje ne koristimo tu povratnu vrijednost. Ne vraćamo nullptr jer LLVM to interpretira kao da je došlo do greške, a nije.

2.4.4 Tipovi

Klasa Tip služi za prevođenje tipova iz jezika C0 u tipove iz LLVM IR.

```
1 llvm::Type* Tip::GenerirajKodTip()
2 {
3   llvm::Type* ret = Context().PrevediTip(m_ime);
4   if (ret == nullptr)
5     throw new SemantickaGreska(Redak(), Stupac(),
6       "Nepoznat tip '" + m_ime + "'");
7   return ret;
8 }
```

Isječak 2.19: Metoda GenerirajKodTip klase Tip

Dva su razloga zašto je metoda GenerirajKodTip toliko jednostavna. Prvi je to što LLVM ima ugrađenu podršku za sve primitivne tipove i vrlo dobre alate za konstruiranje složenih, a drugi je to što smo većinu njenog posla obavili u konstruktoru klase Context.

Pokazivačke tipove rješavamo na sličan način. Na primjer, za int* koristimo poziv llvm::Type::getInt32PtrTy(*m_context).

Polja su nešto kompliciranija jer želimo da se, za svaki tip T, tip T[] razlikuje od tipa T*. Jedna ideja kako bismo ih mogli implementirati je pomoću llvm::ArrayType, ali to ne možemo napraviti jer llvm::ArrayType mora znati svoju duljinu za vrijeme kompajliranja. To nam predstavlja problem jer želimo da se, na primjer, dva polja tipa char[] s različitim brojem elemenata zaista ponašaju kao da su istog tipa — tako se ponašaju u jeziku C0. Čak i ako bismo nekako riješili taj problem, za njim dolazi jedan još veći: nemamo garanciju da

uopće možemo izračunati duljinu polja za vrijeme kompajliranja!

Prisjetimo se odjeljka 1.1.5 — poziv `alloc_array` prima proizvoljni izraz tipa `int` kao duljinu polja. Kako rješavamo slučaj kad je taj izraz poziv funkcije koja nikad ne stane s izvršavanjem? Nikako, nego radije odustajemo od implementacije preko `llvm::ArrayType` i sami konstruiramo strukturu podataka za tip polja. Da bismo znali sve o nekom polju, dovoljno je znati gdje se nalazi u memoriji i koliko elemenata sadrži pa tip T`[]` implementiramo kao `struct{int, char*}*`. Koristimo pokazivač na strukturu umjesto same strukture da bismo lakše prepoznali neinicijalizirano polje; umjesto da provjeravamo vrijednosti članova strukture, samo provjerimo imamo li pokazivač `NULL`. Pritom ne govorimo o `struct{int, char*}*` u jeziku C++, nego o ekvivalentnom „objektu" u LLVM IR. Isto tako, smatramo da je polje neinicijalizirano kad u varijabli tipa T`[]` piše pokazivač `NULL` u LLVM IR, a ne pokazivač `NULL` u jeziku C++, odnosno `nullptr`.

U sljedećem isječku vidimo kako kažemo LLVM da generira takvu strukturu u LLVM IR.

```
llvm::StructType* tempTip = llvm::StructType::create(*m_context,
    "char[]");
std::vector<llvm::Type*> tempTijelo;
tempTijelo.push_back(llvm::Type::getInt32Ty(*m_context));
tempTijelo.push_back(llvm::Type::getInt8PtrTy(*m_context));
static_cast<llvm::StructType*>(tempTip)->setBody(tempTijelo);
m_poznatiTipovi["char[]"] = llvm::PointerType::getUnqual(tempTip);
```

Isječak 2.20: Implementacija tipa `char[]`

Sličan problem imamo s tipom `string` pa njega implementiramo kao `struct{int, char*, bool}*`. U strukturu smo dodali jedan `bool` samo zato što želimo da se tip `string` razlikuje od `char[]`; osim toga on ne služi ničemu.

2.4.5 Strukture

U prethodnom dijelu smo vidjeli kako se neke jednostavne strukture prevode u LLVM IR. Sad ćemo vidjeti kako se to radi s općenitim strukturama. Budući da strukture u jeziku C0 mogu biti deklarirane bez definicije, imamo dvije klase: `DeklaracijaStrukture` i `DefinicijaStrukture`.

Deklariramo strukturu na isti način kao u isječku 2.20, uz par dodatnih koraka.

```
1 llvm::Type* DeklaracijaStrukture::GenerirajKodTip()
2 {
3   llvm::StructType* novaStruktura =
4   llvm::StructType::create(*Context().LLVMContext(), m_ime);
5
6   Context().DodajStrukturu(m_ime);
```

```
7   Context().DodajNoviTip("struct " + m_ime, novaStruktura);
8   return novaStruktura;
9 }
```

Isječak 2.21: Metoda `GenerirajKodTip` klase `DeklaracijaStrukture`

LLVM IR ne pristupa članovima strukture preko njihovih imena, nego im dodjeljuje indekse, počevši od nule. Nama to nije dovoljno dobro jer se u jeziku C0 članovi strukture mogu imenovati i pristupa im se preko imena. Zato klasa `Context` ima popis svih deklariranih struktura (isječak 2.8), koji za dano ime strukture ima popis imena svih njenih članova i odgovarajućih indeksa.

Metoda `Context::DodajStrukturu` samo dodaje novu strukturu u popis i pridružuje joj prazan `std::map<std::string, int>` u koji ćemo popisati imena njenih članova kad je budemo definirali.

Metoda `Context::DodajNoviTip` dodaje tipove `struct` *ime_strukture*, `struct` *ime_strukture*`*` i `struct` *ime_strukture*`[]` u popis poznatih tipova, slično kao u isječku 2.20. Postupak nije kratak, ali smo sve to već vidjeli pa ćemo preskočiti njegovo objašnjenje.

```
1  llvm::Type* DefinicijaStrukture::GenerirajKodTip()
2  {
3    if(Context().StrukturaDefinirana(m_ime))
4      throw SemantickaGreska(Redak(), Stupac(), "Struktura " + m_ime +
5        " je već definirana");
6
7    llvm::StructType* novaStruktura = static_cast<llvm::StructType*>
8      (DeklaracijaStrukture::GenerirajKod());
9
10   ASTList& elementi = *std::dynamic_pointer_cast<ASTList>
11     (m_djeca.front());
12   std::vector<llvm::Type*> tijelo;
13
14   while(!elementi.empty())
15   {
16     llvm::Type* tipElementa = (*elementi.begin())->GenerirajKodTip();
17     tijelo.push_back(tipElementa);
18     elementi.pop_front();
19
20     Context().DodajElementStrukture(m_ime,
21       std::dynamic_pointer_cast<Leaf>(*elementi.begin())->Sadrzaj());
22     elementi.pop_front();
23   }
24   m_djeca.pop_front();
25   novaStruktura->setBody(tijelo);
26   return novaStruktura;
27 }
```

Isječak 2.22: Metoda `GenerirajKodTip` klase `DefinicijaStrukture`

Prvo deklariramo strukturu, nakon čega zapišemo tip i ime svakog njenog člana. Metoda `Context::DodajElementStrukture` sprema ime i odgovarajući indeks u ranije spomenuti `std::map<std::string, int>`, koji kasnije koristimo za operatore dohvata, kako je opisano u sljedećem isječku.

```
1  llvm::Value* Tocka::GenerirajKodIzraz()
2  {
3    llvm::Value* varijabla = m_djeca.front()->GenerirajKodIzraz();
4    std::string imeVarijable = varijabla->getName();
5    m_djeca.pop_front();
6    std::string imeStrukture = Context().PrevediTip(
7      Context().LokalneVarijable()[imeVarijable]->getType());
8
9    llvm::Value* clan = m_djeca.front()->GenerirajKodIzraz();
10   m_djeca.pop_front();
11
12   int indeksClana = Context().DohvatiIndeksClanaStrukture(
13     imeStrukture, clan->getName());
14   llvm::Value* dohvaceniClan = Context().DohvatiClanStrukture(
15     Context().TrenutniBlok()->Trazi(imeStrukture), indeksClana);
16
17   if (dohvaceniClan == nullptr)
18     throw SemantickaGreska(Redak(), Stupac(),
19       "Greška pri dohvaćanju člana " + clan->getName().str() +
20         " strukture " + imeStrukture);
21
22   return dohvaceniClan;
23 }
```

Isječak 2.23: Metoda `GenerirajKodIzraz` klase `Tocka`

Proces je vrlo jednostavan. Prvo pročitamo ime varijable tipa `struct` *ime_strukture,* puno ime strukture i ime člana kojeg želimo dohvatiti. Zatim uz pomoć imena strukture i člana dohvatimo indeks člana u strukturi. Sada pomoću funkcije `Context::DohvatiClanStrukture` i indeksa člana dohvatimo član i vratimo ga. Tu funkciju ćemo često koristiti pa je dobro pogledati što točno radi.

```
1  llvm::Value* Context::DohvatiClanStrukture(llvm::Value* struktura,
2    int indeksClana)
3  {
4    llvm::Value* indeks = llvm::ConstantInt::get(
5      *Context().LLVMContext(), llvm::APInt(32, indeksClana, true));
6
7    std::vector<llvm::Value*> indeksi(2);
8    indeksi[0] = llvm::ConstantInt::get(
9      *Context().LLVMContext().get(), llvm::APInt(32, 0, true));
10   indeksi[1] = indeks;
```

```
11   llvm::Value* clanStrukture = Builder()->CreateGEP(
12     struktura, indeksi, "pointerNaClanStrukture");
13
14   return Builder()->CreateLoad(clanStrukture, "elementStrukture");
15 }
```

Isječak 2.24: Metoda `DohvatiClanStrukture` klase `Context`

Ono što nazivamo strukturom u LLVM je zapravo pokazivač na strukturu pa je naša varijabla tipa `struct` *ime_strukture* zapravo tipa `struct` *ime_strukture*`*`. Dakle, da bismo dohvatili član strukture moramo prvo dereferencirati „strukturu". LLVM IR dozvoljava dereferenciranje pokazivača slično kao jezik C: (`*`*varijabla*)`[`*indeks*`]` vraća isto što i *varijabla*`[0][`*indeks*`]` pa pokazivač na strukturu možemo tretirati kao dvodimenzionalno polje te i-ti član strukture kao $(0, i)$-ti element tog polja. S tim u vidu, spremimo 0 i indeks traženog elementa u `std::vector`, proslijedimo ga funkciji `CreateGEP` (*Get Element Pointer*), pomoću koje dohvatimo pokazivač na $(0, i)$-ti element „polja", kojeg zatim dereferenciramo i vratimo rezultat.

Operator `->` se prevodi vrlo slično pa ćemo ga preskočiti.

2.4.6 Naredba `Typedef`

LLVM IR nema koncept `typedef`a pa `typedef` ne prevodimo, nego samo u klasu `Context` dodamo novi alias za poznati tip.

```
1  llvm::Value* TypeDef::GenerirajKodIzraz()
2  {
3    if (Context().TrenutniBlok()->Roditelj() != nullptr)
4      throw SemantickaGreska(Redak(), Stupac(), "typedef se smije
5        pojavljivati samo u globalnom dosegu");
6
7    Context().AliasiTipova()[m_alias] = m_tip;
8    return llvm::Constant::getNullValue(
9      llvm::Type::getInt32Ty(*Context().LLVMContext()));
10 }
```

Isječak 2.25: Metoda `GenerirajKodTip` klase `Typedef`

2.4.7 Literali

Literali su uvijek listovi apstraktnog sintaksnog stabla pa sve klase literala nasljeđuju od klase `Leaf`.

```
1  class Leaf : public IzrazAST
2  {
```

```
3   protected:
4     Token m_sadrzaj;
5     virtual void Ispisi(std::ostream& out) const override;
6   public:
7     Leaf(Token const& sadrzaj);
8     Leaf(Leaf const& drugi);
9
10    operator Token() const { return m_sadrzaj; }
11    std::string const& Sadrzaj() const
12      { return m_sadrzaj.Sadrzaj(); }
13    TokenTip TipTokena() const { return m_sadrzaj.Tip(); }
14    virtual llvm::Value* GenerirajKodIzraz() override
15      { return nullptr; };
16  };
```

Isječak 2.26: Klasa `Leaf`

`Leaf` se nikako ne prevodi u LLVM IR jer ne predstavlja ništa konkretno; on je samo objekt klase `Token` zapakiran u klasu `AST`. Zato u njegovoj metodi `GenerirajKodIzraz` vraćamo `nullptr`, a ne konstantu 0 — umjesto klase `Leaf` treba koristiti njene izvedene klase.

Literali u jeziku C0 mogu biti tipa `int`, `bool`, `char` ili `string`, ali ćemo pokazati samo kako se literali tipa `bool` i `string` prevode u LLVM IR jer je postupak vrlo jednostavan i sasvim analogan za literale tipova `int`, `bool` i `char`, a za literale tipa `string` je nešto zanimljiviji.

```
1  llvm::Value* BoolLiteral::GenerirajKodIzraz()
2  {
3    return llvm::ConstantInt::get(Context().PrevediTip("bool"),
4      (bool)*this);
5  }
```

Isječak 2.27: Metoda `GenerirajKodIzraz` klase `BoolLiteral`

Deklariramo konstantu odgovarajućeg tipa, inicijaliziramo je i vratimo. Operator konverzije u `bool` vrati `true` ako u sadržaju `Leaf`-a piše "true", a `false` ako piše "false".

```
1  llvm::Value* StringLiteral::GenerirajKodIzraz()
2  {
3    llvm::Function* funkcija = Builder().GetInsertBlock()->getParent();
4    llvm::Value* ret = Context().StackAlokacija(funkcija,
5      Context().PrevediTip("string"), "");
6
7    llvm::Value* size = Context().DohvatiClanStrukture(ret, 0);
8    if (size == nullptr)
9      throw SemantickaGreska(Redak(), Stupac(),
10       "Greška pri spremanju duljine literala tipa string");
11
```

```
12    Builder().CreateStore(llvm::ConstantInt::get(*Context().LLVMContext(),
13      llvm::APInt(32, Sadrzaj().size(), true)), size);
14
15    llvm::Value* data = Context().DohvatiClanStrukture(ret, 1);
16    if data == nullptr)
17      throw SemantickaGreska(Redak(), Stupac(),
18        "Greška pri dohvaćanju sadržaja string literala");
19
20    llvm::Constant* _data = Builder().CreateGlobalString(
21      (const char*)(*this), "noviString(" + Sadrzaj() + ")");
22
23    _data = llvm::ConstantExpr::getBitCast(
24      _data, Context().PrevediTip("char*"));
25    Builder().CreateStore(_data, data);
26    ret = Builder().CreateLoad(ret, "popunjenStringLiteral");
27    return ret;
28  }
```

Isječak 2.28: Metoda `GenerirajKodIzraz` klase `StringLiteral`

Budući da smo tip `string` implementirali kao strukturu, moramo prvo alocirati memoriju za cijelu strukturu. Zatim dohvatimo član strukture u kojem treba pisati duljina, upišemo je te dohvatimo član u kojem treba pisati sadržaj literala. Njega konstruiramo pomoću funkcije `Builder::CreateGlobalString`, vodeći računa o tome da ta funkcija vraća polje vrijednosti tipa `char` i da moramo izvršiti konverziju u tip `char*` prije nego ga spremimo. Nakon konverzija tipa pomoću funkcije `getBitCast`, spremimo dobiveni pokazivač u strukturu i vratimo novokonstruirani literal.

2.4.8 Varijable

Deklaracija varijable nije složena; u grubim crtama, trebamo saznati koliko memorije nam treba, alocirati je na stogu ili na hrpi, dati tom bloku memorije ime i nešto napisati u njega.

```
1  llvm::Value* DeklaracijaVarijable::GenerirajKodIzraz()
2  {
3    bool statickaAlokacija = true;
4    std::string tipSlovima =
5      *std::dynamic_pointer_cast<Tip>(m_djeca.front());
6    llvm::Type* tip = m_djeca.front()->GenerirajKodTip();
7    m_djeca.pop_front();
8
9    std::string ime =
10     std::dynamic_pointer_cast<Leaf>(m_djeca.front())->Sadrzaj();
11   m_djeca.pop_front();
12
```

```
13  if (Context().LokalneVarijable().find(ime) !=
14    Context().LokalneVarijable().end())
15    throw SemantickaGreska(Redak(), Stupac(), "Varijabla " +
16      ime + " je već deklarirana");
17  llvm::Function* funkcija = Builder().GetInsertBlock()->getParent();
18  llvm::Value* inicijalnaVrijednost = nullptr;
19  if (!m_djeca.empty())
20  {
21    if (std::dynamic_pointer_cast<Alokacija>(m_djeca.front()) !=
22      nullptr || std::dynamic_pointer_cast<AlokacijaArray>(
23        m_djeca.front()) != nullptr)
24          statickaAlokacija = false;
25
26    inicijalnaVrijednost = m_djeca.front()->GenerirajKodIzraz();
27  }
28  std::string tipDesno =
29    Context().PrevediTip(inicijalnaVrijednost->getType());
30
31  if(tipDesno != tipSlovima)
32    throw SemantickaGreska(Redak(), Stupac(), "Varijabla tipa " +
33      tipSlovima + " ne može biti inicijalizirana vrijednošću tipa "
34        + tipDesno);
35
36  else if (tipSlovima == "int")
37    inicijalnaVrijednost =
38      llvm::ConstantInt::get(Context().PrevediTip("int"), 0, true);
39
40  else if (tipSlovima == "char")
41    inicijalnaVrijednost =
42      llvm::ConstantInt::get(Context().PrevediTip("char"), '\0');
43
44  else if (tipSlovima == "bool")
45    inicijalnaVrijednost =
46      llvm::ConstantInt::get(Context().PrevediTip("bool"), 0);
47
48  else if (tipSlovima == "string")
49  {
50    StringLiteral literal(Token(STRLIT, "", Redak(), Stupac()));
51    inicijalnaVrijednost = literal.GenerirajKodIzraz();
52  }
53  else if (tipSlovima[tipSlovima.size()-1] == '*')
54    inicijalnaVrijednost =
55      llvm::ConstantPointerNull::get(
56        static_cast<llvm::PointerType*>(tip));
57
58  llvm::Value* vrijednost;
59  if (statickaAlokacija)
60  {
61    llvm::AllocaInst* alokacija = Context().StackAlokacija(funkcija,
```

```
62        tip, ime);
63      Builder().CreateStore(inicijalnaVrijednost, alokacija);
64      vrijednost = alokacija;
65    }
66    else
67      vrijednost = inicijalnaVrijednost;
68    return vrijednost;
69  }
```

Isječak 2.29: Metoda `GenerirajKodIzraz` klase `DeklaracijaVarijable`

U finijm crtama, prvo provjerimo je li već deklarirana varijabla s danim imenom te, ako nije, pogledamo je li varijabla inicijalizirana odmah pri deklaraciji. Ako jest, kompajliramo ono što je s desne strane operatora pridruživanja, provjerimo je li ispravnog tipa i spremimo to na hrpu ako se radi o dinamičkoj alokaciji memorije, a inače na stog. Ako varijabla nije odmah inicijalizirana, dodjeljujemo joj odgovarajuću zadanu vrijednost ovisno o njenom tipu. Uočimo da je tip `string` opet drugačiji od ostalih; zbog njegove složenije implementacije prepuštamo konstrukciju praznog stringa klasi `StringLiteral` umjesto da je obavljamo sami.

Da bismo koristili varijablu nakon deklaracije, dovoljno je dohvatiti pokazivač koji pokazuje na njen blok memorije.

```
1  llvm::Value* Varijabla::GenerirajKodIzraz()
2  {
3    llvm::Value* varijabla = Context().TrenutniBlok()->Trazi(m_ime);
4    if (varijabla == nullptr)
5    {
6      throw SemantickaGreska(Redak(), Stupac(), "Varijabla " +
7        m_ime + " nije deklarirana");
8          return nullptr;
9    }
10   return Builder().CreateLoad(varijabla, m_ime);
11 }
```

Isječak 2.30: Metoda `GenerirajKodIzraz` klase `Varijabla`

2.4.9 Dinamička alokacija memorije

Dinamičku alokaciju memorije, odnosno alokaciju na hrpi, obavljamo uz pomoć klasa `Alokacija` i `AlokacijaArray`.

```
1  llvm::Value* Alokacija::GenerirajKodIzraz()
2  {
3    llvm::Type* tip = m_djeca.front()->GenerirajKodTip();
4    llvm::Type* tipInt = Context().PrevediTip("int");
```

```
5   std::string tipString = Context().PrevediTip(tip);
6   m_djeca.pop_front();
7   llvm::Constant* velicinaObjekta = llvm::ConstantInt::get(
8     tipInt, Module()->getDataLayout().getTypeAllocSize(tip));
9   llvm::Instruction* pozivMalloc = llvm::CallInst::CreateMalloc(
10    Builder().GetInsertBlock(), tipInt, tip, velicinaObjekta,
11      nullptr, nullptr, "alokacija");
12  Builder().Insert(pozivMalloc);
13  return pozivMalloc;
14 }
```

Isječak 2.31: Metoda `GenerirajKodTip` klase `Alokacija`

Ovdje je cilj pozvati LLVM IR-ekvivalent poziva `malloc(sizeof(`*tip*`))` iz jezika C. S tim u vidu, dohvatimo tip vrijednosti za koju alociramo memoriju te tip `int` jer je "veličina objekta" cijeli broj. Zatim pomoću funkcije `getTypeAllocSize` saznamo koliko memorije trebamo alocirati. Na kraju, kreiramo poziv `malloc(sizeof(`*tip*`))` i vratimo pokazivač koji nam on vrati.

Da bismo pristupili vrijednosti na koju pokazivač pokazuje, koristimo operator dereferenciranja, koji navodimo u sljedećem isječku:

```
1  llvm::Value* Dereferenciranje::GenerirajKodIzraz()
2  {
3    llvm::Value* pointerKaoBroj = m_djeca.front()->GenerirajKodIzraz();
4    llvm::Type* tipPointera = Context().PrevediTip(
5      Context().PrevediTip(pointerKaoBroj->getType()) + "*");
6
7    llvm::Value* pointer = Builder().CreateIntToPtr(pointerKaoBroj,
8      tipPointera);
9
10   return Builder().CreateLoad(pointer, tipPointera);
11 }
```

Isječak 2.32: Metoda `GenerirajKodIzraz` klase `Dereferenciranje`

U svakoj varijabli pokazivačkog tipa piše neki broj koji predstavlja lokaciju u memoriji. Pomoću funkcije `CreateIntToPtr`, taj broj pretvorimo u LLVM IR pokazivač te učitamo i vratimo što piše na lokaciji na koju pokazuje.

Alokacija memorije za polja nešto je složenija jer smo složenije implementirali polja u odjeljku 2.4.4. Iako cijela metoda izgleda zastrašujuće, u njoj koristimo samo ideje koje smo već savladali u prethodnim dijelovima.

```
1  llvm::Value* AlokacijaArray::GenerirajKodIzraz()
2  {
3    llvm::Value* ret;
4    llvm::Value* dereferenciraniRet;
5    llvm::Type* tip = m_djeca.front()->GenerirajKodTip();
6    llvm::Type* tipInt = Context().PrevediTip("int");
```

```
7
8   std::string tipSlovima =
9     *std::dynamic_pointer_cast<Tip>(m_djeca.front());
10  m_djeca.pop_front();
11  std::shared_ptr<Leaf> alokacijaTipSlovima =
12    std::make_shared<Leaf>(Token(PRAZNO, tipSlovima + "[]",
13      Redak(), Stupac()));
14
15  std::list<std::shared_ptr<AST>> alokacijaTipDjeca;
16  alokacijaTipDjeca.push_back(std::move(alokacijaTipSlovima));
17  std::shared_ptr<Tip> alokacijaTip = std::make_shared<Tip>(
18    Redak(), Stupac(), std::move(alokacijaTipDjeca));
19
20  std::list<std::shared_ptr<AST>> alokacijaDjeca;
21  alokacijaDjeca.push_back(std::move(alokacijaTip));
22  Alokacija alokacija(Redak(), Stupac(), std::move(alokacijaDjeca));
23
24  ret = alokacija.GenerirajKodIzraz();
25  dereferenciraniRet = Builder().CreateLoad(
26    ret, "dereferencirajStrukturu");
27
28  llvm::Value* size = Context().DohvatiClanStrukture(ret, 0);
29  if (size == nullptr)
30    throw SemantickaGreska(Redak(), Stupac(),
31      "Greška pri spremanju duljine novog polja");
32
33  llvm::Constant* kolicinaObjekata =
34    static_cast<llvm::Constant*>(m_djeca.front()->GenerirajKodIzraz());
35
36  Builder().CreateStore(kolicinaObjekata, size);
37  llvm::Value* elements = Context().DohvatiClanStrukture(ret, 1);
38  if (elements == nullptr)
39    throw SemantickaGreska(Redak(), Stupac(),
40      "Greška pri spremanju sadržaja novog polja");
41
42  llvm::Constant* velicinaObjekta = llvm::ConstantInt::get(
43    tipInt, Module()->getDataLayout().getTypeAllocSize(tip));
44
45  llvm::Constant* velicinaPolja =
46    llvm::ConstantExpr::getMul(velicinaObjekta, kolicinaObjekata);
47
48  velicinaPolja =
49    llvm::ConstantExpr::getTruncOrBitCast(velicinaPolja, tipInt);
50  llvm::Instruction* pozivMalloc = llvm::CallInst::CreateMalloc(
51    Builder().GetInsertBlock(), tipInt, tip, velicinaPolja, nullptr,
52      nullptr, "");
53
54  Builder().Insert(pozivMalloc);
55  Builder().CreateStore(pozivMalloc, elements);
```

```
56    return ret;
57  }
```

Isječak 2.33: Metoda `GenerirajKodIzraz` klase `AlokacijaArray`

Ovdje želimo dvaput alocirati memoriju: jednom za `struct{int, `*tip*`*}` i jednom za *tip*`[`*duljina*`]`. Budući da sad imamo klasu za alokaciju memorije za jedan objekt proizvoljnog tipa, u prvom dijelu ove metode pripremamo podatke za poziv `Alokacija::GenerirajKodIzraz()`. Zatim pomoću klase `Alokacija` alociramo memoriju za strukturu te dereferenciramo povratnu vrijednost koju dobijemo. Moramo dereferencirati dobivenu vrijednost jer, u LLVM IR, kao i u jeziku C, funkcija `malloc` vraća pokazivač na alociranu memoriju.

Kad se domognemo strukture, dohvatimo pokazivač na njen prvi član (onaj s indeksom 0) i spremimo duljinu novog polja u njega. Zatim dohvatimo pokazivač na drugi član strukture i pripremimo podatke za poziv LLVM IR-ekvivalenta poziva `malloc(`*duljina*`*sizeof(`*tip*`))` iz jezika C. Na isti način kao ranije, saznamo koliko nam memorije treba za jedan objekt danog tipa te pomnožimo taj broj sa zadanom duljinom polja pomoću funkcije `getMul`. Dobiveni rezultat pretvorimo iz tipa `const int` u `int`, pozovemo `CreateMalloc`, spremimo pokazivač na alocirani blok memorije u drugi član strukture te vratimo pokazivač na strukturu. Naravno, ne obavljamo množenje ni navedenu pretvorbu tipa za vrijeme kompajliranja, nego zapravo umećemo LLVM IR instrukcije koje će taj posao obaviti za vrijeme izvršavanja kompajliranog programa.

Kao što smo rekli u odjeljku 1.1.5, elemente polja dohvaćamo pomoću operatora `[]`, ali još nismo spremni za njegovu implementaciju, nego ćemo je proučiti u odjeljku 2.4.13, nakon što saznamo nešto o grananjima, iznimkama i operatorima usporedbe.

2.4.10 Grananja

Za razliku od jezika C, u LLVM IR blokovi ne čine stablastu strukturu, nego linearnu. To i činjenica da u LLVM možemo granati samo na početak bloka čini grananja nešto kompliciranijima nego što bismo očekivali. Naime, generirani kôd u LLVM IR neće biti oblika:

```
void funkcija()
{
  // posao prije if
  if(uvjet){ /* posao ako je uvjet==true */ }
  else { /* posao ako je uvjet==false*/ }
  // posao poslije if
},
```

nego će više biti nalik na sljedeće:

```
void funkcija()
{
  // posao prije if
  goto uvjet ? then : else;
}
{
  then: // posao ako je uvjet == true
  goto nastavak;
}
{
  else: // posao ako je uvjet == false
  goto nastavak;
}
{
  nastavak: // posao poslije if
}
```

S tim u vidu, proučimo metodu `GenerirajKodIzraz` klase `IfElse`.

```
1  llvm::Value* IfElse::GenerirajKodIzraz()
2  {
3    llvm::Value* uvjet = m_djeca.front()->GenerirajKodIzraz();
4    m_djeca.pop_front();
5    ProvjeriVrijednost(uvjet);
6    ProvjeriTip(uvjet, Context().PrevediTip("bool"));
7
8    llvm::Function* funkcija = Builder().GetInsertBlock()->getParent();
9    llvm::BasicBlock* ifTrueBlok = llvm::BasicBlock::Create(
10     *Context().LLVMContext(), "ifTrue", funkcija);
11
12   Context().DodajBlok(ifTrueBlok);
13   llvm::BasicBlock* elseBlok = llvm::BasicBlock::Create(
14     *Context().LLVMContext(), "else");
15   llvm::BasicBlock* nastavakBlok = llvm::BasicBlock::Create(
16     *Context().LLVMContext(), "nastavak");
17
18   Builder().CreateCondBr(uvjet, ifTrueBlok, elseBlok);
19   Builder().SetInsertPoint(ifTrueBlok);
20
21   llvm::Value *ifTrueVrijednost =
22     m_djeca.front()->GenerirajKodIzraz();
23   ProvjeriVrijednost(ifTrueVrijednost);
24
25   m_djeca.pop_front();
26   bool imamElse = !m_djeca.empty();
27   Context().ZatvoriBlok();
28
29   Builder().CreateBr(nastavakBlok);
30   ifTrueBlok = Builder().GetInsertBlock();
```

```
31
32    funkcija->getBasicBlockList().push_back(elseBlok);
33    Context().DodajBlok(elseBlok);
34    Builder().SetInsertPoint(elseBlok);
35    llvm::Value* elseVrijednost = nullptr;
36    if (imamElse)
37    {
38      elseVrijednost = m_djeca.front()->GenerirajKodIzraz();
39      ProvjeriVrijednost(elseVrijednost);
40    }
41
42    Builder().CreateBr(nastavakBlok);
43    elseBlok = Builder().GetInsertBlock();
44    Context().ZatvoriBlok();
45    funkcija->getBasicBlockList().push_back(nastavakBlok);
46    Builder().SetInsertPoint(nastavakBlok);
47
48    return llvm::ConstantInt::get(*Context().LLVMContext(),
49      llvm::APInt(1, 0));
50  }
```

Isječak 2.34: Metoda `GenerirajKodIzraz` klase `IfElse`

Objasnimo prvo čemu služe provjere na početku metode.

```
1  void AST::ProvjeraVrijednosti(llvm::Value* prvaVrijednost,
2    llvm::Value* drugaVrijednost, std::string const& imeOperatora = "")
3  {
4    if(prvaVrijednost == nullptr)
5      throw SemantickaGreska(Redak(), Stupac(), "Lijevi operand
6        operatora \"" + imeOperatora "\" je neispravna vrijednost");
7
8    if(drugaVrijednost == nullptr)
9      throw SemantickaGreska(Redak(), Stupac(), "Desni operand
10       operatora \"" + imeOperatora "\" je neispravna vrijednost");
11 }
```

Isječak 2.35: Metoda `ProvjeraVrijednosti` klase `AST`

Svaki poziv funkcije iz prostora imena `llvm` koji vraća pokazivač na neku vrijednost (odnosno vraća `llvm::Value*`) može vratiti `nullptr` — tada tu vrijednost ne možemo koristiti ni za što pa kažemo da je neispravna i prijavljujemo grešku. Ova provjera je nužna na svakom mjestu gdje postoji rizik od pojavljivanja neispravnih vrijednosti jer inače otvaramo vrata nedefiniranom ponašanju.

```
1  void AST::ProvjeraTipa(llvm::Value* vrijednost, llvm::Type*
2    ocekivaniTip)
3  {
4    if(vrijednost->getType() != ocekivaniTip)
5      throw SemantickaGreska(Redak(), Stupac(), "Pogrešan tip!
```

```
6       Očekujem " + Context().PrevediTip(ocekivaniTip));
7  }
```

Isječak 2.36: Metoda `ProvjeraTipa` klase `AST`

Druga provjera vodi računa o tome da vrijednost s kojom radimo ima tip s kojim znamo raditi — u slučaju isječka 2.34, je li `uvjet` tipa `bool`.

Kad smo gotovi s provjerama, prevodimo izraz koji je proslijeđen kao uvjet grananja i, ako to nije neispravna vrijednost, kreiramo blokove `ifTrueBlok`, `elseBlok` i `nastavakBlok`. Nakon toga dodajemo sâmo grananje pozivom funkcije `CreateCondBr` (***Cond**itional **Br**anch*) te se pomoću funkcije `SetInsertPoint` dajemo do znanja LLVM-u da od tog trenutka želimo generirane instrukcije pisati u `ifTrueBlok`. Zatim popunimo `ifTrueBlok` i bezuvjetno granamo u `nastavakBlok` te se prebacimo u njega pomoću funkcije `SetInsertPoint`.

Ako imamo granu `else`, popunimo blok `elseBlok` odgovarajućim naredbama i dodamo bezuvjetno grananje u `nastavakBlok`. Na kraju vratimo LLVM IR konstantu 0 jer nešto moramo vratiti (i ne želimo da je to `nullptr` jer nije došlo do greške).

2.4.11 Naredba `assert` i dizanje iznimke

U ovoj verziji kompajlera za jezik C0 nije implementirano rukovanje iznimkama ni naredba `assert`.

2.4.12 Operatori

U ovom dijelu ćemo obraditi sve unarne i binarne operatore. Iako su razdvojeni u nekoliko klasa, svi se vrlo slično prevode u LLVM IR pa ćemo proučiti funkciju `GenerirajKodIzraz` samo za operatore usporedbe. Također, kako smo rekli u odjeljku 1.1.1, `int` je jedini numerički tip podatka u jeziku C0 pa se bavimo samo operatorima usporedbe cijelih brojeva.

```
1  llvm::Value* OperatorUsporedbe::GenerirajKodIzraz()
2  {
3    llvm::Value* lijevo = m_djeca.front()->GenerirajKodIzraz();
4    m_djeca.pop_front();
5      llvm::Value* desno = m_djeca.front()->GenerirajKodIzraz();
6    m_djeca.pop_front();
7    Leaf const& operacija =
8      *std::dynamic_pointer_cast<Leaf>(m_djeca.front());
9    m_djeca.pop_front();
10   ProvjeraVrijednosti(lijevo, desno, operacija.Sadrzaj());
11   ProvjeraTipa(lijevo, Context().PrevediTip("int"));
12   ProvjeraTipa(desno, Context().PrevediTip("int"));
```

```
13
14    switch (operacija.TipTokena())
15    {
16      case LESS:
17        return Builder().CreateICmpSLT(lijevo, desno,
18          "usporedbaManje");
19
20      case LESSEQ:
21        return Builder().CreateICmpSLE(lijevo, desno,
22          "usporedbaManjeJednako");
23
24      case GRT:
25        return Builder().CreateICmpSGT(lijevo, desno,
26          "usporedbaVece");
27
28      case GRTEQ:
29        return Builder().CreateICmpSGE(lijevo, desno,
30          "usporedbaVeceJednako");
31    }
32  }
```

Isječak 2.37: Metoda `GenerirajKodIzraz` klase `OperatorUsporedbe`

Nakon što se domognemo oba operanda i izvršimo potrebne provjere, već smo obavili većinu posla. Preostaje nam samo pročitati koji operator imamo te pozvati odgovarajuću funkciju za njegovo prevođenje u LLVM IR. Obratimo pažnju na posljednji argument tih funkcija — smijemo ga izostaviti, a govori LLVM-u kako da nazove privremene varijable za rezultate poziva navedenih operatora pri generiranju LLVM IR. Ako ga ne zadamo, zvat će se `%0`, `%1`, itd., kao što smo opisali u točki 2.3. U sljedećoj tablici ćemo pojasniti čudna imena funkcija koje pozivamo i kako izgledaju odgovarajuće funkcije za druge operatore.

operacija	operator	funkcija	objašnjenje
$<$	<	`CreateICmpSLT`	**I**nteger **Comp**are **S**igned **L**ess **T**han
$\leq$	<=	`CreateICmpSLE`	**I**nteger **Comp**are **S**igned **L**ess or **E**qual
$>$	>	`CreateICmpSGT`	**I**nteger **Comp**are **S**igned **G**reater **T**han
$\geq$	>=	`CreateICmpSGE`	**I**nteger **Comp**are **S**igned **G**reater or **E**qual
$=$	==	`CreateICmpEQ`	**I**nteger **Comp**are **Eq**ual
$\neq$	!=	`CreateICmpNE`	**I**nteger **Comp**are **N**ot **Eq**ual

unarni −	`-`	`CreateNeg`	**Neg**ation
¬	`!`	`CreateNot`	**Not**
+	`+`	`CreateAdd`	**Add**ition
binarni −	`-`	`CreateSub`	**Sub**traction
·	`*`	`CreateMul`	**Mul**tiplication
÷	`/`	`CreateSDiv`	**S**igned **Div**ision
modulo	`%`	`CreateSRem`	**S**igned **Rem**ainder
i	`&`, `&&`	`CreateAnd`	**And**
ili	`|`, `||`	`CreateOr`	**Or**
isključivo ili	`~`	`CreateXor`	Exclusive **Or**
pomak ulijevo	`<<`	`CreateShl`	**Sh**ift **L**eft
pomak udesno	`>>`	`CreateAShr`	**A**rithmetic **Sh**ift **R**ight

Tablica 2.1: Funkcije za prevođenje operatora iz jezika C0 u LLVM IR

Sva imena funkcija počinju s „`Create`" — to se odnosi na kreiranje instrukcije jezika LLVM IR koja izvršava danu operaciju. Naravno, ovo nisu sve funkcije za prevođenje operatora koje LLVM podržava, ali preostale su vrlo slične navedenima pa ih nećemo sve posebno navoditi. Umjesto toga, u sljedećoj tablici ćemo ilustrirati sličnosti i razlike preostalih funkcija pomoću par primjera.

funkcija	varijanta	razlika
`CreateICmpEQ`	`CreateFCmpEQ`	uspoređuje decimalne brojeve (**F**loat)
`CreateSDiv`	`CreateUDiv`	dijeli cijele brojeve kao da su nenegativni (**U**nsigned)[6]
`CreateSDiv`	`CreateExactSDiv`	dijeli cijele brojeve, ali se koristi samo kad znamo da neće biti ostatka
`CreateAShr`	`CreateLShr`	radi logički pomak udesno[7]

Tablica 2.2: Neke varijante funkcija za prevođenje operatora u LLVM IR

Neki operatori moraju obaviti još nešto prije ili poslije poziva svoje funkcije za prevođenje u LLVM IR. Prisjetimo se, u odjeljku 1.1.1 smo rekli da operatori pomaka moraju dići iznimku ako im drugi argument nije u $[0, 31]$. Na primjer, za pomak ulijevo to radimo ovako:

[6] Ne odbacuje predznak, nego čita cijeli broj kao *unsigned*, odnosno negativne brojeve neće čitati kao njima suprotne, nego kao velike pozitivne brojeve.

[7] Logički pomak udesno upisuje 0 na novi najznačajniji bit, a aritmetički upisuje bit predznaka.

```
// ...provjera ispravnosti i tipova...
case LSHIFT:
{
  llvm::Value* nula = llvm::ConstantInt::get(
    *Context().LLVMContext(), llvm::ApInt(32, 0, true));
  llvm::Value* tridesetDva = llvm::ConstantInt::get(
    *Context().LLVMContext(), llvm::APInt(32, 32, true));
  llvm::Value* desniVeciOd0 = Builder().CreateICmpSGT(desno,
    nula);
  llvm::Value* desniManjiOd32 = Builder().CreateICmpSLT(desno,
    tridesetDva);
  ProvjeraUvjeta(desniVeciOd0,
    "Drugi operand lijevog pomaka mora biti veći od 0");
  ProvjeraUvjeta(desniManjiOd32,
    "Drugi operand lijevog pomaka mora biti manji od 32");
  return Builder().CreateShl(lijevo, desno, "lijeviPomak");
}
// ...ostali slučajevi...
```

Kod operatora pridruživanja imamo malo više posla — ne možemo samo vratiti dobivenu vrijednost, nego je prvo moramo spremiti na memorijsku lokaciju lijevog operanda.

```
1  llvm::Value* OperatorPridruzivanja::GenerirajKodIzraz()
2  {
3    type_info const& tipLijevo = typeid(*m_djeca.front());
4    if (tipLijevo != typeid(Varijabla) &&
5      tipLijevo != typeid(Dereferenciranje) &&
6      tipLijevo != typeid(UglateZagrade) &&
7      tipLijevo != typeid(Strelica) &&
8      tipLijevo != typeid(Tocka))
9    {
10     throw SemantickaGreska(Redak(), Stupac(), "Prvi operand
11       operatora pridruživanja mora biti lijeva vrijednost");
12   }
13
14   std::shared_ptr<AST> lijevaStrana = m_djeca.front();
15   m_djeca.pop_front();
16   llvm::Value* desno = m_djeca.front()->GenerirajKodIzraz();
17   m_djeca.pop_front();
18   llvm::Value* lijevo;
19   llvm::Value* adresaLijevog;
20   if (tipLijevo == typeid(Varijabla))
21   {
22     lijevo = lijevaStrana->GenerirajKodIzraz();
23     std::string lijevoIme = Context().TrenutniBlok()->Trazi(lijevo);
24     adresaLijevog = Context().TrenutniBlok()->Trazi(lijevoIme);
25   }
26   else
```

```
27   {
28     adresaLijevog = lijevaStrana->GenerirajKodIzraz();
29     lijevo = Builder().CreateLoad(adresaLijevog,
30       "dereferenciranjeAdrese");
31   }
32   ProvjeraVrijednosti(lijevo);
33   ProvjeraVrijednosti(desno);
34   ProvjeraTipa(desno, lijevo->getType());
35   llvm::Value* novoLijevo;
36   Leaf const& operacija =
37     *std::dynamic_pointer_cast<Leaf>(m_djeca.front());
38   switch (operacija.TipTokena())
39   {
40     // ...poziv odgovarajuće funkcije za prevođenje u
41     // LLVM IR ovisno o operatoru...
42   }
43   Builder().CreateStore(novoLijevo, adresaLijevog);
44   return novoLijevo;
45 }
```

Isječak 2.38: Metoda `GenerirajKodIzraz` klase `OperatorPridruzivanja`

Ostala su nam još dva operatora: operator dohvata, koji je tema sljedećeg odjeljka, i ternarni uvjetni operator, za kojeg još nismo spremni. Naime, LLVM nema jednostavno rješenje za uvjetni operator kao što ima za ostale operatore pa ga sami moramo implementirati u LLVM IR. Da bismo to mogli, moramo znati definirati funkcije pa ostavljamo uvjetni operator za odjeljak 2.4.15.

2.4.13 Operator dohvata — []

Sad možemo vratiti dug iz dijela 2.4.9. Naime, operator dohvata elementa iz polja mora dići iznimku ako mu proslijedimo indeks koji nije u intervalu $[0, duljina\_polja - 1]$, što ranije nismo znali prevesti u LLVM IR.

```
1  llvm::Value* UglateZagrade::GenerirajKodIzraz()
2  {
3    llvm::Value* polje = m_djeca.front()->GenerirajKodIzraz();
4    ProvjeraVrijednosti(polje);
5    ProvjeraPolja(polje);
6    m_djeca.pop_front();
7    llvm::Value* indeksPolja = m_djeca.front()->GenerirajKodIzraz();
8    m_djeca.pop_front();
9
10   ProvjeraTipa(indeksPolja, Context().PrevediTip("int"));
11   llvm::Value* nula = llvm::ConstantInt::get(
12     *Context().LLVMContext().get(), llvm::APInt(32, 0, true));
13   llvm::Value* veceOdNule = Builder().CreateICmpSLT(
```

```
14      nula, indeksPolja, "usporedbaManje");
15    ProvjeraUvjeta(veceOdNule, "Indeks polja mora biti veći od 0!");
16    llvm::Value* dohvacenaDuljina =
17      Context().DohvatiClanStrukture(polje, 0);
18    ProvjeraVrijednosti(dohvacenaDuljina);
19
20    llvm::Value* manjeOdDuljine = Builder().CreateICmpSLT(indeksPolja,
21      dohvacenaDuljina);
22    ProvjeraUvjeta(manjeOdDuljine, "Indeks polja mora biti manji
23      od maksimalne duljine polja!");
24
25    llvm::Value* dohvacenoPolje =
26      Context().DohvatiClanStrukture(polje, 1);
27    ProvjeraVrijednosti(dohvacenoPolje);
28
29    std::vector<llvm::Value*> indeksElementa;
30    indeksElementa.push_back(indeksPolja);
31    llvm::Value* dohvaceniElement = Builder().CreateGEP(
32      dohvacenoPolje, indeksElementa, "pointerNaElement");
33    dohvaceniElement = Builder().CreateLoad(dohvaceniElement,
34      "elementPolja");
35    return dohvaceniElement;
36  }
```

Isječak 2.39: Metoda `GenerirajKodIzraz` klase `UglateZagrade`

Ovaj isječak služi kao ponavljanje svega što smo dosad vidjeli — prvo učitamo vrijednosti s kojima radimo i obavimo potrebne provjere, zatim provjerimo je li indeks elementa u intervalu koji nam odgovara, dignemo iznimku ako nije, te na kraju dohvatimo pokazivač koji pokazuje na traženi element.

Jedina novost ovdje je funkcija `ProvjeraPolja`, koja provjerava je li dana vrijednost tipa `string` ili *tip*`[]`. Postupak je isti kao u funkciji `ProvjeraTipa` u isječku 2.36 pa je nećemo posebno ispisivati.

2.4.14 Petlje i kontrola toka

Petlje implementiramo kao poseban slučaj grananja pa ćemo, umjesto analize cijelog kôda, samo riječima opisati slijed blokova koji čini petlju. To ćemo napraviti samo za petlju `for` jer kod nje ima više posla te će nakon toga biti očito kako se isto radi za petlju `while`. Puna implementacija petlji `while` i `for` se može naći na [1] ili prilagoditi iz isječka 2.34. Ideja je da imamo sljedeće blokove:

1. Blok koji sadrži samo inicijalizaciju brojača i grananje u sljedeći blok,
2. Blok koji sadrži provjeru uvjeta te se ovisno o uvjetu grana u sljedeći ili posljednji blok,

3. Blok koji sadrži tijelo petlje i na kraju bezuvjetno odlazi na sljedeći blok,

4. Blok koji sadrži inkrement brojača i bezuvjetno odlazi na blok 2,

5. Blok poslije petlje u koji granamo nakon što petlja završi s radom.

Primijetimo da umjesto bloka 3 možemo imati slijed blokova između 2 i 4 ako program ima ugniježđene petlje ili grananja. Za kontrolu toka imamo naredbe `break` i `continue`. Naredba `continue` radi bezuvjetno grananje u blok 4 u slučaju petlje `for`, odnosno u blok 2 u slučaju petlje `while`, a naredba `break` radi bezuvjetno grananje u blok 5.

2.4.15 Funkcije

Slično kao strukture, funkcije možemo deklarirati bez definicije pa razlikujemo klase `DeklaracijaFunkcije` i `DefinicijaFunkcije`. Budući da funkcije moramo moći pozivati, imamo i posebnu klasu `PozivFunkcije`.

```
1  llvm::Function* DeklaracijaFunkcije::GenerirajKodFunkcija()
2  {
3    llvm::Type* povratniTip = m_djeca.front()->GenerirajKodTip();
4    m_djeca.pop_front();
5
6    std::string ime = std::dynamic_pointer_cast<Leaf>(
7    *m_djeca.begin())->Sadrzaj();
8    m_djeca.pop_front();
9
10   ASTList& argumenti = *std::dynamic_pointer_cast<ASTList>(
11     m_djeca.front());
12   std::vector<llvm::Type*> tipoviArgumenata(argumenti.size() / 2);
13   std::vector<std::string> imenaArgumenata(argumenti.size() / 2);
14   for (int i = 0; !argumenti.empty(); ++i)
15   {
16     tipoviArgumenata[i] = ((*argumenti.begin())->GenerirajKodTip());
17     argumenti.pop_front();
18     imenaArgumenata[i] = (std::dynamic_pointer_cast<Leaf>(
19       (*argumenti.begin()))->Sadrzaj());
20     argumenti.pop_front();
21   }
22   m_djeca.pop_front();
23   llvm::FunctionType* tipFunkcije = llvm::FunctionType::get(
24     povratniTip, tipoviArgumenata, false);
25   llvm::Function* funkcija = llvm::Function::Create(
26     tipFunkcije, llvm::Function::ExternalLinkage, ime, Module().get());
27
28   int i = 0;
29   for (llvm::Function::arg_iterator it = funkcija->arg_begin();
```

```
30      it != funkcija->arg_end(); ++it)
31      it->setName(imenaArgumenata[i++]);
32
33    return funkcija;
34  }
```

Isječak 2.40: Metoda `GeneirajKodFunkcija` klase `DeklaracijaFunkcije`

Da bismo imali ispravnu deklaraciju funkcije, moramo znati njen povratni tip, njeno ime, te broj i tipove njenih argumenata. Njih proslijedimo funkciji `llvm::FunctionType::get`, koja nam vrati tip funkcije. Tip funkcije je definiran kao *povratni_tip*(*tipovi_argumenata*), na primjer, tip funkcije koja prima `char` i `int`, a vraća `bool` je `bool(char, int)`. Nakon što imamo tip funkcije, njenim argumentima dodijelimo ispravna imena. U nastavku slijedi postupak prevođenja definicije funkcije.

```
1  llvm::Function* DefinicijaFunkcije::GenerirajKodFunkcija()
2  {
3    ASTList::iterator it = m_djeca.begin();
4    ++it;
5    std::string ime = std::dynamic_pointer_cast<Leaf>
6    (*it)->Sadrzaj();
7    llvm::Function* funkcija = Module()->getFunction(ime);
8    if (funkcija == nullptr)
9      funkcija = DeklaracijaFunkcije::GenerirajKodFunkcija();
10   else if (!funkcija->empty())
11     throw SemantickaGreska(Redak(), Stupac(),
12       "Funkcija " + ime + " je već definirana");
13
14   else if (funkcija->getReturnType() !=
15     (*m_djeca.begin())->GenerirajKodTip())
16       throw SemantickaGreska(Redak(), Stupac(),
17         "Funkcija '" + ime + "' nema isti povratni tip kao
18         u deklaraciji");
19   else
20   {
21     m_djeca.pop_front();
22     m_djeca.pop_front();
23     ASTList& argumenti = *std::dynamic_pointer_cast<ASTList>
24     (m_djeca.front());
25     std::vector<llvm::Type*> tipoviArgumenata;
26     std::vector<std::string> imenaArgumenata;
27     int maxSize = std::dynamic_pointer_cast<ASTList>
28       (m_djeca.front())->size() / 2;
29     tipoviArgumenata.reserve(maxSize);
30     while (!argumenti.empty())
31     {
32       tipoviArgumenata.push_back(
```

```
33        (*argumenti.begin())->GenerirajKodTip());
34      argumenti.pop_front();
35      imenaArgumenata.push_back(std::dynamic_pointer_cast<Leaf>(
36        *argumenti.begin())->Sadrzaj());
37      argumenti.pop_front();
38    }
39    int i = 0;
40    for (llvm::Function::arg_iterator it = funkcija->arg_begin();
41      it != funkcija->arg_end(); ++it)
42    {
43      if (it->getType() != tipoviArgumenata[i])
44        throw SemantickaGreska(Redak(), Stupac(), "Funkcija '" + ime
45        + "' nema iste argumente kao u deklaraciji");
46
47        it->setName(imenaArgumenata[i]);
48      ++i;
49    }
50    m_djeca.pop_front();
51  }
52
53  llvm::BasicBlock* blok = llvm::BasicBlock::Create(
54    *Context().LLVMContext(), "pocetak funkcije", funkcija);
55  Context().DodajBlok(blok);
56  Builder().SetInsertPoint(blok);
57
58  for (llvm::Function::arg_iterator it = funkcija->arg_begin();
59    it != funkcija->arg_end(); ++it)
60  {
61    llvm::AllocaInst* alokacija = Context().StackAlokacija(
62      funkcija, it->getType(), it->getName());
63    Builder().CreateStore(&(*it), alokacija);
64    Context().TrenutniBlok()->Dodaj(alokacija, it->getName());
65  }
66  try
67  {
68    llvm::Value* povratnaVrijednost =
69      m_djeca.front()->GenerirajKodIzraz();
70    if (povratnaVrijednost != nullptr)
71    {
72      std::string error;
73      llvm::raw_string_ostream errorStream(error);
74      llvm::verifyFunction(*funkcija, &errorStream);
75
76      Context().ZatvoriBlok();
77      return funkcija;
78    }
79    else
80    {
81      Context().ZatvoriBlok();
```

```
82        funkcija->eraseFromParent();
83        return nullptr;
84      }
85    }
86    catch (SemantickaGreska const& e)
87    {
88      funkcija->eraseFromParent();
89      throw;
90    }
91  }
```

Isječak 2.41: Metoda `GeneirajKodFunkcija` klase `DefinicijaFunkcije`

Izgleda kao mnogo posla, ali većina tog posla se svodi na provjere da bismo bili sigurni da smijemo definirati funkciju s danim imenom. Nakon što ih sve obavimo, skoro smo gotovi.

Prvo provjerimo je li istoimena funkcija već deklarirana. Ako jest, provjerimo je li i definirana te u tom slučaju dižemo iznimku — funkcije mogu biti definirane samo jednom. Ako nije definirana, provjerimo slaže li se signatura dane funkcije s onom koja je ranije deklarirana. Ako ne, opet dižemo iznimku.

Na kraju, ako funkcija s danim imenom uopće nije deklarirana, deklariramo je sad i dodijelimo joj blok. Nakon što smo osigurali da pred sobom imamo deklariranu funkciju, u njen doseg stavimo parametre koje je primila te u njen blok upišemo sve naredbe koje treba sadržavati. Ako je sve prošlo bez greške, spremimo novu definiranu funkciju, inače je brišemo.

Pogledajmo još kako u LLVM prevodimo poziv funkcije.

```
1  llvm::Value* PozivFunkcije::GenerirajKodIzraz()
2  {
3    std::string ime =
4    std::dynamic_pointer_cast<Leaf>(m_djeca.front())->Sadrzaj();
5    m_djeca.pop_front();
6
7    llvm::Function* funkcija = Context().Module()->getFunction(ime);
8    if (funkcija == nullptr)
9      throw SemantickaGreska(Redak(), Stupac(), "Ne postoji funkcija
10     s imenom " + ime + "");
11   ASTList& dobiveniArgumenti =
12     *std::dynamic_pointer_cast<ASTList>(m_djeca.front());
13
14   if (dobiveniArgumenti.size() != funkcija->arg_size())
15   {
16     std::stringstream poruka;
17     poruka << "Funkcija " << ime << " prima " << funkcija->arg_size()
18       << " argumenata (poslano " << dobiveniArgumenti.size() << ")";
19     throw SemantickaGreska(Redak(), Stupac(), poruka.str());
20   }
```

```
21
22    std::vector<llvm::Value*> argumenti;
23    for (ASTList::iterator it = dobiveniArgumenti.begin();
24      it != dobiveniArgumenti.end(); ++it)
25      argumenti.push_back((*it)->GenerirajKodIzraz());
26
27    int i = 0;
28    for (llvm::Function::arg_iterator it = funkcija->arg_begin();
29      it != funkcija->arg_end(); ++it)
30    {
31      if (it->getType() != argumenti[i++]->getType())
32      {
33        std::stringstream poruka;
34        poruka << i << ". argument u pozivu funkcije " << ime <<
35        " ne odgovara definiciji. "<< "Očekujem '" <<
36          Context().PrevediTip(it->getType()) << "', dobio ' "
37          << Context().PrevediTip(argumenti[--i]->getType()) << "'";
38        throw SemantickaGreska(Redak(), Stupac(), poruka.str());
39      }
40    }
41    llvm::CallInst* poziv = llvm::CallInst::Create(funkcija,
42      argumenti, "", Context().TrenutniBlok()->Sadrzaj());
43    return poziv;
44  }
```

Isječak 2.42: Metoda `GeneirajKodIzraz` klase `PozivFunckije`

Sada znamo deklarirati, definirati i pozivati funkcije u LLVM pa možemo vratiti dug iz odjeljka 2.4.12 — ternarni uvjetni operator. Naime, za razliku od ostalih logičkih operatora, LLVM nema spremno rješenje za uvjetni operator pa ga moramo napisati sami. Još gore, ne možemo unaprijed znati sve moguće tipove operanada jer korisnik uvijek može definirati nove.

Rješenje je da pri kompajliranju definiramo uvjetne operatore po potrebi. To radimo tako da imamo statički brojač koji nam govori koji je trenutni uvjetni operator po redu te definiramo funkciju oblika

```
T operator?:brojac(bool uvjet, T prvi, T drugi)
{
  if(uvjet)
    return prvi;
  else
    return drugi;
},
```

gdje je T tip operanada, a *brojac* vrijednost statičkog brojača u tom trenutku. Istaknimo da jezik C0 ne dozovljava korištenje znakova '?' i ':' u identifikatorima pa možemo biti sigurni da korisnik nikad neće deklarirati funkciju istog imena i izazvati konflikt. Nakon što definiramo funkciju, na mjesto poziva operatora

stavimo njen poziv.

2.5 Prevođenje u strojni kôd

Sada gotovo svaki element jezika C0 znamo prevesti u LLVM IR pa preostaje još samo prevesti generirani LLVM IR u strojni kôd. Da bismo to napravili, najprije moramo spremiti LLVM IR u datoteku.

```
std::error_code errorCode;

llvm::raw_fd_ostream llvmBC("llvmBC.bc", errorCode);
llvm::raw_fd_ostream llvmIR("llvmIR.ll", errorCode);
// ...
std::shared_ptr<C0Compiler::Program> korijenAST =
    std::make_shared<C0Compiler::Program>();
llvmIR << *korijenAST->Module().get();
llvm::WriteBitcodeToFile(*korijenAST->Module().get(), llvmBC);
llvmIR.flush();
```

Isječak 2.43: Dio funkcije `main` — spremanje LLVM IR u datoteku

Nakon izvršavanja tog dijela kôda će LLVM IR u čitljivom obliku biti spremljen u datoteku „`llvmIR.ll`", a u binarnom obliku u datoteku „`llvmBC.bc`". Time je posao našeg kompajlera završen; sada pomoću alata `llc` i `lld` infrastrukture LLVM možemo iz spremljenog LLVM IR generirati izvršivi program.

Alat `llc` uzima sadržaj datoteke „`llvmBC.bc`" te ga prevodi u strojni jezik. Ako mu ne zadamo arhitekturu, onda ga prevodi u strojni jezik računala na kojem je pokrenut. Više o `llc` i opcijama koje podržava se može naći na [5].

Alat `lld` ima ulogu *linker*a — prima datoteke koje sadrže strojni kôd, objedinjuje ih te od njih generira izvršivi program. Više o `lld` i opcijama koje podržava se može naći na [6].

Bibliografija

[1] Vedran Vinković. *C0-Compiler*. URL: `https://github.com/vvinkov/C0-Compiler` (pogledano 3. 9. 2021.).

[2] Frank Pfenning. *C0 Reference 15-122: Principles of Imperative Computation*. 2010.

[3] Chris Lattner i Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, 2004.

[4] *LLVM Language Reference Manual*. URL: `https://llvm.org/docs/LangRef.html` (pogledano 3. 9. 2021.).

[5] *llc - LLVM static compiler*. URL: `https://www.llvm.org/docs/CommandGuide/llc.html` (pogledano 3. 9. 2021.).

[6] *LLD - The LLVM Linker*. URL: `https://lld.llvm.org/` (pogledano 3. 9. 2021.).

Dodatak A

Popis leksičkih tokena

tip tokena	mogući sadržaj tokena
OOTV	(
OZATV	)
UOTV	[
UZATV	]
VOTV	{
VZATV	}
ZAREZ	,
TZAREZ	;
USKL	!
TILDA	~
MINUS	-
ZVJ	*
TOCKA	.
STRELICA	->
SLASH	/
MOD	%
PLUS	+
LSHIFT	<<
RSHIFT	>>
LESS	<
LESSEQ	<=
GRT	>
GRTEQ	>=
EQ	==
NEQ	!=
BITAND	&

BITXOR	^
BITOR	|
LAND	&&
LOR	||
CONDQ	?
DTOCKA	:
PLUSEQ	+=
MINUSEQ	-=
ZVJEQ	*=
SLASHEQ	/=
MODEQ	%=
LSHIFTEQ	<<=
RSHIFTEQ	>>=
ASSIGN	=
BANDEQ	&=
BXOREQ	^=
BOREQ	|=
DECR	--
INCR	++
BSLASH	\
IF	if
ELSE	else
WHILE	while
FOR	for
ASSERT	assert
ERROR	error
ALLOC	alloc
ALLOCARRAY	alloc_array
NUL	NULL
BREAK	break
CONTINUE	continue
RETURN	return
INT	int
BOOL	bool
CHAR	char
STRING	string
VOID	void
STRUCT	struct
INTPOINT	int*
BOOLPOINT	bool*

CHARPOINT	char*
STRINGPOINT	string*
VOIDPOINT	void*
INTARRAY	int[]
BOOLARRAY	bool[]
CHARARRAY	char[]
STRINGARRAY	string[]
USE	#use
IDENTIFIER	slijed slova, brojeva i znakova '_' koji počinje slovom ili znakom '_'
DEKADSKI	broj u dekadskom zapisu[8]
HEKSADEKADSKI	broj u heksadekadskom zapisu[8]
CHRLIT	ASCII znak[9]
STRLIT	slijed ASCII znakova[10]
BOOLEAN	true ili false
TYPEDEF	typedef
POCETAK	ništa — označava početak datoteke
KRAJ	ništa — označava kraj datoteke

Tablica A.1: Tablica korištenih tokena

[8] Vodeći računa o ograničenjima opisanim u odjeljku 1.1.1.
[9] Vodeći računa o ograničenjima opisanim u odjeljku 1.1.3.
[10] Vodeći računa o ograničenjima opisanim u odjeljku 1.1.4.

Sažetak

U ovom radu proučavamo programski jezik C0, osnove infrastrukture LLVM, te pokazujemo kako se u jeziku C++ može napisati kompajler za jezik C0 uz pomoć infrastrukture LLVM. Prvo poglavlje se isključivo bavi jezikom C0 i njegovim ograničenjima u odnosu na jezik C.

U drugom poglavlju predstavljamo infrastrukturu LLVM i njen skup instrukcija LLVM IR te za svaki element jezika C0 pokazujemo kako se pomoću jezika C++ prevodi u LLVM IR. Na kraju pokazujemo kako se iz LLVM IR, uz pomoć alata infrastrukture LLVM, dobiva izvršivi program.

Summary

In this thesis, we analyze the C0 programming language, the basics of the LLVM compiler infrastructure, and show how to use the LLVM C++ front-end to construct a working compiler for the C0 language. The first chapter focuses on C0 and its limitations with regard to the C programming language.

In the second chapter, we introduce the LLVM infrastructure and its intermediate representation language, LLVM IR, and we illustrate how each element of the C0 language is translated into LLVM IR via the LLVM C++ front-end. Finally, we explain how to construct an executable program once we have successfully generated the LLVM IR.

Životopis

Rođen sam 29. siječnja 1994. godine u Našicama. Pohađao sam Osnovnu školu Vladimira Nazora u Feričancima te opću gimnaziju u Srednjoj školi Isidora Kršnjavoga u Našicama. Po završetku srednjoškolskog obrazovanja 2012. godine, upisao sam preddiplomski sveučilišni studij Matematika na Prirodoslovno-matematičkom fakultetu Sveučilišta u Zagrebu, koji sam završio 2017. godine. Potom sam na istom fakultetu upisao diplomski sveučilišni studij Računarstvo i matematika, koji sam završio 2021. godine.