UNIVERZITET “UNION”
RAČUNARSKI FAKULTET

MASTER RAD

Nenad Božidarević

Beograd, oktobar 2018.

УНИВЕРЗИТЕТ „УНИОН“
РАЧУНАРСКИ ФАКУЛТЕТ
Кнез Михаилова 6/VI
11000 БЕОГРАД

Број:
Датум:

UNIVERZITET UNION
RAČUNARSKI FAKULTET
BEOGRAD
RAČUNARSKE NAUKE

MASTER RAD

Kandidat: **Nenad Božidarević**
Broj indeksa: M-04/15

Tema rada: **Prevođenje programskog jezika C u ezoteričan programski jezik**

Mentor rada: **dr Dušan Vujošević**

Beograd, oktobar 2018.

Apstrakt

Ezoterični programski jezici predstavljaju klasu programskih jezika koji su prvenstveno osmišljeni ne bi li bili demonstrirani različiti aspekti dizajna jezika, odnosno prikazani ekstremni slučajevi dizajna jezika koji, i dalje, mogu biti potpuno funkcionalni.

U radu se analizira ezoterični programski jezik koji se sastoji od svega osam različitih komandi za manipulaciju memorijom i koji je Tjuring potpun. Reč je o programskom jeziku Brejnfak, za koji se u radu implementira programski prevodilac tipa konvertora jezika. Implementirani konvertor prevodi programe napisane na odabranom podskupu programskog jezika C u ekvivalentan Brejnfak kod. Ovaj podskup programskog jezika C odabran je tako da uključuje najčešće korišćene sintaksne konstrukte.

Za parsiranje ulaznog C koda u okviru implementacije prevodioca u radu je korišćen alat ANTLR, dok je za generisanje Brejnfak koda od parsiranjem dobijenog sintaksnog stabla korišćen C++. Pored samog prevodioca, realizovan je i sistem za testiranje, kojim je automatizovana provera korektnosti prevedenih programa – ovim sistemom se proverava da li, za isti ulaz, C i Brejnfak verzija nekog programa daju isti izlaz.

Najzad, u radu je obrađeno i nekoliko ideja vezanih za optimizaciju generisanog koda. Takođe se razmatraju i mogući pristupi prevođenju nekih kompleksnijih funkcionalnosti programskog jezika C.

Sadržaj

Uvod

Kada je reč o programskim jezicima, uglavnom se misli na relativno moderne jezike koji služe za prenos instrukcija računaru, ali razvoj programskih jezika nalik onima danas počeo je već sredinom 20. veka, a neke preteče programskih jezika su starije i od 150 godina.

Jednim od prvih primera programiranja smatra se rad na analitičkoj mašini Čarlsa Bebidža[1] – naime, 1842. godine Ejda Lavlas je uz svoj prevod memoara ovog matematičara pridodala i uputstva za računanje Bernulijevih brojeva upotrebom analitičke mašine, što ju je efektivno učinilo prvim programerom [1]. Ovo je savim adekvatno, jer se, zbog svog rada na diferencijalnoj i analitičkoj mašini, Čarls Bebidž smatra ocem modernih računara[2].

100 godina kasnije, tokom 1940-ih godina, počeli su da se pojavljuju i prvi programski jezici za digitalne računare, velika većina njih ili razvijena za ENIAC[3] ili po ugledu na ENIAC jezike. Najznačajniji od ovih bio je Plankalkul, dizajniran od strane Konrada Zusa za računar Z1, jer je on bio prvi programski jezik višeg nivoa, odnosno koji je apstrakovao detalje i operacije nižeg nivoa, bliže samom računaru [2].

Sledeće decenije pojavio se i prvi komercijalan, široko dostupan programski jezik, FORTRAN[4], razvijen od strane kompanije IBM. Ovaj jezik je i dalje u upotrebi, kao i LISP i COBOL[5] koji su nastali nekoliko godina kasnije, a oni su, zajedno sa raznim idejama koje su cirkulisale u to vreme, doveli do razvijanja jezika ALGOL 68.

ALGOL 68 zbog svoje komplikovane sintakse i velikog broja slabo korišćenih funkcionalnosti nije bio popularan, ali uprkos tome predstavlja korak ka jezicima koji su kasnije razvijeni, pa su se tako tokom sedamdesetih godina počeli pojavljivati jezici sa danas široko rasprostranjenim funkcionalnostima i paradigmama, kao što su na primer objektno-orijentisano programiranje i struktuirano programiranje, odnosno izbegavanje upotrebe

[1] Analitička mašina je mehanički računar opšte namene koji je Čarls Bebidž projektovao nakon diferencijalne mašine. Ova mašina, za razliku od diferencijalne, nikada nije konstruisana.

[2] Slično, Alan Tjuring se smatra ocem računarskih nauka zbog svog rada na algoritmima.

[3] ENIAC (skr.) – Electronic Numerical Integrator and Computer

[4] FORTRAN (skr.) – Formula Translation

[5] COBOL (skr.) – Common Business-Oriented Language

komande `goto`. Najznačajniji od ovih jezika je svakako C i podskup njegovih funkcionalnosti je korišćem u ovom radu.

Osamdesetih i devedesetih godina, zajedno sa popularizacijom i većom dostupnošću računara, kao i sa razvojem Interneta, nastali su mnogi novi programski jezici koji su danas u širokoj upotrebi – C++, Objective-C, Python, Java, JavaScript i PHP su samo neki od njih. Ovi jezici su se sve više i više fokusirali na razvoj velikih, modularnih sistema, i funkcionalnosti koje su oni nudili bile su veoma napredne i kompleksne, ali su sami jezici i dalje bili laki (ako ne i lakši) za korišćenje.

Ovakav trend i smer razvoja programskih jezika, naravno, ima smisla, jer za cilj ima što veću efikasnost programera, pa tako većina jezika ima i sintaksu koja koristi reči engleskog jezika – engleski jezik je *de facto* standard kada je programiranje u pitanju. Međutim, programiranje kao deo računarskih nauka nije razvijano isključivo u ovom smeru, već je osim praktičnih primena posvećeno vreme i samoj teoriji iza funkcionisanja programskih jezika i analizi raznih grana programiranja koje demonstriraju različite aspekte ove teorije.

Jedna od ovih grana programiranja su ezoterični programski jezici. Reč ezoteričnost potiče iz grčkog jezika gde znači "unutar" ili "unutrašnjost", i sredinom 17. veka je dobila svoj današnji oblik koji označava nešto tajnovito ili skriveno, razumljivo malom broju ljudi sa specijalizovanim znanjem ili interesovanjima. U slučaju programskih jezika, ova definicija se direktno primenjuje – ezoterični programski jezici su oni koji demonstriraju granice dizajna programskih jezika i samim tim su teški za razumevanje i nemaju praktičnu primenu, ali su i dalje u potpunosti funkcionalni.

Kako je cilj ovih programskih jezika da budu u potpunosti funkcionalni, u teoriji je moguće prevesti kod napisan u nekom naprednijem jeziku (odnosno jeziku koji nije ezoteričan) u ezoteričan programski jezik u izboru, i ovaj rad se bavi upravo demonstiranjem ovog procesa, odnosno prevođenjem podskupa funkcionalnosti jezika C u ezoteričan programski jezik Brejnfak.

Ezoteričan programski jezik Brejnfak

Brejnfak je ezoteričan programski jezik razvijen 1993. godine od strane Urbana Milera i karakteriše ga velika jednostavnost, odnosno minimalizam – ovaj jezik se sastoji od samo 8 različitih komandi [3]. Uprkos svojoj jednostavnosti, jezik Brejnfak je Tjuring potpun, o čemu će biti više reči kasnije, ali ovo znači da je – u teoriji – upotrebom jezika Brejnfak moguće predstaviti i bilo koji program napisan u nekom kompleksnijem jeziku.

Upravo ova tvrdnja predstavlja okosnicu ovog rada, gde je na primeru prevođenja koda napisanog u jeziku C u Brejnfak prikazano kako se određene relativno kompleksne funkcionalnosti mogu predstaviti upotrebom veoma malog skupa komandi koje Brejnfak nudi.

Kako je programski jezik C uprkos svojoj starosti (prva verzija je objavljena 1972. godine) i dalje jedan od najmoćnijih, najkompleksnijih i najefikasnijih jezika, cilj ovog rada nije mapiranje čitavog skupa funkcionalnosti jezika C u jezik Brejnfak. Umesto toga, odabran je mali podskup funkcionalnosti koje čine okosnicu većine programa napisanih u bilo kojem jeziku, a pritom mogu biti upotrebljene za implementaciju upravo onih kompleksnijih funkcionalnosti koje u ovom radu nisu obrađene. Ovo je detaljnije opisano u sledećem poglavlju.

Okruženje

Kada je u pitanju okruženje u kojem se Brejnfak program izvršava, odnosno stanje koje program održava, ono se sastoji od:

1. Memorije neograničene veličine čije su ćelije sekvencijalno indeksirane počevši od indeksa 0.
2. Pokazivača na određenu memorijsku lokaciju (tj. indeks ćelije).

Ovo se uglavnom predstavlja kao beskonačna traka prikazana u tabeli 1.

Pokazivač	↓									
Memorija	0	0	0	0	0	0	0	0	0	. . .
Indeks	0	1	2	3	4	5	6	7	8	

Tabela 1. Organizacija memorije i pokazivača u Brejnfak programu

Na početku izvršavanja programa, sve memorijske ćelije su inicijalizovane sa vrednošću 0, a pokazivač pokazuje na prvu (tj. nultu) memorijsku lokaciju.

Sva funkcionalnost u jeziku Brejnfak zasniva se na manipulaciji memorijske lokacije na koju pokazivač trenutno pokazuje, pa su tako omogućene sledeće operacije:

- Uvećavanje i umanjivanje vrednosti na trenutnoj memorijskoj lokaciji za 1.
- Čitanje jednog karaktera i upis njegove ASCII[6] vrednosti na trenutnu memorijsku lokaciju.
- Ispis vrednosti trenutne memorijske lokacije kao karakter sa tom ASCII vrednošću.
- Pomeranje pokazivača za jedno mesto u levo ili u desno.

Međutim, ove operacije nisu dovoljne za postizanje Tjuring potpunosti, pa tako Brejnfak uključuje i operacije za kontrolu toka koje, u zavisnosti od vrednosti trenutne ćelije, kontrolišu da li će se određeni deo koda ponovo izvršiti ili ne – ovo je u suštini ekvivalentno jednostavnoj petlji.

Sintaksa

Za realizaciju ovih funkcionalnosti, Brejnfak pruža 8 komandi čije je ponašanje prikazano u tabeli 2.

Komanda	Ponašanje komande
<	Pokazivač se pomera u levo, odnosno indeks trenutne memorijske lokacije se umanjuje za 1.
>	Pokazivač se pomera u desno, odnosno indeks trenutne memorijske lokacije se uvećava za 1.
+	Vrednost trenutne memorijske lokacije se uvećava za 1.

[6] ASCII (skr.) – American Standard Code for Information Interchange

-	Vrednost trenutne memorijske lokacije se umanjuje za 1.
,	Učitava se jedan karakter i vrednost trenutne memorijske lokacije se postavlja na ASCII vrednost tog karaktera.
.	Ispisuje se jedan karakter čija je ASCII vrednost jednaka vrednosti na trenutnoj memorijskoj lokaciji.
[	Ukoliko je vrednost trenutne memorijske lokacije različita od 0, program nastavlja sa normalnim tokom izvršavanja. Ukoliko je vrednost ove lokacije 0, preskaču se sve komande zaključno za zatvorenom zagradom koja je uparena sa ovom otvorenom zagradom.
]	Izvršavanje programa se vraća nazad na otvorenu zagradu koja je uparena sa ovom zatvorenom zagradom.

Tabela 2. Komande dostupne u Brejnfak kodu

Osim ovih 8 komandi, tj. karaktera, svi ostali karakteri se ignorišu, što znači da Brejnfak kod može biti formatiran na bilo koji način[7], kao i da je dodavanje tekstualnih komentara veoma jednostavno. Ovo, očigledno, nije tačno ukoliko komentari sadrže znakove interpunkcije (tj. tačke i zareze), jer ovi karakteri predstavljaju komande u Brejnfaku. Međutim, ovakvi kompleksniji komentari i dalje mogu biti ubačeni u kod ukoliko se nalaze na početku i između uglastih zagrada – kako je memorijska traka inicijalizovana nulama, na početku programa uslov za izvršavanje petlje neće biti ispunjen, pa će tako sve u okviru ove prve petlje biti ignorisano.

[7] Ovo je slučaj kod većine programskih jezika, ali ne i kod, na primer, programskog jezika Pajton koji zahteva određeno formatiranje i indentaciju ne bi li se pravilno izvršavao.

Tjuringova potpunost

Kada je reč o programskim jezicima, termin koji se često pominje jeste Tjuringova potpunost, koji formalno označava sposobnost programskog jezika da simulira bilo koju Tjuringovu mašinu [4][5]. Ova definicija može zvučati prilično apstraktno bez razumevanja teorije iza nje, pa se neretko koristi kolokvijalno, i onda se Tjuring potpunim jezikom smatra onaj jezik koji može da simulira bilo koji drugi programski jezik opšte upotrebe – što je upravo cilj ovog rada. Ipak, korisno je zaći u detalje toga šta se smatra Tjuringovom mašinom i kakav je njen odnos sa programskim jezikom Brejnfak.

Tjuringova mašina predstavlja matematički model apstraktne mašine koja, prateći određena pravila, manipuliše simbolima zapisanima na traci [5]. Već sa samim početkom ove definicije očigledno je da Brejnfak deli bitna svojstva sa opštom Tjuringovom mašinom – ovaj jezik takođe funkcioniše tako što, prateći određena pravila, manipuliše vrednostima na nekoj memorijskoj traci.

Šta više, Tjuringovoj mašini se dalje dodaje svojstvo da je ova traka beskonačna, kao i da je u svakom trenutku glava čitača pozicionirana iznad određene ćelije na traci, što se takođe uklapa u kasnije opisanu implementaciju jezika Brejnfak.

Deo definicije Tjuringove mašine koji se, pak, ne preslikava direktno na Brejnfak jeste sama logika koja se izvršava nakon što je glava čitača postavljenja iznad određene ćelije. Sama logika je veoma jednostavna, i ona se može sastojati od sledeća tri koraka:

1. Upisivanje nove vrednosti na trenutnu memorijsku lokaciju.
2. Pomeranje trake u levo ili desno, ili ostanak na istoj poziciji.
3. Promena internog stanja mašine.

Ovo interno stanje pomenuto u trećem koraku je ono što Tjuring mašinu čini drugačijom od jezika Brejnfak. Naime, logika koje će biti izvršena tokom jednog koraka mašine zavisi ne samo od vrednosti na trenutnoj memorijskoj lokaciji, već i od ovog internog stanja koje se nalazi u konačnom skupu mogućih stanja za tu mašinu, i ovo ne postoji u programskom jeziku Brejnfak.

Naravno, ovakvo ponašanje je i dalje moguće, jer se u suštini svodi na grananje koje proverava dve promenljive, što znači da bi Tjuringovu mašinu i dalje bilo moguće simulirati ukoliko je moguće implementirati ovakvo grananje upotrebom komandi jezika Brejnfak, što je kasnije i demonstrirano.

Varijacije

Kako specifikacija jezika napisana 1993. godine nije bila dovoljno detaljna, ostavljen je prostor za različite interpretacije nekih njenih delova, kao i razne modifikacije, pa je neophodno precizno definisati ponašanje jezika i okruženja koja su korišćena u ovom radu.

Prvo pitanje koje se postavlja jeste koja je veličina memorijske lokacije. Na modernim procesorima, imalo bi smisla da veličina svake lokacije bude 8 bajtova, što odgovara celobrojnoj vrednosti na 64-bitnoj arhitekturi, i znatno olakšava aritmetičke operacije, jer u većini slučajeva nema potrebe brinuti o rezultatima van opsega 64-bitne celobrojne vrednosti. Međutim, ovo nije u skladu sa inicijalnom specifikacijom, tako da je u ovom radu korišćena memorijska lokacije veličine 1 bajt.

U skladu sa ovom odlukom, pri parsiranju i prevođenju biće podržane isključivo promenljive tipa `char`, jer svi ostali tipovi promenljivih (npr. `short`, `int`, `float` itd.) zahtevaju barem dva bajta za predstavljanje. Naravno, ove veće promenljive se mogu predstaviti i implementirati kombinovanjem više memorijskih lokacija veličine 1 bajt uz korišćenje dodatne logike, što je kasnije u radu takođe pokriveno.

S obzirom na malu veličinu memorijskih lokacija (ovo je, naravno, relativno u odnosu na danas standardne veličine osnovnih tipova promenljivih), neophodno je obratiti pažnju na to kako se vrednost na nekoj lokaciji ponaša pri dobijanju vrednosti koja je van opsega koji se može predstaviti upotrebom jednog bajta. Preciznije, potrebno je specificirati šta se dešava ukoliko se lokacija sa vrednošću 0 umanji za 1, odnosno lokacija sa vrednošću 255 uveća za 1. Ponašanje koje je korišćeno u ovom radu jeste tzv. "obmotavanje" gde uvećavanje maksimalne moguće vrednosti rezultuje u minimalnoj mogućoj vrednosti i obrnuto.

Najzad, potrebno je definisati veličinu memorije, odnosno dužinu memorijske trake koja je do sada smatrana beskonačnom. Ovo, očigledno, nije moguće implementirati, jer beskonačni resursi nisu dostupni[8], pa tako originalna specifikacija jezika definiše ovu memorijsku traku kao minimalne dužine od 30.000 ćelija.

Iako ovo na prvi pogled omogućava korišćenje 30.000 različitih promenljivih, u praksi je ovaj broj daleko manji, jer je tokom izvršavanja programa neophodno privremeno koristiti memorijske lokacije za čuvanje među-rezultata prilikom raznih kalkulacija. Šta više, zbog

[8] Postoje razne tehnike koju omogućavaju upotrebu više memorije nego što je dostupno u radnoj memoriji, ali čak i u tom slučaju program bi bio ograničen veličinom adresabilnog memorijskog prostora, odnosno veličinom promenljive u kojoj se čuva trenutni indeks pokazivača. Na današnjim sistemima, ovo bi verovatno bila celobrojna vrednost 64 bita, što omogućava indeksiranje približno $2x10^{19}$ ćelija.

malog broja komandi, broj ovih među-rezultata i operacija neophodnih za realizaciju bilo koje iole kompleksnije funkcionalnosti je daleko veći nego što bi bio kod drugih programskih jezika koji su u širokoj upotrebi, pa je zato pri interpretaciji programa prevedenog iz programskog jezika C neophodno obezbediti veći broj memorijskih lokacija.

Kao što je već objašnjeno, istinski neograničena memorija ne postoji, ali za potrebe ovog rada biće implementiran Brejnfak interpreter koji dozvoljava adresiranje bilo koje memorijske lokacije sa celobrojnim indeksom – detalji implementacije objašnjeni su kasnije.

Ovim se takođe dolazi i do poslednjeg pitanja na koje treba odgovoriti, a to je da li je ova memorijska traka beskonačna u oba smera. Za razliku od većine jezika, Brejnfak nema definisana ponašanja za neispravne operacije – na primer, pristup negativnom indeksu ili deljenje nulom će u programskom jeziku Java izazvati izuzetak koji će onda uticati na tok programa, odnosno prekinuti njegovo izvršavanje ukoliko izuzetak nije ispravno obrađen. U slučaju jezika Brejnfak, izuzetak bi se mogao javiti pomeranjem pokazivača na ćeliju sa negativnim indeksom i ispisom vrednosti te (nepostojeće) ćelije.

Ne bi li bile izbegnute ovakve komplikacije u implementaciji, varijanta Brejnfaka korišćena u ovom radu dozvoljava memorijske lokacije sa negativnim indeksom, odnosno memorijska traka je beskonačna u oba smera.

Parsiranje C programa

Prvi korak bilo kog prevodioca ili kompilatora mora biti učitavanje izvornog koda programa u određenom jeziku i analiza tog koda ne bi li on bio struktuiran, odnosno predstavljen određenom strukturom podataka koja bi dalje mogla da se koristi za prevođenje (tj. kompilaciju) u ciljani jezik – ovaj proces naziva se parsiranje [6].

Bitno je imati na umu da se pri parsiranju ne posmatra sama logika koda, i zbog toga parsiranje predstavlja tek prvi korak u prevođenju nekog programa. Ovo se može predstaviti lingvističkim terminima:

1. Parser se bavi analizom sintakse programa. Sintaksa predstavlja skup pravila za konstrukciju gramatički ispravnih rečenica.
2. Prevodilac se bavi analizom semantike programa. Semantika predstavlja analizu značenja ovih ispravnih rečenica.

Dakle, zadatak parsera je da analizira izvorni kod i, ukoliko je on gramatički ispravan, pripremi podatke neophodne za kasnije semantičku analizu.

Kao što im samo ime kaže, programski jezici zaista jesu jezici, i zato se kod njih mogu koristiti termini kao što su sintaksa i semantika, a programski jezik C korišćen u ovom radu nije izuzetak. Ne bi li kod napisan u jeziku C bio uspešno parsiran, ovaj proces je podeljen na dva dela:

1. Leksička analiza. Bilo koji jezik se može smatrati sekvencom karaktera, ali, ukoliko se uvede dodatni nivo apstrakcije, jezik se takođe može smatrati sekvencom reči. U slučaju programskih jezika, ove kraće sekvence karaktera se smatraju tokenima, i detektovanje tokena će biti prvi korak – na primer, niz cifara će biti broj, dok će niz slova biti reč koja kasnije može biti identifikovana kao ime promenljive, poziv funkcije i slično.
2. Parsiranje. Nakon konstrukcije tokena, sam parser, koji u sebi sadrži gramatička pravila, analizira ove tokene i proverava da li su oni uklopljeni u ispravne "rečenice". Na ovaj način dobija se tražena struktura podataka u kojoj su tokeni organizovani u veće celine (što predstavlja još jedan, nov nivo apstrakcije), ili se javlja greška ukoliko taj kod gramatički nema smisla – na primer, reč praćena znakom = i brojem biće dodela vrednosti promenljivoj, ali broj praćen znakom = i drugim brojem gramatički neće biti ispravno.

Sintaksno stablo

Ukoliko se proces leksičke analize i parsiranja posmatra unazad, on je ekvivalentan razbijanju programa kao jedne celine na manje delove koji su sve jednostavniji i jednostavniji. U oba slučaja, ciljani rezultat ovog procesa je struktura podataka koju prevodilac može efikasno da koristi, i ovo se postiže konstrukcijom sintaksnog stabla [7].

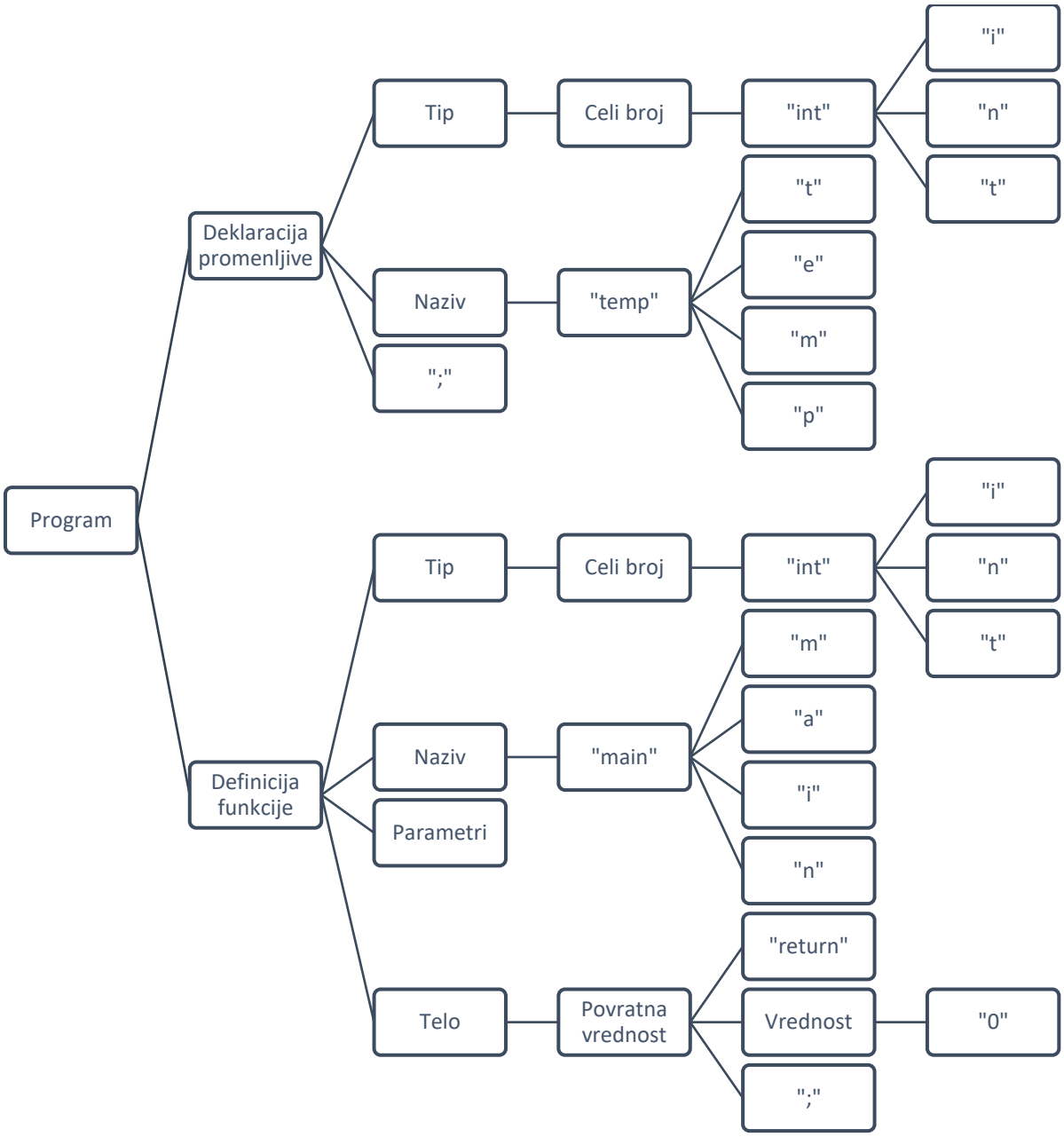
Sintaksno stablo predstavlja stablo u čijem korenu se nalazi sam program, a svaki sledeći nivo predstavlja isti taj program na manjem nivou apstrakcije. Prateći ovo stablo ka listovima, u pretposlednjem nivou svake grane nalazili bi se identifikovani tokeni, dok bi se u listovima stabla nalazili zasebni karakteri. Imajući ovu strukturu, prevodilac je u mogućnosti da je rekurzivno obiđe i iskoristi te informacije za generisanje Brejnfak koda [8]. Pritom, prevodilac neretko neće morati da obiđe čitavo stablo, jer u većini slučajeva nema smisla posećivati listove – umesto toga, reči i brojevi se mogu koristiti kao najmanje gradivne jedinice programa, iako one to zapravo nisu.

Programski jezik C ima veoma kompleksnu strukturu, pa tako ovo stablo – u zavisnosti od toga kako su parsiranje i njegova konstrukcija implementirani – može biti veoma veoma duboko[9]. Na primer, već veoma jednostavan program sa jednom promenljivom i praznom funkcijom prikazan na listingu 1 bi mogao imati 35 čvorova prikazanih na dijagramu 1.

```
int temp;

int main() {
      return 0;
}
```

Listing 1. Primer jednostavnog koda napisanog u programskom jeziku C

[9] Širina stabla takođe može biti velika, ali ovo više govori o kompleksnosti samog programa, a ne jezika.



Dijagram 1. Sintaksno stablo za primer koda napisanog u programskom jeziku C

Imajući na umu da se pri kompilaciji ovo čitavo stablo prevodi u asemblerski jezik sa ograničenim mogućnostima, jasno da se kompleksnije funkcionalnosti mogu razgraditi na daleko jednostavnije. Samim tim, cilj ovog rada nije razvoj prevodioca u Brejnfak koji podržava sve funkcionalnosti jezika C, već prevođenje određenih osnovnih funkcionalnosti koje bi kasnije mogle biti korišćene za implementaciju ostalih, naprednijih funkcionalnosti [9]. Podskup funkcionalnosti jezika C koji je obrađen u ovom radu je:

- Globalno i lokalno deklarisanje promenljivih `char` tipa

- Dodela vrednosti promenljivama
- Aritmetičke operacije
- Bulova logika
- Ispisivanje teksta upotrebom funkcije `printf`
- Učitavanje teksta upotrebom funkcije `scanf`
- Funkcionalnost grananja upotrebom konstrukta `if`/`else`
- Funkcionalnost petlji upotrebom konstrukata `for`, `while`, i `do...while`

ANTLR

U ovom radu za parsiranje koda napisanog u programskom jeziku C korišćen je ANTLR[10], koji izvršava i leksičku analizu i samo parsiranje. Kod nekih starijih alata, kao što su lex i yacc, ova dva koraka su bivala zasebno implementirana, ali ANTLR koristi unifikovanu konfiguracionu datoteku koja pokriva oba procesa [10].

Ova konfiguraciona datoteka koja predstavlja gramatiku sastoji se iz skupa pravila dva tipa:

1. Pravila za leksiranje. Imena ovih pravila počinju velikim slovom, a za definisanje samih pravila koriste se regularni izrazi. U ovim regularnim izrazima mogu da se koriste tekstualni literali i prethodno definisana pravila za leksiranje[11].
2. Pravila za parsiranje. Imena ovih pravila počinju velikim slovom i za njihovo definisanje se takođe koriste regularni izrazi. U ovim izrazima mogu da se koriste prethodno definisana pravila za leksiranje, prethodno definisana pravila za parsiranje, kao i tekstualni literali[12].

Objekti koji su rezultat leksiranja biće tokeni i oni će se nalaziti u listovima sintaksnog stabla. Za razliku od prethodnog primera, zasebni karakteri neće biti uključeni u stablo, jer ne služe nikakvu funkciju za parsiranje i prevođenje. Pravila za parsiranje će koristiti te tokene i

[10] ANTLR (skr.) – Another Tool for Language Recognition

[11] Preciznije, pravila za leksiranje ne mogu referencirati obična pravila za leksiranje, već samo fragmente. Fragmenti predstavljaju posebna pravila za leksiranje koje ne izgrađuju čitav token, ali se dalje koriste za izgradnju token radi pojednostavljivanja gramatike. Za potrebe ovog rada, razlika između fragmenata i samih pravila nije bitna.

[12] Iako u slučaju pravila za parsiranje tekstualni literali ne čine zasebna pravila za leksiranje, oni i dalje rezultuju u tokenima u sintaksnom stablu.

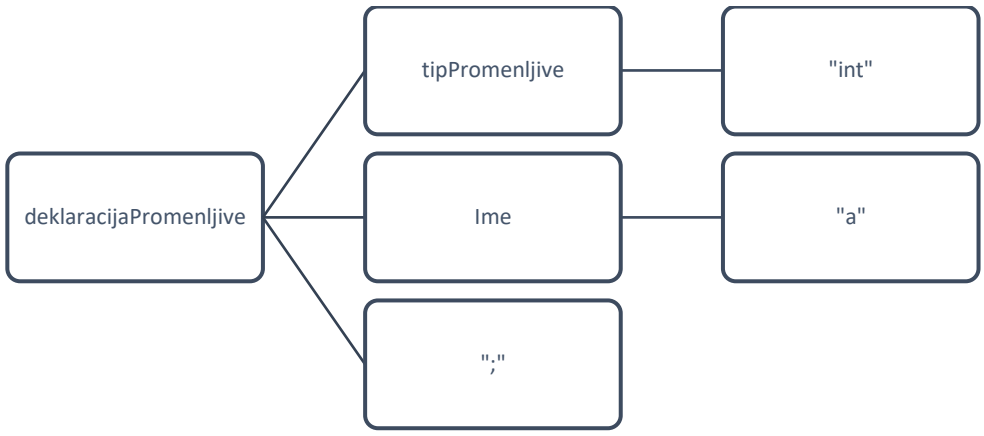
graditi ostatak stabla iznad njih, sve dok se ne dođe do glavnog pravila za parsiranje koje opisuje strukturu čitavog programa i koje rezultuje u korenu stabla.

Gramatika za jednostavno prepoznavanje deklaracije promenljive (`int a;` ili `char b;`) prikazana je na listingu 2.

```
fragment Slovo
      : [a-zA-Z]
      ;

Ime
      : Slovo+
      ;

tipPromenljive
      : 'int'
      | 'char'
      ;

deklaracijaPromenljive
      : tipPromenljive Ime ';'
      ;
```

Listing 2. ANTLR gramatika za parsiranje deklaracije promenljive

Rezultujuće sintaksno stablo za jednu promenljivu prikazano je na dijagramu 2.



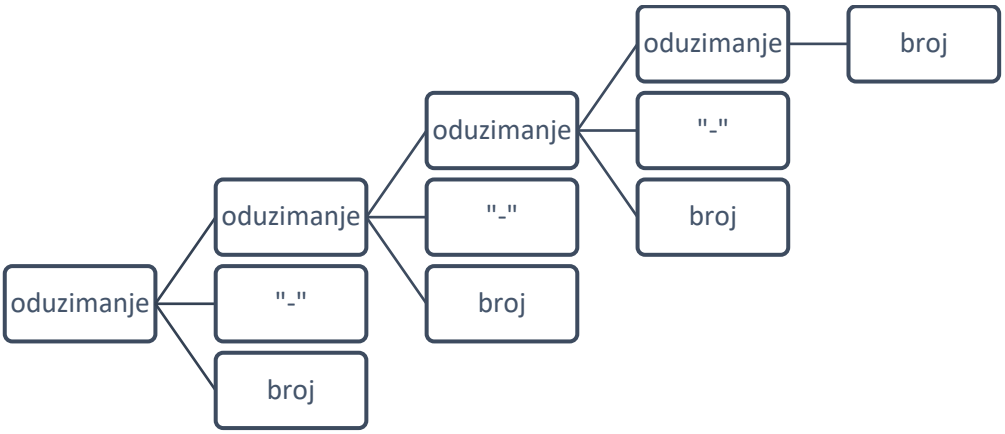
Dijagram 2. Sintaksno stablo deklaracije promenljive

Pri razvoju pravila za parsiranje, u sistemu ANTLR se pored regularnih izraza može koristiti i leva rekurzija, što je veoma praktično za parsiranje određenih izraza, pa je tako ona korišćena u ovom radu. Na primer, ukoliko se koristi leva rekurzija za parsiranje oduzimanja brojeva, rezultujuće stablo je lakše obići rekurizvno, jer svojim oblikom već određuje redosled

operacija. Upotreba rekurzije demonstrirana je na listingu 3, a rezultujeće sintaksno stablo na dijagramu 3.

```
oduzimanje
      : broj
      | oduzimanje '-' broj
      ;
```

Listing 3. ANTLR gramatika za parsiranje operacije oduzimanja



Dijagram 3. Sintaksno stablo operacije oduzimanja

U ovom radu korišćena je gramatika za jezik C dostupna u okviru ANTLR projekta koja implementira podršku za parsiranje C koda napisanog prema standardnu objavljenom 2011. godine [11]. Međutim, ova gramatika je takođe pojednostavljena izbacivanjem nepotrebnih pravila (kako za leksiranje, tako i za parsiranje) koja su namenjena za prepoznavanje naprednih funkcionalnosti, jer ona nisu u okviru ovog rada.

Prevođenje C programa u Brejnfak

Ukoliko pogledamo funkcionalnosti programskog jezika Brejnfak, očigledno je da se određene komande napisene u programskog jeziku C mogu direktno prevesti u ekvivalentne komande u Brejnfaku.

Na primer, inkrementiranje ili dekrementiranje vrednosti promenljive (`a++` i `a--`, respektivno) se direktno mapira na Brejnfak komande `+` i `-`; čitanje i ispis karaktera koji se izvode komandama `.` i `,` u Brejnfaku u jeziku C se vrše pozivom funkcija `getc()` i `putc()`; Brejnfak petlje su ekvivalentne petlji `while (a)`, itd.

Izazov pri prevođenju programa iz jezika C u jezik Brejnfak samim tim leži u predstavljanju kompleksnijih funckionalnosti upotrebom isključivo ovih koje se mogu direktno mapirati. Neke od ovih kompleksnijih funkcionalnosti, međutim, mogu biti razgrađene na druge kompleksnije funkcionalnosti, što će biti korišćeno u opisu određenih postupaka prevođenja.

Usled ovih međuzavisnosti, moguće je da se funkcionalnost A može predstaviti upotrebom funkcionalnosti B, a funkcionalnost B upotrebom funkcionalnosti A. Ovo bi, naravno, učinilo prevođenje nemogućim, tako da je na kraju predstavljen i graf međuzavisnosti između njih kojim je pokazano da se sve one mogu svesti na Brejnfak kod, odnosno da je taj graf acikličan.

Radi jednostavnosti, umesto korišćenja isključivo Brejnfak koda za predstavljanje funkcionalnosti, u nekim slučajevima iskorišćen je pseudokod, poštujući sledeća pravila:

1. U pseudokodu se koristi notacija `memorija[x]`, gde je `x` indeks proizvoljne memorijske lokacije, koja označava pomeranje pokazivača na tu memorijsku lokaciju.
2. Brejnfak petlje i uvećavanje/umanjivanje vrednosti se u pseudokodu predstavljaju notacijama `while (memorija[x]) {}`, `memorija[x]++` i `memorija[x]--`, respektivno.
3. Kod funkcionalnosti koje se predstavljaju upotrebom isključivo Brejnfak koda uvodi se pseudokod notacija koja se kasnije može koristiti umesto tog Brejnfak koda.
4. Kod funkcionalnosti koje se predstavljaju upotrebom pseudokoda takođe se uvodi dodatna pseudokod notacija za predstavljanje te funkcionalnosti.

Promenljive

Prvi i najbitniji aspekat prevođena C programa u Brejnfak jeste rad sa promenljivama. U programskom jeziku C, promenljive mogu biti globalne ili lokalne, u zavisnosti od toga gde su deklarisane. Promenljive deklarisane van funkcije su globalne, dok su one deklarisane u funkciji lokalne. Pritom, promenljive mogu biti deklarisane u okviru bloka komandi (upotrebom vitičastih zagrada), i u tom slučaju su one lokalne za taj blok i dostupne samo u njemu.

Pri prevođenju u Brejnfak, samim tim, neophodno je imati sistem koji bi svakoj promenljivoj dodelio određenu memorijsku lokaciju, a promenljiva bi bila jedinstveno određena kombinacijom njegog imena i bloka komandi u kojem je definisana. U samoj implementaciji, dovoljno je svakom bloku komandi dodeliti jedinstven indeks – ovi indeksi mogu biti sekvencijalni, a nulti indeks korišćen za globalne promenljive – i onda taj broj zajedno sa imenom same promenljive koristiti za nalaženje indeksa te promenljive u Brejnfak memoriji.

Dalje, tokom procesa prevođenja, prevodilac je u svakom trenutku svestan trenutne pozicije memorijskog pokazivača, što znači da je upotrebom komandi `<` i `>` uvek moguće pomeriti pokazivač na željenu memorijsku lokaciju (tj. promenljivu), bilo da je to radi čitanja njene vrednosti ili upisivanja nove – ovo se, kao što je pomenuto, u pseudokodu predstavlja upotrebom notacije `memorija[x]`.

Najzad, za dodeljivanje vrednosti promenljivoj, odnosno trenutnoj memorijskoj lokaciji, dovoljno je upotrebiti komandu `+` N puta, gde je N ciljana vrednost promenljive. Ovo će, doduše, ispravno funkcionisati isključivo pri prvoj dodeli vrednosti promenljivoj, jer je njena početna vrednost nula – ne bi li dodela vrednosti uvek funkcionisala, neophodno je vrednost promenljive prvo vratiti na nulu, što se jednostavno postiže upotrebom petlje koja njenu vrednost umanjuje sve dok ne postane nula, odnosno komandama `[-]`.

Imajući ovo prevođenje na umu, za dodeljivanje konstantne vrednosti `y` promenljivoj na memorijskoj lokaciji `x` biće korišćena pseudokod notacija `memorija[x] = y`.

Aritmetičke operacije

Pri izvršavanju aritmetičkih operacija, jedan od problema predstavlja činjenica da je za bilo koji promenu vrednosti određene memorijske lokacije neophodno promeniti (tj. "uništiti") vrednost neke druge lokacije, ne bi li operacija mogla da se realizuje upotrebom

petlje. Samim tim, okosnicu osnovnih aritmetičkih operacija činiće pravljenje kopija vrednosti koje učestvuju u binarnoj operaciji i menjanje tih kopija prilikom vršenja operacije.

Pravljenje ovih kopija se postiže upotrebom petlje koja umanjuje vrednost originalne promenljive, i pri svakom umanjivanju ove vrednosti uvećava vrednost proizvoljnog broja drugih memorijskig lokacija. U većini slučajeva, ovo će značiti uvećavanje vrednosti dve druge lokacije – ovo je neophodno ne bi li se jedna kopija kasnije koristila za računanje, a druga kopija premestila nazad u originalnu promenljivu (pošto je ona prethodno obrisana). Ukoliko je potrebno napraviti kopiju promenljive smeštene u prvoj memorijskoj lokaciji, postupak je prikazan u tabeli 3.

Memorija	12	0	0	0	0	0	0	0	0	...

Memorija	0	12	12	0	0	0	0	0	0	...

Memorija	12	12	0	0	0	0	0	0	0	...

Tabela 3. Promene memorije Brejnfak programa pri pravljenju kopije promenljive

Ukoliko se ovo predstavi Brejnfak kodom, implementacija je prikazana na listingu 4.

```
[->+>+<<]
>>
[-<<+>>]
```

Listing 4. Brejnfak program za pravljenje kopije promenljive

Ova tri reda redom predstavljaju:

1. Petlju u kojoj se prva lokacija umanjuje za 1, a druga i treća lokacija uvećavaju za po 1.
2. Pomeranje pokazivača na treću lokaciju radi premeštanja njene vrednosti na prvu lokaciju.

3. Premeštanje vrednosti treće lokacije na prvu lokaciju korišćenjem istog postupka kao i u prvom redu komandi.

Imajući ovaj mehanizam, koji će biti predstavljen pseudokod notacijom `memorija[x] = memorija[y]`, lakše je implementirati aritmetičke operacije. Kod svake od ovih operacija postoje tri slučaja:

1. Oba operanda mogu biti konkretne brojevne vrednosti.
2. Jedan operand može biti brojevna vrednost, a drugi promenljiva.
3. Oba operanda mogu biti promenljive.

S obzirom da se prva dva slučaja mogu predstaviti trećim nakon smeštanja brojevnih vrednosti na određenu memorijsku lokaciju, biće objašnjena implementacija ovih operacija nad dve promenljive.

Sabiranje i oduzimanje su veoma slične operacije, i one počinju sa dve vrednosti – radi jednostavnijeg objašnjenja, ove vrednosti su smeštene u susednim memorijskim lokacijama, što je prikazano u tabeli 4.

Memorija	12	5	0	0	0	0	0	0	0	...

Tabela 4. Stanje memorije pre sabiranja prve dve vrednosti

Prvi korak će biti kopiranje ovih vrednosti u nove ćelije, kao što je ranije objašnjeno, ne bi li bile sačuvane originalne vrednosti, što je prikazano u tabeli 5.

Memorija	12	5	12	5	0	0	0	0	0	...

Tabela 5. Stanje memorije nakon kopiranja promenljivih radi njihovog sabiranja

Nakon toga, koristi se još jedna petlja koja umanjuje kopiju druge promenljive i dodaje je na (ili oduzima od) kopiju prve promenljive, što je prikazano u tabeli 6.

Memorija	12	5	17	0	0	0	0	0	0	...

Tabela 6. Stanje memorije nakon sabiranja dve promenljive

Rezultat operacije se nalazi u trećoj ćeliji i on onda može biti dodeljen odgovarajućoj memorijskoj ćeliji za dalju upotrebu.

Pseudokod koji vrši sabiranje na ovaj način prikazan je na listingu 5.

```
memorija[0] = prvi sabirak
memorija[1] = drugi sabirak
memorija[2] = memorija[0]
memorija[3] = memorija[1]
while (memorija[3]) {
  memorija[2]++
  memorija[3]--
}
```

Listing 5. Pseudokod za sabiranje dve promenljive

Brejnfak verzija ovog koda prikazana je na listingu 6.

```
[->>+>+<<<]>>>[-<<<+>>>]    Kopiranje prve
                            promenljive
<<[->>+>+<<<]>>>[-<<<+>>>]  Kopiranje druge
                            promenljive
<[-<+>]                     Sabiranje
```

Listing 6. Brejnfak kod za sabiranje dve promenljive

Množenje je implementirano na sličan način, jer se u suštini sastoji od ponovljenog sabiranja. U slučaju množenja, rezultat neće biti dobijen menjanjem kopije prve promenljive, već će se vrednost prve promenljive dodavati na praznu memorijsku lokaciju onoliko puta koliko je vrednost kopije druge lokacije. Kako se pri svakom koraku kopija prve promenljive uništava, neophodno je napraviti novi kopiju svaki put. Ukoliko se množioci nalaze u prve dve memorijske ćelije, pseudokod – u kojem sada može biti korišćena operacija sabiranja – je prikazan na listingu 7.

```
memorija[0] = prvi mnozilac
memorija[1] = drugi mnozilac
memorija[2] = 0
memorija[3] = memorija[0]
memorija[4] = memorija[1]
while (memorija[4]) {
  memorija[2] = memorija[2] + memorija[3]
  memorija[4]--
}
```

Listing 7. Pseudokod za množenje dve promenljive

Kao što je prikazano, sabiranje i oduzimanje se mogu obaviti upotrebom jednostavnog Brejnfak koda, dok je za množenje neophodno sabiranje. Celobrojno deljenje je još kompleksnija operacija, jer se može implementirati upotrebom oduzimanja i `while` petlje sa bulovom logikom. Ukoliko se vrednost prve memorijske lokacije deli vrednošću druge memorijske lokacije, pseudokod je prikazan na listingu 8.

```
memorija[0] = deljenik
memorija[1] = delilac
memorija[2] = 0
memorija[3] = memorija[0]
memorija[4] = memorija[1]
while (memorija[3] ≥ memorija[4]) {
  memorija[2]++
  memorija[3] = memorija[3] - memorija[4]
}
```

Listing 8. Pseudokod za deljenje dve promenljive

Ovde je korišćena notacija za poređenje vrednosti dve memorijske lokacije iako ona još uvek nije objašnjena, ali je na kraju u grafu međuzavisnosti pokazano da je ovakva njena upotreba validna.

Najzad, nakon implementacije celobrojnog deljenja i ostalih operacija, lako je implementirati i ostatak pri deljenju, jer se on može predstaviti upotrebom ovih osnovnih artitmetičkih operacija, što je prikazano na listingu 9.

```
memorija[0] = deljenik
memorija[1] = delilac
memorija[2] = memorija[0] / memorija[1]
memorija[2] = memorija[2] * memorija[1]
memorija[2] = memorija[0] - memorija[2]
```

Listing 9. Pseudokod za računanje ostatka pri deljenju

Grananje

U jeziku C postoje dva glavna mehanizma za grananje koda – `if` grananje i `switch` grananje. Uprkos veoma različitoj sintaksi i situacijama gde ih ima smisla primeniti, `switch` grananje je lako pretvoriti u niz `if` grananja upotrebom `if` grane za svaki slučaj u `switch` grananju i korišćenjem `else` grane za slučajeve koji se završavaju upotrebom komande `break`. Na primer, dva koda prikazana na listingu 10 i listingu 11 su ekvivalentna:

```
switch (x) {
  case 'a':
    // Kod A
  case 'b':
    // Kod B
    break;
  case 'c':
    // Kod C
}
```

Listing 10. switch grananje u programskom jeziku C

```
if (x == 'a') {
  // Kod A
}
if (x == 'a' || x == 'b') {
  // Kod B
} else if (x == 'c') {
  // Kod C
}
```

Listing 11. Predstavljanje switch grananja upotrebom if grananja u programskom jeziku C

Imajući ovo na umu, nema potrebe direktno prevoditi `switch` grananje u Brejnfak, već je dovoljno predstaviti ga ovim nizom `if` grananja i onda ta grananja prevesti u Brejnfak.

Kada su `if` grananja u pitanju, prevođenje se sastoji iz dva dela:

1. Računanje vrednosti izraza u `if` uslovu
2. Biranja grane koda koja će se izvršiti

Izrazi bulove logike – koji čine `if` uslov – se mogu izračunati upotrebom Brejnfak koda, i ovo je pokazano kasnije u radu, pa je prvi korak moguće izvesti i njegov rezultat (odnosno vrednost 0 ili 1) smestiti u pomoćnu promenljivu koja će tokom određivanja grane koda biti modifikovana. U jeziku C ne postoje promenljive `bool` tipa, već se bilo koja vrednost različita od nule smatra istinitom, tako da grananja funkcionišu i sa vrednostima različitim od 0 ili 1, i to je ovde uzeto u obzir.

Ukoliko, nakon računanja vrednosti izraza u uslovu grananja, je ta vrednost smeštena u prvu memorijsku ćeliju – nezavisno od toga da li je u pitanju aritmetički ili izraz bulove logike – `if` grananje se može realizovati na način prikazan na listingu 12.

```
memorija[0] = vrednost izraza
memorija[1] = 1
while (memorija[0]) {
```

```
  memorija[0] = 0
  memorija[1] = 0
  // IF grana koda
}
while (memorija[1]) {
  memorija[1] = 0
  // ELSE grana koda, ukoliko postoji
}
```

Listing 12. Pseudokod za realizaciju if grananja

U ovom kodu, koriste se dve promenljive čije vrednosti određuju da li će biti izvršena `if` ili `else` grana koda. Ove grane koda su predstavljene upotrebom petlji, a te petlje se izvršavaju u zavisnosti od vrednosti dve promenljive. Prva promenljiva sadrži rezultat izraza iz uslova grananja, dok druga promenljiva sadrži 1, odnosno bilo koju pozitivnu vrednost. U ovom trenutku, postoje dve mogućnosti:

1. Ukoliko je vrednost prve promenljive pozitivna, izvršava se prva petlja koja predstavlja `if` granu, i u njoj se vrednosti obe promenljive postavljaju na nulu, ne bi li bilo izbegnuto ponavljanje prve petlje, kao i ulazak u drugu petlju, što je ekvivalentno preskakanju `else` grane.
2. Ukoliko je vrednost prve promenljive nula, prva petlja se neće izvršiti, što znači da će vrednost druge promenljive ostati 1, i usled toga biće izvršena druga petlja, odnosno `else` grana koda.

Bulova logika

Operacije bulove logike su jedan od najbitnijih aspekata programiranja, jer one omogućavaju menjanje ponašanja koda (tj. biranja dela koda koji će se izvršavati) u zavisnosti od raznih ulaznih parametera. Ove operacije mogu biti veoma kompleksne, jer se mogu kombinovati upotrebom komandi "i" i "ili (tj. `&&` i `||`), pa je njihovo prevođenje predstavljeno u nekoliko koraka.

Za početak, kako u programskom jeziku C ne postoje promenljive tipa `bool`, što znači da se bilo koja promenljiva može upotrebiti na mesti gde se očekuje logički izraz – ukoliko je njena vrednost različita od nule, izraz će se smatrati istinitim. U ovoj situaciji, nisu potrebne nikakve operacije bulove logike, i prevođenje upotreba ovih vrednosti u komandi `if`/`else` je već objašnjeno. Ovo je bitno, međutim, jer je za računanje izraza bulove logike neophodno korišćenje komande `if`.

Imajući mogućnost da se izvrši provera da li je određena vrednost jednaka nuli ili ne, prva stvar koju je moguće implementirati jeste negacija. Rezultat negacije pozitivne vrednosti je 0, a negacije nula 1, pa može biti korišćen kod prikazan na listingu 13.

```
memorija[0] = vrednost
if (memorija[0]) {
  memorija[1] = 0
} else {
  memorija[1] = 1
}
```

Listing 13. Pseudokod za realizaciju logičke negacije

Ova operacija je dalje predstavljena upotrebom notacije `!memorija[x]`.

Dalje, može se implementirati provera jednakosti dve promenljive (što istovremeno pokriva i proveru jednakosti između promenljive i konstante vrednosti, slično aritmetičkim operacijama). Ukoliko se rezultat provere jednakosti smešta u novu promenljivu, može biti korišćen kod prikazan na listingu 14.

```
memorija[0] = prva vrednost
memorija[1] = druga vrednost
memorija[2] = 0
while (memorija[0]) {
  memorija[0]--
  memorija[1]--
}
if (!memorija[0]) {
  if (!memorija[1]) {
    memorija[2] = 1
  }
}
```

Listing 14. Pseudokod za proveru jednakosti između dve promenljive

Prva petlja u ovom kodu koristi se da prvu vrednost umanji do nule, istovremeno umanjujući i drugu vrednost. Ukoliko su početne vrednosti jednake, one će istovremeno biti umanjene do nule, i onda će oba `if` uslova biti zadovoljena, a rezultat postavljen na 1. Kombinovanjem ovog koda i koda za negaciju moguće je implementirati i proveru različitosti dve promenljive.

Sledeće operacije koje mogu biti prevedene su `&&` i `||`. Ovo se postiže upotrebom ugnježdenih `if` grananja – za operaciju `&&` obe vrednosti moraju biti pozitivne, dok je za `||` potrebno da barem jedna vrednost bude pozitivna. Ovo je prikazano na listingu 15.

```
memorija[0] = prva vrednost
memorija[1] = druga vrednost
memorija[2] = 0 // Rezultat operacije &&
memorija[3] = 0 // Rezultat operacije ||
if (memorija[0]) {
  if (memorija[1]) {
    memorija[2] = 1
  }
}
if (memorija[0]) {
  memorija[3] = 1
}
if (memorija[1]) {
  memorija[3] = 1
}
```

Listing 15. Pseudokod za realizaciju logičkog "i" i "ili"

Najzad, neophodno je prevesti i operacije poređenja, odnosno `<`, `≤`, `>` i `≥`. Operacija `>` se može predstaviti upotrebom operacije `<` i zamenom pozicija promenljivih, dok se `≤` i `≥` mogu predstaviti kombinacijom `<`, `||` i `==` (odnosno `>`, `||` i `==`). Za operaciju `<` se može koristiti kod prikazan na listingu 16.

```
memorija[0] = prva vrednost
memorija[1] = druga vrednost
memorija[2] = 0 // Rezultat < poređenja
while (memorija[0] != 0 && memorija[1] != 0) {
  memorija[0]--
  memorija[1]--
}
if (memorija[0] == 0 && memorija[1] != 0) {
  memorija[2] = 1
}
```

Listing 16. Pseudokod za proveru relacije "manje od"

Petlje

Kao što je pomenuto, Brejnfak petlje su ekvivalentne veoma jednostavnoj `while` petlji u jeziku C, ali ovaj jezik pored ove podržava još dva tipa petlji. U pitanju su:

1. `while` petlja
2. `do...while` petlja
3. `for` petlja

Prevođenje same `while` petlje je veoma jednostavno, jer je jedini bitan deo računanje vrednosti samog uslova, što je već objašnjeno. Osim ovoga, neophodno je ponovo izračunati vrednost uslova pre kraja petlje, pa je pseudokod prevođenja `while` petlje prikazan na listingu 17.

```
memorija[0] = vrednost uslova
while (memorija[0]) {
  // Kod WHILE petlje
  memorija[0] = vrednost uslova
}
```

Listing 17. Pseudokod za realizaciju jednostavne while petlje

`do...while` petlja se ponaša veoma slično, s tim što se uslov proverava tek na kraju petlje. Posledica ovoga je da se petlja uvek izvršava barem jednom, i to se može postići postavljanjem inicijalne vrednosti izraza na 1, što je prikazano na listingu 18.

```
memorija[0] = 1
while (memorija[0]) {
  // Kod WHILE petlje
  memorija[0] = vrednost uslova
}
```

Listing 18. Pseudokod za realizaciju while petlje sa kompleksnim logičkim uslovom

Najzad, `for` petlja je nešto komplikovanija, ali ona zapravo može biti razložena na nekoliko komandi koje je sve moguće prevesti u Brejnfak i ovo je prikazano na listingu 19 i listingu 20.

```
for (inicijalizacija; uslov; izraz) {
  // Kod FOR petlje
}
```

Listing 19. Pseudokod strukture for petlje

```
inicijalizacija
while (uslov) {
  // Kod FOR petlje
  izraz
}
```

Listing 20. Pseudokod for petlje predstavljene upotrebom while petlje

Nakon ove transformacije i upotrebom prethodno objašnjenog prevođenja `while` petlji, `for` petlju je jednostavno prevesti u Brejnfak.

Ulaz/Izlaz

Pri prevođenju funkcije `printf()`, ukoliko se ispisuju određeni karakteri, funkciju je lako prevesti upotrebom niza komadi za ispis tih pojedinačnih karaktera. Međutim, takođe je potrebno podržati i ispis promenljivih, odnosno ispis vrednosti u formatu `%c` (tj. kao karakter za tom ASCII vrednošću) i u formatu `%d` (tj. kao broj).

Ispisivanje promenljivih u formatu `%c` je jednostavno, jer se ponovo svodi na direktnu upotrebu komande za ispis karaktera. Ispis brojevnih vrednosti je nešto komplikovaniji, jer je neophodno ispisati svaku cifru broja zasebno, i to učiniti u ispravnom redosledu. Ovo se postiže upotrebom petlje koja deli vrednost brojem 10 sve dok ne preostane jedna cifra, koja se onda ispisuje, a vrednost za sledeću iteraciju petlje postavlja na vrednost broja sa tom ispisanom cifrom uklonjenom, što je prikazano na listingu 21.

```
memorija[0] = vrednost
memorija[1] = memorija[0]
while (memorija[1]) {
  memorija[2] = 1
  while (memorija[1] > 9) {
    memorija[1] = memorija[1] / 10
    memorija[2] = memorija[2] * 10
  }
  putc(memorija[1] + '0')
  memorija[1] =
    memorija[0] - memorija[1] * memorija[2]
}
```

Listing 21. Pseudokod za ispisivanje brojevnih vrednosti

Slično, funkcija `scanf()` podržava učitavanje karaktera i brojeva – karakteri se direktno učitavaju upotrebom Brejnfak komande, dok se brojevi učitavaju u petlji koja, nakon što u ulazu naiđe na cifru, množi početnu vrednost brojem 10 i dodaje učitanu cifru, efektivno nižući nove cifre na do sada učitan deo tog broja, što je prikazano na listingu 22.

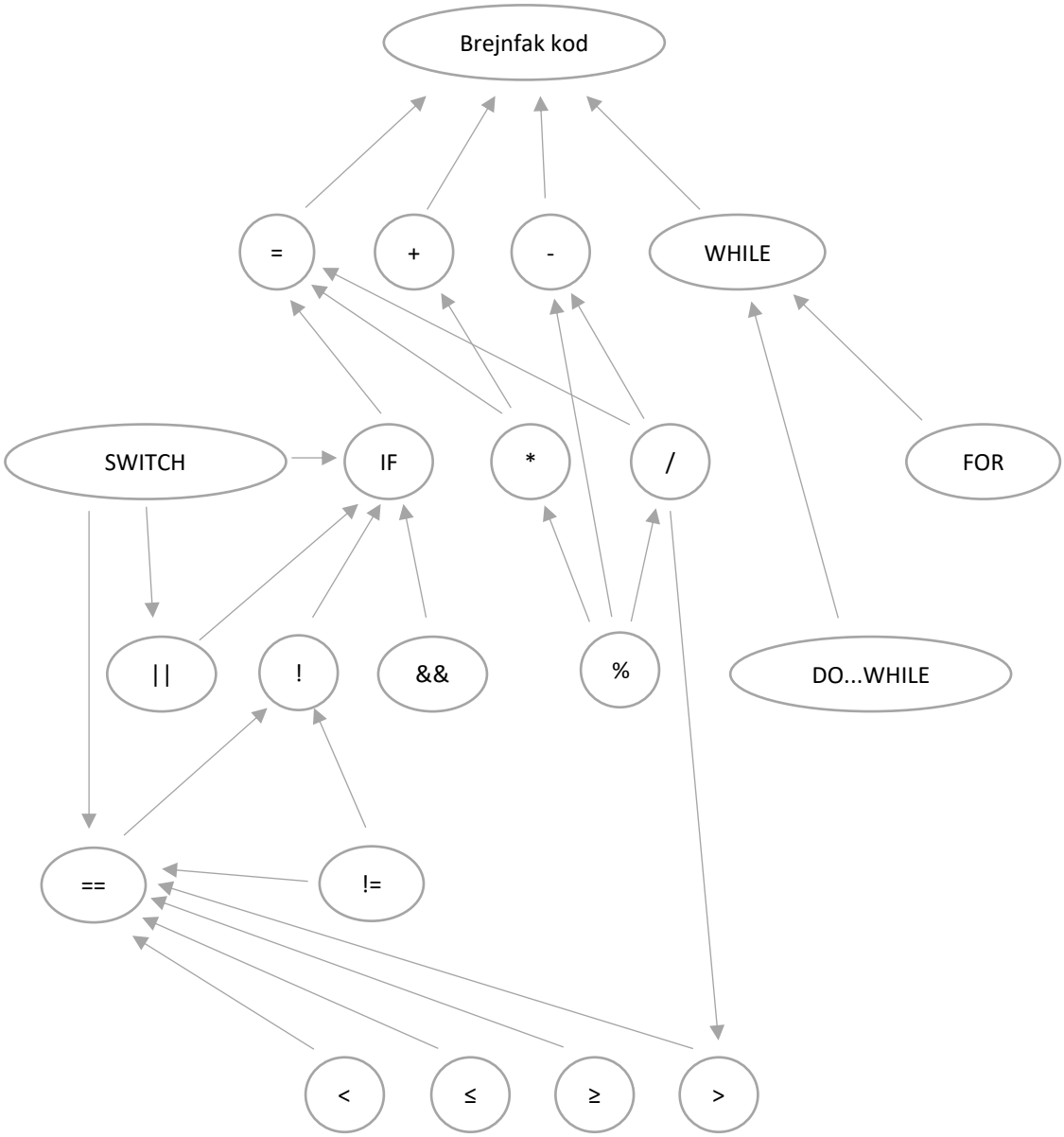
```
memorija[0] = 0
do {
  getc(memorija[1])
} while (
  memorija[1] < '0' || memorija[1] > '9'
)
while (
  memorija[1] >= '0' && memorija[1] <= '9'
) {
```

```
  memorija[0] = memorija[0] * 10
    + memorija[1] - '0'
  getc(memorija[1])
}
```

Listing 22. Pseudokod za učitavanje brojevnih vrednosti

Graf međuzavisnosti

Kao što je pokazano, neke od ovih funkcionalnosti moguće je prevesti razlaganjem na druge funkcionalnost – `switch` grananje se može predstaviti upotrebom `if` grananja, `for` petlja se može predstaviti upotrebom `while` petlje, itd. Na dijagramu 4 prikazan je graf međuzavisnosti korišćenih u ovom poglavlju ne bi li bilo pojašnjeno da se njihovom upotrebom sva funkcionalnost svodi na Brejnfak kod, odnosno da ne postoje ciklične zavisnosti.



Dijagram 4. Međuzavisnosti između implementacija različitih funkcionalnosti programskog jezika C

Implementacija prevodioca

Sistem za samo prevođenje iz programskog jezika C u programski jezik Brejnfak je u potpunosti implementiran upotrebom programskog jezika C++ [12], a zajedno sa dostupnom C++ bibliotekom sistema ANTLR.

Pri učitavanju koda napisanog u programskom jeziku C, kao što je pomenuto, prvi korak neophodan pre prevođenja predstavljaju leksiranje i parsiranje. Ovu funkcionalnost izvršava ANTLR, i to na osnovu prethodno napisane gramatike koja definiše leksička i parserska pravila. Međutim, ANTLR ne obrađuje ovu gramatiku dinamički, tj. pri obradi ulaznog koda, već se ona obrađuje statički pre same implementacije prevodioca, i ovaj proces automatski generiše četiri C++ klase:

1. klasu za leksiranje,
2. klasu za parsiranje,
3. osnovnu klasu za obilazak sintaksnog stabla i
4. izvedenu klasu za obilazak sintaksnog stabla.

Klase za leksiranje i parsiranje se generišu na osnovu prosleđene gramatike jezika i one definišu svu C++ logiku za izgradnju sintaksnog stabla određenog C koda. U ovom radu ove dve klase su posmatrane kao crna kutija, jer je rezultat njihovog korišćenja sintaksno stablo i nisu potrebne nikakve izmene u tom kodu (niti je predviđeno da se on menja) ne bi li ovo stablo bilo upotrebljivo za kasnije prevođenje.

Osnovna klasa za obilazak sintaksnog stabla predstavlja klasu sa isključivo apstraktnim funkcijama (tj. funkcijama bez implementacije) gde je definisana po jedna funkcija za svaki tip unutrašnjeg čvora (tj. čvora koji nije list) sintaksnog stabla. Ukoliko se posmatra definisana gramatika, generisana je po jedna funkcija za svako parsersko pravilo, ali ne i za leksička pravila – kao što je objašnjeno, leksička pravila su predstavljena listovima u sintaksnom stablu, jer su ona nedeljive celine koje predstavljaju tokene i koje dalje ne treba obilaziti rekurzivno.

Cilj ovih funkcija je da obezbede C++ logiku koja će se izvršavati u trenutku obilaženja čvora određenog tipa. Ne bi li se obišlo čitavno sintaksno stablo, neophodno je zasebno implementirati sve ove funkcije, što nije uvek praktično – na primer, čvor čija su deca funkcije napisane u jeziku C ne mora da ima nikakvu posebnu logiku, već je dovoljno da samo nastavi sa rekurzivnim obilaskom tih čvorova koji predstavljaju funkcije. Ne bi li razvoj prevodioca bio olakšan, generiše se i izvedena klasa za obilazak sintaksnog stabla koja nasleđuje osnovnu klasu i implementira sve funkcije upravo na ovaj način, odnosno pozivanje funkcije korena nad sintaksnim stablom rekurzivno poziva odgovarajuće funkcije za sve ostale čvorove.

Imajući ovu izvedenu klasu koja omogućava potpun obilazak sintaksnog stabla bez dodatne logike, moguće je selektivno implementirati isključivo one funkcije koje rezultuju u generisanju Brejnfak koda. Pri implementaciji ovih funkcija, dostupni su razni atributi trenutnog čvora, ali su za prevođenje bitna samo sledeća tri:

1. tip čvora,
2. kod (tj. tekst) na osnovu kog je čvor generisan i
3. niz čvorova koji su deca trenutnog čvora.

Tip čvora je određen samom funkcijom koja se implementira, pa se na osnovu njega određuje koja će se logika izvršiti; kod na osnovu kog je čvor generisan je bitan kod određenih pravila koja rade ili sa imenima promenljivih i funkcija, ili sa konstantnim vrednostima (bilo da su to brojevi ili karakteri); najzad, niz čvorova koji su deca trenutnog čvora je neophodan ne bi li se nastavio obilazak, a njihovi tipovi često mogu da utiču i na logiku trenutan čvora. Takođe, u zavisnosti od tipa čvora, ova funkcija može imati povratnu vrednost, pa tako neke funkcije vraćaju memorijsku lokaciju, dok neke vraćaju niz drugih čvorova, ili pak niz koji je mešavina vrednosti i memorijskih lokacija.

Na primer, funkcija koja obrađuje čvor sa brojevnom vrednošću će generisati Brejnfak kod za smeštanje te vrednosti u novu memorijsku lokaciju i onda tu lokaciju vratiti svom roditelju. Ukoliko je roditeljski čvor dodeljivanje vrednosti `a = 5`, on će obilaskom svoje dece dobiti memorijsku lokaciju promenljive `a`, kao i lokaciju gde je privremeno smeštena vrednost 5, i na osnovu toga generisati Brejnfak kod koji tu vrednost iz privremene lokacije premešta u lokaciju promenljive `a`.

U nekom drugom čvoru može se desiti obrada funkcije `printf()`, i u tom slučaju je neophodno obići čvorove koji predstsavljaju parametre funkcije i obraditi ih na određen način – prvi čvor će uvek biti format ispisa u tekstualnom obliku, dok će ostali čvorovi biti memorijske lokacije sa kojih treba uzeti vrednosti za ispis, pa je neophodno da se pri obilaženju tih čvorova vrate odgovorajuće vrednosti ne bi li bile ispravno upotrebljene na višim nivoima stabla.

Kao što je ranije objašnjeno, jednostavnije funkcionalnosti (koje su neretko na nižim nivoima sintaksnog stabla) implementirane su direktnim ispisivanjem Brejnfak koda, dok su kompleksnije funkcionalnosti implementirane kompozicijom ovih jednostavnijih.

Imajući sve ove elemente, odnosno implementacije funkcija, prevodilac funkcioniše tako što instancira klasu za parsiranje, prosleđuje joj ulazni C kod ne bi li dobio sintaksno stablo i onda instanciranoj klasi za obilazak sintaksnog stabla prosleđuje to stablo. Rezultat izvršavanja klase za obilazak sintaksnog stabla je krajnji Brejnfak kod sa istim ponašanjem kao i ulazni C kod.

Testiranje

Kako je Brejnfak ezoteričan programski jezik, veoma je teško proveriti korektnost određene implementacije isključivo čitajući kod, jer bi ovo zahtevalo puno vremena i najverovatnije bi podleglo raznim ljudskim greškama. Šta više, prevođenje iz jezika C u Brejnfak je – kao što je pokazano – veoma kompleksno, i samim tim postoje desetine graničnih slučajeva koji neretko nisu očigledni, ali je njihovo korektno obrađivanje neophodno za uspešan rad prevodioca.

Iz ovih razloga, nameće se jednostavna odluka da bi prevodilac i sve prataće komponente trebalo testirati automatski, ne bi li se postigla veća efikasnost, pokrio što veći broj gorepomenutih graničnih slučajeva i smanjio broj problema u daljem razvoju. Naime, ukoliko se pri svakoj promeni izvršava čitav skup testova, neće se desiti da nov kod prouzrokuje probleme sa prethodnim, korektnim ponašanjima. Ukoliko se, pak, desi da nov kod prouzrokuje probleme sa određenim ponašanjima koja nisu pokrivena postojećima testovima, lako je dodati nov test koji bi se onda uvek izvršavao u budućnosti.

Sam prevodilac predstavlja samo jedan deo implementiranog sistema, pa je tako izvršeno testiranje tri različita dela:

1. Interpreter. Ne bi li bilo moguće testirati korektnost C programa prevednih u Brejnfak, neophodno je da taj program može da se izvrši, pa je za ovo u okviru sistema implementiran jednostavan Brejnfak interpreter.
2. Formater. Iako, kao što je napomenuto, čitanje koda nije najprecizniji način za njegovu validaciju, za jednostavnije programe i njihovo lakše razumevanje može biti korisno ukoliko je kod formatiran na određen način, pa je tako implementiran Brejnfak formater.
3. Prevodilac. Najzad, prevodilac o kojem je već bilo reči je glavni deo sistema, a samim tim i najkompleksniji, pa je za njega testiranje najbitnije.

Testiranje interpretera

S obzirom da je funkcionalnost jezika Brejnfak veoma jednostavna, ne postoji potreba da se kod prevodi u neki drugi (npr. mašinski) jezik, već se komande mogu obrađivati sekvencijalno u realnom vremenu, što je upravo definicija interpretera.

Kao što je već objašnjeno, okosnicu izvršavanja Brejnfak programa predstavlja memorija teoretski neograničene dužine koja se sastoji od ćelija veličine 1 bajt i pokazivača

koji kreće od nulte lokacije i može da se pomera ka lokacijama većeg ili manjeg indeksa – za razliku od stvarne memorije, ova memorija može imati i negativne indekse.

Imajući ovo na umu, zadatak interpetera je da simulira ovu memoriju i vrši Brejnfak komande nad njom. Interpreter je implementiran upotrebom programskog jezika C++, pa je tako bilo moguće koristiti običnu heš mapu koja mapira cele brojeve na karaktere, odnosno `map<int, char>`. Pored ovoga, korišćena je i jedna celobrojna vrednost za trenutnu lokaciju pokazivača, i ona je inicijalizovana sa vrednošću 0. Ova struktura podataka nam omogućava veoma jednostavno interpretiranje većine Brejnfak komandi koje se može predstaviti C++ kodom prikazanim na listingu 23.

```
void interpretiraj(char komanda) {
  map<int, char> memorija;
  int pokazivac = 0;
  char c;
  switch (komanda) {
    case '<':
      --pokazivac;
      break;
    case '>':
      ++pokazivac;
      break;
    case '+':
      ++memorija[pokazivac];
      break;
    case '-':
      --memorija[pokazivac];
      break;
    case ',':
      cin.get(memorija[pokazivac]);
      break;
    case '.':
      cout.put(memorija[pokazivac]);
      break;
  }
}
```

Listing 23. Deo Brejnfak interpretera implementiranog u programskom jeziku C

Međutim, kod interpretiranja petlji, odnosno komandi `[` i `]`, javlja se dodatna kompleksnost, jer se komande ne mogu obrađivati sekvencijalno i onda odbaciti, već je neophodno čuvati ih za ponovnu obradu ukoliko se one nalaze u okviru neke petlje. Ovo se postiže upotrebom niza u kojem se čuvaju sve komande počevši od početka prve petlje (ovaj

niz se može smatrati svojevrsnim baferom[13]) i steka[14] na koji se dodaju indeksi početka petlji ne bi li bilo moguće vratiti se do odgovarajuće komande kada se dođe do kraja petlje. Ukoliko je petlja završila se izvršavanjem, indeks njenog početka se uklanja sa vrha steka, a kada stek postane prazan briše se i čitav niz komandi i vraća na sekvencijalno interpretiranje komandi bez njihovog čuvanja, jer one nisu u okviru petlje.

Pojasnivši ove detalje, javlja se nekoliko oblasti za koje su napisani automatski testovi:

1. Čitanje i ispis vrednosti, što istovremeno zahteva i testiranje pomeranja pokazivača, kao i menjanja memorijskih vrednosti.
2. Pristup memoriji sa negativnim indeksima.
3. Rad se petljama, kako jednostavnim tako i ugnježdenim.

Pri testiranju interpretera, ulazne podatke svakog testa predstavljaju Brejnfak kod, ulaz za program, i njegov očekivani izlaz.

Testiranje formatera

Jedan od glavnih aspekata bilo kojeg većeg projekta, bilo da na njemu radi jedna ili više osoba, jeste jasan i pregledan kod koji je u budućnosti lako menjati. Nažalost, zbog prirode jezika Brejnfak, ovaj kod svakako neće biti veoma jasan, ali ovu prepreku je delimično moguće savladati preglednošću, odnosno automatskim formatiranjem koda.

Zajedno sa jednostavnim funkcionalnostima Brejnfaka, i sama implementirana logika za formatiranje je veoma jednstavna, i sastoji se od nekoliko pravila:

1. Osnovu preglednosti koda predstavljaju linije koje su više uvučene od drugih, i u većini jezika uvlače se linije koje su u okviru zagrada. U slučaju Brejnfaka, ovo možemo primeniti na uglaste zagrade (tj. petlje), pa će tako početak i kraj petlje biti postavljeni u posebne redove, a sve linije između će na svom početku imati dva razmaka više nego linije van petlje. U slučaju ugnježdenih petlji, dodaju se po dva razmaka za svaki nivo dubine.
2. Komande za pomeranje pokazivača se postavljaju u novi red, osim ukoliko im prethodi ista komanda, u kojem slučaju se one ispisuju jedna do druge.

[13] Bafer (*eng.* buffer) – struktura podataka u kojoj se podaci privremeno čuvaju za kasnije obrađivanje.

[14] Stek (*eng.* stack) – struktura podataka u kojoj se vrednosti ubacuju redom, a pri izbacivanju vrednosti one se izbacuju u suprotnom redosledu.

3. Komande za čitanje, ispis, i promenu vrednosti memorije se uvek ispisuju u nastavku trenutnog reda.

Program koji ispisuje tekst "Hello, World!" bi, na osnovu ovih pravila, bio formatiran na način prikazan na listingu 24.

```
++++++++
[
  >++++
  [
    >++
    >+++
    >+++
    >+
    <<<<-
  ]
  >+
  >+
  >-
  >>+
  [
    <
  ]
  <-
]
>>.
>---.+++++++..+++.
>>.
<-.
<.+++.----- -.----- ---.
>>+.
```

Listing 24. Brejnfak program koji ispisuje "Hello, World!"

Pri testiranju formatera, ulazne podatke svakog testa predstavljaju Brejnfak kod i njegova formatirana varijanta koju bi formater trebalo da vrati kao izlaz.

Testiranje prevodioca

Kako je prevodilac centralni deo sistema i glavni deo ovog rada, testiranje njegove korektnosti je najbitniji deo sistema za testiranje, pa su tako datoteke sa ulaznim podacima veoma raznovrsne. Da bi moglo biti rečeno da prevodilac ispravno funkcioniše, neophodno je da se testiraju barem sledeći aspekti:

- Deklarisanje promenljivih i dodela vrednosti
- Aritmetičke operacije
    - Sabiranje, oduzimanje, množenje, deljenje i ostatak pri deljenju
    - Između dva broja, promenljive i broja ili dve promenljive
- Bulove operacije
- Učitavanje i ispis karaktera
- `if` i `switch` grananja
- `for` i `while` petlje

Na prvi pogled, jednostavno je sve ovo uključiti u jedan program, jer su zaista u pitanju jednostavne operacije koje bi se našle u bilo kakvom iole kompleksnijem kodu, ali bi to umanjilo vrednost koju testiranje pruža. Umesto toga, potrebno je napisati zaseban test za svaku (koliko god malu) funkcionalnost – na primer, jedan takav test bi se sastojao od sabiranja dva broja, smeštanja rezultata u promenljivu, i ispisa tog rezultata. Imajući takvu organizaciju testova, daleko je jednostavnije odrediti u kom delu prevodioca se nalazi problem koji uzrokuje probleme sa određenim testom.

Jedna ideja koja se nameće jeste da se ovo testiranje vrši tako što se kod napisan u jeziku C prevede u Brejnfak i onda uporedi sa očekivanim Brejnfak kodom. Ne bi li bili izbegnuti problemi sa formatiranjem, oba Brejnfak koda mogu biti puštena kroz formater, što bi ih onda dovelo do identičnog oblika.

Međutim, cilj prevodica nije da dovede do veoma precizno definisanog Brejnfak koda, već do koda koji funkcioniše isto kao i onaj napisan u C jeziku. Tokom bilo kojih daljih unapređena prevodioca može se desiti da se promeni način na koji se određena funkcionalnost jezika C predstavlja u jeziku Brejnfak (npr. na optimalniji način koji zahteva manje komandi) i kao posledica ovoga veliki broj testova bi verovatno prestao da radi.

Imajući ovo na umu, odabran je pristup u kojem se testira isključivo izlaz programa za određeni ulaz, i ovaj izlaz – za razliku od Brejnfak koda – mora da bude identičan očekivanom. Ne bi li ovakav test bio implementiran, neophodne su dve datoteke:

1. Izvorni kod programa u jeziku C
2. Ulazni podaci za taj program

Definisanje datoteke sa očekivanim izlazom nije potrebno, jer je sistem za testiranje u mogućnosti da, nakon kompilacije, pokrene program napisan u jeziku C i sačuva njegov izlaz za kasnije poređenje. Samim tim, testiranje prevodioca se sastoji iz sledećih koraka:

1. Izvorni kod programa u jeziku C se kompajlira i pokreće sa datim ulaznim podacima.
2. Izlaz ovog pokretanja se čuva za kasnije poređenje.

3. Izvorni kod programa u jeziku C se prosleđuje prevodiocu koji vraća Brejnfak kod za ovaj program.
4. Brejnfak kod se prosleđuje interpreteru zajedno sa istim ulaznim podacima iz koraka (1).
5. Izlaz ovog pokretanja se poredi za izlazom sačuvanim u koraku (2).

Implementacija

Za realizaciju sistema za testiranje korišćen je programski jezik Pajton [13] usled svoje jednostavne sintakse koja, zajedno sa velikim brojem biblioteka, omogućava brz razvoj samog sistema i lako dodavanje novih testova.

Za početak, za svaki od tri skupa testova određena je putanja na disku u kojoj će podaci za te testove biti smešteni. Nazivi datoteka sa ovim podacima mogu da se razlikuju između testova, ali u okviru jednog testa svi nazivi moraju da budu isti, sa isključivo različitim ekstenzijama. Za testove su odabrane sledeće šeme:

1. Interpreter
    - `x.bf` predstavlja izvorni kod Brejnfak programa
    - `x.in` predstavlja ulazne podatke prosleđene programu
    - `x.out` predstavlja izlazne podatke očekivane nakon izvršavanja programa
2. Formater
    - `x.bf` predstavlja izvorni kod Brejnfak programa
    - `x.out` predstavlja očekivani formatiran kod ovog programa
3. Prevodilac
    - `x.c` predstavlja izvorni kod C programa
    - `x.in` predstavlja ulazne podatke prosleđene programu

Nakon što je sistem za testiranje pokrenut, on za svaki od skupova testova pronalazi sve datoteke sa odgovarajućom ekstenzijom, i na osnovu njih traži datoteke sa ostalim ekstenzijama. Ukoliko su za određeno ime pronađene datoteke sa svim neophodnim ekstenzijama, testovi se izvršavaju nad tim podacima i korisniku se prikazuje rezultat testiranja.

Pri izvršavanju testova, glavno pitanje na koje treba odgovoriti jeste da li je izlaz programa isti kao očekivan izlaz. Međutim, može biti korisno i da, osim te binarne informacije, sistem za testiranje prikaže i informacije o tome kako se očekivani izlaz razlikuje od dobijenog.

Za ovo je upotrebljena biblioteka `difflib` koja analizira redove teksta u dobijenom i očekivanom izlazu i prikazuje koji redovi su uklonjeni ili dodati u jednoj od ove dve datoteke.

Upotrebom ove biblioteke, pogrešno formatiran kod bi bio ispisan na način prikazan u tabeli 7.

Očekivan kod	Dobijen kod	difflib prikaz

`>++++`
`[`
`  >++`
`  >+++`
`  >+++`
`  >+`
`  <<<<-`
`]`
	`>++++`
	`[`
	`  >++`
	`  >+++>+++`
	`  >+`
	`  <<<<-`
	`]`
		`- >+++`
		`- >+++`
		`+ >+++>+++`

Tabela 7. Primer funkcionisanja Pajton biblioteke difflib

Dalja unapređenja i optimizacije

Jedan od aspekata ovog prevodica koji nije analiziran jeste veličina prevedenog koda u bajtovima. S obzirom da Brejnfak izvršava kompleksne operacije upotrebom veoma jednostavnih komandi, za očekivati je da ovaj kod bude primetno veći od, na primer, binarne datoteke dobijene kompilacijom C koda. Međutim, iako fokus ovog ranije nije bio na efikasnosti (kako prostornoj, tako ni vremenskog), razmotreno je nekoliko metoda čijom upotrebom bi Brejnfak kod dobijen prevođenjem mogao značajno da se pojednostavi. Pored toga, razmotreni su i određeni pristupi za prevođenje kompleksnijih funkcionalnosti koje bi dalje unapredile ovaj sistem.

Tipovi promenljivih

Glavni problem podskupa funkcionalnosti jezika C obrađenog u ovom radu jeste to što je on ograničen na korišćenje isključivo promenljivih `char` tipa, jer su one veličine 1 bajt, što je istovremeno i veličina memorijske ćelije. Međutim jezik C pored ovog tipa podržava i razne druge tipove od koji su najbitniji `int` (celobrojna vrednost) i `float` (realna vrednost), kao i njihove varijacije koje koriste različite brojeve bajtova.

Ne bi li podrška za ove promenljive bila imprementirana, prevodilac mora da koristi po nekoliko ćelija za predstavljanje ovih tipova podataka. Broj ovih ćelija će zavisiti od arhitekture procesora [14] – na primer, za int tip je garantovano da se čuva u 4 bajta, odnosno 4 Brejnfak ćelije, ali će na 64-bitnim procesorima ova vrednost biti veća, i biće korišćeno 8 ćelija.

Nezavisno od samog broja ćelija, operacije koje bi inače bile obavljane nad jednom ćelijom mogu se prilagoditi većem broju ćelija tako što se one posmatraju kao cifre broja zapisanog u sistemu sa osnovom 256. Na primer:

- Ukoliko se proverava jednakost dva broja od 4 bajta (odnosno 4 ćelije), ovo se izvršava proverom jednakosti zasebnih ćelija u svakom od brojeva.
- Ukoliko se proverava relacija između brojeva, počinje se proverom te relacije na prvoj ćeliji, i ukoliko su te vrednosti jednake, proces se ponavlja za sledeću ćeliju, itd.
- Ukoliko se vrši sabiranje dve vrednosti, počinje se sabiranjem poslednje ćelije, i ukoliko se detektuje premašivanje vrednosti 255 (što je slučaj ukoliko je dobijena vrednost manja od jednog ili oba sabirka) sledeća ćelija se uvećava za

1 i postupak se ponavlja – ovo su upravo koraci koji se koriste za ručno sabiranje dva broja u decimalnom sistemu, itd.

Rad sa realnim brojevima je nešto kompleksniji, zbog formata u kojem se oni čuvaju u memoriji, ali se slično može razložiti na više jednostavnijih operacija.

Aritmetičke operacije

U radu su obrađene samo osnovne aritmetičke operacije, ali njihovom upotrebom lako je implementirati i ostale, kao što su:

- Stepenovanje, koje je moguće implementirati ponovljenim množenjem.
- Korenovanje, koje je moguće implemnetirati upotrebom Njutnovog metoda.
- Logaritam, koji je takođe moguće implementirati upotrebom Njutnovog metoda.
- Trigonometrijske funkcije, koje je moguće implementirati kombinacijom sačuvanih trigonometrijskih tablica za određene uglove i Volderovog algoritma.

Osim ovoga, moguće su i optimizacije postojećih aritmetičkih operacija, odnosno rada sa brojevima. Na primer, pri ubacivanju vrednosti X u promenljivu, prevodilac će upotrebiti X komandi tipa `+`, ali kod većih brojeva ovo se može realizovati upotrebom množenja, što bi dovelo do manjeg broja komandi. Na primer, vrednost 200 se može izračunati upotrebom izraza $14 \times 14 * 4$, pa bi umesto 200 komandi tipa `+` bio korišćen kod prikazan na listingu 25.

```
// Prvi množilac, tj. 14
++++++++++++++
[
  // Umanjivanje prvog množioca
     radi kontrole petlje
  -
  // Uvećavanje rezultata za
     14, tj. drugi množilac
  >++++++++++++++<
]
// Dodavanje 4 rezultat množenja
>++++
```

Listing 25. Primer optimizovanja koda za dodelu vrednosti promenljivoj

U ovom obliku, upis broja 200 zahteva samo 38 komandi.

Funkcije

Funkcije u programskom jeziku C imaju 3 različita apsekta koji ih čine korisnima:

1. Kod je laške organizovati ukoliko su funkcionalne celine izdvojene u zasebne blokove.
2. Upotrebom komande `return` jednostavno je prekinuti izvršavanje funkcije – i vratiti vrednost, ukoliko je to potrebno – pre kraja koda.
3. Pozivanjem funkcije unutar iste te funkcije moguće je izvršiti rekurziju.

Prvi od ovih aspekata nema direktnog uticaja na prevođenje u Brejnfak, ali može informisati potencijalan postupak prevođenja. Naime, kako je ova bolja organizacija koda namenjena korisniku (tj. programeru), nema potrebe da ona bude očuvana nakon prevođenja u Brejnfak, pa se ovo može postići jednostavnom zamenom poziva funkcije samim kodom funkcije – ovo je čak i podržano u jeziku C, gde je moguće dodati ključnu reč `inline` koja kompilatoru govori da poziv funkcije zameni njenim telom radi bržeg izvršavanja.

Za razliku od prvog, drugi aspekat ima uticaja na prevođenje C koda, jer u jeziku Brejnfak ne postoji koncept vraćanja vrednosti iz funkcije, odnosno prekidanja izvršavanja programa – program se uvek izvršava od početka do kraja, sekvencijalno. Međutim, iako je komanda `return` specifična za funkcije, ona se u suštini može zameniti smeštanjem povratne vrednosti u određenu promenljivu i komandom `goto` koja bi onda skočila na kraj pozvane funkcije i tu izazvala vraćanje prethodno sačuvane povratne vrednosti [15].

Jedan od načina implementiranja `goto` komande zahteva da se kod prvo podeli u disjunktne celine između kojih se nalaze ove `goto` komande i odgovarajuće labele (tj. destinacije). Postupak je onda sledeći:

1. Svakoj ovoj celini dodeli se indeks, i to sekvencijalno počevši od vrednosti 1.
2. Uvede se promenljiva koja označava koju celinu treba izvršiti u sledećem koraku, i njena vrednost je 0 ukoliko je program završen.
3. Svaka celina se smesti u `if`/`else` grananje kojim se proverava da li ta celina treba da se izvrši. Ukoliko ne treba, vrednost promenljive se umanjuje i prelazi se na sledeću celinu.

Ovo ponašenje se može pokazati na jednostavnom primeru sa pseudokodom prikazanim na listingu 26.

```
labela1:
  // Kod
  GOTO labela2
```

```
  // Kod
labela2:
  // Kod
```

Listing 26. Pseudokod za korišćenje komande goto

Podelom koda u blokove dobija se kod prikazan na listingu 27.

```
indeks = 1

while (indeks > 0) {
  indeks = index - 1

  if (indeks > 0) {
    indeks = index - 1
  } else {
    // Kod prve celine
    // Između prve labele i GOTO poziva
    sledeciIndeks = brojPreostalihCelina + 1
  }

  if (indeks > 0) {
    indeks = index - 1
  } else {
    // Kod druge celine
    // Između GOTO poziva i druge labele
  }

  if (indeks > 0) {
    indeks = index - 1
  } else {
    // Kod treće celine
    // Nakon druge labele
  }
}
```

Listing 27. Pseudokod za realizaciju komande goto upotrebom petlji i grananja

Na samom početku, indeks je postavljen na 1, jer izvršavanje počinje od prve celine. Nakon ulaska u petlju, ovaj indeks se umanjuje za 1 ne bi li sledeće grananje ispravno funkcionisalo. Ukoliko početni indeks ima veću vrednost (npr. usled upotrebe `goto` komande), celine će biti preskakane sve dok taj indeks ne postane 0, i u tom trenutku će se izvršiti odgovarajuća celina. Pritom, sve celine nakon nje će takođe biti izvršene, jer će vrednost indeksa ostati 0, što i jeste očekivano ponašanje – pri izvršavanju koda labele se ignorišu, odnosno ne utiču na tok programa.

Međutim, za razliku od labela, sami `goto` pozivi utiču na tok programa, pa je, u slučaju kada je celina završena `goto` pozivom, neophodno postaviti indeks na vrednost ciljane celine uvećane za broj preostalih celina u programu. Ovo uvećanje je neophodno, jer će interpreter nastaviti sa izvršavanjem ostalih grananja, umanjujući vrednost indeksa sve dok ne dođe do kraja petlje, i u tom trenutku će vrednost indeksa biti tačna broj ciljane celine.

Imajući ovu implementaciju komande `goto`, ona takođe omogućava i realizaciju trećeg bitnog aspekta funkcija, odnosno podršku za rekurziju. Glavni problem sa do sada opisanim pristupom jeste to što bi samo prevođenje rekurzivne funkcije izazvalo rekurzivne pozive tokom prevođenja i dovelo do rekurzije prevelike dubine i premašivanja kapaciteta steka. Međutim, sama činjenica da rekurzija kao koncept ne postoji nakon prevođenja programa u mašinski kod govori da bi je moglo biti moguće implementirati i u jeziku Brejnfak. Ovime se, između ostalog, bavi i Čurč-Tjuring hipoteza koja, doduše, još uvek nije dokazana.

Ukoliko se posmatra način funkcionisanja rekurzije, njena glavna osobina je upotreba steka za smeštanje parametera koji su prosleđeni funkciji, ali i pokazivača na deo koda gde izvršavanje treba da se vrati nakon završetka rekurzivnog poziva. Stek je prilično jednostavna struktura podataka koju je moguće implementirati upotrebom funkcionalnosti koje su već objašnjene, ali se problem javlja kod skakanja na određeni deo koda, jer Brejnfak ovo ne podržava – i za to se može koristiti prethodno objašnjena implementacija komande `goto`.

Statička analiza

Najzad, jedan od bitnih aspekata kompilatora koji rade sa naprednijim jezicima je statička analiza. Ovaj proces predstavlja analizu ponašanja i toka određenog programa bez izvršavanja samog koda – u slučaju izvršavanja koda, u pitanju bi bio proces dinamičke analize.

Statička analiza ima razne upotrebe, kao što su bojenje sintakse koda radi lakšeg razvoja, pa čak i formalno dokazivanje korektnosti programa. Aspekat statičke analize koji bi mogao doprineti optimizaciji ovog prevodioca jeste identifikovanje delova C koda koji mogu biti pojednostavljeni.

U pitanju je pristup koji većina kompilatora uveliko primenjuje ne bi li veličina programa bila što manja, i zasniva se na principima različitih složenosti. Neki od primera bi bili identifikovanje aritmetičkih izraza koji koriste isključivo konstante vrednosti i koji se, samim tim, mogu izračunati i pre izvršavanja programa, ili identifikovanje grana programa koje se (često greškom ili nepažnjom programera) nikada neće izvršiti. Upravo ovi procesi bi mogli znatno umanjiti veličinu Brejnfak koda, jer je, kao što je pokazano, i za veoma jednostavne C komande koje bi ovim procesom bile dalje pojednostavljene ili uklonjene neophodna nesrazmerno velika količina Brejnfak koda.

Zaključak

Kao što je prikazano kroz razne primere, kako Brejnfak koda tako i same logike za izvršavanje kompleksnijih operacija, ovaj jezik je veoma daleko od toga da bude upotrebljiv u bilo kojoj realnoj primeni – što je i za očekivati, jer je jezik razvijen sa ciljem da bude minimalistički i doveden do apsurda. Međutim, uprkos ovome, pokazano je da je sasvim moguće predstaviti podskup funkcionalnosti jezika C upotrebom jezika Brejnfak, što stvara sasvim korisnu osnovu za implementiranje i još kompleksnijih funkcionalnosti.

Ovo kombinovanje operacija (odnosno razlaganje kompleksnih operacija na jednostavnije) je već uveliko primenjeno u ovom radu. Počevši isključivo od komandi jezika Brejnfak, moguće je izgraditi jednostavan kod za dodelu promenljivih i aritmetičke operacije. Koristeći ove, sada dostupne, gradivne blokove, mogu se uvesti grananja, a onda upotrebom grananja i petlje. Dalje, sa ovim blokovima je lako uvesti podršku za promenljive veće od jednog bajta, pa čak i nizove, rekurziju, itd.

Iako je u ovom radu korišćena u teoriji neograničena količina memorije, ovo nije preduslov za uspešan razvoj potpunog prevodioca iz jezika C u jezik Brejnfak, jer moderni računari obrađuju podatke upotrebljujući velike količine radne memorije (reda veličine nekoliko desetina gigabajta), a takođe imaju i mehanizme za virtuelno proširenje te memorije, pa bi nedostatak istinski neograničene memorije retko – ako ikada – bio realan problem.

Najzad, ovaj rad za cilj nije imao formalan dokaz Tjuring potpunosti jezika Brejnfak, ali se većina analiza i primera bavi činjenicom da je moguće prevesti kompleksan, široko korišćen jezik u Brejnfak, što je, u slučaju Tjurning potpunog jezika kao što je C, svojevrsan informalan dokaz da je Brejnfak takođe Tjuring potpun.

Literatura

[1] M. Campbell-Kelly, W. Apsray, 2004. *Computer: A History of the Machine*. Westview Press, Boulder, Colorado, USA.

[2] J. Steinberg, 2013. *"Hello, World!": The History of Programming*. CreateSpace Independent Publishing Platform, Scotts Valley, California, USA.

[3] B. Raiter. *The Brainfuck Programming Language*. http://www.muppetlabs.com /~breadbox/bf/, 26.9.2018.

[4] J.E. Hopcroft, R. Motwani, J.D. Ullman, 2007. *Automata Theory, Languages, and Computation (3rd Edition).* Pearson Education Inc., Boston, Massachusetts, USA.

[5] Z. Ognjanović, N. Krdžavac, 2004. *Uvod u Teorijsko Računarstvo*. Fakultet Organizacionih Nauka, Belgrade, Serbia.

[6] T.Æ. Mogensen, 2010. *Basics of Compiler Design (Anniversary Edition)*. University of Copenhagen, Copenhagen, Denmark.

[7] M.L. Scott, 2009. *Programming Language Pragmatics (3rd Edition)*. Morgan Kaufmann Publishers, Burlington, Massachusetts, USA.

[8] W.M. Waite, G. Goos, 1984. *Compiler Construction*. Springer-Verlag, Berlin, Germany.

[9] B.W. Kernighan, D. Ritchie, 1988. *The C Programming Language (2nd Edition)*. Prentice Hall, Upper Saddle River, New Jersey, USA.

[10] T. Parr, 2013. *The Definitive ANTLR 4 Reference (2nd Edition)*. Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[11] T. Parr. *GitHub - antlr/antlr4: ANTLR (ANother Tool for Language Recognition)*. https://github.com/antlr/antlr4, 26.9.2018.

[12] B. Stroustrup, 2014. *Programming: Principles and Practice Using C++ (2nd Edition)*. Addison Wesley, Boston, Massachusetts, USA.

[13] Python Software Foundation. *The Python Language Reference*. https://docs.python.org /2/reference/index.html, 26.8.2018.

[14] W. Stallings, 2015. *Computer Organization and Architecture (10th Edition)*. Pearson Education Inc., Boston, Massachusetts, USA.

[15] A.V. Aho, M.S. Lam, R. Sethi, J. Ullman, 2006. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison Wesley, Boston, Massachusetts, USA.