5. Četvrta laboratorijska vježba

Tema četvrte laboratorijske vježbe je generiranje koda. Vježba se u velikoj mjeri naslanja na treću laboratorijsku vježbu, ali i djelomično riješena treća vježba može poslužiti kako bi se djelomično riješila četvrta vježba.

U praksi je generiranje *kvalitetnog* koda (učinkovitog po raznim parametrima) vrlo složen problem koji se i sam tipično obavlja u više faza, kao što je i objašnjeno u udžbeniku i na predavanjima. U ovoj laboratorijskoj vježbi je generiranje koda maksimalno pojednostavljeno, pa se tako ne ocjenjuje kvaliteta generiranog koda nego isključivo njegova točnost.

Jedan od problema u prijašnjim inačicama ovih laboratorijskih vježbi bila je činjenica da većina studenata ili ne zna ili slabo poznaje mnemonički jezik nekog stvarnog procesora. Kako bi se izbjegao taj problem, u ovoj vježbi će se generirati mnemonički jezik procesora FRISC koji se uči na predmetu *Arhitektura računala* FER-2 programa. Simulator procesora FRISC u web-pregledniku dostupan je ovdje[1]. Isti simulator može se pokrenuti i iz naredbenog retka, koristeći Node.js platformu. Uz instaliran Node.js, bit će potrebno u jedan direktorij dohvatiti i datoteke friscasm.js, friscjs.js i main.js. Ako je FRISC mnemonički program zapisan u datoteci `a.frisc`, program se može simulirati naredbom `node main.js a.frisc`.

Alternativno, za ispitivanje ispravnosti generiranih programa može se koristiti i ATLAS kao na predmetu *Arhitektura računala*. Ako naiđete na neke razlike u funkcionalnosti između ATLAS-a i gore opisanog simulatora, javite se na e-mail listu predmeta *ppj@zemris.fer.hr*.

5.1 Ulaz i izlaz generatora koda

Ulaz u četvrtu laboratorijsku vježbu bit će isti kao ulaz u treću vježbu, tj. generativno stablo *ppjC* programa. Važno je uočiti da je ulaz "obično" generativno stablo i nad njim treba provesti semantičku analizu. Svi zadani programi bit će semantički ispravni, ali je semantičku analizu ipak potrebno provesti kako bi bilo moguće generirati kod. Iz tog razloga je predviđeni način rješavanja ove laboratorijske vježbe proširenje rješenja treće laboratorijske vježbe u kojoj se provodi semantička analiza.

Rješenje četvrte laboratorijske vježbe treba zapisati generirani mnemonički program za

[1] Preciznije rečeno, uneseni program pisan mnemoničkim jezikom procesora FRISC prevodi se mnemoničkim asemblerom u strojne naredbe procesora FRISC. Generirane strojne naredbe pune se u memoriju od adrese `0` (ako nije drugačije zadano asemblerskom naredbom `` `ORG ``) i simulira se rad procesora FRISC. U nastavku upute ćemo zbog jednostavnosti o cijelom ovom postupku govoriti kao o simulatoru.

procesor FRISC u datoteku `a.frisc` u istom direktoriju u kojem se nalazi sam generator koda. Kako zbog jednostavnosti jezik *ppjC* ne podržava unos i ispis podataka, kao rezultat izvođenja programa promatrat će se povratna vrijednost iz funkcije `main` koju generirani mnemonički program prije izvođenja instrukcije `HALT` mora zapisati u registar `R6`.

Važna napomena: Iz prethodno navedenog razloga, moguće je "optimizirati" cijeli ulazni program u dvije instrukcije—jednu koja puni registar `R6` rezultatom izvođenja i jednu `HALT` instrukcij—tako da generator jednostavno odsimulira ulazni program. Iako bi se moglo reći da je to najbolji mogući generirani kod za zadani program, to očito nije smisao ove laboratorijske vježbe i takva rješenja bit će bodovana s 0 bodova. S druge strane, dozvoljeno je (iako ne i nužno) tijekom prevođenja izračunati konstante izraze, tj. izraze koji sadrže samo konstante i eventualno `const`-kvalificirane varijable. Naprednije postupke kojima bi se moglo izračunati i izraze koji sadrže varijable koje nisu `const`-kvalificirane **nemojte** provoditi.

Prethodno navedeni simulator za naredbeni redak na standardni izlaz ispisuje vrijednost u registru `R6` nakon izvođenja instrukcije `HALT`, a isti simulator koristit će se pri automatskoj evaluaciji rješenja. Drugim riječima, u ovoj vježbi će se automatski provjeravati samo rezultat izvođenja FRISC programa kojeg generira rješenje ove laboratorijske vježbe, a ne i sam generirani FRISC program.

Konačno, u jednoj rečenici bi se moglo reći da je zadatak četvrte laboratorijske vježbe napisati program koji će, na osnovi zadanog generativnog stabla na standardnom ulazu, u datoteku `a.frisc` generirati program u mnemoničkom jeziku procesora FRISC koji povratnu vrijednost funkcije `main` pohranjuje u registar `R6`. Sve ostalo odradit će sustav SPRUT.

5.2 Značajke simulatora

Simulator koji će se koristiti pri evaluaciji rješenja ove laboratorijske vježbe bi po svim funkcionalnim značajkama trebao biti jednak onom koji se koristi na predmetu *Arhitektura računala*. Drugim riječima, simulator podržava sve mnemoničke naredbe procesora FRISC i sve *pseudonaredbe* kao što su `` `ORG `` i `` `DW ``.

Na procesor je spojena memorija od `256KB` na adresama `0`–`3FFFF`. Iz tog razloga je preporučljivo pokazivač stoga (registar `R7`) na početku programa inicijalizirati na adresu $\texttt{40000}_{16} = \texttt{262144}_{10}$ (to je prva memorijska adresa koja ne postoji).

5.3 Karakteristike ispitnih primjera za četvrtu laboratorijsku vježbu

S ciljem omogućavanja djelomičnog bodovanja rješenja ove vježbe, veliki dio ispitnih primjera bit će vrlo kratki i fokusirani na neku specifičnu značajku jezika. Na taj način će i s rješenjem koje ne podržava dio jezika (na primjer, neke operatore, rekurzivno pozivanje funkcija i slično) biti moguće ostvariti dio bodova. Ipak, važno je imati na umu da svaki *ppjC* program ima barem jednu funkciju (funkciju `main`) i provjerava se povratna vrijednost funkcije pa je za bilo kakve bodove nužno generirati kod barem za jednostavne

funkcije bez parametara i lokalnih varijabli koje vraćaju vrijednost.

5.4 Predaja četvrte laboratorijske vježbe

Ulazna točka za Javu treba biti u razredu `GeneratorKoda`, a ulazna točka za Python treba biti u datoteci `GeneratorKoda.py`. Ostala pravila za predaju četvrte laboratorijske vježbe jednaka su kao za treću vježbu.

5.5 Proširenje nekih semantičkih pravila

U ovom poglavlju su ukratko navedena još neka semantička pravila koja se odnose na generiranje koda. Drugim riječima, u semanatičkoj analizi ne treba ništa novo provjeravati, nego pravila definiraju kakav kod treba generirati.

5.5.1 Semantička pravila inkrement i dekrement operatora

Isto kao *C*, jezik *ppjC* ima postfiks i prefiks inačice inkrement i dekrement operatora. Precizan opis značenja tih operatora u *C*-u, točnije trenutak u kojem se događa promjena vrijednosti varijable, relativno je složen i sličan opis ovdje ne bi imao smisla jer se radi o detalju koji nije previše bitan.

Zato ćemo za jezik *ppjC* definirati da se vrijednost varijable mijenja neposredno prije čitanja vrijednosti za prefiks inkrement i dekrement operatore i neposredno poslije čitanja vrijednosti za postfiks inkrement i dekrement. Na primjer, ako se izrazi izračunavaju koristeći stog[2], za izraz `i++` (što može biti dio nekog većeg izraza) treba generirati kod koji dohvaća vrijednost varijable `i` u registar, pohranjuje tu vrijednost na stog, povećava vrijednost u registru i zapisuje vrijednost u memoriju na adresu koja odgovara varijabli `i`.

Dodatno, definirat ćemo da sva korištenja ovih operatora koja imaju nedefinirano značenje u jeziku *C* imaju nedefinirano značenje i u jeziku *ppjC*, tj. neće se pojavljivati u ispitnim primjerima[3]. Ovo drugim riječima znači da zbog ovih operatora ne treba posebno brinuti o tome kojim redoslijedom se izračunavaju lijeva i desna strana binarnih operatora, vrijednosti argumenata funkcija i slično.

5.5.2 Semantička pravila operatora %

Zbog jednostavnosti, operator `%` ćemo definirati samo za pozitivne operande i za slučaj kada je lijevi operand `0`, a desni operand pozitivan broj. Rezultat je ostatak pri dijeljenju lijevog operanda s desnim.

[2] Vidi poglavlje 5.6.3.

[3] Primjeri takvih naredbi su: `a[i] = i++; x = --i + i; i = i--; x = f(i, i++);` i slično.

5.5.3 Evaluacija operanada logičkih operatora

Logički operatori u jeziku *ppjC*, isto kao u *C*-u, **ne evaluiraju** desni operand ako se rezultat može jednoznačno odrediti na osnovi vrijednosti lijevog operanda[4]. Konkretno, za operator `&&`, ako lijevi operand nije istinit, rezultat primjene operatora će sigurno biti neistina, bez obzira na vrijednost desnog operanda. U tom slučaju, desni operand se ne evaluira.

Ispravnost mnogih programa u jezicima koji definiraju ovakvo značenje logičkih operatora ovisi upravo o tome da se desni operand konjunkcije ne evaluira kada je lijevi operand neistinit. Pretpostavimo da je definiran neki niz znakova `a` duljine `n`. Ovaj način evaluacije operanada bio bi važan, na primjer, za izraz `i<n && a[i]='x'`. Ako je indeks `i` prevelik za indeksiranje u niz `a`, tj. lijevi operand operatora `&&` je lažan, važno je da se desni operand ne evaulira[5].

Slično, za operator `||`, desni operand se ne evaluira ako je lijevi operand istinit jer je tada rezultat sigurno istina, bez obzira na vrijednost desnog operanda.

5.6 Savjeti za implementaciju

U ovom poglavlju su navedeni neki savjeti za implementaciju generatora koda. Preporučamo da savjete pročitate, a po želji možete neke ili sve u potpunosti zanemariti. Svi primjeri FRISC koda u ovom poglavlju su ilustrativni, tj. ni u kom slučaju nije nužno da generator generira *točno takav* kod. Štoviše, većina generatora generirat će za ove primjere kod sa više instrukcija i to nije nikakav problem ako je funkcionalnost izvornog *ppjC* programa očuvana.

5.6.1 Organizacija generiranog programa

Prije početka rada na generiranju koda, treba razmisliti kako će generirani program biti organiziran, tj. gdje će se nalaziti potprogrami, gdje će biti globalne varijable, na kojoj adresi počinje stog i slično. U nastavku poglavlja je načelno i na kratkim primjerima opisana jedna moguća organizacija generiranog programa.

Kako izvođenje programa počinje od adrese `0`, na početku generiranog programa moraju biti instrukcije (a ne podaci). Svaki program mora inicijalizirati stog, pozvati proceduru koja je generirana na osnovi funkcije `main`, povratnu vrijednost pohraniti u registar `R6` i izvesti instrukciju `HALT`. Kao što je spomenuto u poglavlju 5.2, preporučljivo je pokazivač stoga inicijalizirati na adresu $\texttt{40000}_{16}$ jer stog procesora FRISC raste prema nižim adresama. Nadalje, pretpostavit ćemo da svi potprogrami vraćaju vrijednost putem registra `R6`[6]. Uz ove dvije odluke, početak i kraj izvođenja programa ostvaruju se sa tri instrukcije na adresama `0`, `4` i `8`, prikazane u ispisu 5.1.

[4] Ovakav način evaluacije operanada se u literaturi na engleskom jeziku obično naziva *short-circuit evaluation*.

[5] U primjeru je namjerno kao desni operand korišten izraz pridruživanja kako bi bila jasna razlika između rezultata izvođenja u slučaju da se taj desni operand evaluira i u slučaju da se ne evaluira, iako je jednako neispravno samo i pročitati vrijednost izvan granica niza.

[6] Vidi poglavlje 5.6.2.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT
```

Ispis 5.1: Instrukcije za inicijalizaciju i završetak izvođenja programa.

Labela `F_MAIN` u ispisu predstavlja adresu potprograma koji je generiran iz funkcije `main` ulaznog programa u jeziku *ppjC*. Na primjer, za jednostavan ulazni program u ispisu 5.2 mogli bismo generirati program sličan onom u ispisu 5.3.

```
int main(void) {
  return 42;
}
```

Ispis 5.2: Jednostavan *ppjC* program.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  MOVE %D 42, R6
        RET
```

Ispis 5.3: Mogući generirani FRISC mnemonički program za ispis 5.2.

Nakon generiranih instrukcija za inicijalizaciju, mogu se generirati potprogrami za svaku funkciju u programu, slično kao što je prikazano u prethodnom primjeru za funkciju `main`. Svakom potprogramu se pridijeli jedinstvena labela koja omogućuje pozivanje tog potprograma instrukcijom `CALL`.

Nakon koda potprograma, mogu se rezervirati memorijske lokacije za globalne varijable. Na primjer, za program u ispisu 5.4 mogao bi se generirati program sličan onom u ispisu 5.5.

```
int x = 12;
int main(void) {
  return x;
}
```

Ispis 5.4: Jednostavan *ppjC* program s globalnom varijablom.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  LOAD R6, (G_X)
        RET

G_X     DW %D 12
```

Ispis 5.5: Mogući generirani FRISC mnemonički program za ispis 5.4.

Konačno, prostor nakon globalnih varijabli može biti koristan za neke konstante u programu, a s kraja memorije ga puni i rastući stog.

5.6.2 Preslikavanje funkcije jezika *ppjC* u FRISC potprogram

Kako svaki *ppjC* program sadrži barem jednu funkciju, važno je razmisliti o različitim pitanjima koja se pojavljuju pri preslikavanju funkcija u potprograme u mnemoničkom jeziku.

Prenošenje argumenata i povratne vrijednosti

U ostvarenju preslikavanja funkcija u potprograme, ključno je odlučiti na koji način će potprogram komunicirati sa svojim pozivateljem, tj. na koji način pozivatelj potprogramu prenosi argumente i na koji način potprogram vraća povratnu vrijednost pozivatelju. I za prenošenje argumenata i za povratnu vrijednost postoje barem tri mogućnosti: korištenje registara, korištenje stoga i korištenje poznatih memorijskih lokacija tj. adresa. Štoviše, različiti potprogrami mogu na različite načine komunicirati s pozivateljem.

Zbog jednostavnosti, preporuča se za sve potprograme koristiti stog za argumente i registar `R6` za povratnu vrijednost. Uz takvu organizaciju, kako bi pozvao potprogram, pozivatelj na stog mora postaviti argumente i izvesti instrukciju `CALL` s odgovarajućom labelom, a povratna vrijednost se nakon povratka iz potprograma nalazi u registru `R6`. Potprogram pristupa predanim argumentima koristeći pokazivač stoga (registar `R7`) i registarsko indirektno adresiranje s odmakom. Naime, kako generator koda generira sve ove instrukcije, lako je *tijekom prevođenja* odrediti potrebne odmake za sve parametre.

Preostaje odlučiti što učiniti s argumentima na stogu nakon završetka izvođenja potprograma. Jedna mogućnost je da pozivatelj nakon instrukcije `CALL` uveća pokazivač stoga za odgovarajuću vrijednost i na taj način "obriše" argumente sa stoga. Ova mogućnost prikazana je u ispisu 5.7.

```
1 int f(int y) {
2   return y;
3 }
4 int x = 12;
5 int main(void) {
6   return f(x);
7 }
```

Ispis 5.6: Jednostavan *ppjC* program s pozivom funkcije.

```
1         MOVE 40000, R7
2         CALL F_MAIN
3         HALT
```

```
4
5  F_MAIN LOAD R0, (G_X)
6         PUSH R0
7         CALL F_F
8         ADD R7, 4, R7
9         RET
10
11 F_F    LOAD R6, (R7+4)
12        RET
13
14 G_X    DW %D 12
```

Ispis 5.7: Mogući generirani FRISC mnemonički program za ispis 5.6.

Naredba koja pomiče pokazivač stoga nalazi se u retku 8. U ovom slučaju, pokazivač stoga se pomiče za `4` jer je postojao samo jedan četverobajtni argument. Uočite da bi se bez ove naredbe vrijednost argumenta (u ovom slučaju `12`) interpretirala kao povratna adresa potprograma `F_MAIN` u instrukciji `RET`.

Druga mogućnost je da sam potprogram prije završetka rada ukloni argumente sa stoga *pazeći na svoju povratnu adresu*. I ovu mogućnost je relativno jednostavno ostvariti, ali nije ovdje prikazana. Preporuča se konzistentno korištenje jednog od ova dva načina obnavljanja stoga za sve potprograme.

Lokalne varijable

Lokalne varijable se slično kao i parametri mogu pohranjivati na stog, u registre ili na određene memorijske lokacije. Kako bi se omogućilo rekurzivno pozivanje potprograma, najjednostavnije je lokalne varijable pohraniti na stog, na početku potprograma. Lokalne varijable se onda u potprogramu koriste isto kao i parametri, registarskim indirektnim adresiranjem s odmakom.

Eventualne inicijalizacije lokalnih varijabli mogu se obaviti i nakon što su za sve parametre osigurana mjesta na stogu kako bi odmaci od pokazivača stoga bili konstantni. Mjesto se na stogu može osigurati jednostavnim smanjenjem pokazivača stoga, pri čemu je po pravilima jezika dozvoljeno da neinicijalizirane lokalne varijable imaju neodređene vrijednosti (vrijednosti će ovisiti o sadržaju stoga u nekom prethodnom izvođenju nekog potprograma). Prije izvođenja instrukcije `RET`, potprogram mora vratiti pokazivač stoga na originalno mjesto, tj. ukloniti lokalne varijable sa stoga.

```
1 int f(int y) {
2   int x = y;
3   int z;
4   return x;
5 }
6 int x = 12;
7 int main(void) {
8   return f(x);
9 }
```

Ispis 5.8: Funkcija s lokalnom varijablom.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  LOAD R0, (G_X)
        PUSH R0
        CALL F_F
        ADD R7, 4, R7
        RET

F_F     SUB R7, 8, R7 ; mjesto za x i z
        LOAD R0, (R7+0C)
        STORE R0, (R7+4) ; inicijalizacija lokalne varijable x
        LOAD R6, (R7+4) ; povratna vrijednost
        ADD R7, 8, R7 ; skidanje lokalnih varijabli sa stoga
        RET

G_X     DW %D 12
```

Ispis 5.9: Mogući generirani FRISC mnemonički program za ispis 5.8.

Čuvanje konteksta

Ako se u generiranju koda koristi neki algoritam za dodjelu registara (što se u ovoj vježbi **ne preporuča**; vidi poglavlje 5.6.3), onda može biti važno da potprogrami tijekom svog rada ne promijene sadržaj određenih registara. Kontekst se tipično čuva tako da se na početku potprograma vrijednosti nekih ili svih registara zapišu na stog te se prije završetka rada potprograma te iste vrijednosti pohrane nazad u registre.

Dodatno, treba razmisliti o tome hoće li potprogram čuvati i vrijednost statusnog registra ili ne. Čuvanje vrijednosti statusnog registra može pozitivno djelovati na neke postupke optimiranja, ali za potrebe ove vježbe se ne preporuča.

Nedostatak čuvanja konkteksta je potrošnja procesorskog vremena, ali takva organizacija potprograma bitno olakšava neke postupke dodjele registara i općenito omogućuje učinkovitije korištenje registara jer pozivi potprograma ne utječu na vrijednosti registara (osim eventualno na vrijednost registra koji se koristi za prijenos povratne vrijednosti).

5.6.3 Jednostavno izračunavanje izraza korištenjem stoga

Kako je generiranje koda koji učinkovito koristi registare čak i u jednostavnim inačicama relativno složen problem, za potrebe ove laboratorijske vježbe se preporuča izračunavanje izraza korištenjem stoga. Pritom se registri koriste samo privremeno, kako bi se dohvatile

odgovarajuće vrijednosti sa stoga i kako bi se obavila odgovarajuća operacija, nakon čega se izračunata vrijednost opet pohranjuje na stog.

Pokazivač stoga će se tijekom ovog postupka mijenjati što otežava pristupanje parametrima i lokalnim varijablama potprograma. Ispravni odmaci se svakako daju izračunati, a alternativa je u svakom potprogramu rezervirati jedan registar (na primjer `R6` u koji se na kraju potprograma pohranjuje povratna vrijednost) za adresu *okvira funkcije* koja se može inicijalizirati, na primjer, nakon zauzimanja prostora za lokalne varijable i nakon toga se ne mijenja do kraja izvođenja potprograma. U tom slučaju, vrijednost tog registra treba pohraniti na stog prije svake `CALL` instrukcije u potprogramu i dohvatiti ju sa stoga nakon te instrukcije.

Na primjer, za računanje vrijednosti izraza `x + y` gdje su `x` i `y` globalne varijable, mogao bi se generirati FRISC kod prikazan u ispisu 5.10.

```
1         LOAD R0, (G_X)
2         PUSH R0
3
4         LOAD R0, (G_Y)
5         PUSH R0
6
7         POP R1
8         POP R0
9         ADD R0, R1, R0
10        PUSH R0
```

Ispis 5.10: Računanje izraza `x + y` primjenom stoga.

Pri obradi primarnih izraza `x` i `y` generirale bi se po dvije instrukcije u retcima 1–2 i 4–5, a pri obradi operatora zbrajanja (u produkciji znaka `<aditivni_izraz>`) generirale bi se četiri instrukcije u retcima 7–10.

5.6.4 Množenje, dijeljenje i operator ostatka

Kako FRISC nema instrukcije za obavljanje ovih operacija, njihovo provođenje mora se riješiti ručno što čini implementaciju ovih operatora znatno složenijom od implementacije ostalih operatora. Najjednostavniji način za ostvariti ove operatore je sa dva potprograma koji se mogu ugraditi u svaki program koji koristi operatore (ili čak i u programe koji ih ne koriste). Pritom ne treba paziti na rezultate koji izlaze iz opsega tipa `int` jer su takvi rezultati u pravilima jezika ostavljeni nedefiniranima.

Implementacija ovih operatora nije previše važna (pogotovo s obzirom na relativnu složenost u odnosu na implementaciju ostalih operatora) i preporuča se implementirati ove operatore pri kraju rada na generatoru koda, ako za to bude vremena.

5.6.5 Kako početi rješavati laboratorijsku vježbu

Prošlih godina je jedno od čestih pitanja bilo upravo kako započeti rad na generatoru koda. Dobra strategija je početi od najjednostavnijeg primjera *ppjC* programa i za njega napisati

odgovarajući FRISC kod te ga isprobati na simulatoru. Iz jednostavnih primjera moguće je osmisliti općenite pristupe za generiranje različitih dijelova programa i to inkrementalno implementirati u generator.

Pri pisanju odgovarajućeg FRISC koda, bitno je pokušati "imitirati" trenutni mentalni model generatora koda, što smo pokušali ilustrirati u ovim savjetima za implementaciju. Kada se taj model promijeni, treba revidirati prethodno obrađene primjere i provjeriti je li sve u redu. Na primjer, ako bismo odlučili da će se izrazi izračunavati koristeći stog, čini se da bismo uniformnom primjenom tog modela za jednostavni program iz ispisa 5.2 u kojem funkcija `main` jednostavno vraća konstantu `42` mogli umjesto instrukcije `MOVE %D 42, R6` generirati tri instrukcije, kao što je prikazano u ispisu 5.11.

```
1         MOVE 40000, R7
2         CALL F_MAIN
3         HALT
4
5 F_MAIN  MOVE %D 42, R0
6         PUSH R0
7         POP R6
8         RET
```

Ispis 5.11: FRISC mnemonički program za ispis 5.2 uz konzistentno računanje izraza pomoću stoga.

Instrukcije u retcima 5 i 6 mogle bi se generirati pri obradi izraza `42`, a instrukcije u retcima 7 i 8 pri obradi `return` naredbe.

Bibliografija

[1] Siniša Srbljić: *Prevođenje programskih jezika*, Element, 2007.

[2] Siniša Srbljić: *Uvod u teoriju računarstva*, Element, 2007.