4. Treća laboratorijska vježba

Tema treće laboratorijske vježbe je semantička analiza. Za razliku od prve dvije vježbe gdje je cilj bio ostvariti generator leksičkog odnosno sintaksnog analizatora, u ovoj vježbi je cilj ostvariti semantički analizator za jedan zadani programski jezik. Važno je uočiti da je ova vježba potpuno neovisna o rješenjima prve dvije vježbe.

Kako je semantička analiza tipično složenija od leksičke i sintaksne analize, među ostalim i zbog teže primjenjive formalne podloge, programski jezik koji će se koristiti u ovoj vježbi je semantički relativno jednostavan. Dodatno, neka ograničenja koja bi tipično bila u domeni semantičke analize prebačena su u sintaksnu analizu, opet s ciljem pojednostavljivanja semantičke analize. Nadalje, iako se semantička analiza tipično provodi nad sintaksnim stablom[1], za potrebe laboratorijskih vježbi ćemo ovaj korak preskočiti, tj. semantička analiza će se provoditi nad generativnim stablom. U ovoj uputi navode se samo semantička pravila koja su važna za semantičku analizu, a pravila će biti dodatno proširena u uputi za sljedeću laboratorijsku vježbu za potrebe generiranja koda.

Uz sva pojednostavljenja, opis semantike svakog programskog jezika (pa tako i ovog) je inherentno relativno složen. Na primjer, u *C* standardu opis jezika (bez standardnih biblioteka) ima nešto manje od 200 stranica, a od toga se većina bavi semantikom. Naravno, *C* je daleko složeniji jezik od jezika kojim se bave ove laboratorijske vježbe i opis ovog jezika nije ni približno toliko potpun, ali je istovremeno znatno jednostavniji za potrebe implementacije semantičkog analizatora. Cilj upute je da većina pravila bude očita nakon prvog čitanja, a da istovremeno što veći dio jezika bude pokriven, tj. da ne bude nužno postavljati mnogo pitanja tipa "A što ako...?".

4.1 Ulaz i ispis semantičkog analizatora

Semantički analizator će na stadardni ulaz dobiti generativno stablo u istom obliku kao ispis sintaksnog analizatora iz druge laboratorijske vježbe. Usprkos tome, u većini primjera kojima se u uputi ilustrira neko semantičko pravilo prikazuje se program a ne generativno stablo. Program **nije** ulaz u rješenje ove laboratorijske vježbe, ali je semantička pravila lakše objasniti nad programom nego nad relativno nepreglednim generativnim stablom. Iz tog razloga se za testiranje rješenja tijekom razvoja preporuča korištenje leksičkog i sintaksnog analizatora kako biste mogli rad semantičkog analizatora provjeravati nad programima i kako ne biste morali ručno mijenjati generativno stablo.

Za ovaj način rada ne morate mijenjati rješenja prošlih laboratorijskih vježbi ni na

[1] Tipično sintaksni analizator izravno generira sintaksno stablo ili se sintaksno stablo izgradi naknadnim obilaskom generativnog stabla.

koji način. Ulazna datoteka koja opisuje leksički analizator jezika *ppjC* nalazi se ovdje[2], a ulazna datoteka koja opisuje sintaksni analizator jezika *ppjC* nalazi se ovdje[3]. Koristeći ove dvije datoteke i rješenja prethodnih laboratorijskih vježbi možete generirati leksički i sintaksni analizator za jezik *ppjC*. Nadalje, kako analizatori čitaju ulaz sa stadnardnog ulaza i ispisuju rezultat izvođenja na standardni izlaz, a ulazi i izlazi su im kompatibilni, analizatore možete povezati u "lanac" u naredbenom retku.

Ako pretpostavimo da je *LA* ime izvodivog leksičkog analizatora, *SA* ime izvodivog sintaksnog analizatora i *SemantickiAnalizator* ime izvodivog semantičkog analizatora (na kojem radite u ovoj vježbi), onda bi naredba kojom možete testirati rad semantičkog analizatora nad programom izgledala na primjer ovako:
LA `<program.c | SA |` *SemantickiAnalizator*

Semantički analizator treba obići generativno stablo i na standardni izlaz "ispisati produkciju" u kojoj je otkrivena prva semantička pogreška, kao što je precizno definirano u poglavlju 4.4.2. Semantički analizator ne treba provoditi nikakav postupak oporavka od pogreške. Drugim riječima, čim otkrije prvu semantičku pogrešku i obavi odgovarajući ispis, semantički analizator prestaje s radom.

Ako je program prikazan generativnim stablom na ulazu semantički ispravan, semantički analizator ne smije ispisati ništa na standardni izlaz[4].

4.2 Predaja treće laboratorijske vježbe

Za predaju treće laboratorijske vježbe, sve datoteke s programskim kodom treba zapakirati u `zip`-arhivu. Ostala pravila jednaka su kao za prethodne vježbe, osim što više nije potreban direktorij `analizator`.

Ulazna točka za Javu treba biti u razredu `SemantickiAnalizator`, a ulazna točka za Python treba biti u datoteci `SemantickiAnalizator.py`. Za ostale jezike imena datoteka i razreda su proizvoljna.

4.3 Opis jezika *ppjC*

U ovom poglavlju dan je pregled jezika kako bi se olakšalo praćenje semantičkih pravila u poglavlju 4.4. Neki dijelovi ovog opisa možda neće biti odmah jasni, ali bit će dodatno razjašnjeni kroz opis semantičkih pravila.

4.3.1 Tipovi podataka

Jezik *ppjC* ima vrlo jednostavan sustav tipova što značajno olakšava semantičku analizu. U nastavku poglavlja navedeni su tipovi podataka u jeziku i objašnjeno je na koje načine

[2]Ovo je ista datoteka kao `simplePpjLang.lan`

[3]Ovo je malo pojednostavljena inačica datoteke `simplePpjLang.san`. Generator generira ε-NKA s 2936 stanja i DKA s 543 stanja.

[4]Isto kao u prethodnim vježbama, sav ostali ispis, na primjer smislenije poruke o pogrešci namijenjene korisnicima cjelovitog kompilatora, mora ići na standardni izlaz za greške, tj. `stderr`.

se vrijednosti jednog tipa mogu implicitno ili eksplicitno (*cast* operatorom) pretvoriti u vrijednosti drugog tipa.

Brojevni tipovi

Jezik *ppjC* ima dva brojevna tipa: `char` i `int`. Raspon vrijednosti tipa `char` je 0–255 (8b NBC). Tip `int` koristi 32 bita i predstavlja i nenegativne cijele brojeve (pozitivne cijele brojeve i nulu) i negativne cijele brojeve u dvojnom komplementu. Drugim riječima, raspon vrijednosti tipa `int` je $-2^{31} = -2147483648 \leq v \leq 2147483647 = 2^{31} - 1$.

U ostatku upute će se brojevni tipovi označavati sa T, tj. T može biti `char` ili `int`.

const-kvalifikator

Brojevni tip prefiksiran ključnom riječi `const` (uniformni znak `KR_CONST`) naziva se `const`-kvalificiran tip i u ostatku upute će biti označavan sa *const*(T) gdje je T ili `char` ili `int`. Varijablu `const`-kvalificiranog tipa obavezno se mora inicijalizirati pri definiciji i tada pridružena vrijednost ne može se promijeniti tijekom izvođenja programa.

U ostatku upute će se brojevni tip s opcionalnim `const`-kvalifikatorom označavati znakom X, tj. X može biti `char`, `int`, *const*(`char`) ili *const*(`int`). Koristeći oznaku T, X može biti T ili *const*(T).

Nizovi (engl. *array*)

Jezik podržava nizove isključivo brojevnih tipova (`const`-kvalificiranih ili ne). Drugim riječima, ne postoje nizovi nizova (tj. višedimenzionalni nizovi), a ovo ograničenje je osigurano u gramatici, tj. sintaksnoj analizi. Tip *niz* se u uputi označava sa *niz*(X) gdje je X neki (možda `const`-kvalificiran) brojevni tip.

void

Ključna riječ `void` ima vrlo sličnu ulogu kao u jeziku *C*, a kao tip može biti isključivo povratni tip funkcije.

Tip funkcija

Tip funkcije određen je tipom povratne vrijednosti i svih formalnih parametara. Na primjer, funkcije u ispisu 4.1 redom imaju tipove *funkcija*(`[char, char]` → `int`), *funkcija*(`[int]` → `void`) i *funkcija*(`void` → `void`). Tipovi parametara organizirani su u listu, a ako funkcija nema parametara, to je označeno sa `void` (ali ne sa `[void]`, što ne bi imalo smisla).

```
1 int f1(char a, char b) {
2     return a == b;
3 }
```

```
4
5  void f2(int x) {
6      return;
7  }
8
9  void f3(void) {
10     return;
11 }
```

Ispis 4.1: Primjer jednostavne funkcije.

Implicitne promjene tipa

Sve vrijednosti tipa $const(T)$ mogu se implicitno pretvoriti u vrijednost tipa T[5]. Vrijedi i obrat, tj. sve vrijednosti tipa `int` ili `char` mogu se implicitno pretvoriti u vrijednosti odgovarajućeg `const`-kvalificiranog tipa.

Nadalje, sve vrijednosti tipa `char` mogu se implicitno pretvoriti u vrijednost tipa `int`. Kako `int` sadrži raspon tipa `char` kao podskup svog raspona, pri ovoj implicitnoj promjeni sama vrijednost se ne mijenja.

Konačno, vrijednost tipa $niz(T)$ gdje T **nije** `const`-kvalificiran tip može se pretvoriti u vrijednost tipa $niz(const(T))$.

U semantičkim pravilima u nastavku upute, izraz $U \sim V$ je zadovoljen ako se vrijednost tipa U može implicitno pretvoriti u vrijednost tipa V, a inače se radi o semantičkoj pogrešci. Relacija $\sim$ je refleksivna i tranzitivna.

Logički podtip tipa `int`

Logički tip u jeziku *ppjC* nije poseban tip nego je podtip tipa `int` s vrijednostima `0` i `1`, pri čemu `0` predstavlja logičku neistinu, a `1` logičku istinu. Sve vrijednosti različite od nule pretvaraju se u logičku vrijednost `1`, a nula se "pretvara" u logičku vrijednost `0`[6].

Eksplicitne promjene tipa

Eksplicitne promjene tipa dozvoljene su samo nad vrijednostima brojevnih tipova, a zadaju se *cast* operatorom. Drugim riječima, jedina promjena tipa koju je moguće ostvariti samo eksplicitno je promjena iz vrijednosti tipa `int` u vrijednost tipa `char`. Ako iznos `int` vrijednosti nije u opsegu tipa `char`, **rezultat operacije nije definiran**[7], a u suprotnom se iznos ne mijenja.

[5] Važno je uočiti da se promjene tipa (i implicitne i eksplicitne) odvijaju nad **vrijednostima** a ne nad **varijablama**. Tip varijable koji je određen njenom definicijom nije moguće nikako promijeniti.

[6] Ova rečenica ni na koji način ne utječe na semantičku analizu u ovoj laboratorijskoj vježbi, ali će biti važna za generiranje koda u sljedećoj vježbi.

[7] Vidi 4.4.1.

4.3.2 Konstante

Brojevne konstante (uniformni znak `BROJ`) su tipa `int` i moraju imati vrijednost u dozvoljenom rasponu za tip `int`.

Znakovne konstante (uniformni znak `ZNAK`) su tipa `char` i imaju vrijednost jednaku ASCII kodu znaka. Kroz leksičku analizu proći će i neke znakovne konstante koje nisu ispravne pa takve pogreške treba otkriti u semantičkoj analizi. Konkretno, od znakovnih konstanti koje između jednostrukih navodnika imaju više od jednog znaka (tj. imaju znak prefiksiran znakom \), *ppjC* dozvoljava isključivo znakove `'\t'`, `'\n'`, `'\0'`, `'\''` (jednostruki navodnik), `'\"'` (dvostruki navodnik) i `'\\'`. Sve ostale dvoznačne konstante predstavljaju semantičku grešku[8]. Nadalje, znak dvostrukog navodnika se može pojaviti neprefiksiran (dakle `'"'`) sa istim značenjem.

Konstantni znakovni nizovi (uniformni znak `NIZ_ZNAKOVA`) su tipa *niz*(*const*(`char`)) i implicitno završavaju znakom `'\0'` (kao u *C*-u). Ispravni konstantni znakovini nizovi počinju i završavaju dvostrukim navodnikom (ovo je osigurano leksičkom analizom), i mogu sadržavati sve ispisive ASCII znakove, a znakom \ mogu biti prefiksirani samo znakovi koji su navedeni u prethodnom odlomku. Slično kao za znakovne konstante, leksička analiza će propustiti neke konstantne znakovne nizove koji nisu ispravni, npr. nizove `"\"` i `"\x"`.[9]

4.3.3 Djelokrug deklaracija i životni vijek varijabli

Jezik *ppjC* po klasifikaciji iz udžbenika koristi *statičko pravilo djelokruga bez ugniježđenih procedura* [1, str. 223–224], ali podržava ugniježđene blokove, isto kao *C*. Kako je sintaksa procedura u udžbeniku bliska *Pascalu*, a taj stil danas više nije previše zastupljen i mnogi od vas se možda s njim nisu prije susreli, za potrebe laboratorijskih vježbi se oslonite više na djelokrug deklaracija *C*-a i na primjere u nastavku.

Životni vijek globalnih varijabli počinje u trenutku njihove definicije, a završava na kraju izvođenja programa. Za varijable lokalne nekom bloku (što uključuje i parametre funkcija), životni vijek počinje njihovom definicijom i završava na kraju tog bloka. Varijable koje nisu eksplicitno inicijalizirane (prilikom definicije ili kasnije) imaju *neodređenu vrijednost* (drugim riječima, mogu imati bilo koju vrijednost) i rezultat korištenja takve vrijednosti u bilo kakvom izrazu nije definiran[10].

```
1 int x = 3;
2 int main(void) {
3     int y = x + 1; // 4
4     return 0;
```

[8] Ovo pravilo se dakako moglo osigurati i u leksičkoj analizi, ali je namjerno ostavljeno za semantičku analizu kako bi se ilustriralo da leksička, sintaksna i semantička pravila nisu zadana nekim univerzalnim kanonom, nego autor jezičnog procesora mora odlučiti koje značajke će provjeravati na koji način. Ovo svakako nije previše uobičajen primjer jer je vrlo lako uključiti ovo pravilo u regularne izraze, ali na primjer provjera raspona brojevnih konstanti je značajno jednostavnija u domeni semantičke analize nego u ovakvom modelu leksičke analize.

[9] Zbog ovog propuštanja neispravnih nizova, pod određenim uvjetima nije moguće koristiti veći broj konstantnih znakovnih nizova u istom retku, ali s tim se nećemo zamarati.

[10] Za razliku od jezika *C*, ovo se odnosi i na varijable u globalnom djelokrugu.

```
}
```

Ispis 4.2: Jednostavan primjer djelokruga deklaracija.

Jednostavan primjer prikazan je u ispisu 4.2. Varijabla `x` deklarirana je u globalnom djelokrugu u kojem je deklarirana i funkcija `main`. Tijelo funkcije `main` je blok, tj. složena naredba, te kao takvo ima vlastiti djelokrug deklaracija u kojem je deklarirana varijabla `y`. U tijelu funkcije `main`, prethodno deklarirana imena iz globalnog djelokruga (u ovom slučaju `x`) također su dostupna.

```
1 int main(void) {
2     int y = x + 1; // greska
3     return 0;
4 }
5 int x = 3;
```

Ispis 4.3: Primjer pogreške u djelokrugu—varijabla `x` nije deklarirana prije korištenja.

Primjer u ispisu 4.3 sadrži semantičku pogrešku u retku 2 jer varijabla `x` nije prethodno deklarirana.

```
1 int x = 3;
2 int main(void) {
3     int x = 5;
4     int y = x + 1; // 6
5     return 0;
6 }
```

Ispis 4.4: Sakrivanje globalne deklaracije.

Lokalno ime može sakriti ime iz ugniježđujućeg djelokruga. Primjer tog slučaja prikazan je u ispisu 4.4. Deklaracija varijable `x` u retku 3 skriva globalnu varijablu `x`.

```
1  int main(void) {
2      int x = 3;
3      int z;
4      {
5        int x = 5;
6        int y = x + 1; // 6
7        z = y + 1; // 7
8      }
9      z = x + 1; // 4
10     return 0;
11 }
```

Ispis 4.5: Sakrivanje deklaracije u bloku.

Isto pravilo zajedno s pravilom o životnom vijeku varijabli određuje značenje programa u ispisu 4.5. Važno je uočiti da je moguće sakriti samo deklaraciju iz ugniježđujućeg djelokruga, ali ne i iz istog djelokruga (u tom slučaju radi se o pokušaju redeklaracije što za varijable nije dozvoljeno, a pravila za funkcije su objašnjena u poglavlju 4.3.4).

Primjer ove greške prikazan je u ispisu 4.6.

```
1 int main(void) {
2     int x = 5;
3     int x = 6; // greska
4     return 0;
5 }
```

Ispis 4.6: Nedozvoljena redeklaracija varijable.

4.3.4 Funkcije

Svaki program u jeziku *ppjC* mora imati funkciju s prototipom `int main(void)`. Ova funkcija je ulazna točka za izvođenje programa.

Sve funkcije koje ne primaju argumente, u deklaraciji između oblih zagrada moraju imati navedenu ključnu riječ `void`. Drugim riječima, deklaracija funkcije oblika `int f();` je neispravna, a to je osigurano u gramatici tj. u sintaksnoj analizi.

Svaka funkcija prije korištenja mora biti **deklarirana**, pri čemu je **definicija** funkcije ujedno i njena deklaracija. Deklaracija je u tom slučaju u potpunosti završena prije znaka lijeve vitičaste zagrade koji započinje tijelo funkcije. Funkcija može biti deklarirana proizvoljan broj puta, a mora biti definirana točno jednom. Pritom, sve deklaracije (uključujući i definiciju) moraju imati identične povratne tipove i tipove formalnih parametara.

U deklaracijama je (za razliku od *C*-a) obvezno navesti imena formalnih parametara, ali ta imena nisu važna tj. mogu se razlikovati među različitim deklaracijama iste funkcije (drugim riječima, treba ih ignorirati).

```
1 int main(void) {
2   return f(); // greska
3 }
4 int f(void) {
5   return 0;
6 }
```

Ispis 4.7: Funkcija `f` nije deklarirana prije korištenja.

U ispisu 4.7, semantička pogreška pojavljuje se pri pozivu funkcije `f` u retku `2`, jer funkcija `f` nije prethodno deklarirana. Grešku je moguće ispraviti na više načina, a tri načina prikazana su u nastavku.

```
1 int f(void);
2 int main(void) {
3   return f();
4 }
5 int f(void) {
6   return 0;
```

```
}
```

Ispis 4.8: Dodatak deklaracije funkcije `f` prije definicije funkcije `main`.

Prvo, kao što je prikazano u ispisu 4.8, moguće je prije definicije funkcije `main` (koja poziva funkciju `f`) dodati deklaraciju funkcije `f`.

Drugo, za razliku od definicija funkcija koje su dozvoljene isključivo u globalnom djelokrugu, funkcija se može deklarirati i unutar (bilo kojeg) bloka. U ispisu 4.9 deklaracija funkcije `f` prebačena je iz globalnog djelokruga u tijelo funkcije `main`.

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

Ispis 4.9: Deklaracija funkcije `f` neposredno prije poziva u tijelu funkcije `main`.

Kako je definicija funkcije ujedno i deklaracija, u ovom slučaju je moguće definiciju funkcije `f` prebaciti prije definicije funkcije `main`, kao što je prikazano u ispisu 4.10.

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

Ispis 4.10: Definicija funkcije `f` je ujedno i njena deklaracija.

Nadalje, s obzirom na to da deklaracija funkcije završava prije lijeve vitičaste zagrade koja započinje tijelo funkcije, za ostvarenje rekurzije nije potrebno koristiti eksplicitnu deklaraciju funkcije. Primjer semantički ispravne rekurzivne funkcije prikazan je u ispisu 4.11.

```
int fact(int n) {
  if (n > 0) {
    return n * fact(n-1);
  } else {
    return 1;
  }
}
```

Ispis 4.11: Rekurzivne funkcije ne treba eksplicitno deklarirati prije definicije.

Ipak, u slučaju da se dvije funkcije međusobno pozivaju, nije moguće izbjeći eksplicitnu deklaraciju jedne od njih. Primjer takvog slučaja prikazan je u ispisu 4.12.

```
1  int bar(int x);
2  int foo(int a, int b) {
3    if (a > 0) {
4      return a + bar(b);
5    } else {
6      return 0;
7    }
8  }
9  int bar(int a) {
10   return foo(a, a-1);
11 }
```

Ispis 4.12: Nužna deklaracija funkcije `bar` prije definicije funkcije `foo`.

U deklaraciji funkcije `bar` u retku `1`, navedeno je drugačije ime formalnog parametra nego u definiciji funkcije u retku `9`, ali to ne predstavlja grešku jer se imena parametara u deklaraciji ignoriraju.

Povratni tip funkcije može biti `char` ili `int` **bez** `const`-kvalifikatora, ili `void` što znači da funkcija ne vraća ništa. Semantička je pogreška ako funkcija deklarirana da vraća `void` pokuša vratiti neku vrijednost. Nadalje, ako funkcija deklarirana da vraća neku vrijednost brojevnog tipa u nekom slijedu izvođenja ne vrati vrijednost, rezultat izvođenja nije definiran (ali se ne radi o semantičkoj pogrešci, tj. semantički analizator ne treba provjeravati vraća li funkcija stvarno neku vrijednost u svim sljedovima izvođenja). Ako funkcija vraća neku vrijednost, tip vrijednosti mora se moći implicitno pretvoriti u deklarirani povratni tip funkcije (ovo pravilo pokriva i `void` povratni tip s obzirom na to da se niti jedan tip vrijednosti ne može implicitno pretvoriti u `void`).

U ispisu 4.13 je prikazana semantički ispravna funkcija `foo` i semantički neispravna funkcija `bar`. Funkcija `foo` je ispravna jer vrijedi `char` $\sim$ `int`, a funkcija `bar` je neispravna jer obrat ne vrijedi pa nije moguće vratiti `int` vrijednost iz funkcije kojoj je povratni tip `char`.

```
1  int foo(void) {
2    return 'a';
3  }
4
5  char bar(void) {
6    return 97; // greska
7  }
```

Ispis 4.13: Ispravna funkcija `foo` i neispravna funkcija `bar`.

Funkciju `bar` moglo bi se ispraviti korištenjem *cast* operatora, kao što je prikazano u ispisu 4.14.

```
1  char bar(void) {
2    return (char)97;
3  }
```

Ispis 4.14: Ispravljena funkcija `bar`.

Nadalje, hoće li se `return` naredba ikada izvesti ili ne nema utjecaja na semantičku ispravnost programa. Na primjer, funkcija u ispisu 4.15 također je neispravna, iako bi tijekom izvođenja uvijek vraćala vrijednost ispravnog tipa.

```
char bar(void) {
  if (0) {
    return 97; // greska
  } else {
    return (char)97;
  }
}
```

Ispis 4.15: Ispravnost tipova povratnih vrijednosti ne ovisi o tome hoće li se određena `return` naredba ikada izvršiti ili ne.

Prilikom poziva funkcije, *vrijednost* argumenata brojevnog tipa prenosi se u odgovarajuće parametre[11]. S druge strane, u formalne parametre tipa $niz(X)$ koji se deklariraju sintaksom X `a[]` prenosi se *adresa* argumenta. U oba slučaja, tip vrijednosti argumenta mora se moći implicitno pretvoriti u tip odgovarajućeg parametra.

Razlika u načinu prenošenja argumenata u funkciju između brojevnih tipova i nizova prikazana je u ispisu 4.16.

```
void f(int x, int a[]) {
  x = x + 1;
  a[0] = a[0] + 1;
}

int main(void) {
  int x = 3;
  int a[8] = {0};
  f(x, a); // x == 3, a[0] == 1
  return 0;
}
```

Ispis 4.16: Brojevni tipovi se prenose *razmjenom vrijednosti*. Nizovi se prenose *razmjenom adresa*.

4.3.5 Operatori

Većina operatora u jeziku *ppjC* definirana je isključivo nad tipom `int`, a i rezultat je tipa `int`. Jedna donekle iznenađujuća posljedica ovog pravila u kombinaciji s pravilima o pretvorbi tipova je da je program u ispisu 4.17 neispravan. Naime, kako operator zbrajanja

[11] Način prenošenja argumenata ne utječe na ovu laboratorijsku vježbu, ali je ovdje naveden zbog cjelovitosti pregleda funkcija u jeziku *ppjC*.

očekuje operande tipa `int` i daje rezultat tipa `int`, vrijednost varijable `c` se implicitno pretvara u tip `int` i zbraja se s konstantom `1` (koja je tipa `int`). Rezultat operacije je opet tipa `int`, a kako ne vrijedi `int` $\sim$ `char`, radi se o semantičkoj pogrešci u pridruživanju.

```
1 int main(void) {
2   char c = 'a';
3   c = c + 1; // greska
4   return 0;
5 }
```

Ispis 4.17: Neispravan program s operatorom zbrajanja.

Pogreška se može lako ispraviti korištenjem eksplicitne promjene tipa, kao što je prikazano u ispisu 4.18.

```
1 int main(void) {
2   char c = 'a';
3   c = (char)(c + 1);
4   return 0;
5 }
```

Ispis 4.18: Ispravljen program iz ispisa 4.17.

Ako pri aritmetičkim operacijama dođe do preljeva ili podljeva, rezultat operacije nije definiran.

4.4 Semantička pravila jezika *ppjC*

Semantička pravila jezika definirana su u nastavku prirodnim jezikom i uz pomoć formalne notacije slične notaciji atributne prijevodne gramatike, a istovremeno je objašnjena i gramatika koja opisuje sintaksu jezika *ppjC*. Sve produkcije gramatike u BNF obliku nalaze se ovdje. Važno je razumjeti gramatiku ovog jezika jer se semantička analiza provodi nad generativnim stablom čiji oblik izravno ovisi o gramatici. Preporučljivo je prije čitanja semantičkih pravila pogledati produkcije gramatike i razmisliti o ulogama pojedinih znakova i produkcija. Gramatika se sastoji od tri skupine produkcija: produkcije za izraze, produkcije za naredbenu struktura programa i produkcije za deklaraciju i definiciju varijabli i funkcija. Opisa semantičkih pravila u nastavku prati ovu strukturu.

4.4.1 Razlika između semantičke pogreške i *nedefiniranog ponašanja*

U nastavku ove upute, važno je razlikovati slučaj u kojem neki jezični konstrukt nije semantički ispravan i slučaj u kojem nije definiran *rezultat izvođenja* tog jezičnog konstrukta.

Kao primjer iz jezika *C*, semantička je pogreška pokušati zbrojiti dva pokazivača. Svaki ispravni *C* kompilator mora odbiti prevođenje programa koji sadrži ovu semantičku pogrešku. S druge strane, oduzimanje dva pokazivača na kompatibilne tipove semantički

je ispravno i svaki ispravni *C* kompilator program koji sadrži takvo oduzimanje mora prevesti (naravno, ako program ne sadrži druge pogreške). Međutim, pojednostavljeno rečeno, ako pokazivači ne pokazuju u isti niz, *rezultat oduzimanja* nije definiran (točnije, ponašanje takvog programa u cijelosti postaje nedefinirano)[12].

Drugi sličan primjer je dereferenciranje *null* pokazivača. U mnogim slučajevima kompilator ne može provjeriti hoće li neki pokazivač imati vrijednost *null* pokazivača tijekom izvođenja programa i samim time tijekom prevođenja ne može upozoriti programera da će se to dogoditi. Međutim, ako tijekom rada programa dođe do referenciranja *null* pokazivača, ponašanje programa postaje nedefinirano s aspekta *C* standarda.

To što je ponašanje programa u nekom slučaju nedefinirano u kontekstu izrade kompilatora znači da se o tim slučajevima ne treba posebno razmišljati (barem u okviru ove vježbe) — tijekom izvođenja prevedenog programa, može se dogoditi bilo što, uključujući i prestanak rada kompilatora, tj. u kontekstu ove vježbe, semantičkog analizatora. Posljedica ove činjenice je da vaša rješenja nećemo testirati sa generativnim stablima programa koji sadrže jezične konstrukte nedefiniranog ponašanja.

4.4.2 Obilazak stabla i opis semantičkih pravila

Kako generativno stablo prikazuje na koji način je iz gramatike generiran neki niz završnih znakova, struktura stabla u svakom čvoru jedinstveno određuje koja produkcija je primijenjena kako bi se znak u korijenu svakog podstabla zamijenio znakovima kojima su u stablu označena djeca tog korijena. U ovoj laboratorijskoj vježbi se pretpostavlja provjera semantičkih pravila dubinskim pretraživanjem generativnog stabla s lijeva na desno, pri čemu se teži provjeri semantičkog pravila što je prije moguće. To znači da se u produkcijama koje sadrže više završnih i nezavršnih znakova s desne strane produkcije, podstabla koja odgovaraju tim znakovima provjeravaju s lijeva na desno. Nakon svake provjere nekog podstabla s desne strane produkcije, provjeravaju se sva pravila u samoj produkciji za koja su dostupna sva potrebna svojstva znakova desne strane. Kako je redoslijed provjera važan jer o njemu ovisi ispis analizatora, uz svaku produkciju gramatike naveden je redoslijed provjere pravila. Drugim riječima, opis obilaska u ovom odlomku je čisto informativan, tj. služi kako bi bilo jasno zašto je u određenoj produkciji odabran baš taj redoslijed provjera pravila.

Za opis semantičkih pravila koriste se većinom izvedena i nekoliko nasljednih svojstava. Njihova uloga je opis pravila i nije obavezno u implementaciji koristiti ista svojstva ili svojstva uopće koristiti. Nadalje, semantička pravila nisu u potpunosti opisana pravilima računanja svojstava nego su dobrim dijelom opisana i prirodnim jezikom. Rješenje se testira isključivo na principu ulaz/izlaz, a način ostvarenja je proizvoljan dok god će rezultat biti jednak ovom konceptualnom modelu koji je opisan u uputi. Dakako, jedna mogućnost je implementirati upravo ovaj model obilaska stabla.

Semantička pravila navedena su po produkcijama, a produkcije su grupirane po nezavršnom znaku s lijeve strane. Produkcije za određene grupe operatora koje imaju identičnu strukturu i semantička pravila, ali različite operatore (na primjer produkcije koje generiraju operatore zbrajanja i oduzimanja), opisane su u okviru jedne produkcije koja na mjestu operatora ima regularan izraz koji pokriva sve moguće vrijednosti (na

[12]Tehnički pojam za ovakav slučaj je *nedefinirano ponašanje* (engl. *undefined behavior*).

primjer (PLUS | MINUS)).

Nezavršni znakovi su naslovi odjeljaka, nakon čega slijedi kratak opis uloge nezavršnog znaka u gramatici i popis produkcija s pripadnim semantičkim pravilima. Neposredno ispod svake produkcije navedena su pravila računanja izvedenih svojstava nezavršnog znaka s lijeve strane produkcije, a samo ime znaka zbog čitljivosti nije navedeno. Velika većina pravila računanja bi trebala biti sasvim očita nakon prvog čitanja, a složenija pravila su objašnjena riječima ispod produkcije.

Ispod pravila računanja su uz redne brojeve nabrojana semantička pravila koja je potrebno provjeriti i opisan je način obilaska tog podstabla. Za provjeru svih semantičkih pravila u podstablu čiji korijen je označen nekim nezavršnim znakom, koristi se oznaka *provjeri*(<ime_znaka>). Nužno je pratiti redoslijed kojim su pravila navedena, a cilj je da taj redoslijed bude očit po prethodno opisanom obrazloženju. Ako bilo koje pravilo nije zadovoljeno, semantički analizator treba ispisati danu produkciju u skladu s prethodnim opisom i završiti s radom.

Pravila računanja izvedenih svojstava nezavršnog znaka lijeve strane produkcije konceptualno se izvode nakon provjere svih pravila te produkcije i koriste vrijednosti svojstava izračunate tijekom provjere semantičkih pravila s desne strane produkcije. Kada u pravilima može doći do zabune zbog ponavljanja istog znaka više puta, uz znakove je naveden subskript.

Provjera pravila nad generativnim stablom započinje od korijena koji je uvijek označen nezavršnim znakom <prijevodna_jedinica>.

4.4.3 Ispis semantičkog analizatora u slučaju greške

Čim naiđe na prvu semantičku pogrešku, analizator treba ispisati produkciju u kojoj je pogreška otkrivena i završiti s radom. Uz završne znakove gramatike treba u zagradi ispisati i broj retka i pripadnu leksičku jedinku.

Na primjer, za pogrešku u naredbi `int x = 1 + "abc"` u drugom retku programa, ispis bi bio kao u nastavku.

```
1 <aditivni_izraz> ::= <aditivni_izraz> PLUS(2,+) <multiplikativni_izraz>
2
```

Ispis 4.19: Primjer ispisa.

Zašto se ispisuje ta produkcija bit će jasno kada pročitate semantička pravila, ali važno je uočiti da se uz uniformni znak operatora zbrajanja (PLUS) u zagradi ispisuje redak i leksička jedinka koju uniformni znak predstavlja. Prazan drugi redak znači samo to da je na kraju prvog retka znak za novi redak, kao i u svim vježbama do sada.

Dodatno, nakon obilaska stabla, analizator treba provjeriti još neka pravila koja su zajedno s pripadnim ispisom u slučaju greške opisana u poglavlju 4.4.7.

4.4.4 Izrazi

U gramatici jezika *ppjC* postoji veći broj nezavršnih znakova koji prikazuju neku vrstu izraza, a služe prvenstveno za osiguravanje ispravnog prioriteta operatora u izrazima. U nastavku poglavlja prikazane su različite vrste izraza i pripadna semantička pravila koja proizlaze iz značenja operatora.

Izrazi imaju izvedeno svojstvo *tip* koje sadrži oznaku tipa vrijednosti izraza. Dodatno, većina izraza i završni znak `IDN` imaju izvedeno svojstvo *l-izraz* koje označava da je izrazu moguće pridružiti neku vrijednost, tj. da izraz (među ostalim) može stajati s lijeve strane operatora pridruživanja[13]. Ovo svojstvo je logičko, tj. može imati vrijednosti `0` (ako izraz *nije l-izraz*) ili `1` (ako izraz *je l-izraz*). Od završnih znakova gramatike, jedino `IDN` (identifikator) može biti *l-izraz* i to samo ako predstavlja varijablu brojevnog tipa (`char` ili `int`) bez `const`-kvalifikatora. Identifikator koji predstavlja funkciju ili niz nije *l-izraz*.

`<primarni_izraz>`

Nezavršni znak `<primarni_izraz>` generira najjednostavnije izraze koji se sastoje od jednog identifikatora, neke vrste konstante ili izraza u zagradi.

- `<primarni_izraz> ::= IDN`
  tip ← `IDN`.*tip*
  l-izraz ← `IDN`.*l-izraz*

  1. `IDN`.*ime* je deklarirano

  U skladu s pravilima o djelokrugu deklaracija, provjera u točki 1 provodi se u lokalnom djelokrugu, prvom ugniježđujućem djelokrugu, i tako dalje sve do (uključujući) globalnog djelokruga (ili dok se deklaracija ne pronađe, što god bude prije). Tip identifikatora i je li identifikator *l-izraz* određuje se pomoću tablice znakova.

- `<primarni_izraz> ::= BROJ`
  tip ← `int`
  l-izraz ← `0`

  1. vrijednost je u rasponu tipa `int`

- `<primarni_izraz> ::= ZNAK`
  tip ← `char`
  l-izraz ← `0`

  1. znak je ispravan po 4.3.2

- `<primarni_izraz> ::= NIZ_ZNAKOVA`
  tip ← *niz*(*const*(`char`))
  l-izraz ← `0`

[13] Svojstvo *l-izraz* u skladu s pravilima računanja efektivno označava da je izraz cjelobrojna varijabla bez `const`-kvalifikatora, opcionalno u zagradi. Ovo bi se moglo provjeriti i bez svojstva gdje je to potrebno, ali je ovo svojstvo uključeno kako bi se ilustriralo što se sa svojstvima može raditi u semantičkoj analizi osim praćenja tipova.

1. konstantni niz znakova je ispravan po 4.3.2

- `<primarni_izraz> ::= L_ZAGRADA <izraz> D_ZAGRADA`
  tip ← `<izraz>`.*tip*
  l-izraz ← `<izraz>`.*l-izraz*

  1. *provjeri*(`<izraz>`)

`<postfiks_izraz>`

Nezavršni znak `<postfiks_izraz>` generira neki primarni izraz s opcionalnim postfiks-operatorima.

- `<postfiks_izraz> ::= <primarni_izraz>`
  tip ← `<primarni_izraz>`.*tip*
  l-izraz ← `<primarni_izraz>`.*l-izraz*

  1. *provjeri*(`<primarni_izraz>`)

- `<postfiks_izraz> ::= <postfiks_izraz> L_UGL_ZAGRADA <izraz> D_UGL_ZAGRADA`
  tip ← X
  l-izraz ← $X \neq const(T)$

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*tip* $= niz(X)$
  3. *provjeri*(`<izraz>`)
  4. `<izraz>`.*tip* $\sim$ `int`

  Ova produkcija omogućuje indeksiranje nizova. Izraz kao što je `a[1][2]` je sintaksno dozvoljen, što je vidljivo iz ove i prethodne produkcije, ali predstavlja semantičku pogrešku, čak i ako je `a` tipa $niz(X)$. Naime, tip izraza `a[1]` će tada po pravilima računanja svojstva *tip* biti X. Novo indeksiranje cjelobrojnog tipa nije dozvoljeno i u koraku 2 dolazi do greške.

  U ispisu 4.20 prikazan je program koji sadrži ovu semantičku pogrešku, a u ispisu 4.21 je prikazan očekivani ispis semantičkog analizatora. Drugo indeksiranje je namjerno napisano u novom retku kako bi se ilustriralo da do semantičke pogreške dolazi upravo tada.

```
1 int main(void) {
2   int a[10];
3   a[1]
4    [2];
5   return 0;
6 }
```

Ispis 4.20: Program sa semantičkom pogreškom višestrukog indeksiranja niza.

```
1 <postfiks_izraz> ::= <postfiks_izraz> L_UGL_ZAGRADA(4,[) <izraz>
     D_UGL_ZAGRADA(4,])
2
```

Ispis 4.21: Ispis semantičkog analizatora za generativno stablo programa 4.20.

Redak je prelomljen jer ne stane na stranicu, ali između znaka `<izraz>` i znaka `D_UGL_ZAGRADA` treba se nalaziti točno jedan razmak.

- `<postfiks_izraz> ::= <postfiks_izraz> L_ZAGRADA D_ZAGRADA`
  tip ← *pov*
  l-izraz ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*tip* = *funkcija*(`void` → *pov*)

  Ova produkcija omogućuje pozivanje funkcija bez parametara, što se upravo i provjerava u točki 2. Nadalje, nije potrebno provjeriti je li funkcija koja se poziva deklarirana, zato jer će se to sigurno provjeriti u točki 1, tj. tijekom obrade nezavršnog znaka s desne strane produkcije. Povratnoj vrijednosti, bez obzira na tip, nije moguće ništa pridružiti, pa ona nikada nije *l-izraz*.

- `<postfiks_izraz> ::= <postfiks_izraz> L_ZAGRADA <lista_argumenata> D_ZAGRADA`
  tip ← *pov*
  l-izraz ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. *provjeri*(`<lista_argumenata>`)
  3. `<postfiks_izraz>`.*tip* = *funkcija*(*params* → *pov*) i redom po elementima *arg-tip* iz `<lista_argumenata>`.*tipovi* i *param-tip* iz *params* vrijedi *arg-tip* ∼ *param-tip*

  Ova produkcija omogućuje poziv funkcije s argumentima i razlikuje se od prethodne produkcije po tome što se tipovi argumenata pokušavaju implicitno pretvoriti u tipove parametara funkcije.

- `<postfiks_izraz> ::= <postfiks_izraz> (OP_INC | OP_DEC)`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*l-izraz* = `1` i `<postfiks_izraz>`.*tip* ∼ `int`

  I prefiks i postfiks inkrement operatori u dijelu promjene vrijednosti varijable imaju značenje kao naredba `v = (v.`*tip*`)(v + 1);`. Za dekrement operatore značenje je isto uz zamjenu operatora zbrajanja s operatorom oduzimanja. Drugim riječima, moguće je inkrementirati ili dekrementirati varijable tipova `int` i `char`. Provjera

vrijednosti svojstva *l-izraz* u točki dva osigurava da se radi o varijabli bez `const`-kvalifikatora, a zajedno s drugim uvjetom osigurava se da se radi o varijabli brojevnog tipa. Važno je uočiti da rezultat primjene ovih operatora više nije *l-izraz*, nego je vrijednost tipa `int`[14].

`<lista_argumenata>`

Nezavršni znak `<lista_argumenata>` generira listu argumenata za poziv funkcije, a za razliku od nezavršnih znakova koji generiraju izraze, imat će svojsto *tipovi* koje predstavlja listu tipova argumenata, s lijeva na desno.

- `<lista_argumenata> ::= <izraz_pridruzivanja>`
  tipovi ← `[ <izraz_pridruzivanja>.` *tip* `]`

  1. *provjeri*(`<izraz_pridruzivanja>`)

  Ova produkcija generira krajnje lijevi (moguće i jedini) argument i postavlja njegov tip kao jedini element liste u svojstvu *tipovi*.

- `<lista_argumenata> ::= <lista_argumenata> ZAREZ <izraz_pridruzivanja>`
  tipovi ← `<lista_argumenata>.` *tipovi* `+ [ <izraz_pridruzivanja>.` *tip* `]`

  1. *provjeri*(`<lista_argumenata>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

  Ova produkcija omogućuje nizanje argumenata odvojenih zarezom. Tip novog argumenta koji je predstavljen nezavršnim znakom `<izraz_pridruzivanja>` dodaje se na desni kraj liste tipova koji su određeni za prethodne argumente.

`<unarni_izraz>`

Nezavršni znak `<unarni_izraz>` generira izraze s opcionalnim prefiks unarnim operatorima.

- `<unarni_izraz> ::= <postfiks_izraz>`
  tip ← `<postfiks_izraz>.` *tip*
  l-izraz ← `<postfiks_izraz>.` *l-izraz*

  1. *provjeri*(`<postfiks_izraz>`)

- `<unarni_izraz> ::= (OP_INC | OP_DEC) <unarni_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<unarni_izraz>`)
  2. `<unarni_izraz>.` *l-izraz* $=$ `1` i `<unarni_izraz>.` *tip* $\sim$ `int`

[14] Značenje operatora u širem kontekstu izraza je vrlo slično kao u *C*-u, ali nije važno za potrebe semantičke analize.

Prefiks inkrement i dekrement imaju analogna semantička pravila postfiks inačicama istih operatora.

- `<unarni_izraz> ::= <unarni_operator> <cast_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<cast_izraz>`)
  2. `<cast_izraz>`.*tip* ∼ `int`

  Unarni operatori primjenjivi su na vrijednosti tipa `int` što se provjerava u točki 2, a rezultat je opet tipa `int`. Iako je u produkciji nezavršni znak `<unarni_operator>`, nije potrebno provjeravati nikakva semantička pravila u toj grani stabla jer taj nezavršni znak jednostavno generira neki od unarnih operatora (što je prikazano u nastavku). Konačno, bez obzira na to je li `<cast_izraz>` *l-izraz* ili ne, rezultat primjene unarnog operatora je samo *vrijednost* i nije *l-izraz*. Na primjer, naredba `+a = 3;` je semantički neispravna zbog ovog pravila.

`<unarni_operator>`

Nezavršni znak `<unarni_operator>` generira aritmetičke (`PLUS` i `MINUS`), bitovne (`OP_TILDA`) i logičke (`OP_NEG`) prefiks unarne operatore. Kako u ovim produkcijama u semantičkoj analizi ne treba ništa provjeriti, produkcije ovdje nisu navedene.

`<cast_izraz>`

Nezavršni znak `<cast_izraz>` generira izraze s opcionalnim *cast* operatorom.

- `<cast_izraz> ::= <unarni_izraz>`
  tip ← `<unarni_izraz>`.*tip*
  l-izraz ← `<unarni_izraz>`.*l-izraz*

  1. *provjeri*(`<unarni_izraz>`)

- `<cast_izraz> ::= L_ZAGRADA <ime_tipa> D_ZAGRADA <cast_izraz>`
  tip ← `<ime_tipa>`.*tip*
  l-izraz ← `0`

  1. *provjeri*(`<ime_tipa>`)
  2. *provjeri*(`<cast_izraz>`)
  3. `<cast_izraz>`.*tip* se može pretvoriti u `<ime_tipa>`.*tip* po poglavlju 4.3.1

  Ova produkcija omogućuje eksplicitne promjene tipa vrijednosti. Kako se nezavršni znak `<ime_tipa>` koristi na više mjesta u gramatici, on može generirati i tip `void` iako će to u kontekstu *cast* operatora uvijek biti semantička greška. Nadalje, vrijednost kojoj se tip pokušava promijeniti može biti bilo kojeg tipa, pa se u točki 2 mora provjeriti da se radi o brojevnom tipu, u skladu s pravilima iz poglavlja 4.3.1.

  Na primjer, svi izrazi u ispisu 4.22 su semantički ispravni, pod pretpostavkom da su varijable `x` i `y` brojevnog tipa.

```
(int)'a'
(const char)x
(const int)'a'
(char)((const int)300 + (int)'a')
(int)(char)(const int)(const char)(x + y)
```

Ispis 4.22: Ispravne primjene *cast*-operatora.

`<ime_tipa>`

Nezavršni znak `<ime_tipa>` generira imena opcionalno `const`-kvalificiranih brojevnih tipova i ključnu riječ `void`. U ovim produkcijama će se izračunati izvedeno svojstvo *tip* koje se koristi u produkcijama gdje se `<ime_tipa>` pojavljuje s desne strane i dodatno će se onemogućiti tip `const void` (koji je sintaksno ispravan, ali nema smisla).

- `<ime_tipa> ::= <specifikator_tipa>`
  tip ← `<specifikator_tipa>`.*tip*

  1. *provjeri*(`<specifikator_tipa>`)

  Prva produkcija koristi se za tipove koji nisu `const`-kvalificirani.

- `<ime_tipa> ::= KR_CONST <specifikator_tipa>`
  tip ← *const*(`<specifikator_tipa>`.*tip*)

  1. *provjeri*(`<specifikator_tipa>`)
  2. `<specifikator_tipa>`.*tip* $\neq$ `void`

  U drugoj produkciji generiraju se `const`-kvalificirani tipovi. Kao što je prije spomenuto, u točki 2 se onemogućuje tip `const void`.

`<specifikator_tipa>`

Nezavršni znak `<specifikator_tipa>` generira jedan od tri završna znaka `KR_VOID`, `KR_CHAR` i `KR_INT`. U semantičkoj analizi ćemo iz završnog znaka odrediti vrijednost svojstva *tip* nezavršnog znaka, ali u ovim produkcijama ne može doći do semantičke pogreške.

- `<specifikator_tipa> ::= KR_VOID`
  tip ← `void`

- `<specifikator_tipa> ::= KR_CHAR`
  tip ← `char`

- `<specifikator_tipa> ::= KR_INT`
  tip ← `int`

`<multiplikativni_izraz>`

Nezavršni znak `<multiplikativni_izraz>` generira izraze u kojima se opcionalno koriste operatori množenja, dijeljenja i ostatka. Struktura produkcija osigurava da sva tri operatora imaju isti prioritet i to manji od unarnih (prefiks i postfiks) operatora, a veći od ostalih operatora. Nadalje, lijeva asocijativnost ovih operatora osigurana je lijevom rekurzijom u produkcijama.

Svi ostali binarni operatori u jeziku (čija pravila su prikazana kasnije) ostvareni su sličnim produkcijama i provjeravaju se slična pravila.

- `<multiplikativni_izraz> ::= <cast_izraz>`
  tip ← `<cast_izraz>`.*tip*
  l-izraz ← `<cast_izraz>`.*l-izraz*

  1. *provjeri*(`<cast_izraz>`)

- `<multiplikativni_izraz> ::= <multiplikativni_izraz> (OP_PUTA | OP_DIJELI | OP_MOD) <cast_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<multiplikativni_izraz>`)
  2. `<multiplikativni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<cast_izraz>`)
  4. `<cast_izraz>`.*tip* ∼ `int`

  Ova produkcija generira operator množenja, dijeljenja ili ostatka između dva podizraza. Kao i većina binarnih operatora u jeziku, ovi operatiri su definirani samo nad vrijednostima tipa `int` i rezultat provođenja operacije je opet tipa `int`. U točkama 2 i 4 provjerava se mogu li se lijevi i desni operand pretvoriti u vrijednosti tipa `int`. Važno je uočiti da znak `<multiplikativni_izraz>` može imati bilo koji tip (iako je rezultat primjene bilo kojeg od ova tri operatora tipa `int`) zato što u prethodnoj produkciji `<multiplikativni_izraz>` preuzima tip od znaka `<cast_izraz>` i potencijalno tako dalje sve do znaka `<primarni_izraz>`.

`<aditivni_izraz>`

Nezavršni znak `<aditivni_izraz>` generira izraze u kojima se opcionalno koriste operatori zbrajanja i oduzimanja.

- `<aditivni_izraz> ::= <multiplikativni_izraz>`
  tip ← `<multiplikativni_izraz>`.*tip*
  l-izraz ← `<multiplikativni_izraz>`.*l-izraz*

  1. *provjeri*(`<multiplikativni_izraz>`)

- `<aditivni_izraz> ::= <aditivni_izraz> (PLUS | MINUS) <multiplikativni_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<aditivni_izraz>`)
  2. `<aditivni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<multiplikativni_izraz>`)
  4. `<multiplikativni_izraz>`.*tip* ∼ `int`

`<odnosni_izraz>`

Nezavršni znak `<odnosni_izraz>` generira izraze u kojima se opcionalno koriste odnosni operatori `<` (uniformni znak `OP_LT`), `>` (uniformni znak `OP_GT`), `<=` (uniformni znak `OP_LTE`) i `>=` (uniformni znak `OP_GTE`).

- `<odnosni_izraz> ::= <aditivni_izraz>`
  tip ← `<aditivni_izraz>`.*tip*
  l-izraz ← `<aditivni_izraz>`.*l-izraz*

  1. *provjeri*(`<aditivni_izraz>`)

- `<odnosni_izraz> ::= <odnosni_izraz> (OP_LT | OP_GT | OP_LTE | OP_GTE) <aditivni_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<odnosni_izraz>`)
  2. `<odnosni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<aditivni_izraz>`)
  4. `<aditivni_izraz>`.*tip* ∼ `int`

`<jednakosni_izraz>`

Nezavršni znak `<jednakosni_izraz>` generira izraze u kojima se opcionalno koriste jednakosni operatori `==` (uniformni znak `OP_EQ`) i `!=` (uniformni znak `OP_NEQ`).

- `<jednakosni_izraz> ::= <odnosni_izraz>`
  tip ← `<odnosni_izraz>`.*tip*
  l-izraz ← `<odnosni_izraz>`.*l-izraz*

  1. *provjeri*(`<odnosni_izraz>`)

- `<jednakosni_izraz> ::= <jednakosni_izraz> (OP_EQ | OP_NEQ) <odnosni_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<jednakosni_izraz>`)

2. `<jednakosni_izraz>`.*tip* ∼ `int`
3. *provjeri*(`<odnosni_izraz>`)
4. `<odnosni_izraz>`.*tip* ∼ `int`

`<bin_i_izraz>`

Nezavršni znak `<bin_i_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `&` (uniformni znak `OP_BIN_I`)[15]. Bitovni operatori imaju različite prioritete pa zato svaki operator ima pripadni nezavršni znak.

- `<bin_i_izraz> ::= <jednakosni_izraz>`
  tip ← `<jednakosni_izraz>`.*tip*
  l-izraz ← `<jednakosni_izraz>`.*l-izraz*
  1. *provjeri*(`<jednakosni_izraz>`)
- `<bin_i_izraz> ::= <bin_i_izraz> OP_BIN_I <jednakosni_izraz>`
  tip ← `int`
  l-izraz ← `0`
  1. *provjeri*(`<bin_i_izraz>`)
  2. `<bin_i_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<jednakosni_izraz>`)
  4. `<jednakosni_izraz>`.*tip* ∼ `int`

`<bin_xili_izraz>`

Nezavršni znak `<bin_xili_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `^` (uniformni znak `OP_BIN_XILI`).

- `<bin_xili_izraz> ::= <bin_i_izraz>`
  tip ← `<bin_i_izraz>`.*tip*
  l-izraz ← `<bin_i_izraz>`.*l-izraz*
  1. *provjeri*(`<bin_i_izraz>`)
- `<bin_xili_izraz> ::= <bin_xili_izraz> OP_BIN_XILI <bin_i_izraz>`
  tip ← `int`
  l-izraz ← `0`
  1. *provjeri*(`<bin_xili_izraz>`)
  2. `<bin_xili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_i_izraz>`)
  4. `<bin_i_izraz>`.*tip* ∼ `int`

[15] Smislenije ime nezavršnog i završnog znaka sadržavalo bi *bit* umjesto *bin*, ali ova greška u imenu postoji zbog konzistentnosti s drugim materijalima.

`<bin_ili_izraz>`

Nezavršni znak `<bin_ili_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `|` (uniformni znak `OP_BIN_ILI`).

- `<bin_ili_izraz> ::= <bin_xili_izraz>`
  tip ← `<bin_xili_izraz>`.*tip*
  l-izraz ← `<bin_xili_izraz>`.*l-izraz*

  1. *provjeri*(`<bin_xili_izraz>`)

- `<bin_ili_izraz> ::= <bin_ili_izraz> OP_BIN_ILI <bin_xili_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<bin_ili_izraz>`)
  2. `<bin_ili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_xili_izraz>`)
  4. `<bin_xili_izraz>`.*tip* ∼ `int`

`<log_i_izraz>`

Nezavršni znak `<log_i_izraz>` generira izraze u kojima se opcionalno koristi logički operator konjunkcije `&&` (uniformni znak `OP_I`). Slično kao za bitovne operatore, kako logički operatori imaju različite prioritete svakom je pridružen vlastiti nezavršni znak.

- `<log_i_izraz> ::= <bin_ili_izraz>`
  tip ← `<bin_ili_izraz>`.*tip*
  l-izraz ← `<bin_ili_izraz>`.*l-izraz*

  1. *provjeri*(`<bin_ili_izraz>`)

- `<log_i_izraz> ::= <log_i_izraz> OP_I <bin_ili_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<log_i_izraz>`)
  2. `<log_i_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_ili_izraz>`)
  4. `<bin_ili_izraz>`.*tip* ∼ `int`

`<log_ili_izraz>`

Nezavršni znak `<log_ili_izraz>` generira izraze u kojima se opcionalno koristi logički operator disjunkcije `||` (uniformni znak `OP_ILI`).

- `<log_ili_izraz> ::= <log_i_izraz>`
  tip ← `<log_i_izraz>`.*tip*
  l-izraz ← `<log_i_izraz>`.*l-izraz*

  1. *provjeri*(`<log_i_izraz>`)

- `<log_ili_izraz> ::= <log_ili_izraz> OP_ILI <log_i_izraz>`
  tip ← `int`
  l-izraz ← `0`

  1. *provjeri*(`<log_ili_izraz>`)
  2. `<log_ili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<log_i_izraz>`)
  4. `<log_i_izraz>`.*tip* ∼ `int`

`<izraz_pridruzivanja>`

Nezavršni znak `<izraz_pridruzivanja>` generira izraze u kojima se neka vrijednost opcionalno pridružuje varijabli koristeći operator pridruživanja `=` (uniformni znak `OP_PRIDRUZI`). Za razliku od prethodno prikazanih binarnih operatora, operator pridruživanja je desno asocijativan što je u gramatici osigurano primjenom desne rekurzije. Desna asocijativnost omogućuje nizanje pridruživanja, npr. `a = b = c = 42;`.

- `<izraz_pridruzivanja> ::= <log_ili_izraz>`
  tip ← `<log_ili_izraz>`.*tip*
  l-izraz ← `<log_ili_izraz>`.*l-izraz*

  1. *provjeri*(`<log_ili_izraz>`)

- `<izraz_pridruzivanja> ::= <postfiks_izraz> OP_PRIDRUZI <izraz_pridruzivanja>`
  tip ← `<postfiks_izraz>`.*tip*
  l-izraz ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*l-izraz* = `1`
  3. *provjeri*(`<izraz_pridruzivanja>`)
  4. `<izraz_pridruzivanja>`.*tip* ∼ `<postfiks_izraz>`.*tip*

  Na primjer, u skladu s pravilima računanja svojstva *l-izraz*, za varijablu `a` tipa `int` semantički su ispravna pridruživanja `a = 'a';` i `(a) = 10;`.

`<izraz>`

Nezavršni znak `<izraz>` omogućuje opcionalno nizanje izraza koristeći operator `,` (uniformni znak `ZAREZ`). Vrijednost takvog složenog izraza jednaka je vrijednosti krajnje desnog izraza u nizu.

- `<izraz> ::= <izraz_pridruzivanja>`
  tip ← `<izraz_pridruzivanja>`.*tip*
  l-izraz ← `<izraz_pridruzivanja>`.*l-izraz*

  1. *provjeri*(`<izraz_pridruzivanja>`)

- `<izraz> ::= <izraz> ZAREZ <izraz_pridruzivanja>`
  tip ← `<izraz_pridruzivanja>`.*tip*
  l-izraz ← `0`

  1. *provjeri*(`<izraz>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

4.4.5 Naredbena struktura programa

Programi u jeziku *ppjC* se sastoje od deklaracija i definicija varijabli i funkcija. Tijelo svake funkcije je *složena naredba* tj. blok. Blokovi se sastoje od deklaracija funkcija, definicija (ujedno i deklaracija) varijabli i niza naredbi. Ova struktura programa ostvarena je nezavršnim znakovima i produkcijama opisanim u nastavku. Skup produkcija za naredbenu strukturu programa semantički je znatno jednostavniji od produkcija za izraze.

`<slozena_naredba>`

Nezavršni znak `<slozena_naredba>` predstavlja blok naredbi koji opcionalno počinje listom deklaracija. Svaki blok je odvojeni djelokrug, a nelokalnim imenima se pristupa u ugniježđujućem bloku (i potencijalno tako dalje sve do globalnog djelokruga).

- `<slozena_naredba> ::= L_VIT_ZAGRADA <lista_naredbi> D_VIT_ZAGRADA`

  1. *provjeri*(`<lista_naredbi>`)

  Ova produkcija generira blok koji nema vlastite deklaracije (ali neka od naredbi u bloku može biti novi blok koji ima lokalne deklaracije).

- `<slozena_naredba> ::= L_VIT_ZAGRADA <lista_deklaracija> <lista_naredbi> D_VIT_ZAGRADA`

  1. *provjeri*(`<lista_deklaracija>`)
  2. *provjeri*(`<lista_naredbi>`)

  S druge strane, ova produkcija generira blok s lokalnim deklaracijama. Kao i u jeziku *C*, deklaracije su dozvoljene samo na početku bloka.

`<lista_naredbi>`

Nezavršni znak `<lista_naredbi>` omogućuje nizanje naredbi u bloku.

- `<lista_naredbi> ::= <naredba>`

1. *provjeri*(<naredba>)

- <lista_naredbi> ::= <lista_naredbi> <naredba>

  1. *provjeri*(<lista_naredbi>)
  2. *provjeri*(<naredba>)

<naredba>

Nezavršni znak <naredba> generira blokove (<slozena_naredba>) i različite vrste jednostavnih naredbi (<izraz_naredba>, <naredba_grananja>, <naredba_petlje> i <naredba_skoka>). Kako su sve produkcije jedinične (s desne strane imaju jedan nezavršni znak) i u svim produkcijama se provjeravaju semantička pravila na znaku s desne strane, produkcije ovdje nisu prikazane.

<izraz_naredba>

Nezavršni znak <izraz_naredba> generira opcionalni izraz i znak ; (uniformni znak TOCKAZAREZ) i predstavlja jednostavnu naredbu. Za potrebe uvjeta u for-petlji, znaku <izraz_naredba> se pridjeljuje izvedeno svojstvo *tip*.

- <izraz_naredba> ::= TOCKAZAREZ
  tip ← int

  Ova produkcija generira "praznu naredbu" koja može biti korisna kao tijelo petlje koja ne radi ništa i slično. Prazna naredba će imati tip int kako bi se mogla koristiti kao uvijek zadovoljen uvjet u for-petlji (na primjer u kakonskoj beskonačnoj petlji for(;;)).

- <izraz_naredba> ::= <izraz> TOCKAZAREZ
  tip ← <izraz>.*tip*

  1. *provjeri*(<izraz>)

  Za zadani izraz, svojstvo *tip* se preuzima od znaka <izraz>.

<naredba_grananja>

Nezavršni znak <naredba_grananja> generira if naredbu u jeziku.

- <naredba_grananja> ::= KR_IF L_ZAGRADA <izraz> D_ZAGRADA <naredba>

  1. *provjeri*(<izraz>)
  2. <izraz>.*tip* ∼ int
  3. *provjeri*(<naredba>)

  Ova produkcija generira if naredbu bez else dijela.

- `<naredba_grananja> ::= KR_IF L_ZAGRADA <izraz> D_ZAGRADA <naredba>`$_1$ `KR_ELSE <naredba>`$_2$

  1. *provjeri*(`<izraz>`)
  2. `<izraz>`.*tip* $\sim$ `int`
  3. *provjeri*(`<naredba>`$_1$)
  4. *provjeri*(`<naredba>`$_2$)

`<naredba_petlje>`

Nezavršni znak `<naredba_petlje>` generira `while` i `for` petlje.

- `<naredba_petlje> ::= KR_WHILE L_ZAGRADA <izraz> D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz>`)
  2. `<izraz>`.*tip* $\sim$ `int`
  3. *provjeri*(`<naredba>`)

- `<naredba_petlje> ::= KR_FOR L_ZAGRADA <izraz_naredba>`$_1$ `<izraz_naredba>`$_2$ `D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz_naredba>`$_1$)
  2. *provjeri*(`<izraz_naredba>`$_2$)
  3. `<izraz_naredba>`$_2$.*tip* $\sim$ `int`
  4. *provjeri*(`<naredba>`)

  Ova produkcija generira `for`-petlju bez opcionalnog izraza koji se tipično koristi za promjenu indeksa petlje dok sljedeća produkcija generira petlju sa tim izrazom.

- `<naredba_petlje> ::= KR_FOR L_ZAGRADA <izraz_naredba>`$_1$ `<izraz_naredba>`$_2$ `<izraz> D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz_naredba>`$_1$)
  2. *provjeri*(`<izraz_naredba>`$_2$)
  3. `<izraz_naredba>`$_2$.*tip* $\sim$ `int`
  4. *provjeri*(`<izraz>`)
  5. *provjeri*(`<naredba>`)

`<naredba_skoka>`

Nezavršni znak `<naredba_skoka>` generira `continue`, `break` i `return` naredbe.

- `<naredba_skoka> ::= (KR_CONTINUE | KR_BREAK) TOCKAZAREZ`

  1. naredba se nalazi unutar petlje ili unutar bloka koji je ugniježđen u petlji

I naredba `continue` (uniformni znak `KR_CONTINUE`) i naredba `break` (uniformni znak `KR_BREAK`) dozvoljene su isključivo unutar neke petlje, a imaju isto značenje kao u jeziku *C*.

- `<naredba_skoka> ::= KR_RETURN TOCKAZAREZ`

  1. naredba se nalazi unutar funkcije tipa *funkcija*(*params* $\rightarrow$ `void`)

  Naredba `return` bez povratne vrijednosti može se koristiti jedino u funkcijama koje ne vraćaju ništa.

- `<naredba_skoka> ::= KR_RETURN <izraz> TOCKAZAREZ`

  1. *provjeri*(`<izraz>`)
  2. naredba se nalazi unutar funkcije tipa *funkcija*(*params* $\rightarrow$ *pov*) i vrijedi `<izraz>`.*tip* $\sim$ *pov*

`<prijevodna_jedinica>`

Nezavršni znak `<prijevodna_jedinica>` je početni nezavršni znak gramatike i generira niz nezavršnih znakova `<vanjska_deklaracija>` koji generiraju definicije (i deklaracije) u globalnom djelokrugu programa.

- `<prijevodna_jedinica> ::= <vanjska_deklaracija>`

  1. *provjeri*(`<vanjska_deklaracija>`)

- `<prijevodna_jedinica> ::= <prijevodna_jedinica> <vanjska_deklaracija>`

  1. *provjeri*(`<prijevodna_jedinica>`)
  2. *provjeri*(`<vanjska_deklaracija>`)

`<vanjska_deklaracija>`

Nezavršni znak `<vanjska_deklaracija>` generira ili definiciju funkcije (znak `<definicija_funkcije>`) ili deklaraciju varijable ili funkcije (znak `<deklaracija>`). Obje produkcije su jedinične i u obje se provjeravaju pravila u podstablu kojem je znak s desne strane korijen.

4.4.6 Deklaracije i definicije

Preostale produkcije u gramatici odnose se na deklaracije i definicije. U semantičkoj analizi se na osnovi dijela generativnog stabla označenog ovim produkcijama gradi (ili dopunjuje) tablica znakova.

`<definicija_funkcije>`

- `<definicija_funkcije> ::= <ime_tipa> IDN L_ZAGRADA KR_VOID D_ZAGRADA <slozena_naredba>`

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>`.*tip* $\neq$ *const*(T)
  3. ne postoji prije definirana funkcija imena `IDN`.*ime*
  4. ako postoji deklaracija imena `IDN`.*ime* u globalnom djelokrugu onda je pripadni tip te deklaracije *funkcija*(`void` $\rightarrow$ `<ime_tipa>`.*tip*)
  5. zabilježi definiciju i deklaraciju funkcije
  6. *provjeri*(`<slozena_naredba>`)

  Točka 2 u skladu s definicijom znaka `<ime_tipa>` osigurava da je povratni tip funkcije `int`, `char` ili `void`. U točki 3 osigurava se pravilo da svaka funkcija može biti definirana najviše jednom. Konačno, u točki 4 se provjerava da prethodne deklaracije te funkcije u **globalnom djelokrugu** imaju isti tip kao i sama definicija funkcije. Provjerava se globalni djelokrug jer sintaksa jezika osigurava da se sve funkcije definiraju u globalnom djelokrugu.

  Ako su sva ova pravila zadovoljena, definicija i deklaracija funkcije se zapisuju u odgovarajuće strukture podataka **prije** obrade podstabla kojem je korijen označen znakom `<slozena_naredba>`. Drugim riječima, funkcija je u svom tijelu već deklarirana i može se rekurzivno pozivati.

  Djelokrug definiran složenom naredbom, tj. tijelom funkcije, pridružuje se definiciji funkcije, npr. kako bi bilo moguće provjeriti pravila vezana uz `return` naredbu.

- `<definicija_funkcije> ::= <ime_tipa> IDN L_ZAGRADA <lista_parametara> D_ZAGRADA <slozena_naredba>`

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>`.*tip* $\neq$ *const*(T)
  3. ne postoji prije definirana funkcija imena `IDN`.*ime*
  4. *provjeri*(`<lista_parametara>`)
  5. ako postoji deklaracija imena `IDN`.*ime* u globalnom djelokrugu onda je pripadni tip te deklaracije *funkcija*(`<lista_parametara>`.*tipovi* $\rightarrow$ `<ime_tipa>`.*tip*)
  6. zabilježi definiciju i deklaraciju funkcije
  7. *provjeri*(`<slozena_naredba>`) uz parametre funkcije koristeći `<lista_parametara>`.*tipovi* i `<lista_parametara>`.*imena*.

  Ova produkcija generira definicije funkcija s listom parametara, tj. funkcije koje primaju jedan ili više argumenata. Za točku 7 je zato važno osigurati da se prije provjere pravila u tijelu funkcije u lokalni djelokrug ugrade parametri funkcije.

`<lista_parametara>`

Nezavršnom znaku `<lista_parametara>` pridružit ćemo svojstvo *tipovi* koje sadrži listu tipova parametara i svojstvo *imena* koje sadrži imena parametara. Vrijednosti svojstva grade se analogno kao kod znaka `<lista_argumenata>`.

- `<lista_parametara> ::= <deklaracija_parametra>`
  tipovi ← `[ <deklaracija_parametra>.`*tip*` ]`
  imena ← `[ <deklaracija_parametra>.`*ime*` ]`

  1. *provjeri*(`<deklaracija_parametra>`)

- `<lista_parametara> ::= <lista_parametara> ZAREZ <deklaracija_parametra>`
  tipovi ← `<lista_parametara>.`*tipovi* `+ [ <deklaracija_parametra>.`*tip*` ]`
  imena ← `<lista_parametara>.`*imena* `+ [ <deklaracija_parametra>.`*ime*` ]`

  1. *provjeri*(`<lista_parametara>`)
  2. *provjeri*(`<deklaracija_parametra>`)
  3. `<deklaracija_parametra>.`*ime* ne postoji u `<lista_parametara>.`*imena*

  U točki 3 se provjerava jedinstvenost imena parametara u jednoj deklaraciji funkcije.

`<deklaracija_parametra>`

Nezavršni znak `<deklaracija_parametra>` služi za deklaraciju jednog parametra i ima svojstva *tip* i *ime*.

- `<deklaracija_parametra> ::= <ime_tipa> IDN`
  tip ← `<ime_tipa>.`*tip*
  ime ← `IDN.`*ime*

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>.`*tip* $\neq$ `void`

  Ova produkcija generira deklaraciju cjelobrojnog parametra.

- `<deklaracija_parametra> ::= <ime_tipa> IDN L_UGL_ZAGRADA D_UGL_ZAGRADA`
  tip ← *niz*(`<ime_tipa>.`*tip*)
  ime ← `IDN.`*ime*

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>.`*tip* $\neq$ `void`

  Ova produkcija generira parametre koji su nizovi.

`<lista_deklaracija>`

Nezavršni znak `<lista_deklaracija>` generira jednu ili više deklaracija na početku bloka.

- `<lista_deklaracija> ::= <deklaracija>`
  1. *provjeri*(`<deklaracija>`)
- `<lista_deklaracija> ::= <lista_deklaracija> <deklaracija>`
  1. *provjeri*(`<lista_deklaracija>`)
  2. *provjeri*(`<deklaracija>`)

`<deklaracija>`

Nezavršni znak `<deklaracija>` generira jednu naredbu deklaracije.

- `<deklaracija> ::= <ime_tipa> <lista_init_deklaratora> TOCKAZAREZ`
  1. *provjeri*(`<ime_tipa>`)
  2. *provjeri*(`<lista_init_deklaratora>`) uz nasljedno svojstvo `<lista_init_deklaratora>`.*ntip* ← `<ime_tipa>`.*tip*

Specifičnost točke 2 je nasljedno svojstvo *ntip* nezavršnog znaka `<lista_init_deklaratora>`. Svojstvo *ntip* služi za prijenos jednog dijela informacije o tipu u sve deklaratore. Za varijable brojevnog tipa *ntip* će biti cijeli tip, za nizove će biti tip elementa niza, a za funkcije će biti povratni tip.

`<lista_init_deklaratora>`

Nezavršni znak `<lista_init_deklaratora>` generira deklaratore odvojene zarezima. Na primjer, u naredbi `int x, y=3, z=y+1;`, znak `<lista_init_deklaratora>` generira `x, y=3, z=y+1` dio (dakako, generira odgovarajuće uniformne znakove).

- `<lista_init_deklaratora> ::= <init_deklarator>`
  1. *provjeri*(`<init_deklarator>`) uz nasljedno svojstvo `<init_deklarator>`.*ntip* ← `<lista_init_deklaratora>`.*ntip*
- $\texttt{<lista\_init\_deklaratora>}_1$ `::=` $\texttt{<lista\_init\_deklaratora>}_2$ `ZAREZ <init_deklarator>`
  1. *provjeri*($\texttt{<lista\_init\_deklaratora>}_2$) uz nasljedno svojstvo $\texttt{<lista\_init\_deklaratora>}_2$.*ntip* ← $\texttt{<lista\_init\_deklaratora>}_1$.*ntip*
  2. *provjeri*(`<init_deklarator>`) uz nasljedno svojstvo `<init_deklarator>`.*ntip* ← $\texttt{<lista\_init\_deklaratora>}_1$.*ntip*

`<init_deklarator>`

Nezavršni znak `<init_deklarator>` generira deklarator s opcionalnim inicijalizatorom.

- `<init_deklarator> ::= <izravni_deklarator>`

  1. *provjeri*(`<izravni_deklarator>`) uz nasljedno svojstvo
     `<izravni_deklarator>`.*ntip* ← `<init_deklarator>`.*ntip*
  2. `<izravni_deklarator>`.*tip* $\neq$ *const*(T)
     i
     `<izravni_deklarator>`.*tip* $\neq$ *niz*(*const*(T))

  Točka 2 osigurava da `const`-kvalificirane varijable i nizovi `const`-kvalificiranih tipova moraju biti inicijalizirani pri definiciji kako bi imali određenu vrijednost. Važno je uočiti da se provjerava izvedeno svojstvo *tip* znaka `<izravni_deklarator>`, a ne nasljedno svojstvo *ntip*.

- `<init_deklarator> ::= <izravni_deklarator> OP_PRIDRUZI <inicijalizator>`

  1. *provjeri*(`<izravni_deklarator>`) uz nasljedno svojstvo
     `<izravni_deklarator>`.*ntip* ← `<init_deklarator>`.*ntip*
  2. *provjeri*(`<incijalizator>`)
  3. ako je `<izravni_deklarator>`.*tip* T ili *const*(T)
     `<inicijalizator>`.*tip* $\sim T$
     inače ako je `<izravni_deklarator>`.*tip* *niz*(T) ili *niz*(*const*(T))
     `<inicijalizator>`.*br-elem* $\leq$ `<izravni_deklarator>`.*br-elem*
     za svaki U iz `<inicijalizator>`.*tipovi* vrijedi $U \sim T$
     inače *greška*

  Točka 3 provjerava semantičku ispravnost inicijalizacije. Za brojevne tipove je dovoljno da se tip inicijalizatora može implicitno pretvoriti u odgovarajući tip **bez** `const`-kvalifikatora (inače ne bi bilo moguće inicijalizirati varijable `const`-kvalificiranog tipa vrijednostima koje nisu `const`-kvalificiranog tipa; npr. `const int x = 3;` bi bila pogreška).

  Nadalje, za nizove treba provjeriti je li broj elemenata u inicijalizatoru manji ili jednak broju elemenata niza i mogu li se elementi inicijalizatora implicitno pretvoriti u odgovarajući tip, isto kao u slučaju brojevnih tipova.

  Konačno, u svim ostalim slučajevima (a to je jedino deklaracija funkcije), program ima semantičku pogrešku.

`<izravni_deklarator>`

Nezavršni znak `<izravni_deklarator>` generira deklarator varijable ili funkcije. Znak ima nasljedno svojstvo *ntip* i izvedeno svojstvo *tip* u koje se pohranjuje potpuni tip varijable ili funkcije. Ako se deklarira niz, znak dodatno ima i izvedeno svojstvo *br-elem* koje označava broj elemenata niza.

- `<izravni_deklarator> ::= IDN`
  tip $\leftarrow$ *ntip*

  1. *ntip* $\neq$ `void`
  2. `IDN.`*ime* nije deklarirano u lokalnom djelokrugu
  3. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom

  Ova produkcija služi za generiranje varijabli cjelobrojnog tipa. Važno je uočiti da je varijabla deklarirana odmah nakon navedenog identifikatora, a prije opcionalnog inicijalizatora. To znači da je inicijalizacija `int x = x + 1;` semantički ispravna, ali rezultat nije definiran jer `x` na desnoj strani ima neodređenu vrijednost.

- `<izravni_deklarator> ::= IDN L_UGL_ZAGRADA BROJ D_UGL_ZAGRADA`
  tip $\leftarrow$ *niz*(*ntip*)
  br-elem $\leftarrow$ `BROJ.`*vrijednost*

  1. *ntip* $\neq$ `void`
  2. `IDN.`*ime* nije deklarirano u lokalnom djelokrugu
  3. `BROJ.`*vrijednost* je pozitivan broj (> 0) ne veći od `1024`
  4. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom

  Ova produkcija služi za deklariranje nizova. Obavezno mora biti naveden broj elemenata niza (to je sintaksno osigurano ovom produkcijom) i taj broj mora biti pozitivan i maksimalnog iznosa `1024`.

  Sintaksno nije moguće broj elemenata zadati nekakvim izrazom, čak ni ako se on u cijelosti sastoji od konstanti.

- `<izravni_deklarator> ::= IDN L_ZAGRADA KR_VOID D_ZAGRADA`
  tip $\leftarrow$ *funkcija*(`void` $\rightarrow$ *ntip*)

  1. ako je `IDN.`*ime* deklarirano u lokalnom djelokrugu, tip prethodne deklaracije je jednak *funkcija*(`void` $\rightarrow$ *ntip*)
  2. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom ako ista funkcija već nije deklarirana u lokalnom djelokrugu

- `<izravni_deklarator> ::= IDN L_ZAGRADA <lista_parametara> D_ZAGRADA`
  tip $\leftarrow$ *funkcija*(`<lista_parametara>.`*tipovi* $\rightarrow$ *ntip*)

  1. *provjeri*(`<lista_parametara>`)
  2. ako je `IDN.`*ime* deklarirano u lokalnom djelokrugu, tip prethodne deklaracije je jednak *funkcija*(`<lista_parametara>.`*tipovi* $\rightarrow$ *ntip*)
  3. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom ako ista funkcija već nije deklarirana u lokalnom djelokrugu

  Ova i prethodna produkcija generiraju deklaracije funkcija. Za razliku od varijabli, funkcije se mogu deklarirati u istom djelokrugu proizvoljan broj puta (zato jer je deklaracija varijable ujedno i njena definicija, što kod funkcija nije slučaj).

`<inicijalizator>`

Nezavršni znak `<inicijalizator>` generira izraz ili složeni inicijalizator za nizove. Za inicijalizaciju varijabli brojevnog tipa, `<inicijalizator>` će imati izvedeno svojstvo *tip* koje će sadržavati tip izraza s kojim se varijabla pokušava inicijalizirati. Za inicijalizaciju nizova, `<inicijalizato>` će imati izvedena svojstva *br-elem* i *tipovi*. Svojstvo *br-elem* sadržavat će broj elemenata inicijalizatora, a svojstvo *tipovi* će biti lista tipova izraza u složenom inicijalizatoru.

- `<inicijalizator> ::= <izraz_pridruzivanja>`
  ako je `<izraz_pridruzivanja>` $\overset{*}{\Rightarrow}$ `NIZ_ZNAKOVA`
  br-elem $\leftarrow$ duljina niza znakova `+ 1`
  tipovi $\leftarrow$ lista duljine *br-elem*, svi elementi su `char`
  inače
  tip $\leftarrow$ `<izraz_pridruzivanja>.` *tip*

  1. *provjeri*(`<izraz_pridruzivanja>`)

  Kao podsjetnik iz UTR-a, simbol $\overset{*}{\Rightarrow}$ označava da se iz lijeve strane primjenom proizvoljnog broja produkcija generira desna strana. U ovoj gramatici, navedena relacija je zadovoljena samo ako se iz znaka `<izraz_pridruzivanja>` primjenom jediničnih produkcija konačno generira `NIZ_ZNAKOVA`, tj. slijed generiranja je `<izraz_pridruzivanja>` $\Rightarrow$ `<log_ili_izraz>` $\Rightarrow$ ... $\Rightarrow$ `<primarni_izraz>` $\Rightarrow$ `NIZ_ZNAKOVA`. Taj poseban slučaj u pravilima računanja svojstava u ovoj produkciji služi za poistovjećivanje inicijalizacije znakovnih nizova konstantnim nizom znakova (uniformni znak `NIZ_ZNAKOVA`) sa istovjetnim složenim inicijalizatorom.

```
1 char a[10] = "abc"; // isto kao redak 2
2 const char b[10] = { 'a', 'b', 'c', '\0' };
3 char c[10] = a; // greska
```

Ispis 4.23: Dva istovjetna načina inicijalizacije niza znakova i greška.

Nadalje, važno je uočiti da se nizovi ne mogu inicijalizirati drugim nizom koji nije konstanta (potpuno neovisno o tome jesu li elementi niza `const`-kvalificirani ili ne). Dodatno, ova pravila omogućuju i (neobičnu) inicijalizaciju niza `int` elemenata konstantnim nizom znakova (što nije dozvoljeno u *C*-u), kao što je prikazano u ispisu 4.24.

```
1 int a[10] = "abc"; // isto kao redak 2
2 int b[10] = { 'a', 'b', 'c', '\0' };
3 int c[10] = { 97, 98, 99, 0 };
4 int d[10] = a; // greska
```

Ispis 4.24: Tri istovjetna načina inicijalizacije niza brojeva i greška.

Drugi slučaj u pravilima računanja ove produkcije odnosi se na sve ostale tipove vrijednosti (uključujući i nizove znakove koji nisu konstante, tj. nisu označeni uniformnim znakom `NIZ_ZNAKOVA`). Samim tim, svojstva *br-elem* i *tipovi* tada neće

postojati pa će po pravilima za `<init_deklarator>` doći do semantičke greške u točki 3 druge produkcije.

- `<inicijalizator> ::= L_VIT_ZAGRADA <lista_izraza_pridruzivanja> D_VIT_ZAGRADA`
  br-elem ← `<lista_izraza_pridruzivanja>`.*br-elem*
  tipovi ← `<lista_izraza_pridruzivanja>`.*tipovi*

  1. *provjeri*(`<lista_izraza_pridruzivanja>`)

`<lista_izraza_pridruzivanja>`

Nezavršni znak `<lista_izraza_pridruzivanja>` generira jedan ili više izraza odvojenih zarezima. U produkcijama se računaju izvedena svojstva *br-elem* i *tipovi*, s istim značenjem kao i do sada.

- `<lista_izraza_pridruzivanja> ::= <izraz_pridruzivanja>`
  tipovi ← `[ <izraz_pridruzivanja>`.*tip* `]`
  br-elem ← `1`

  1. *provjeri*(`<izraz_pridruzivanja>`)

  `<lista_izraza_pridruzivanja> ::= <lista_izraza_pridruzivanja> ZAREZ`
  `<izraz_pridruzivanja>`
  tipovi ← `<lista_izraza_pridruzivanja>`.*tipovi* `+ [ <izraz_pridruzivanja>`.*tip* `]`
  br-elem ← `<lista_izraza_pridruzivanja>`.*br-elem*`+ 1`

  1. *provjeri*(`<lista_izraza_pridruzivanja>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

4.4.7 Provjere nakon obilaska stabla

Konačno, nakon obilaska stabla i provjere svih navedenih semantičkih pravila, semantički analizator treba provjeriti još dva pravila[16]

1. u programu postoji funkcija imena `main` i tipa *funkcija*(`void` → `int`)

   Ako ovo pravilo nije zadovoljeno, semantički analizator treba na standardni izlaz ispisati samo `main` u vlastiti redak i završiti s radom.

2. svaka funkcija koja je deklarirana bilo gdje u programu (u bilo kojem djelokrugu) mora biti definirana

   Ako ovo pravilo nije zadovoljeno, semantički analizator treba na standardni izlaz ispisati samo `funkcija` u vlastiti redak i završiti s radom.

[16] Ova pravila tipično ne bi bila u domeni semantičke analize, ali kako jezik *ppjC* ne podržava više od jedne datoteke s izvornim programom i samim time ne postoji potreba za povezivačem (engl. *linker*), semantički analizator treba provjeriti i ova pravila.

4.5 Savjeti za implementaciju

U ovom poglavlju navedene su neke napomene koje mogu pomoći u ostvarenju ove laboratorijske vježbe. Poglavlje će biti prošireno ako se za tim pokaže potreba u skladu sa čestim pitanjima.

4.5.1 Djelokrug deklaracija i tablica znakova

U semantičkoj analizi jezika *ppjC*, naglasak je na provjeri tipova podataka i djelokruga deklaracija. Obje provjere se ostvaruju korištenjem tablice znakova. Na primjer, pri deklaraciji varijable, tip varijable se zapisuje u tablicu znakova, a kada se varijabla koristi u istom ili ugniježđenom djelokrugu, tip varijable se određuje na osnovi zapisa u tablici znakova.

Kako je djelokrug deklaracija u jeziku *ppjC* određen hijerarhijom blokova tj. složenim naredbama (što je vrlo uobičajeno), prirodno je tablicu znakova također organizirati hijerarhijski, kao raspodijeljenu strukturu nad generativnim stablom. Pojedini dijelovi takve tablice znakova također čine stablastu strukturu koja u potpunosti odgovara blokovskoj strukturi programa koja je prikazana u generativnom stablu.

Svakom djelokrugu deklaracija u programu, dakle globalnom djelokrugu i svim blokovima (što uključuje i tijela funkcija) pridružen je jedan čvor stablaste tablice znakova. Svaki čvor logički ima dva dijela: tablicu loklanih imena i kazaljku na čvor koji predstavlja ugniježđujući blok (ako takav postoji). Korijen stablaste tablice znakova očito predstavlja globalni djelokrug deklaracija i jedini je čvor koji nema kazaljku na ugniježđujući blok.

Koristeći takvu strukturu tablice znakova, provjera mnogih semantičkih pravila jezika *ppjC* može se pojednostaviti u odnosu na korištenje jednorazinske tablice znakova. Na primjer, u jednostavnoj tablici znakova osim tipa varijable (i eventualno još nekih značajki) nužno je na neki način zapisati u kojem djelokrugu je varijabla deklarirana. Nadalje, ako u dva ili više djelokruga postoji varijabla istog imena (što je semantički ispravno), u takvoj jednostavnoj tablici znakova trebalo bi biti više zapisa za isto ime.

S druge strane, semantička pravila jezika garantiraju da su imena u svakom čvoru stablaste tablice znakova jedinstvena. Pretraživanje tablice svodi se na traženje imena u tablici lokalnih imena, a u slučaju da se ime tamo ne nalazi, pretražuje se čvor roditelj i tako sve do globalnog djelokruga.

Konačno, čvor stablaste tablice znakova može se pri provjeri semantičkih pravila tretirati kao nasljedno svojstvo većine znakova gramatike. Prije obrade bloka (tj. složene naredbe), gradi se novi čvor koji se koristi pri obradi tog bloka naredbi. Kazaljka na ugniježđujući blok novog čvora usmjeri se na nasljedno svojstvo složene naredbe.