Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

# Prevođenje programskih jezika

*Ak. god. 2021./2022.*

## 4. laboratorijska vježba

Tema četvrte laboratorijske vježbe je generiranje koda. Vaš zadatak je programski ostvariti generator mnemoničkog koda procesora FRISC za jezik *PJ*.
**Važna napomena:** Kako jezik *PJ* nema mogućnost učitavanja podataka tijekom izvođenja, moguće je rezultat izvođenja cijelog programa izračunati tijekom prevođenja i generirati program koji izravno "vraća" (što to znači je definirano kasnije u uputi) tako izračunati rezultat. Takav pristup u rješavanju ove vježbe je **strogo zabranjen**. Drugim riječima, nemojte provoditi nikakve optimizacije broja aritmetičkih operacija koje se trebaju izvesti u programu.

## Ulaz i izlaz programa generatora koda

Ulaz u generator koda (stdin) bit će generativno stablo *PJ* programa, tj. isti kao ulaz u semantički analizator u trećoj laboratorijskoj vježbi. To znači da ćete prije/tijekom generiranja FRISC koda morati napraviti semantičku analizu ulaznog programa tj. odrediti na koju varijablu se odnosi pojedini identifikator u naredbama programa. Preporuča se da rad na ovoj vježbi započnete od rješenja treće laboratorijske vježbe tj. da rješenje treće laboratorijske vježbe dopunite generiranjem koda. Za potrebe ove vježbe, pretpostavite da generativno stablo na ulazu prikazuje semantički ispravan program tj. u ispitnim primjerima neće biti semantičkih pogrešaka.

Generator treba **u datoteku `a.frisc` u istom direktoriju kao i generator** zapisati FRISC kod za zadani program. Sustav SPRUT će tada pokrenuti generirani FRISC kod i usporediti njegov izlaz (definirano kasnije) s očekivanim izlazom. Drugim riječima, provjera točnosti generatora je neizravna što omogućuje slobodu u samom generiranju koda.

## Simulator procesora FRISC

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`**, njegovo izvođenje može se simulirati naredbom **`node main.js a.frisc`**. SPRUT koristi ovaj isti simulator za evaluaciju rješenja.

Na procesor je u simulatoru spojena memorija od **256KB** na adresama **0-3FFFF**.

Alternativno, za ispitivanje 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, molimo vas da nam tu razliku dojavite na e-mail listu predmeta.

---

[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.

## Izlaz generiranog FRISC programa

Za potrebe ove vježbe, svaki *PJ* program imat će globalnu varijablu imena `rez`. Generirani FRISC program treba izračunatu vrijednost te varijable zapisati u registar **`R6`** prije izvođenja instrukcije **`HALT`**.

Sve konstante i rezultati svih aritmetičkih operacija u svim primjerima prikazani u dvojnom komplementu stat će u 32 bita.

## Predaja rješenja na SPRUT

Rješenje se predaje kao zip arhiva u kojoj se nalazi sav potreban kod za prevođenje/izvođenje. Vrijede ista pravila kao na UTR-u (specifično, u **Javi nemojte koristiti pakete**). Ulazna točka za rješenja u Javi je razred FRISCGenerator, a za Python datoteka FRISCGenerator.py. Za ostale jezike je organizacija koda proizvoljna (naravno, mora postojati točno jedna funkcija/metoda main itd.). Vremensko ograničenje za izvođenje generatora je jedna minuta (što je daleko više od očekivanog vremena izvođenja). Vrijeme simulacije generiranog FRISC programa također je ograničeno na jednu minutu, ali očekivano vrijeme za ispitne program je do nekoliko sekundi.

## Savjeti za implementaciju

Tekst u ovom poglavlju je informativne prirode i nije obvezujuć tj. ostvarenje generatora je u velikoj mjeri proizvoljno, uz ogradu vezanu za “optimiranje” (tj. izračunavanje tijekom prevođenja) navedenu na početku upute i uobičajenu ogradu da nije dozvoljeno koristiti gotove alate/biblioteke koje odrađuju dijelove zadatka koji su tema predmeta PPJ.

U poglavlju su također prikazani i neki primjeri *PJ* programa i mogućih FRISC ekvivalenata. Pri čitanju primjera, važno je na umu imati dvije stvari. Prvo, u primjerima se pojavljuju *PJ* programi, a ulaz u rješenje ove laboratorijske vježbe će biti isključivo generativno stablo (kao u 3. vježbi). Drugo, prikazani FRISC programi su ilustrativne prirode i nije ni u kom slučaju nužno da vaš generator generira točno takav kod.

Preporuča se da ovo poglavlje jednom pročitate i onda pojedine dijelove po potrebi detaljnije proučite u razmišljanju o svom generatoru.

### Pohrana vrijednosti varijabli

Jedna od ključnih odluka u rješavanju ove vježbe je gdje će se i kako pohranjivati vrijednosti varijabli. Načelno, vrijednosti varijabli mogu se pohranjivati u memoriju na fiksne memorijske lokacije, na stog ili u registre (za različite varijable se, očito, mogu koristiti različiti pristupi). Zbog jednostavnosti jezika *PJ*, u ovoj vježbi se preporuča pohranjivanje vrijednosti varijabli na fiksne memorijske lokacije. Drugim riječima, svakoj varijabli u programu se tijekom generiranja koda može dodijeliti neka adresa u memoriji i ta adresa se onda koristi za dohvaćanje i spremanje vrijednosti varijable.

Kod ovog pristupa, važno je uočiti da ime varijable ne određuje jedinstveno adresu u memoriji jer u programu u prisustvu `za`-petlji može biti više različitih varijabli istog imena (tada će adresa za određeno ime ovisiti o kontekstu u kojem se ime koristi tj. koristi li se ime unutar neke petlje, u globalnom djelokrugu itd.)

Alternativa fiksnim adresama je pohrana vrijednosti varijabli na stogu. U tom slučaju bi za definiciju varijable trebalo rezervirati prostor na stogu i u generatoru zapamtiti odmak od fiksne adrese vrha stoga pri ulasku u određeni blok. Ovaj pristup je nesto složeniji od pohranjivanja vrijednosti na fiksne memorijske adrese i ovdje se neće dalje razmatrati.

Dodjeljivanje registara varijablama se, usprkos svojoj velikoj važnosti u praksi, **ne preporuča** za ovu vježbu zbog složenosti postupaka koje treba provoditi.

Kao što je opisano ranije u uputi, rezultat programa je vrijednost spremljena u globalnoj varijabli `rez` na kraju izvođenja programa. Ta vrijednost treba se nalaziti u registru **`R6`** prije izvođenja instrukcije **`HALT`**. Jedno od mogućih ostvarenja ovog cilja je vrijednost varijable `rez` cijelo vrijeme držati u registru **`R6`** (a ne na fiksnoj memorijskoj lokaciji ili na stogu kao što je ranije opisano). Međutim, imajte na umu da je moguće bilo koju globalnu varijablu (pa tako i varijablu `rez`) sakriti brojačem petlje. Iz tog razloga se preporuča koristiti varijablu `rez` kao i sve ostale varijable i onda na kraju učitati njenu vrijednost u registar **`R6`**.

### Izračunavanje vrijednosti izraza

Za izračunavanje vrijednosti izraza preporuča se korištenje stoga u skladu s jednostavnim algoritmom koji je kroz primjere opisan u nastavku[2]. Zbog veličine memorije priključene na procesor u simulatoru i načina rada instrukcija PUSH i POP kod procesora FRISC, preporučljivo je pokazivač stoga (registar R7) na početku programa inicijalizirati na adresu $\mathbf{40000_{16} = 262144_{10}}$ (to je prva memorijska adresa koja ne postoji).

Osnovna ideja jednostavnog algoritma za izračunavanje vrijednosti izraza korištenjem stoga je stavljati operande i sve međurezultate na stog. U obradi izraza susrećemo se s dvije osnovne situacije - obradom operanada i obradom aritmetičkih operatora. Operandi se stavljaju na stog. Pri obradi aritmetičkog operatora, operandi će se već nalaziti na stogu (na primjer desni operand na vrh stoga, lijevo operand odmah ispod desnog). To znači da ćemo za izračunavanje vrijednosti aritmetičke operacije skinuti operande sa stoga (pohraniti ih u registre), izračunati rezultat operacije odgovarajućom instrukcijom i rezultat vratiti na stog.

Na primjer, jedan od najjednostavnijih izraza je sama konstanta 0, na primjer na desnoj strani u nekoj naredbi pridruživanja. Za obradu izraza 0, generirale bi se instrukcije

```
MOVE %D 0, R0
PUSH R0
```

Za nešto složeniji izraz `1 + 2`, generirale bi se instrukcije

```
MOVE %D 1, R0
PUSH R0

MOVE %D 2, R0
PUSH R0

POP R1
POP R0
ADD R0, R1, R2
PUSH R2
```

Ključno je uočiti da je na kraju ovog odsječka programa opet osigurana istinitost invarijante da se na vrhu stoga nalazi izračunata vrijednost izraza što omogućuje izračunavanje proizvoljno složenih izraza konzistentnom primjenom ovog pristupa.

---

[2] Nedostatak opisanog algoritma je što se generira znatno veći broj instrukcija nego je to nužno, ali to nije bitan nedostatak za potrebe ove vježbe.

Za ilustraciju rada s varijablama, pogledajmo sljedeći primjer *PJ* programa.

```
x = 3
rez = x - 2
```

Cijeli FRISC program koji bi se mogao generirati za navedeni primjer dan je u nastavku (komentari iznad skupa instrukcija prikazuju dio izvornog *PJ* programa na osnovu kojeg je skup instrukcija generiran):

```
    MOVE 40000, R7 ; init stog

    ; 3
    MOVE %D 3, R0
    PUSH R0

    ; x = 3
    POP R0
    STORE R0, (V0)

    ; x
    LOAD R0, (V0)
    PUSH R0

    ; 2
    MOVE %D 2, R0
    PUSH R0

    ; x - 2
    POP R1
    POP R0
    SUB R0, R1, R2
    PUSH R2

    ; rez = x - 2
    POP R0
    STORE R0, (V1)

    ; "vrati" rez
    LOAD R6, (V1)
    HALT

    ; varijable
V0  DW 0 ; x
V1  DW 0 ; rez
```

### Množenje i dijeljenje

Kako FRISC nema instrukcije za množenje i cjelobrojno dijeljenje, ove operacije treba ostvariti potprogramima. Preporuča se da operacije ostvarite uzastopnim zbrajanjem/oduzimanjem jer će brojevi u primjerima biti dovoljno mali da se ti algoritmi množenja i dijeljenja izvrše u kratkom vremenu.

Primjer potprograma koji ostvaruju ove operacije dan je u nastavku (u svom rješenju labosa možete koristiti

dane potprograme ili napisati svoje). Potprogrami bez čuvanja konteksta koriste sve registre osim **R5**. Za množenje dva broja potrebno je pozvati potprogram **MUL**, a za dijeljenje potprogram **DIV**. U oba slučaja, potprogram će skinuti operande sa stoga (drugi operand je na vrhu stoga, a prvi operand je ispod njega) i izračunati rezultat staviti na vrh stoga. Ostali potprogrami su pomoćni i ne bi ih se trebalo izravno pozivati.

```
        ; ako je R0*R1 ili R0/R1 negativno ili 0
        ; R6 <- 1 (inace 0)
        ; nakon potprograma R0>=0 i R1>=0
MD_SGN  MOVE 0, R6
        XOR R0, 0, R0
        JP_P MD_TST1
        XOR R0, -1, R0
        ADD R0, 1, R0
        MOVE 1, R6
MD_TST1 XOR R1, 0, R1
        JP_P MD_SGNR
        XOR R1, -1, R1
        ADD R1, 1, R1
        XOR R6, 1, R6
MD_SGNR RET

MD_INIT POP R4 ; MD_INIT ret addr
        POP R3 ; M/D ret addr
        POP R1 ; op2
        POP R0 ; op1
        CALL MD_SGN
        MOVE 0, R2 ; init rezultata
        PUSH R4 ; MD_INIT ret addr
        RET

MD_RET  XOR R6, 0, R6 ; predznak?
        JP_Z MD_RET1
        XOR R2, -1, R2 ; promijeni predznak
        ADD R2, 1, R2
MD_RET1 POP R4 ; MD_RET ret addr
        PUSH R2 ; rezultat
        PUSH R3 ; M/D ret addr
        PUSH R4 ; MD_RET ret addr
        RET

MUL     CALL MD_INIT
        XOR R1, 0, R1
        JP_Z MUL_RET ; op2 == 0
        SUB R1, 1, R1
MUL_1   ADD R2, R0, R2
        SUB R1, 1, R1
        JP_NN MUL_1 ; >= 0?
MUL_RET CALL MD_RET
        RET

DIV     CALL MD_INIT
```

```
        XOR R1, 0, R1
        JP_Z DIV_RET ; op2 == 0
DIV_1   ADD R2, 1, R2
        SUB R0, R1, R0
        JP_NN DIV_1
        SUB R2, 1, R2
DIV_RET CALL MD_RET
        RET
```

Preporuča se da dani FRISC kod uključite u svoj generirani kod nakon koda generiranog za ulazni *PJ* program (FRISC programi se izvode od adrese `0` i na toj adresi treba biti prva instrukcija "glavnog programa" što će vjerojatno biti inicijalizacija registra `R7`).

Konačno, dani potprogram za dijeljenje daje rezultat 0 za dijeljenje s nulom. Dozvoljeno je da implementacija operacije dijeljenja za dijeljenje s nulom napravi bilo što, uključujući beskonačnu petlju, vraćanje pogrešnog rezultata i slično.

### Petlje

Osnovna ideja za generiranje koda za petlje je prije petlje izračunati `od`-izraz i dodijeliti njegovu vrijednost brojaču petlje. Tijelo petlje počinje instrukcijom označenom jedinstvenom labelom. Nakon svih naredbi u tijelu petlje, generiraju se instrukcije za inkrement brojača i za izračun `do`-izraza iz zaglavlja petlje. Povećani brojač se uspoređuje s izračunatim `do`-izrazom i petlja se uvjetno ponavlja. Ova ideja prikazana je na sljedećem primjeru:

```
x = 3
y = 5
rez = 0
za x od 1 do y
  rez = rez + x
az
rez = rez + x
```

Generirani FRISC kod mogao bi izgledati ovako:

```
    MOVE 40000, R7 ; init stog

    ; 3
    MOVE %D 3, R0
    PUSH R0
    ; x = 3
    POP R0
    STORE R0, (V0)

    ; 5
    MOVE %D 5, R0
    PUSH R0
    ; y = 5
    POP R0
    STORE R0, (V1)

    ; 0
    MOVE %D 0, R0
    PUSH R0
```

```
    ; rez = 0
    POP R0
    STORE R0, (V2)

    ; 1
    MOVE %D 1, R0
    PUSH R0
    ; x od 1
    POP R0
    STORE R0, (V3)

    ; rez
L0  LOAD R0, (V2)
    PUSH R0
    ; x
    LOAD R0, (V3)
    PUSH R0
    ; rez + x
    POP R1
    POP R0
    ADD R0, R1, R2
    PUSH R2
    ; rez = rez + x
    POP R0
    STORE R0, (V2)
    ; inkrement x
    LOAD R0, (V3)
    ADD R0, 1, R0
    STORE R0, (V3)
    ; y
    LOAD R0, (V1)
    PUSH R0
    ; x do y
    LOAD R0, (V3)
    POP R1
    CMP R0, R1
    JP_SLE L0

    ; rez
    LOAD R0, (V2)
    PUSH R0; rez
    ; x
    LOAD R0, (V0)
    PUSH R0
    ; rez + x
    POP R1
    POP R0
    ADD R0, R1, R2
    PUSH R2
    ; rez = rez + x
    POP R0
```

```
    STORE R0, (V2)

    ; "vrati" rez
    LOAD R6, (V2)
    HALT

    ; varijable
V0  DW 0 ; x
V1  DW 0 ; y
V2  DW 0 ; rez
V3  DW 0 ; x L0
```

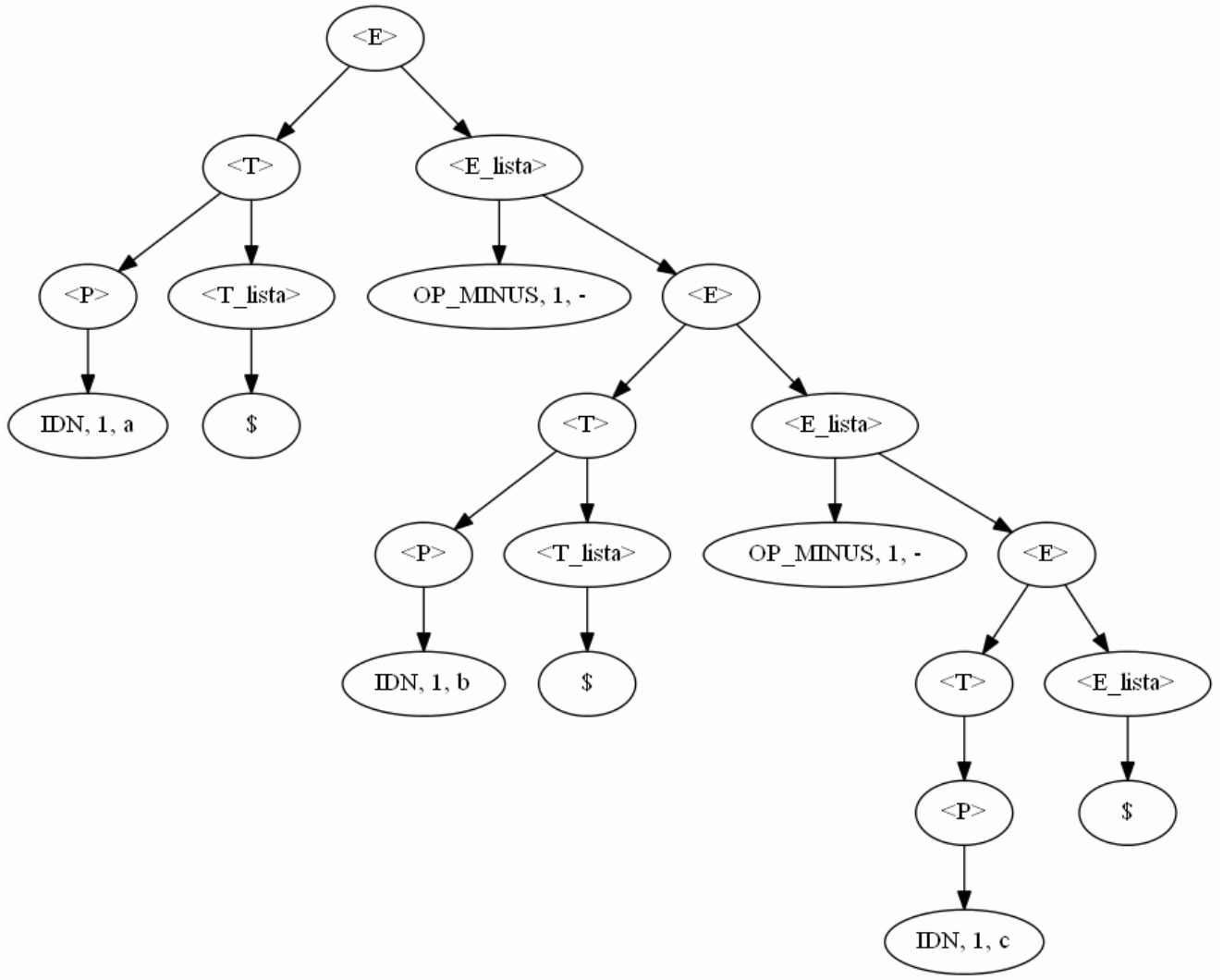
### Asocijativnost aritmetičkih operatora (opcionalno/složeno)

Kao što je spomenuto u uputi za drugu laboratorijsku vježbu, LL(1)-gramatika pogrešno prikazuje asocijativnost aritmetičkih operatora. Preciznije, svi aritmetički operatori koje koristimo u jeziku *PJ* su lijevo asocijativni, a u gramatici su prikazani kao desno asocijativni. Ova se razlika očituje u generativnom stablu koje dobivate na ulazu. U ispitnim primjerima će maksimalno 10% primjera ovisiti o asocijativnosti aritmetičkih operatora tako da ostatak ovog poglavlja možete smatrati opcionalnim. U nastavku je ukratko skiciran jedan način rješavanja problema asocijativnosti.

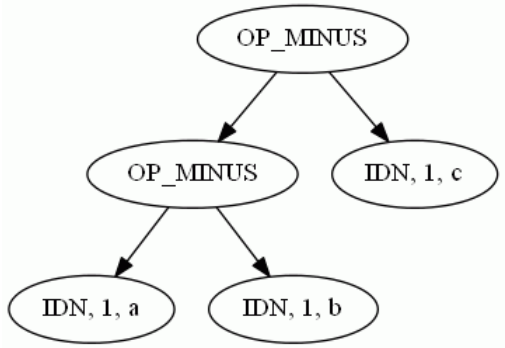
Konceptualno, ideja je iz podstabla s korijenom označenim znakom `<E>` generirati sintaksno stablo za aritmetičke izraze s ispravnom asocijativnošću. Sintaksno stablo će u unutrašnjim čvorovima imati operatore, a u listovima operande. Generiranje koda za takvo sintaksno stablo je onda vrlo jednostavno - prvo se rekurzivno generira kod za lijevo podstablo, onda za desno podstablo i konačno kod za operaciju u korijenu stabla (dakle instrukcije koje skidaju operande sa stoga i odgovarajuća ALU instrukcija, kao što je prikazano u primjerima).

Postupak izgradnje sintaksnog stabla tipično je dio sintaksne analize, tj. parser odmah generira sitaksno stablo za izraze (umjesto generativnog). U nastavku poglavlja je opisan postupak pretvorbe generativnog stabla za izraze u sintaksno stablo.

Na primjer, za izraz `a - b - c` u prvom retku nekog *PJ* programa dobili bismo sljedeće generativno stablo:




Sintaksno stablo koje želimo dobiti izgledalo bi ovako:




Definirat ćemo funkcije koje obrađuju čvorove generativnog stabla označene znakovima `<E>`, `<E_lista>`, `<T>`, `<T_lista>` i `<P>` i vraćaju sintaksno stablo. Sintaksno stablo cijelog izraza dobiva se onda obradom korijenskog `<E>`-čvora generativnog stabla cijelog izraza.

Funkcija za znak `<P>` jednostavno vraća list stabla označen identifikatorom ili konstantom ili cijelo stablo dobiveno obradom generativnog stabla za znak `<E>` u oblim zagradama (ovisno o produkciji znaka <P> koja je primijenjena).

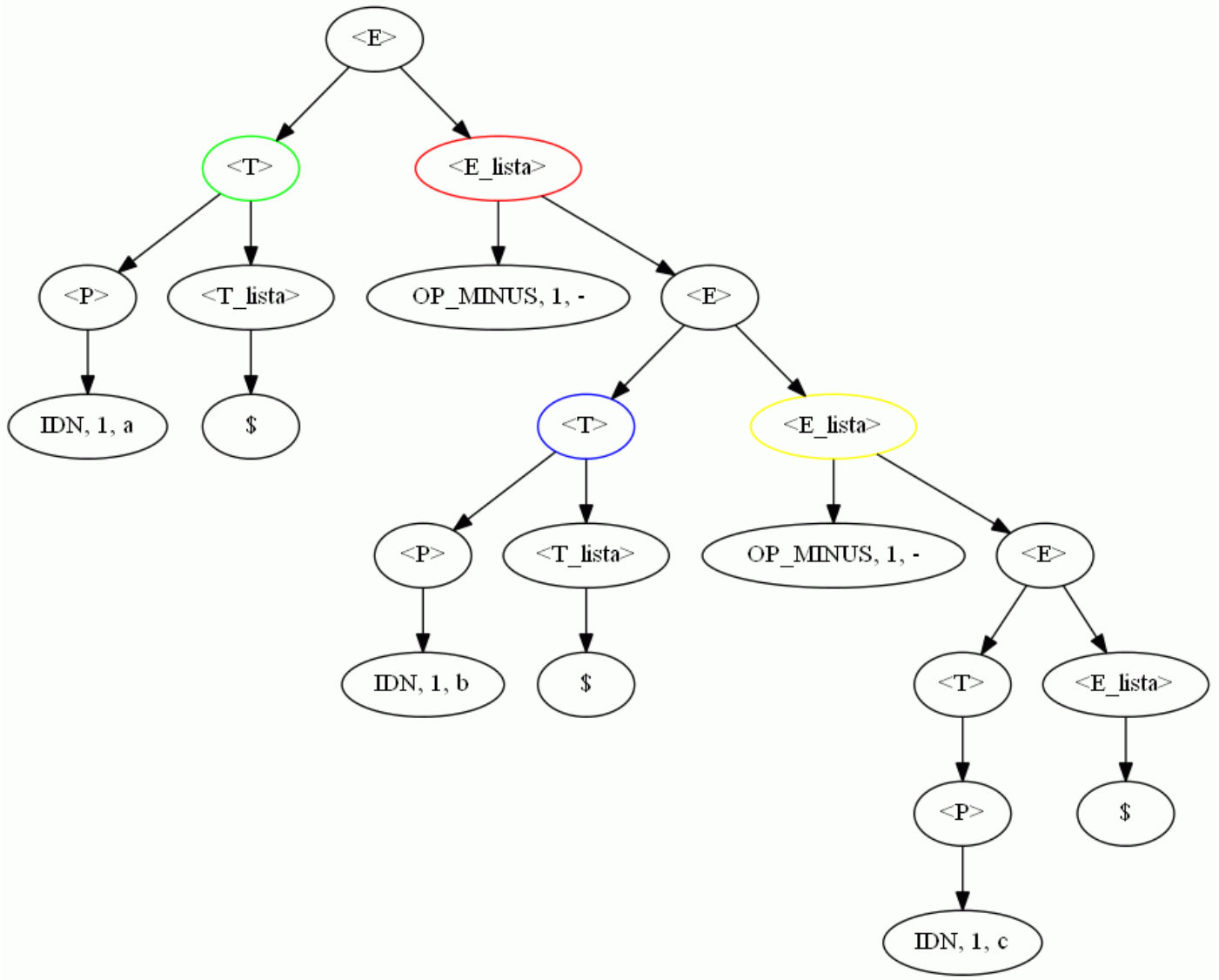
Funkcije za znakove `<T>` i `<T_lista>` analogne su funkcijama za znakove `<E>` i `<E_lista>` i neće biti posebno opisivane.

Za znak `<E>` moguća je samo jedna produkcija (`<E> ::= <E> <E_lista>`) pa je njegova obrada vrlo

jednostavna - rekurizvno se obrađuje lijevo podstablo (s korijenom <E>) i rezultat se predaje kao argument funkciji za obradu desnog podstabla (s korijenom <E_lista>).

Za znak <E-lista>, moguća su dva oblika produkcija - ili ε-produkcija ili produkcija oblika <E_lista> ::= OPERATOR <E> (pri čemu je OPERATOR u ovom slučaju OP_MINUS ili OP_PLUS) U slučaju ε-produkcije, funkcija jednostavno vraća svoj parametar kao rezultantno sintaksno stablo.

Za produkciju oblika <E_lista> ::= OPERATOR <E>, vrijednost parametra predstavlja lijevi operand za zadani operator. Desni operand nalazi se u lijevom podstablu znaka <E>.




Na primjer, pri obradi znaka <E_lista> označenog crvenom bojom na gornjoj slici, lijevi operand pripadnog operatora oduzimanja nalazi se u podstablu s korijenom u znaku <T> zelene boje. Sintaksno stablo za lijevi operand već je prethodno izgrađeno (sastoji se samo od lista označenog identifikatorom a) i predano kao argument funkciji koja obrađuje znak <E_lista>.

S druge strane, desni operand nalazi se u podstablu čiji korijen je označen znakom <T> plave boje. Dakle, kako bi se izgradilo sintaksno stablo za operaciju oduzimanja, funkcija obrađuje lijevo podstablo svog desnog podstabla (desno podstablo označeno je znakom <E>, a njegovo lijevo podstablo znakom <T> plave boje). Lijevi i desni operand (tj. njihova sintaksna stabla) povezuju se kao djeca čvora označenog znakom OPERATOR (koji god operator to bio). Tako dobiveno sintaksno stablo predaje se kao argument u funkciju za znak <E-lista> u desnom podstablu znaka <E> (jedini preostali dio stabla koji još nije

obrađen - čvor je označen žutom bojom u primjeru na slici).

Konačno, opisani postupak moguće je prilagoditi tako da umjesto gradnje sintaksnog stabla izravno generira kod za izraz sa ispravnom asocijativnošću.