Prevodjenje programskih jezika – beleške sa predavanja
Uvod

Milan Banković

*Matematički fakultet,
Univerzitet u Beogradu

Jesenji semestar 2024/25.

Pregled

Motivacija

Čime se bavimo?

- Do sada ste naučili da programirate na programskim jezicima visokog nivoa (C, Java, C++)
- Naučili ste i kako računar radi na niskom nivou (UOAR2), i na koji način se sa njim može komunicirati (mašinski jezik, asembler)
- Svaki program na visokom nivou se mora prevesti na mašinski jezik koji procesor razume, kako bi mogao da se izvrši
- Ovaj deo je ostao nerazjašnjen: kako da npr. C program prevedemo na ekvivalentan asemblerski program?
- Time se bavimo na ovom predmetu

Sadržaj predmeta

Teme kojima se bavimo

- Uvod u teoriju formalnih jezika
- Regularni izrazi
- Formalne gramatike
- Konačni automati
- Leksička analiza
- Potisni automati
- Sintaksna analiza (naniže i naviše)
- Elementi semantičke analize

Teme kojima se ne bavimo

- Optimizacija koda
- Generisanje koda
- O tome ćete učiti na predmetu Konstrukcija kompilatora

Obaveze studenata

Nastava

- 2 časa predavanja
- 3 časa vežbi
- 6 ESPB

Predispitne obaveze

- Aktivno prisustvo na časovima (5 + 5 poena)
- Testovi na predavanjima (10 poena)

Ispit

- Praktični ispit (55 poena, prag 22 poena)
  - važi do kraja školske godine
- Teorijski ispit (35 poena, prag 14 poena)
  - uslov za izlazak je ostvaren prag na praktičnom

Pregled

O jezicima uopšte

Osnovne karakteristike jezika

- Sintaksa jezika
  - Skup pravila koja definišu ispravne jezičke konstrukcije
- Semantika jezika
  - Pravila koja definišu značenje ispravnih jezičkih konstrukcija

Kakvi jezici mogu biti?

- Prirodni jezici (srpski, engleski, kineski,...)
  - Sintaksa nedovoljno precizno definisana
  - Semantika nije jednoznačna (postoje dvosmislene i besmislene rečenice)
- Formalni jezici (programski jezici, jezici za obeležavanje, jezici matematčke logike, ...)
  - Sintaksa precizno definisana odgovarajućim formalizmom (gramatika, Bekus-Naurova notacija,...)
  - Semantika jednoznačna (svaka ispravna konstrukcija ima jedinstveno značenje)

Opšta struktura prevodioca

Delovi prevodioca

- Prednji deo: obavlja etapu analize
  - Ulaz prednjeg dela je program na višem programskom jeziku (izvorni jezik)
  - Izlaz prednjeg dela je stablo apstraktne sintakse sa pridruženim semantiškim informacijama
- Zadnji deo: obavlja etapu sinteze
  - Ulaz zadnjeg dela je izlaz prednjeg dela
  - Izlaz zadnjeg dela je program na jeziku niskog nivoa (objektni jezik; tipično asemblerski jezik)

Etapa analize

Iz čega se sastoji etapa analize?

- Leksička analiza
  - Razlaže izvorni kod programa na lekseme („reči" jezika)
  - Primeri leksema su identifikatori, celobrojne konstante, ključne reči, separatori, operatori, i td.
  - Svaka leksema se klasifikuje po svojoj vrsti i zamenjuje odgovarajućim tokenom koji predstavlja tu klasu (identifikator, celobrojna konstanta, itd.)
  - Dobijeni niz tokena se prosledjuje sintaksnom analizatoru
- Sintaksna analiza
  - Dobija na ulazu niz tokena i u njemu prepoznaje ispravne jezičke konstrukcije („rečenice" jezika)
  - Proverava da li je raspored tokena na ulazu u skladu sa sintaksnim pravilima jezika
  - Pravila jezika najčešće su opisana u nekom formalnom sistemu (gramatike, Bekus-Naurova notacija)
  - Na osnovu pravila jezika generiše se stablo izvođenja, kao i apstraktno sintaksno stablo
- Semantička analiza
  - Razmatraju se dodatna jezička pravila koja nije moguće opisati gramatikom (tipovi, dometi, prava pristupa, i sl.)
  - Odgovarajuće semantičke informacije se pridružuju čvorovima sintaksnog stabla i ono se po potrebi modifikuje

Primer – leksička analiza

Primer

Ako na ulazu imamo:

```
Povrsina = (OsnovicaA + OsnovicaB) * Visina / 2.0;
```

tada će leksički analizator redom prepoznati sledeći niz leksema:

`Povrsina`, `=`, `(`, `OsnovicaA`, `+`, `OsnovicaB`, `)`, `*`, `Visina`, `/`, `2.0`, `;`

Primetimo da se razmaci ignorišu. Ovaj niz leksema se konvertuje u sledeći niz tokena:

`<id>`, `<op_dodela>`, `<lz>`, `<id>`, `<op_sabiranje>`, `<id>`, `<dz>`, `<op_mnozenje>`, `<id>`, `<op_deljenje>`, `<realna_konstanta>`, `<tz>`

Primer – leksička analiza

Primer

Ako na ulazu imamo:
`01pera01`
tada će leksički analizator redom prepoznati sledeći niz leksema:
`01`, `pera01`
kom odgovara niz tokena:
`<oktalna_konstanta>`, `<id>`

Primer

Ako na ulazu imamo:
`x+++y`
tada će leksički analizator prepoznati sledeći niz leksema:
`x`, `++`, `+`, `y`
(tzv. gramzivi algoritam, uzima najdužu moguću leksemu koju prepoznaje). Sa druge strane, ako imamo:
`x + ++y`
tada imamo niz leksema:
`x`, `+`, `++`, `y`

Primer – sintaksna analiza

Primer

Vratimo se na primer:

```
Povrsina = (OsnovicaA + OsnovicaB) * Visina / 2.0;
```

i dobijeni niz tokena:

<id>, <op_dodela>, <lz>, <id>, <op_sabiranje>, <id>, <dz>, <op_mnozenje>, <id>, <op_deljenje>, <realna_konstanta>, <tz>

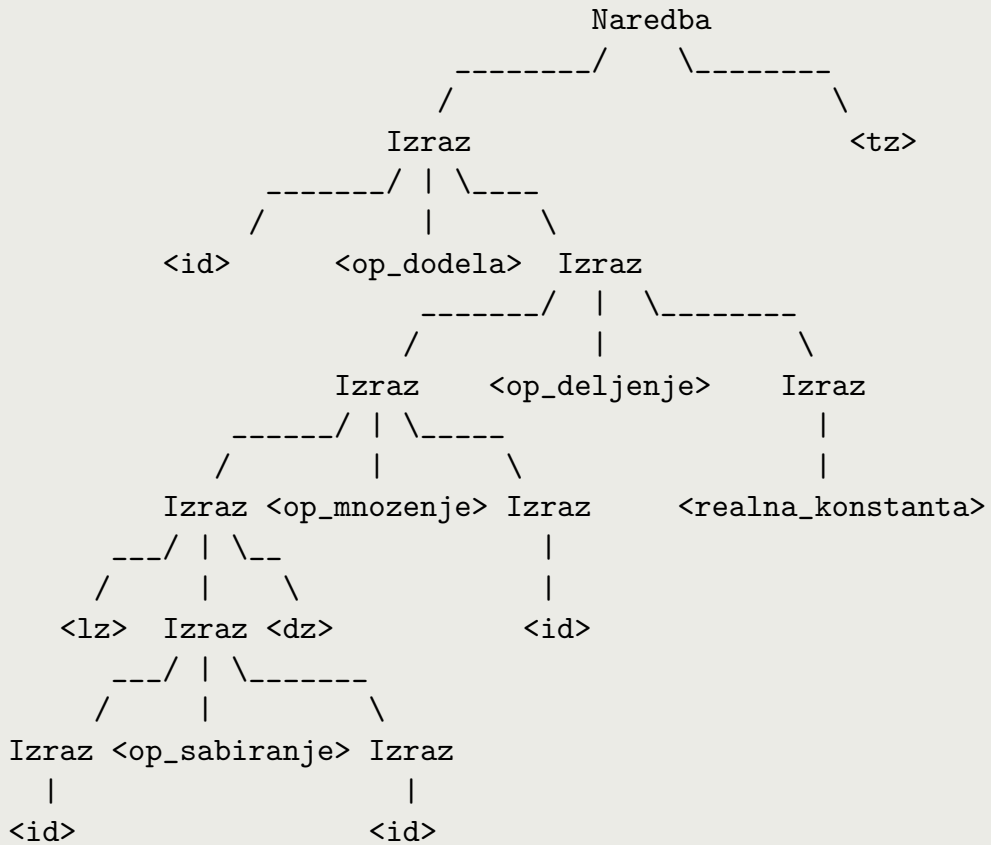
koji prosledjujemo sintaksnom analizatoru. Pretpostavimo da imamo sledeća sintaksna pravila jezika:

```
Naredba ::= Izraz <tz>
Izraz ::= Izraz <op_sabiranje> Izraz
Izraz ::= Izraz <op_oduzimanje> Izraz
Izraz ::= Izraz <op_mnozenje> Izraz
Izraz ::= Izraz <op_deljenje> Izraz
Izraz ::= <lz> Izraz <dz>
Izraz ::= <id> <op_dodela> Izraz
Izraz ::= <id>
Izraz ::= <celobrojna_konstanta>
Izraz ::= <realna_konstanta>
```

Primer – sintaksna analiza (nastavak)

Primer

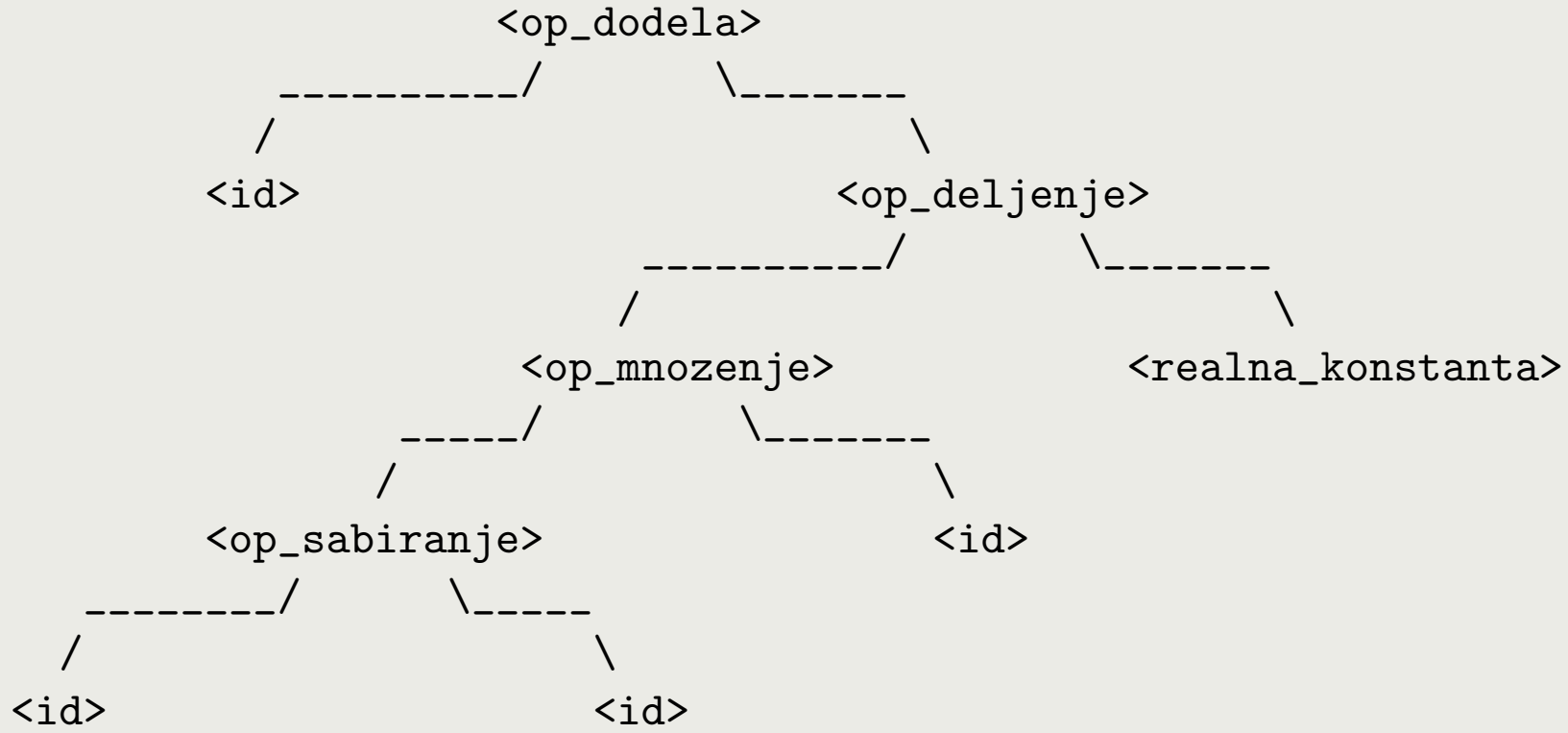
Pomoću ovih pravila, od datog niza tokena se može formirati sledeće stablo izvodjenja:



Primer – sintaksna analiza (nastavak)

Primer

Apstrahujući detalje konkretne sintakse, dobijamo sledeće stablo apstraktne sintakse:



Primer – sintaksna analiza

Primer

Naredba:

```
if(x > y) max = x; else max = y;
```

je sintaksno ispravna konstrukcija u jeziku C. Sa druge strane:

```
if)x > y) max = x; else max = y;
```

nije sintaksno ispravna konstrukcija, jer se, po pravilima jezika, nakon tokena `<if>` *mora nalaziti token* `<lz>` *(leva zagrada), a ne token* `<dz>` *(desna zagrada). Slično, deklaracija:*

```
int x[] = {1, 2, 3};
```

je sintaksno ispravna u C-u, dok deklaracija:

```
int [] x = {1, 2, 3};
```

nije sintaksno ispravna u jeziku C, jer se po pravilima jezika u deklaratoru specifikator dimenzije niza navodi iza promenljive (ili nekog drugog složenog deklaratora), a ne ispred.

Tabela simbola

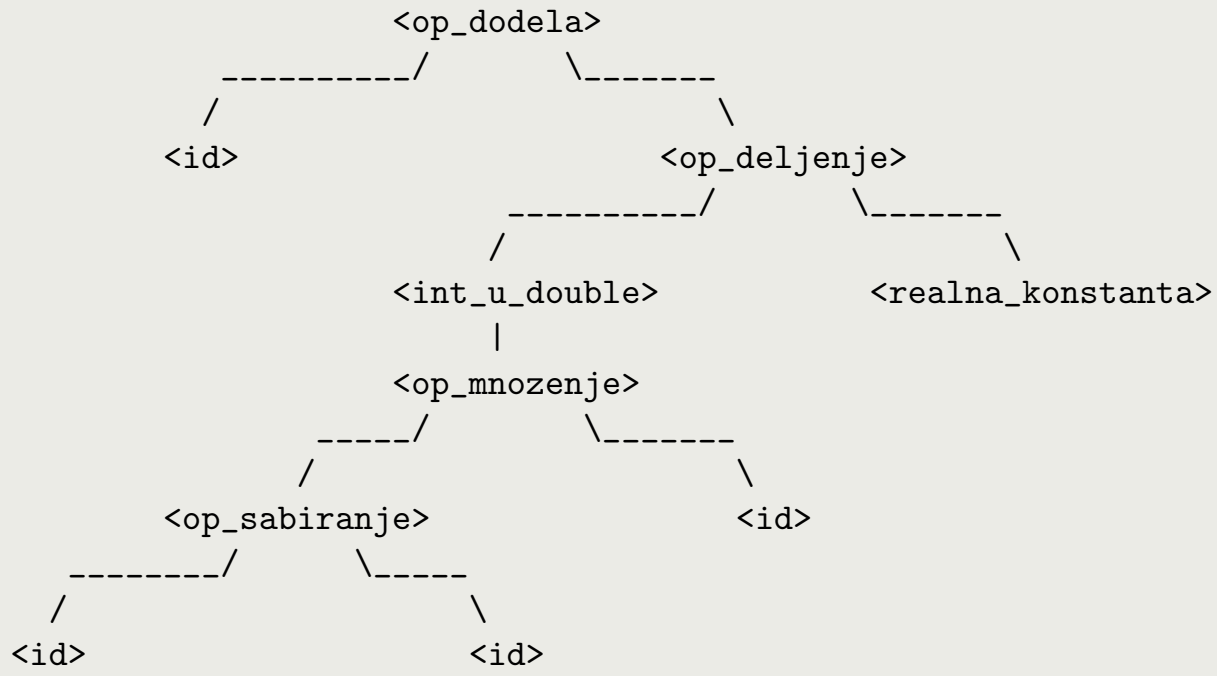
Tabela simbola

- U fazi leksičke analize apstrahovali smo konkrente lekseme i zamenili ih tokenima
- Ovo je zato što u fazi sintaksne analize nije bitno da li je npr. identifikator x ili y, već je samo bitno da li identifikator može da stoji na tom mestu
- U fazi semantičke analize (a i kasnije, u etapi sinteze) biće nam veoma bitno koji konkretan identifikator stoji na kom mestu
- Zbog toga se te apstrahovane informacije ne odbacuju, već se čuvaju uz svaki token i koriste se kasnije za pristup informacijama u tabeli simbola
- Tabela simbola identifikatorima pridružuje informacije koje se određuju u fazi semantičke analize
- Informacije koje se pridružuju identifikatorima u tabeli simbola su npr. njihov tip i domet koji se odredjuje njihovom deklaracijom
- Ove informacije se prosleđuju dalje zadnjem delu prevodioca (etapa sinteze)

Primer – semantička analiza (nastavak)

Primer

Pretpostavimo u prethodnom primeru da su promenljive `OsnovicaA`, `OsnovicaB` *i* `Visina` *deklarisane kao* `int` *promenljive, a* `Povrsina` *kao* `double` *promenljiva. U fazi semantičke analize će se prepoznati da operator deljenja ima operande različitog tipa, te će se u stablo umetnuti operator konverzije* `int_u_double` *(implicitna konverzija):*



NAPOMENA: Kod strogo tipiziranih jezika (poput Pascal-a), biće prijavljena semantička greška.

Uloge semantičkog analizatora

Šta sve radi semantički analizator?

- Obradjuje naredbe deklaracija i informacije o tipovima promenljivih i funkcija smešta u tabelu simbola
- Utvrdjuje domete deklaracija, kao i prava pristupa u OOP jezicima (`private`, `protected`, `public`)
- Na osnovu prikupljenih informacija iz deklaracija proverava ispravnost upotrebe identifikatora u različitim kontekstima
- U programskim jezicima gde je to dozvoljeno, umeće implicitne konverzije gde je to moguće
- Odredjuje tipove složenih izraza
- Proverava ispravnost funkcijskih poziva (s obzirom na tipove argumenata i tip povratne vrednosti)
- Proverava ispravnost upotrebe pojedinih naredbi u odredjenim kontekstima (poput `break` i `continue` u C-u)
- Proverava da li se tip izraza koji funkcija vraća (npr. naredbom `return` u C-u) poklapa sa deklarisanim povratnim tipom i po potrebi umeće konverziju (ako jezik to dozvoljava)
- U jezicima koji to omogućavaju, vrši dedukciju tipova identifikatora (npr. u C++-u, Haskell-u itd.)
- ...

Primer – semantička analiza

Primer

Pretpostavimo da u jezik C imamo izraz `a+b` *pri čemu su promenljive a i b strukturnog tipa. Ovaj izraz je sintaksno ispravan, ali je semantički neispravan, jer strukture nije moguće sabirati u jeziku C.*

Primer

Pretpostavimo da u jeziku C imamo naredbu `x = 3;`*, pri čemu deklaracija promenljive x nije u dometu (ili ne postoji). Naredba je sintaksno ispravna, ali je semantički neispravna, jer se u C-u promenljive mogu koristiti samo u dometu odgovarajuće deklaracije.*

Primer

Pretpostavimo da u jeziku Java imamo naredbu `c.x = 5;`*, gde je c instanca klase MojaKlasa, a x njen privatni član tipa* `int`*. Ukoliko se naredba nalazi u metodi neke druge klase, tada će biti prijavljena greška, iako je naredba sintaksno ispravna.*

Primer

Pretpostavimo da u jeziku C imamo naredbu `continue;` *koja se ne nalazi unutar petlje. Iako je ovo sintaksno ispravna naredba (u skladu sa pravilima gramatike jezika C), na tom mestu ona semantički nema smisla, te će semantički analizator prijaviti grešku.*

Primer – semantička analiza

Primer

Naredba:

```
if(x == y) continue;
```

je sintaksno ispravna konstrukcija u jeziku C, iako ne mora biti i semantički ispravna (jer se naredba `continue` *ne može koristiti van petlje). Sa druge strane, naredba:*

```
if(x = y) x++;
```

je sintaksno (čak i semantički) ispravna konstrukcija u jeziku C, iako obično nije ono što želimo.

Etapa sinteze

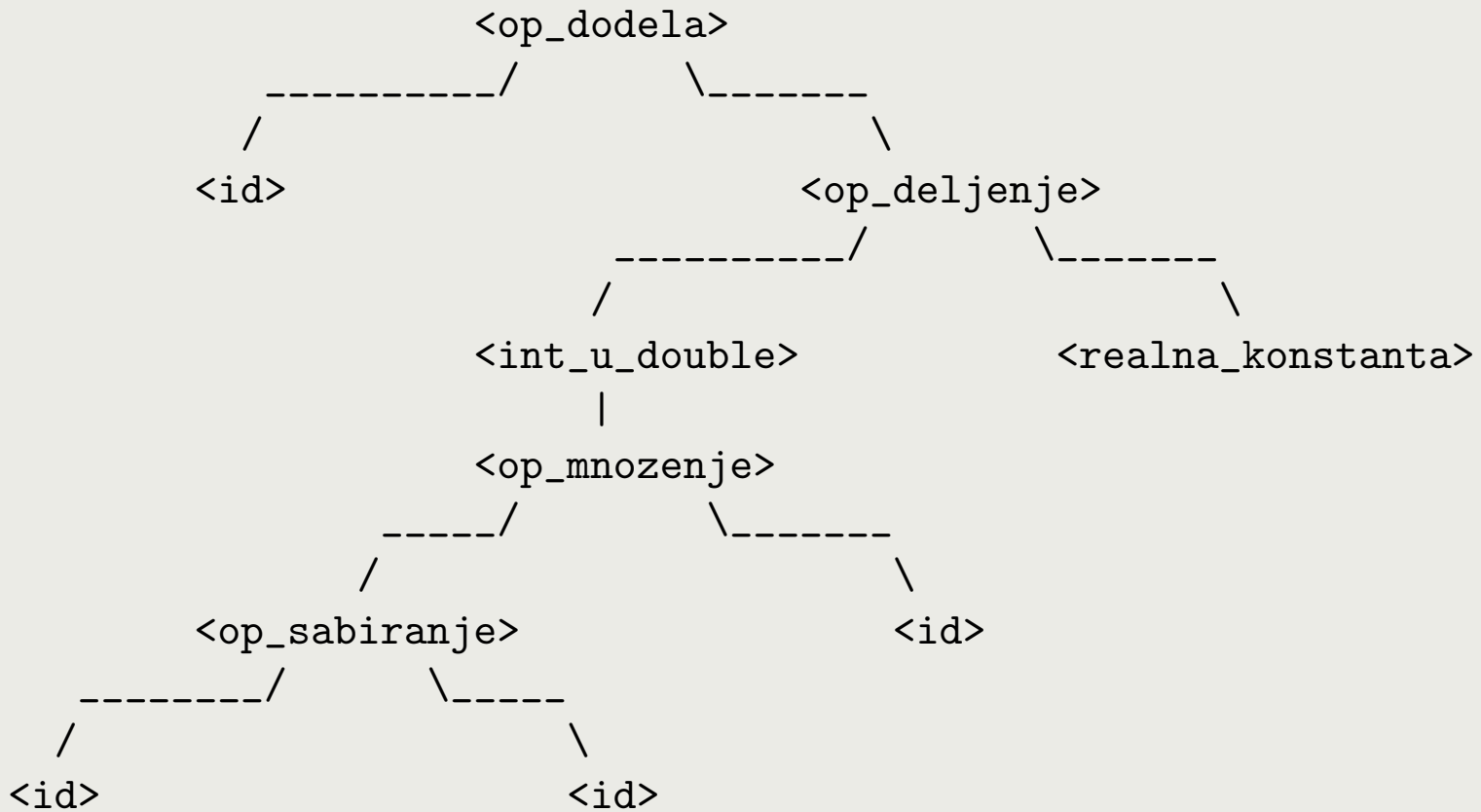
Faze u etapi sinteze

- Generisanje koda na medjujeziku
  - Ovaj jezik je blizak asembleru, ali je nezavisan od konkretnog hardvera i pogodan je za optimizaciju
  - Tipično je u pitanju troadresni kod ili stek-zasnovani kod
- Optimizacija koda
  - Evaluacija konstantnih izraza
  - Eliminacija neproduktivnog koda
  - Optimizacija petlji
  - Eliminacija skupih operacija
  - Eliminacija zajedničkih podizraza
  - Eliminacija repne rekurzije
  - ...
- Generisanje objektnog koda
  - Alokacija registara
  - Optimizacija zavisna od mašine

Primer

Primer

Podsetimo se dobijenog stabla apstraktne sintakse u našem primeru:



Primer (nastavak)

Primer

Na osnovu ovog stabla, polazeći od listova ka korenu, generišemo kôd na medjujeziku poput sledećeg:

```
t1 := OsnovicaA INT_ADD OsnovicaB
t2 := t1 INT_MUL Visina
t3 := INT_TO_REAL t2
t4 := t3 REAL_DIV 2.0
Povrsina := t4
```

Tokenima `<id>` *i* `<realna_konstanta>` *u listovima stabla pridružuju se identifikatori i konstante koje im odgovaraju u tabeli simbola. Promenljive* t_1, t_2, t_3, t_4 *su pomoćne promenljive koje odgovaraju vrednostima unutrašnjih čvorova stabla. Svakom unutrašnjem čvoru odgovara jedna operacija u medjukôdu. Izbor operacije zavisi od tipa vrednosti na koje se primenjuje, što opet znamo na osnovu informacija prikupljenih u fazi semantičke analize.*

Primer (nastavak)

Primer

Nakon optimizacije, gornji medjukôd se prevodi u kôd na konkretnom asemblerskom jeziku. U zavisnosti od arhitekture, svakoj naredbi medjukôda može odgovarati jedna ili više instrukcija asemblerskog jezika. Veoma bitan postupak ovde je alokacija registara, kojima se promenljivama iz prethodnog kôda pridružuju registri. Cilj je da što više promenljivih budu u registrima, kako bi se smanjio broj pristupa memoriji. Na x86-64 arhitekturi, gornji kod bi se mogao prevesti na sledeći način:

```
two: .double 2.0

     mov eax, OsnovicaA
     add eax, OsnovicaB
     imul dword ptr Visina
     cvtsi2sd xmm0, eax
     divsd xmm0, two
     movsd Povrsina, xmm0
```

Pregled

Interpretacija i kompilacija

Kompilacija

- Prethodno opisani postupak se naziva kompilacija
- Ulaz kompilatora (ili kompajlera) je program na izvornom jeziku (izvorni kôd)
- Izlaz kompilatora je program na objektnom jeziku (objektni kôd)
- Kompilator ne izvršava program, već generiše semantički ekvivalentan program na drugom jeziku (prevod izvornog programa)

Interpretacija

- Interpretacija podrazumeva izvršavanje operacija koje u semantičkom smislu odgovaraju naredbama datog programa na odredjenoj platformi
- Ulaz interpretatora je program koji treba izvršiti, kao i ulaz na koji taj program treba primeniti
- Izlaz interpretatora je izlaz programa za dati ulaz
- Interpretator ne generiše kôd na drugom jeziku, već samo tumači naredbe izvornog programa i preduzima akcije kojima se simulira njihov efekat na datoj platformi
- Za razliku od kompilacije, interpretacija uključuje samo etapu analize, dok etapa sinteze ne postoji

Interpretacija i kompilacija

Još o interpretaciji

- Prethodna definicija interpretatora je veoma opšta i uključuje mnogo različitih stvari
- Na primer, procesor se može smatrati (hardverskim) interpretatorom mašinskog jezika
- U tom slučaju, platforma na kojoj procesor izvršava semantičke akcije koje odgovaraju mašinskim instrukcijama je putanja podataka u njemu (engl. *datapath*)
- U klasičnom smislu, (softverski) interpretator predstavlja program koji analizira naredbe nekog drugog programa, tumači ih i izvršava njihov efekat na datoj hardverskoj arhitekturi
- Interpretator implementira etapu analize izvornog programa, a nakon što se dobije sintaksno stablo, umesto generisanja kôda izvršava se efekat odgovarajuće programske konstrukcije
- Interpretator u tabeli simbola održava i vrednosti promenljivih koje se menjaju tokom izvršavanja naredbi programa
- U našem primeru, interpretator bi, prateći stablo izraza `Povrsina = (OsnovicaA + OsnovicaB) * Visina / 2.0`, a na osnovu trenutnih vrednosti promenljivih u tabeli simbola, izračunao vrednost desne strane i upisao je u tabelu simbola kao novu vrednost promenljive `Povrsina`

Interpretacija i kompilacija

Kompilacija

- Primeri tipičnih kompilatorskih jezika su C, C++, Fortran, i sl.
- Programi koji se kompiliraju prevode se na mašinski jezik, a onda se direktno izvršavaju na procesoru (potencijalno veliki broj puta)
- Ponovno prevodjenje je neophodno samo ako se nešto menja u izvornom programu
- Kompilirani programi su obično znatno brži od interpretiranih
- Nedostatak je često manjak fleksibilnosti (npr. kompilatorski jezici su obično statički tipizirani)

Interpretacija

- Primeri tipičnih interpretatorskih jezika su Perl, PHP, Python, JavaScript, i sl.
- Interpretacija je drastično sporija od direktnog izvršavanja prevedenog kôda (program se svaki put iznova analizira)
- Interpretatorski programski jezici su često fleksibilniji i jednostavniji za programiranje
- Zbog toga su naročito pogodni za brzo rešavanje rutinskih zadataka
- Usled napretka interpretatora, u novije vreme interpretatorski jezici se koriste i kod obimnijih softverskih projekata
- Ipak, ograničeni su na primene kod kojih brzina nije tako bitna

Interpretacija i kompilacija

I još malo o interpretaciji

- Postoje i jezici kod kojih se koristi hibridni pristup:
  - program se sa izvornog jezika prevodi na neki medjujezik i generiše se odgovarajući medjukôd koji se čuva na disku
  - posebnim interpretatorom se vrši interpretacija tog medjukôda kad god se pokrene program
- Tipični primeri ovakvih jezika su Java i C#:
  - Java programi se prevode na medjukôd koji se naziva još i Java bajtkod
  - Java virtuelna mašina (JVM) se koristi da se prilikom pokretanja dobijenog programa interpetira bajtkod na procesoru računara
  - Slično, C# programi se prevode na jezik poznat kao CIL (Common Intermediate Language).
  - Interpretator CIL jezika koji je deo .NET okruženja intepretira ovako dobijene programe i izvršava ih na procesoru računara
- Čak i klasični interpretatorski jezici (poput PHP-a ili Perl-a) danas uglavnom koriste kombinaciju interpretacije i kompilacije, tako što se kôd interno prevodi na neki medjujezik koji se zatim interpretira
- Ovakve tehnike omogućavaju veći stepen optimizacije kôda i poboljšavaju efikasnost interpretacije

Zaključak

Čime se mi bavimo?

- U okviru ovog kursa detaljno se obradjuju faze analize koda
- Znanje stečeno na ovom predmetu je, stoga, dovoljno da se konstruišu jednostavni interpretatori
- Etapa sinteze se ne obradjuje u okviru ovog predmeta
- Otuda, na ovom predmetu nećete naučiti da konstruišete kompilatore
- Predmet Konstrukcija kompilatora se dataljnije bavi etapom sinteze