Programski prevodioci

Contents

[Glava 1 – Uvod 1](#_Toc12816980)

[1. Hibridni prevodioci 1](#_Toc12816981)

[2. Visejezicni kompilatori 2](#_Toc12816982)

[3. Obelezeno sintaksno stablo 3](#_Toc12816983)

[Glava 2 – Elementi teorije formalnih jezika 4](#_Toc12816984)

[1. Formalna azbuka 4](#_Toc12816985)

[2. Formalna definicija reci 4](#_Toc12816986)

[3. Formalni jezici 4](#_Toc12816987)

[4. Operacije nad jezicima 5](#_Toc12816988)

[5. Definicija formalne gramatike 5](#_Toc12816989)

[6. Gramatike po Comskom 5](#_Toc12816990)

[Glava 3 - Automati kao uredjaji za prepoznavanje jezika 6](#_Toc12816991)

[1. Definicija automata za prepoznavanje jezika 6](#_Toc12816992)

[2. Razlika izmedju deterministickih automata i nedeterministickih 7](#_Toc12816993)

[3. Tjuringova masina 7](#_Toc12816994)

[4. Linearno ograniceni automat 8](#_Toc12816995)

[5. Magacinski automat 9](#_Toc12816996)

[6. Konacni automat 10](#_Toc12816997)

[7. Pojam regularnih izraza 10](#_Toc12816998)

[8. Pravila kod regularnih izraza 11](#_Toc12816999)

[9. Izvodjenje regularnih izraza na osnovu konacnog automata 11](#_Toc12817000)

[Glava 4 – Leksicki analazitor 11](#_Toc12817001)

[1. Leksicki analizator 11](#_Toc12817002)

[2. Razlika izmedju lekseme, tokena i sablona 12](#_Toc12817003)

[Glava 5 – Sintaksni analizator: 12](#_Toc12817004)

[1. Zadatak sintaksnog analizatora i podela algoritama sintaksne analize 12](#_Toc12817005)

[2. Opsti oblik top-down algoritma 12](#_Toc12817006)

[3. Opsti bottom-up algoritam 13](#_Toc12817007)

[Glava 6 – Prediktivna analiza – LL1 analizatori 13](#_Toc12817008)

[1. Formalna definicija LL1 proste gramatike 13](#_Toc12817009)

[2. LL(1) sintaksna tabela 14](#_Toc12817010)

[3. Leva faktorizacija 14](#_Toc12817011)

[4. First i Follow 15](#_Toc12817012)

[5. Formalna definicija LL1 bez e 15](#_Toc12817013)

[6. Formalna definicja LL1 sa e 15](#_Toc12817014)

[Glava 7 – Analiza zasnovana na relacijama prvenstva 16](#_Toc12817015)

[1. Operatorska gramatika prvenstva 16](#_Toc12817016)

[2. Algoritam sintaksne analize kod operatorskih gramatika 16](#_Toc12817017)

[3. Funkcije prvenstva 16](#_Toc12817018)

[Glava 8 – LR analizatori 17](#_Toc12817019)

[1. LR algoritam 17](#_Toc12817020)

[2. LR sintaksna tabela preko grafa automata 18](#_Toc12817021)

[Glava 9 – Tabele simbola 18](#_Toc12817022)

[1. Tabela simbola - implementacija (lista, stablo, hash) 18](#_Toc12817023)

[Glava 10 - Genaratori leksickih i sintaksnih analizatora 18](#_Toc12817024)

[1. Definicija prioriteta i asocijativnosti operatora (presedance, levo/desno/neasocijativni) 18](#_Toc12817025)

[2. JFlex pravila (Lex) 18](#_Toc12817026)

1. [3. Metakarakteri za regularne izraze u jflex-u (Lex) 19](#_Toc12817027)

[4. Definicija specijalnih pocetnih stanja u jflex 20](#_Toc12817028)

[5. JFlex directive 21](#_Toc12817029)

[6. Makroi u JFlexu 22](#_Toc12817030)

[7. Definisanje gramatike u CUP specifikaciji 23](#_Toc12817031)

[8. Error handlovanje u sintaksnoj analizi u CUP 23](#_Toc12817032)

[Glava 11 – Atributne gramatike 24](#_Toc12817033)

[1. Atributne gramatike 24](#_Toc12817034)

[2. Navesti i objasniti tipove atributa atributnih gramatika 24](#_Toc12817035)

[3. Semanticka rutina 25](#_Toc12817036)

[4. Zadaci semanticke analize (prezentacija 09) 25](#_Toc12817037)

[5. Definisati sintaksno upravljanje (prezentacija 09) 25](#_Toc12817038)

[6. Metode za sintaksno upravljano prevodjenje (prezentacija 09) 25](#_Toc12817039)

[7. Type checking (prezentacija 09) 26](#_Toc12817040)

[Glava 12 – Medjukodovi 26](#_Toc12817041)

[1. Vrste medjukodova 26](#_Toc12817042)

[2. Razlika izmedju sintaksnog stabla i AST 27](#_Toc12817043)

[3. Nacini memorisanja AST 27](#_Toc12817044)

[4. Troadresni medjukod - nacini memorisanja 27](#_Toc12817045)

[Glava 13 – Optimizacija koda 27](#_Toc12817046)

[1. Optimizacija toka programa 27](#_Toc12817047)

[2. Podela optimizacija 28](#_Toc12817048)

[3. Eliminacija reduntantnih naredbi 28](#_Toc12817049)

[4. Koriscenje masinskih idioma,nabrojati ih 28](#_Toc12817050)

[5. Nabrojati vrste osnovnih optimizacija 28](#_Toc12817051)

[6. Zajednicki podizrazi 29](#_Toc12817052)

[7. Direktna redukcija u okviru petlje 29](#_Toc12817053)

[8. Indukcione promenljive 29](#_Toc12817054)

[Glava 14 – Generisanje koda 30](#_Toc12817055)

[1. Komponenta za upravljanje memorijom 30](#_Toc12817056)

[2. Strategije za organizovanje memorije u toku izvrsenja. 30](#_Toc12817057)

[3. Aktivacioni slogovi 31](#_Toc12817058)

[4. Objasti pojam aktivacionog stabla 32](#_Toc12817059)

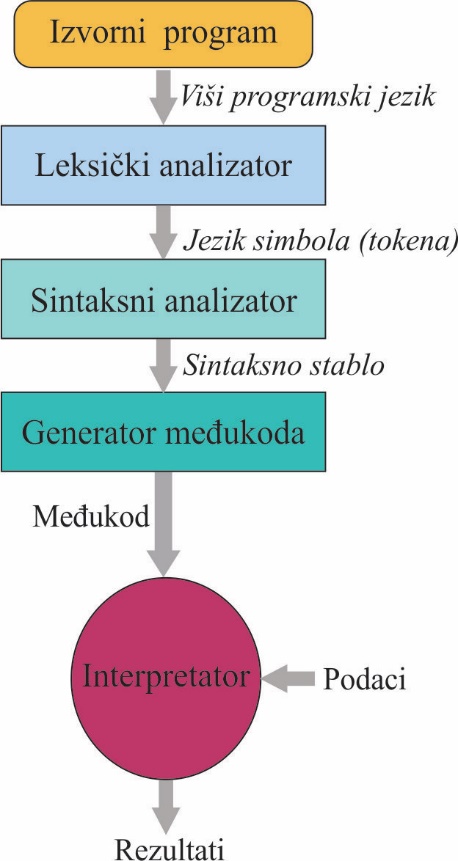
[5. Prevodjenje prilikom poziva 32](#_Toc12817060)

[6. Graf toka (278 str.) 33](#_Toc12817061)

# Glava 1 – Uvod

## Hibridni prevodioci

Pored čistih kompilatora i interpretatora danas se koriste i hibridni prevodioci, koji predstavljaju neku vrstu mešavine kompilatora i interpretatora, vrše analizu koda kao kompilatori, generišu moeđukod programa i nakon toga interpretiraju taj međukod.



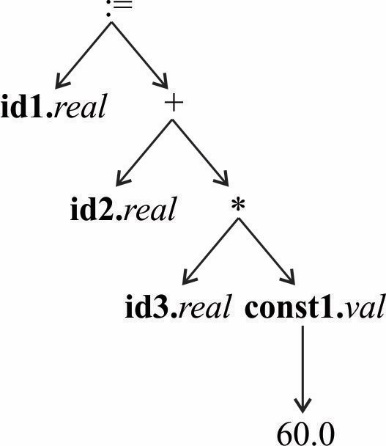
## Visejezicni kompilatori

Korišćenje međukoda takođe omogućava realizaciju višejezičnih kompilatora. Naime to su kompilatori koji imaju poseban deo za analizu programa za više različitih programskih jezika, pri čemu se kod svih ovih analizatora generiše isti međukod programa. Posle toga taj među kod može da se prevodi u mašinski jezik određenog računara, jednog ili više različitih procesora.



## Obelezeno sintaksno stablo

Sintaksni analizator može da bude obogaćen dodatnim semantičkim procedurama kojima se proveravaju neka dodatna, semantička pravila kojima je dopunjen opis jezika. U našem slučaju semantički analizator može da proverava da li su u naredbi dodeljivanja ispoštovana pravila jakih tipova podataka. U te svrhe analizator koristi Apstrakno sintaksno stablo obogaćeno atributima. Ovakvo stablo se naziva Označeno sintaksno stablo (*Annotated Syntax Tree*) za naš primer.



# Glava 2 – Elementi teorije formalnih jezika

## Formalna azbuka

Prilikom opisa jezika polazi se od azbuke kao osnovnog pojma. Neka je V konačan neprazan skup elemenata. Elemente skupa V nazivamo simbolima, slovima, a sam skup apstraktnom azbukom ili samo azbukom.

Primeri azbuka

V={0,1} Azbuci V pripadaju samo cifre 0 i 1;

V={*a,b,c*} Azbuci pripadaju samo slova *a*,*b* i *c*

Za opis programskih jezika se koriste nešto složenije azbuke.

Na primer jednu takvu azbuku mogu da čine sledeći simboli:

- Velika i mala slova Abecede: *A,a,B,b,C,c* ...

- Specijalni znaci: +, -,\*, :=, ...

- Reči kao što su: begin, end, if, then, ...

## Formalna definicija reci

Reč u kontekstu formalnih jezka se definiše kao konačan niz simbola azbuke V. Niz koji ne sadrži ni jedan simbol naziva se prazna reč i označava sa .

Primer Reči

Neka je V={*a,b,c*}. Sledeći nizovi su reči azbuke V: , *a, b, c, aa, bb, cc, ab, ac, abc, aabc*

Reč se može i formalno definisati sledećim skupom pravila:

* + 1.  reč nad azbukom V
    2. Ako je x reč azbuke V i ako je *a* element azbuke V tada je i x*a* reč azbuke V.
    3. Niz y je reč nad azbukom V ako i samo ako je dobijen pomoću pravila 1. i 2.

Za označavanje reči koristićemo završna mala slova Abecede napisana boldirano: u,v,w,x,y,z

Broj slova (simbola) u reči definiše se kao dužina reči. Za označavanje dužine reči **x** koristi se simbol |**x**|.

## Formalni jezici

Formalnim jezikom L nad azbukom V naziva se bilo koji skup reči nad tom azbukom.

Prema ovoj definiciji formalni jezik je i prazan skup reči kao i skup { ε } koji sadrži samo reč ε.

Formalni jezik nad azbukom V je bilo koji podskup skupa V\*. Bilo koji podskup, bilo da je konačan ili beskonačan, predstavlja jezik. Kako je V\* uvek beskonačan skup broj njegovih podskupova je takođe beskonačan.

## Operacije nad jezicima

Unija jezika

L i M L U M => L U M = { x | x ∈ L ∨ x ∈ M }

Nadovezivanje konkatenacija

L i M LM => LM = {xy | x ∈ L ∧ y ∈ M }

Potpuno zatvaranje

L\*=> U i=0;n Li

Pozitivno zatvaranje

L +=> Ui=1;n Li

## Definicija formalne gramatike

Formalna gramatika je sredstvo za opis jezika na konačan način.

Gramatika jezika opisuje kako se generišu reči koje pripadaju određenom jeziku.

Svaka formalna gramatika **G** se može definisati kao skup **G**=(**Vt**, **Vn**, S, **P**) gde je **Vt** skupterminalnih sibola**, Vn** skup neterminalnih simbola**, S** startni simbol i **P** skup smena definaisan na sledeći način:



Pri cemu vazi: V = Vt Vn  i važi Vt Vn  = 

Uočimo da je uslov da reč na levoj strani pravila mora da sadrži bar jedan neterminalni simbol.

## Gramatike po Comskom

Gramatike tipa 0

G = ( Vn, Vt, S, P) u kojoj su sve smene iz skupa P oblika:

x → y, gde je x∈V\*VnV\* ∧ y ∈V\*

Gramatika tipa 1

Konteksna gramatika

Za x → y vazi | y |≥| x | Kako je |x| >= 1 sledi da je i |y| >= 1,

što znači da na desnoj strani pravila ne može da bude prazan niz ε.

Gramatike tipa 2

Beskonteksne gramatike

Ako u gramatici G svaka smena ima oblik:

A → y, A∈Vn , y ∈V\*

iZa ove gramatike se koristi i naziv Bekusova normalna forma BNF i najčešće se koriste za opis sintakse programskih jezika.

Gramatike tipa 3.

Regularne gramatike

Gramatika G je gramatika tipa 3 ako je svaka njena smena oblika:

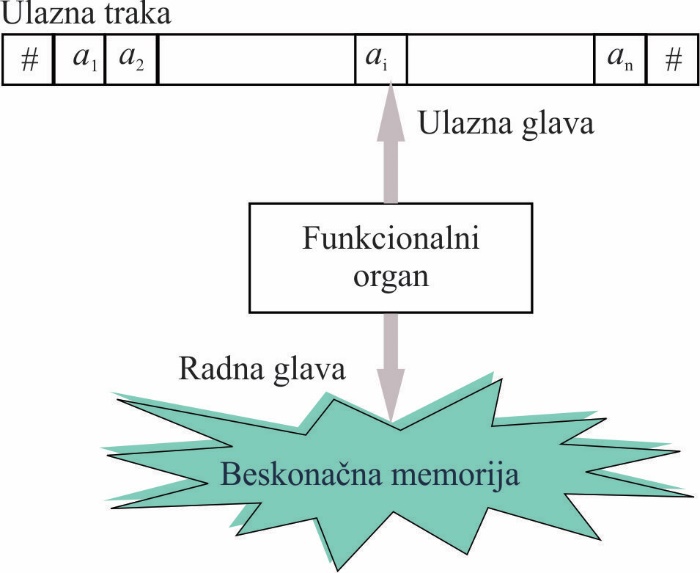
A → aB v A → a, A, B ∈Vn ∧ a∈ Vt U {ε}

Za ove gramatike se koriste još i nazivi Regularne gramatike, Gramatike sa konačnim brojem stanja i Automatne gramatike.

Služe za opis leksičkih elemenata jezika.

# Glava 3 - Automati kao uredjaji za prepoznavanje jezika

## Definicija automata za prepoznavanje jezika



Po ulaznoj traci se kreće ulazna glava koja u jednom trenutku čita jedan simbol sa trake, Ova glava se u opštem slučaju može kretati i napred i nazad (desno i lrvo).

Funkcionalni organ automata je njegov najznačajniji deo. Može da se nađe u nizu različitih stanja, u zavisnosti od toga kako je opisan jezik koji se prepoznaje. Funkcionalni organ beleži predistoriju prepoznavanja i za to koristi memoriju koja je u ovom opštem modelu predstavljena kao beskonačna memorija. Automat sa beskonačnom memorijom nije praktično izvodliv tako da se realni automati razlikuju od ovog modela po tome što koriste neki određeni tip memorije.

Proces prepoznavanja reči započinje tako što se ulazna glava pozicionira na startni simpol (#), a funkcionalni organ se nalazi u svom početnom stanju. U svakom koraku prepoznavanja, u zavisnosti od toga gde je pozicionirana ulazna glava (koji simbol čita) i stanja funkcionalnog organa, kao i simbola u radnoj memoriji koji čita radna glava, događaju se sledeće promene u automatu:

menja se stanje funkcionalnog organa

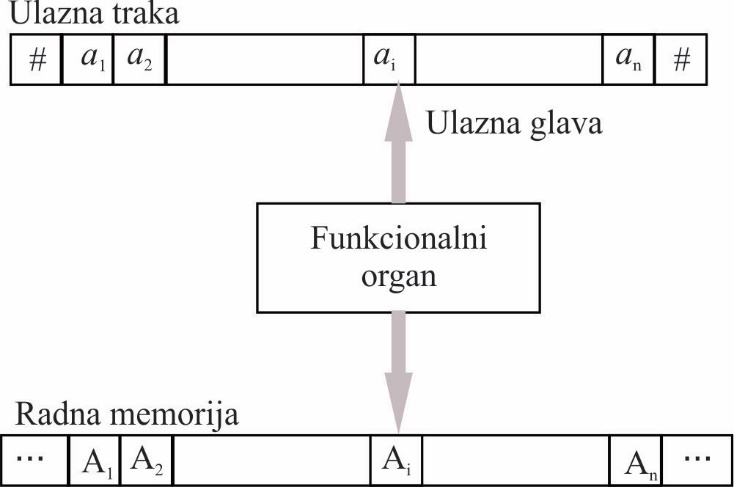
u radnu memoriju se upisuje jedan ili više simbola

**radna glava se pomera napred, nazad ili ostaje na poziciji na kojoj se nalazi.**

## Razlika izmedju deterministickih automata i nedeterministickih

Podela na determinističke i nedeterminističke izvršena je u odnosu na preskikavanje stanja funkcionalnog organa koje se vrši u toku prepoznavanja . Ukoliko je stanje u koje prelazi automat za određeni ulazni simbol i određeno zatečeno stanje, jednoznačno definisano onda je to deterministički automat. Ukoliko je ovo preslikavanje višeznačno onda je to nedeterministički automat.

## Tjuringova masina



Operativni organ menja stanje.

Ulazna glava se pomera.

Na radnu traku se upisuje novi simbol.

Pomera se radna glava.

2N Tjuringova mašina se može opisati sledećom entorkom: Tm=(Q, V, S, P, q0, *b* F) gde je:

Q – Skup stanja operativnog organa

V – Skup simbola na ulaznoj traci

S – Skup simbola na radnoj traci

F – Skup završnih, krajnjih stanja (Podskup skupa Q)

q0 – Početno stanje operativnog organa

*b* – Oznaka za prazno slovo

P – Preslikavanje definisano sa:

Ako je (qj, *B*, d1, d2) P(qi, *a*, *A*), gde je qj, qi  Q, *a*V, *A*S i d1,d2 {-1, 0, 1},

odnosno ako je ulazna glava pozicionirana na simbolu *a*, funkcionalni organ se nalazi u stanju qi, a radna glava iznad simbola *A*, automat će preći u stanje qj, na radu traku će upisti simbol B i izvršiće pomeranje ulazne glave i radne glave u skladu sa vrednostima d1 i d2. Pri tome važi sledeća notacija: Ako je d=0 nema pomeranja radne glave, za d=1 glava se pomera za jedno mesto napred (u desno), a za d=-1 glava se pomera za jedno mesto nazad (u levo).

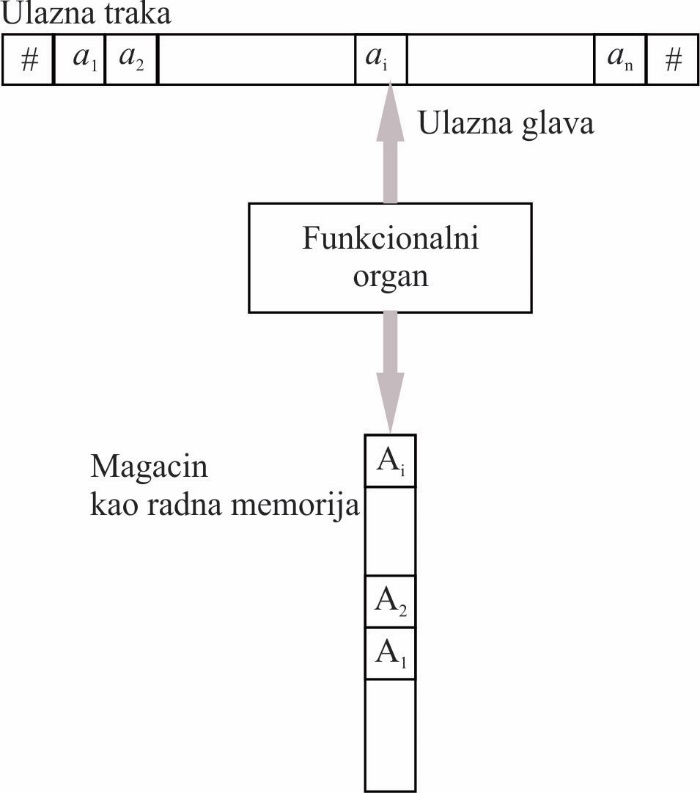
## Linearno ograniceni automat

Linearno ograničen automat je 2N Tjuringova mašina za koju može da se odredi konstanta k takva da se prilikom prepoznavanja reči: na radnoj traci upisuje niz r , odnosno, ako se prepoznaje reč wV\*, gde je |w|=*n*, na radnu traku se upisuje reč dužine *r,* pri čemu je |*r| n\*k*

Za prepoznavanje svih reči jezka dovoljna je radna traka k puta duža od najduže reči jezika.

1. Klase 2N i 1N Linearno ograničenih automata su međusobno ekvivalentne i svaka od njih prepoznaje jezike tipa jedan. Drugim rečima, za svaki jezik tipa jedan može se definisati nedeterministički linearno ograničeni automat koji ga prepoznaje.
2. Klase 2D i 1D linearno ograničenih automatu su međusobno ekvivalentne. To znači da se za svaki 2D linearno ograničeni automat može definisati ekvivalentan 1D linearno ograničeni automat.
3. Ekvivalentnost 2N i 2D LOA do sada nije dokazana.
4. Svaka klasa LOA ne prepoznaje sve jezike tipa jedan.

## Magacinski automat



2N Magacinski automat je definisan entorkom Ma=(Q, V, S, P, q0, *Z*0, F) gde je:

Q – Skup stanja operativnog organa

V – Skup simbola na ulaznoj traci

S – Skup magacinskih simbola

F – Skup završnih, krajnjih stanja (Podskup skupa Q)

q0 – Početno stanje operativnog organa

*Z*0 – Početni simbol na vrhu magacina

P – Preslikavanje definisano sa:



Za opis konfiguracije magacinskog automata u određenom trenutku obično se koristi sledeća struktura:

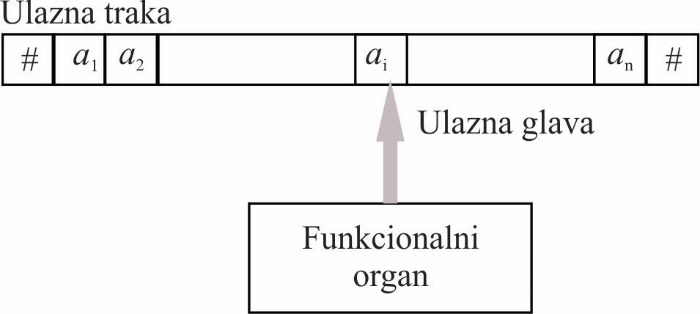
U magacin može da se upiše reč, da se ništa ne upiše i da se izbaci jedno slovo kada je r=e.

Magacinski automati su pogodni za prepoznavanje bezkonteksnih jezika. Dokazana su sledeća tvrđenja:

* 1N magacinski automat prepoznaje jezik L ako i samo ako je L beskonteksni jezik tipa 2.
* 2N magacinskim automatima mogu se prepoznavati svi beskontesni i neki, ali ne svi, konteksni jezici.
* Postoje beskonteksni jezici koji se ne mogu prepoznati 1D magacinskim automatima.
* 2D magacinski automati pored bezkonteksnih mogu da prepoznaju i neke konteksne jezike.
* Magacinski automati se koriste u sintaksnoj analizi programskih jezika.

## Konacni automat

Na osnovu stanja operastivnog organa i simbola koji čita ulazna glava, menja se stanje automata i ulazna glava pomera za jedno mesto ulevo ili udesno ili ostaje na svom mestu.



Konačan automat je opisan entorkom Ka=(**Q**, **V**, **P**, *q0*, **F**), gde je:

**Q** – Skup stanja operativnog organa

**V** – Skup simbola na ulaznoj traci

**F** – Skup završnih, krajnjih stanja (Podskup skupa **Q**)

*q0* – Početno stanje operativnog organa

**P** – Preslikavanje pravilima oblika: **Q**  {**V** #} **Q**

Za neki ulazni niz w ∈V\* kazemo da je prepoznat automatom ako je g(q0,w) ∈ F.

## Pojam regularnih izraza

Skup svih reči koje prepoznaje KA nazivamo regularnim i označavaćemo ih sa T(k):

T (k) = {x| g( q0, x) ∈F}

Sami regularni izrazi su nizovi formirani od slova neke azbuke V i skupa specijalnih simbola. . Regularni izraz obično skraćeno definišpe skup reči jezika definisanog nad azbukom V.

## Pravila kod regularnih izraza

* 1.  je regularni izraz.
  2.  je regularni izraz.
  3. Ako je *a*V, onda je *a* regularni izraz.
  4. Ako su r1 i r2 regularni izrazi, onda su i (r1| r2) i (r1 r2) regularni izrazi.
  5. Ako je r regularni izraz onda je i r\* takođe regularni izraz
  6. Nema drugih regularnih izraza nad azbukom V.

## Izvodjenje regularnih izraza na osnovu konacnog automata

# Glava 4 – Leksicki analazitor

## Leksicki analizator

Leksičkim analizatorom se realizuje faza leksičke analize u procesu prevođenja jezika. Odnosno, u kodu se identifikuju leksičke celine koje imaju neki sintaksni smisao, transformišu se u simbole (tokene) i prosleđuju se sintaksnom analizatoru.

Postoji više razloga zbog kojih se leksički analizator izdvaja od sintaksnog analizatora. Možda najvažniji su:

* Jednostavnija realizacija.
* Tako se dobija mogućnost da se tehnike za sintaksnu analizu razvijaju nezavisno od jezika. Ulaz u sintaksni analizator je niz simbola definisan određenim formalnim jezikom, nije zavisan od programskog jezika koji se prevodi.
* Prenosivost kompilatora – U fazi leksičke analize eliminišu se svi mašinski zavisni elementi i generiše mašinski nezavisna sekvenca simbola koja se prosleđuje sintaksnom analizatoru
* Povećanje efikasnosti kompilatora. Posebnim tehnikama baferisanja koje se koriste u realizaciji leksičkog analizatora doprinosi se njegovoj efikasnosti a time i efikasnosti kompilatora.

## Razlika izmedju lekseme, tokena i sablona

* LEKSEMA – Ulazni niz koji se prepoznaje na osnovu formalnog opisa pomoću šablona i za koji se generiše određeni token.
* TOKEN – Izlazni simbol koji se generiše kada je prepoznat određeni ulazni niz znakova. Primer : Integer ,Ključna reč: else, or, if, begin ....
* ŠABLON (pattern) – Potreban nam je način da jednoznačno definišemo koji nizovi odgovaraju kom tokenu. Regularni izraz, formalni opis ulaznih nizova za koje se generiše odeđeni token.

# Glava 5 – Sintaksni analizator:

## Zadatak sintaksnog analizatora i podela algoritama sintaksne analize

Proverava da li je program napisan u skladu sa formalnim opisom jezika.

Ulaz: niz tokena koje dobija od leksičkog analizatora.

Izlaz: sintaksno stablo koje pokazuje kako je program izgrađen

Top-down

• S leva na desno (krajnje levo izvođenje)

• S desna na levo (krajnje desno izvođenje)

Bottom-up

## Opsti oblik top-down algoritma

1. U izvedenu sekvencu upisati startni simbol gramatike, proglasiti ga za tekuci u izvedenoj sekvenci i pročitati prvi simbol iz ulaznog koda.

2. Ukoliko je tekuci simbol u izvedenoj sekvenci neterminalni simbol, zameniti ga desnom stranom prve smene na čijoj je levoj strani taj neterminalni simbol.

3. Ukoliko je tekuci simbol u izvedenoj sekvenci terminalni simbol jednak tekućem ulaznom simbolu, prihvati ga (preći na analizu sledećeg simbola).

4. Ukoliko je tekuci simbol u izvedenoj sekvenci terminalni simbol različit od tekućeg ulaznog simbola, poništiti dejstvo poslednje primenjene smene. Ukoliko postoji još koja smena za preslikavanje istog neterminalnog simbola, pokušati sa primenom sledeće smene, u suprotnom vratiti se još jedan korak nazad.

5. Ukoliko se vraćanjem došlo do startnog simbola i ne postoji više smena za njegovo preslikavanje, ulazni kod sadrži sintaksnu grešku.

6. Ukoliko nakon prihvatanja poslednjeg ulaznog simbola, ni u izvedenoj sekvenci nema neobradjenih simbola, kod je sintaksno korektan.

## Opsti bottom-up algoritam

1. Pročitati prvi simbol iz ulaznog koda i upisati ga u radni magacin.

2. Ponavljati sledeće korake dok se ne dođe do kraja ulaznog koda:

a) Ukoliko se na vrhu radnog magacina nalazi desna strana neke smene, redukovati frazu sa vrha radnog magacina (tj. frazu sa vrha radnog magacina zameniti simbolom sa leve strane odgovarajuće smene);

b) u suprotnom pročitati novi simbol iz ulaznog koda i smestiti ga u radni magacin.

3. Kada se dođe do kraja ulaznog koda, ukoliko je u radnom magacinu samo startni simbol gramatike, analiza je uspešno završena (kod je prihvaćen).

4. Kada se dođe do kraja ulaznog koda, ukoliko kod nije redukovan na startni simbol gramatike, a usput su izvršene neke redukcije, vratiti se na korak kada je izvršena poslednja redukcija, poništiti njeno dejstvo i preći na korak 2.

5. Ako se došlo do kraja ulaznog koda i pri tom nije izvršena ni jedna redukcija, analiza je završena neuspešno, ulazni kod je sintaksno neispravan.

# Glava 6 – Prediktivna analiza – LL1 analizatori

## Formalna definicija LL1 proste gramatike

Prosta LL(1) gramatika – Beskonteksna gramatika u kojoj sve smene za isti neterminalni simbol počinju različitim terminalnim simbolima:



Gramatika G=({*S*, *A*}, {*a*, *b*, *c*, *d*}, *S* , P}, gde je P sledeći skup pravila:

*S→aS*

*S→bA*

*A→d*

*A→ccA*

**Reč *a a b c c d* # pripada jeziku definisanom ovom gramatikom i izvodi se na sledeći način:

## LL(1) sintaksna tabela

Definiše akciju koja će se izvršiti u postupku analize zavisno od simbola sa vrha radnog magacina i tekućeg ulaznog simbola.



Opisani magacinski automat prepoznaje reč koju analizira tako što se u magacin ubacuje startni niz *S*# i u svakom koraku vrši preslikavanje određeno tablicom sintaksne analize, odnosno:

## Leva faktorizacija

Smene oblika:

gde su:  i 

Takva gramatika sigurno nije LL-1 gramatika. Međutim ona se lako transformiše u sledeću LL-1 gramatiku:



Neka je gramtika zadata sledećim skupom pravila:

*Nar  if Izraz then Nar else Nar | if Izraz then Nar*

*Izraz b*

Kako u datoj gramatici, za neterminal *Nar,* postoje dva previla koja na desnoj imaju reči sa istimprefiksom: *if izraz then nar,* ova gramatika nije LL-1*.*  Njoj ekvivalentna LL-1 gramatika bi bila:

*Nar  if Izraz then Nar else Nar Nar’*

*Nar’ else Nar| *

*Izraz b*

## First i Follow

FIRST

Skup terminalnih simbola sa kojima počinju reči izvedene iz reči α.



FOLLOW

Funkcija FOLLOW se definiše za neterminalne simbola, kao skup terminalnih simbola koji mogu u toku izvođenja da se nađu iza tog neterminalnog simbola, ili:



## Formalna definicija LL1 bez e

Gramatika bez ε pravila je LL(1) gramatika ako su z a sva pravila oblika :



Skupovi:

FIRST(α ), FIRST(α ), ,FIRST(α )

Disjuknktni po pravilima, odnosno:



## Formalna definicja LL1 sa e

Beskonteksna gramatika koja sadrži i pravila je LL-1 gramatika ako i samo ako za sva pravila oblika pravila  i  važi:





# Glava 7 – Analiza zasnovana na relacijama prvenstva

## Operatorska gramatika prvenstva

Operatorska gramatika u kojoj za svaka dva terminalna simbola važi najviše jedna relacija prvenstva naziva se *Operatorska gramatika prvenstva*.

Kod ovih gramatika moguće je primeniti algoritam za *Bottom-up* analizu u kome neće biti povratnih petlji.

## Operatorska gramatika

U jednoj smeni ne mogu da se pojave dva neterminalna simbola jedan do drugog.

## Algoritam sintaksne analize kod operatorskih gramatika

Skup smena gramatike se modifikuje tako da se svaki neterminalni simbol zameni jedinstvenim simbolom „p“.

• U radni magacin se upisuje simbol #,

• Čita se jedan po jedan simbol iz ulaznog niza i smešta u radni magacin sve dok izmeĎu poslednjeg terminalnog simbola u magacimu i pročitanog simbola važi relacija < ili =.

• Kada izmeĎu poslednjeg terminalnog simbola u magacinu pročitanog simbola važi relacija >, sa vrha magacina se izbacuje fraza „obuhvaćena relacijama < i >“ i ukoliko takva fraza postoji u skupu smena, u radni magaci se ubacuje oznaka neterminalnog simbola „p“. Ukoliko takva fraza ne postoji u skupu smena, u ulaznom nizu postoji greska i dalja analiza se prekida.

• Ulazni niz je uspešno prepoznat ukoliko se redukuje na jednu frazu, tj. ukoliko u radnom magacinu ostanu samo simboli # i „p“, a sledeći ulazni simboj je #.

## Funkcije prvenstva

Za određivanje funkcije prvenstva formira se orijentisani graf koji ima 2*n* čvorova. Potezi u grafu se određuju na sledeći način:

Ako važi relacija *a*⋅>*b*, tada postoji poteg od čvora *f*(*a*) prema čvoru *g*(*b*).

Ako važi relacija *a*<⋅*b*, tada postoji poteg od čvora *g*(*b*) prema čvoru *f*(*a*).

Ako važi relacija *a**b*, tada postoje potezi od čvora *f*(*a*) prema čvoru *g*(*b*) i od čvora *g*(*b*) prema čvoru *f*(*a*).

Vrednost funkcije prvenstva se odrdjuje kao dužina najdužeg puta koji polazi iz čvora koji odgovara toj funkciji. Pod dužinom puta podrazumeva se broj čvorova kroz koje put prolazi uključujući sam taj čvor.

# Glava 8 – LR analizatori

## LR algoritam

LR analizatori su u suštini i kao *Shift-reduce* analizatori, što znači da se kroz ulazni niz prolazi sleva u desno i nastoji se da se izvrši redukcija (*reduce*), ako redukcija nije moguća prelazi se na sledeći znak u nizu (*shift*). Za razliku od osnovnog algoritma za *Bottom-up* analizu koji je praktično *brut force* algoritam, kod ovih analizatora se analizom smena gramatike generiše LR sintaksna tabela koja određuje promenu stanja u automatu u zavisnosti od ulaznog tekućeg ulaznog simbola.

Algoritam:

Postaviti ulazni pokazivač *ip* na početak niza w#, koji se analizira.

repeat forever

begin

Neka je *q* znak u vrhu magacina i *a* znak na koji pokazuje ulazni pokazivač *ip*.

if *action*(*q*, *a*) = *sift* *k* then

begin

Ubaciti u magacin *a*, a zatim i *k* i pomeriti ulazni pokazivač *ip* za jedno mesto udesno.

end

else if *action*(*s*, *a*) = *reduce* *k*, pri čemu je *k*-ta smena gramatike:

A**  then

Izbaciti 2\*|**| simbola iz steka, ubaciti *A* u stek, a zatim i oznaku stanja koja se dobija na osnovu goto(s’, *A*), gde je s’ oznaka stanja u koje smo se vratili posle redukcije . Pri tome se generiše i izlaz A**.

end

else if *action*(*s*, #) = *accept* then

return

else *error*()

end

## LR sintaksna tabela preko grafa automata

# Glava 9 – Tabele simbola

# Tabela simbola - implementacija (lista, stablo, hash)

161 str

# Glava 10 - Genaratori leksickih i sintaksnih analizatora

## Definicija prioriteta i asocijativnosti operatora (presedance, levo/desno/neasocijativni)

Operatori mogu biti levo-asocijativni, desno-asocijativni i neasocijativni. Za svaki tip asocijativnosti postoji posebna Yacc definicja (%left, %right, %nonassoc). Prioritet operatora je definisan redosledom navođenja njihovih asocijativnosti i to tako što se uvek navode najpre asocijativnosti operatori najnižeg prioritera. Npr. asocijativnost i prioriteti osnovnih aritmetičkih operatora u yacc specifikaciji mogu biti definisani na sledeći način:

%left ‘+’, ‘-’

%left ‘\*’, ‘/’

%right STEPEN

Operatori, koji su navedeni u definicjama asocijativnosti i prioriteta, ne moraju biti navedeni u listi tokena.

## JFlex pravila (Lex)

I pravila u jflex specifikaciji se definišu na isti način kao i u lex-u. Jedina razlika u tome je što akcija mora da bude navedena kao blok naredbi (izmedju zagrada {} i kada ona sdrži samo jednu java naredbu).

Centralni deo Lex specifikacije su pravila i to je jedini obavezni deo specifikacije. Minimalni sadržaj Lex specifikacije je:

%%

<pravila>

Jedno pravilo Lex specifikacije sadrži dva dela:

1. Uzorak – regularni izraz koji definiše sekvencu simbola koja se traži u ulaznom tekstu,
2. Akciju – kod na odredišnom jeziku (na programskom jeziku na kojem alat generiše izlazni kod) koji će se izvršiti kada se navedeni uzorak prepozna. Akcija se od uzorka odvaja bar jednim belim simbolom (blanko znakom, tabulatorom, ili prelazom na novi red).

Kada se Lex koristi za generisanje leksičkog analizatora, uzorcima se definišu regularni izrazi reči (leksema) programskog jezika za koji se analizator pravi, a akcija se obično završava *return* naredbom koja vraća token koji odgovara prepoznatoj reči.

U regularnim izrazima za defininisanje uzorka mogu da se koriste tekst simboli (koji se baš takvi kakvi jesu porede sa znacima iz ulaznog teksta) i metasimboli koji imaju svoje specijalno značenje.

## Metakarakteri za regularne izraze u jflex-u (Lex)

|  |  |
| --- | --- |
| Znak | Značenje |
| + | Prethodni znak ili grupa se ponavlja jednom ili više puta. |
| \* | Prethodni znak ili grupa se ponavlja nijednom ili više puta. |
| ? | Prethodni znak može ali i ne mora da se pojavi. |
| {n} | Prethodnji znak ili grupa se ponavlja tačno n puta. |
| {n,m} | Prethodni znak ili grupa se ponavlja minimalno n, a maksimalno m puta |
| {name} | Poziv makroa (ime lex promenljive) |
| [ ] | Alternativa – izbor jednog od navedenih simbola unutar zagrade |
| - | Sa prethodnim i narednim znakom definiše opseg simbola - može se koristiti samo unutar alternative |
| ^ | Koristi se na početku alternative ili na početku regularniog izrara. Ukoliko se nalazi na početku alternative, označava negaciju skupa znakova koji sledi – vrši se izbor jednog od simbola koji nisu navedeni u alternativi. Ukoliko se nalazi na početku regularnog izraza, označava da se taj regularni izraz primenjuje samo za izdvajanje podstringova (particija) sa početka linije. |
| $ | Koristi se na kraju regularnog izraza i označava da se taj regularni izraz primenjuje samo za prepoznavanje podnizova koji se nalaze na kraju linije. |
| / | Označava da se podnizovi definisani regularnim izrazom koji prethodi izdvajaju samo ukoliko ukoliko se iza njih nađe podniz definisan regularnim izrazom koji sledi. |
| ( ) | Grupisanje simbola. |
| \ | Poništava specijalno značenje metakaraktera koji sledi. |
| “ “ | Poništavaju specijalna dejstva svih metasimbola između. |
| | | Izbor alternative (bira se između simbola (ili grupe) koji prethodi i simbola (ili grupe) koji sledi. |
| ! | Negira znak koji sledi - na toj poziciji se može pojaviti svaki znak osim navedenog. |
| ~ | Prihvata sve simbole do pojave znaka ili grupe koja sledi, uključujući i taj znak/grupu. |
| . | Zamenjuje bilo koji znak. |
| <name> | Koristi se na početku regularnog izraraza i označava da se primenjuje samo ukoliko prepoznavanje podniza polčinje od navedenog stanja. |

## Definicija specijalnih pocetnih stanja u jflex

Prilikom prepoznavanja svake nove reči, polazi se od unapred definisanog početnog stanja konačnog automata. Po default-u, početno stanje generisanog konačnog automata je stanje koje ima redni broj 0. Nekada se javlja situacija da iste reči imaju različita značenja zavisno od konteksta u kojem su upotrebljene. Da bi se ta specijalna značenja pojedinih reči prepoznala uvode se nova početna stanja konačnog automata koji prepoznaje reči jezika.

Specijalna početna stanja mogu da budu:

* Potpuno definisana specijalna početna stanja – za njih treba definisati prelaze za svaki simbol ulazne azbuke. U engleskoj literaturi se ovakva stanja nazivaju *exclusive states*. Simbolička imena takvih stanja se u lex specifikaciji navode u liniji koja počinje sa %x.
* Parcijalno definisana specijalna početna stanja – za njih se mogu definisati samo prelazi za one simbole koje se nalaze na početku reči koje imaju drugačije značenje od onog koje imaju kada analiza počinje od početnog stanja 0. Za prepoznavanje onih reči koje imaju isto značenje i u jednom i u drugom slučaju koristiće se uzorci navedeni za *default*-no početno stanje. Za ovakva stanja se u engleskoj literaturi koristi termine incluseive states. Simbolička imena takvih stanja se u lex specifikaciji navode u liniji koja počinje sa %s.

Prelaz na korišćenje novog početnog stanja prilikom izdvajanja sledeće reči (podniza) koristi se funkcija begin.

## JFlex directive

Direktive kojima se podešavaju parametri generisane klase – leksičkog analizatora:

* + %class <ImeKlase>

Definiše ime generisane klase – ukoliko ova ocija nije navedena ime generisane klase je Yylex.

* + %implements <intrefejs1>[,<interfejs2>][,…]

Ukoliko je ova direktiva navedena generisana klasa implementira navedene interfejse.

* + %extends <ImeKlase>

Ukoliko je ova direktiva navedena generisana klasa nasledjuje navedenu klasu.

Direktive kojima se podešavaju parametri funkcije koja vrši izdvajanje sledeće reči iz ulaznog teksta:0

* + %function <ImeFunkcije>

Definiše ime generisane funkcije– ukoliko ova direktiva nije navedena ime generisane funkcije je yylex().

* + %int ili %integer

Postavlja povratni tip funkcije koja vrši leksičku analizu na tip int.

* + %type <ImeTipa>

Postavlja povratni tip funkcije koja vrši leksičku analizu na navedeni tip.

Ukoliko ni jedna od prethodne dve direktive nije navedena, podrazumenvani povratni tip leksičkog analizatora je Yytoken.

Direktive kojima se uključuju brojači linija i karaktera:

* + %line

Uključuje brojač linija u izvornom fajlu. Generiše se promenljiva yyline (tipa int) koja pamti redni broj tekuće linije (brojanje počinje od 0).

* + %char

Uključuje brojač karaktera u izvornom fajlu. Generiše se promenljiva yychar (tipa int) koja pamti redni broj prvog karaktera izdvojene reči (brojanje počinje od 0).

* + %column

Uključuje brojač karaktera u tekućoj liniji. Generiše se promenljiva yycolumn (tipa int) koja pamti redni broj prvog karaktera izdvojene reči u tekućoj liniji (brojanje počinje od 0).

Direktive kojima se nalaže generisanje samostalne aplikacije (tj. generisanje main metoda):

* + %debug

Generiše se main metod koji poziva leksički analizator sve dok se ne dodje do kraja ulaznog fajla. Na standardni izlaz (tj. u Java konzoli) ispisuju se svi izdvojeni tokeni kao i kod akcije koja je tom prilikom izvršena.

* + %standalone

Generiše se main metod koji poziva leksički analizator sve dok se ne dodje do kraja ulaznog fajla. Ispravno prepoznati tokeni se ignorišu, a na standardni izlaz se ispisuju neprepoznati delovi ulaznog fajla.

## Makroi u JFlexu

U jflex specifikaciji se makroi definišu na sledeći način:

<ImeMakroa> = <RegularniIzraz>

(Za razliku od lex specifikacije, ovde se nakon imena makroa navodi znak ’=’.)

## Definisanje gramatike u CUP specifikaciji

Sadrži (opciono) definiciju startnog simbola gramatike i (obavezno) definicje njenih smena.

Format definicija startnog simbola gramatike je:

start with <ImeSimbola>;

Ukoliko je definicija srtartnog simbola gramatike izostavljena, kao i u Yacc specifikaciji, podrazumeva se da je startni simbol gramatike simbol koji se nalazi na levoj strani prve navedene smene.

Smene gramatike se u Cup specifikaciji definišu parvilima koja se navode u formatu:

<LevaStrana> ::= <DesnaStrana> [ {: <Akcija> :} ];

Akcija je java kod koji se izvršava u trenutku kada se vrši redukcija po navedenoj smeni.

Više smena koje na levoj strani imaju isti neterminalni simbol se mogu skraćeno navoditi na isti način kao u Yacc specifikaciji.

Za oporavak od grešaka se, takodje, koristi ista notacija kao u Yacc specifikaciji (uvode se pomoćene smene sa error simbolom).

Ukoliko u akcijama treba koristiti dodatne atribute simbola koji se nalaze na desnoj strani smene, uvode se reference na te atribute na sledeći način:

<ImeSimbola> : <Ime>

<Ime> je u ovom slučaju referenca na član value objekta klase Symbol.

value atributu simbola koji se nalazi na levoj strani smene pristupa se korišćenjem reference RESULT

## Error handlovanje u sintaksnoj analizi u CUP

Sintaksni analizator generisan na osnvu specifikacije iz primera 9.11 će samo utvrditi da li je izraz korektno zapisan ili ne. U trenutku kada se u kodu pojavi prva greška, analiza će se prekinuti, biće pozvana funkcija yyerror koja će prikazati poruku “Syntax error”. Nedostatak ovog analizatora je što neće lokalizovati grešku, tj. neće nam dati informacije gde se ona nalazi, ni o kakvoj se grešci radi; a neće ni otkriti sve greške u kodu. Cilj analizatora je da prijavi sve greške i da korisniku ukaže gde se nalaze i kako da ih ispravi. Postupak koji nam omogućava da i kada se greška otkrije, analizator nastavi svoj rad se popularno naziva “Oporavak od greške”.

Pretpostavimo da je u nekom izrazu (koji se prepoznaje gramatikom koja je definisana u primeru X.11) izostavljena zatvorena zagrada. Prilikom prepoznavanja izraza sa zagradama koristilo bi se pravilo:

E : ‘(‘ E ‘)’

Oporavak od ove greške bi podrazumevao da kada zagrada nedostaje prijavimo gresku a da se analiza ipak nastavi. To znači da treba uvesti fiktivnu smenu po kojoj će se vrsiti redukcija ako zagrada nedostaje. U takvim smenama koristi se ključna reč error koja zamenjuje skup simbola proizvoljne dužine za koji akcija u odredjenom stanju nije definisana. U ovom slučaju, ako zagrada nedostaje, vršiće se redukcija po smeni:

E : ‘(‘ E error { /\* prijava greske \*/ };

Pri definisanju ovih fiktivnih smena treba voditi računa da se što je moguće više spreči višestruko prijavljivanje iste greške (što je nemoguće potpuno izbeći) i da se ne desi da se u istom stanju pri pojavi istog simbola na ulazu može vršiti redukcija po većem broju smena. U takvim slučajevima yacc će prijaviti takozvane *reduce/reduce* konflikte (koje on ne može da prevazidje) i kod neće generisati. Ako se prilikom kreiranja sintaksne tabele pojave *shift/reduce* konflikti (njihov uzrok je objašnjen u poglavlju o LR sintaknim analizatorima), prijaviće se semo upozorenje, a u sintaksnoj tabeli biće upisana *shift* akcija.

# Glava 11 – Atributne gramatike

## Atributne gramatike

Da bi se proverila semantička ispravnost (ili realizovala bilo koja naredna faza prvođenja) često je potrebno da se uz simbole koji učestvuju u smenama pamte i neke dodatne informacije o njima.

• Dodatne informacije o simbolima se nazivaju atributi simbola, a ovakve gramatike atributne gramatike.

• Sintaksno stablo obogaćeno vrednostima atributa simbola se naziva obeleženo (notirano) sintaksno stablo.

## Navesti i objasniti tipove atributa atributnih gramatika

Generisani(sintetizovani) atribut

Vrednosti atributa terminalnih simbola određuje leksički analizator.

` • Vrednosti atributa neterminalnih simbola se izračunavaju kao funkcije atributa simbola koji se nalaze u čvorovima potomcima odgovarajućeg čvora.

• Vrednosti ovih atributa se propagiraju kroz sintaksno stablo (odozdo naviše).

X→Y1 Y2 ...Yn

X.a = f(Y1 .a, Y2 .a, ..., Yk .a, ..., Yn .a)

Nasleđeni atributi

Vrednosti ovih atributa se izračunavaju kao funkcije atributa simbola koji se nalazi u roditeljskom čvoru i atributa simbola na istom nivou levo od simbola čiji se atribut određuje.

* + Ovi atributi se propagiraju kroz sintaksno stablo odozgo-naniže.

X→Y1 Y2 ...Yn

Yk .a = f(Y1 .a, Y2 .a, ..., Yk-1 .a)

## Semanticka rutina

Opis naredbi jezika preko bezkonteksnih gramatika obično nije dovoljan za uspešno prevođenje jezika. Postoji mnogo problema koji se ne mogu predstaviti formalnim opisom jezika. Zbog toga se kod praktičnih rešenja pravilima kojima se definiše jezik pridružuje skup dodatnih informacija tako što se simbolima gramatike dodaju određeni atributi. Vrednosti ovih atributa se izračunavaju preko semantičkih rutina koje se pridružuju pravilima gramatike.

Koriste se dva načina da se semantičke rutine pridruže pravilima gramatike:

* Sintaksno upravljane definicije (Syntax Directed Definitions)
* Translacione šeme (Translation Shemes)

Sintaksno upravljane definicije su jednostavno definicije rutina pridružene pravilima. Daju se na dosta visokom nivou i ne sadrže informaciju o tome kako se realizuju i kojim redosledom se primenjuju. Translacione šeme pokazuju redosled u kome se primenjuju semantička pravila što na način utiče i na njihovu implementaciju. To znači da su nešto nižeg nivoa od definicija.

Na osnovu semantičkih rutina mogu da se generišu grafovi zavisnosti (*Dependency Graphs*) koji pokazuju zavisnost između atributa i određuju kojim redosledom se izračunavaju vrednosti atributa.

## Zadaci semanticke analize (prezentacija 09)

Sintaksna analiza- utvrdjuje da li je struktura programa korektna

Semantička analiza – utvrđuje:

* 1. Da li su različiti strukturni elementi medjusobno usaglašeni
* Zato se naziva još i kontekstno-zavisna analiza (context-sensitive)
  1. Da li su sve promenljive koje se koriste u programu deklarisane,
  2. Da li je promenljivoj dodeljena vrednost pre njenog korišćenja,
  3. Da li je promenljiva „vidljiva“ u tački u kojoj se koristi (opseg važenja)
  4. Da li su operatori primenjeni nad operandima odgovarajućeg tipa,
  5. Da li je određeno ime, ime funkcije, promenljive, klase...
  6. Da li je lista stvarnih parametara u pozivu funkcije usaglašena sa listom fiktivnih parametara u definiciji funkcije,
  7. Da li klasa sartži član koji se koristi,
  8. Da li je korišćenje određenog člana klase u skladu sa njegovim pravom pristupa

## Definisati sintaksno upravljanje (prezentacija 09)

Rutine za proveru semantičke ispravnosti koda se pridružuju pravilima (smenama gramatike), odnosno semantičke rutine se izvršavaju u neterminalnim čvorovima sintaksnog stabla.

• Generalizacija ove ideje – sintaksno-upravljano prevođenje - sve naredne faze prevođenja se realizuju tako što se odgovarajuće rutine izvršavaju u čvorovima sintaksnog stable

## Metode za sintaksno upravljano prevodjenje (prezentacija 09)

Jedno-prolazni prevodioci

* + Svi atributi su istog tipa (nasleđeni ili generisani) prilagođeni algoritmu sintaksne analize
  + Kod top-down algoritama se koriste nasleđeni atributi
  + Kod bottom-up algoritama generisani

Više-prolazni prevodioci

* + Mogu da budu definisani i nasleđeni i generisani atributi
  + U prvom prolazu se generiše samo sintaksno stablo
  + Sintaksno stablo se obilazi više puta i prilikom svakog obilaska se izračunavaju novi atributi

## Type checking (prezentacija 09)

Najbitnija komponenta semantičkog analizatora

Gruba definicija: Type checker proverava da li su broj operanada i njihovi tipovi u izrazima odgovarajući .

Tip definiše oseg vrednosti podataka koje se mogu predstaviti i skup operacija koje se nad njima mogu izvoditi

Tip imaju:

* Promenljive
* Konstante
* Funkcije
* Izrazi

Postoje ugrađeni tipovi podataka i korinički definisani tipovi.

Zadaci : (ne mora valjda)

* 1. Da li su oba operanda u binarnim aritmetičkim izrazima numeričkog tipa,
  2. Da li je indeks polja celobrojnog tipa,
  3. Da li tip svakog stvarnog parametra u pozivu funkcije odgovara tipu formalnog parametra,
  4. Da li postoji klasa navedena kao roditeljska u definiciji izvedene klase,
  5. Da li klasa koja implementira interfejs sadrži sve funkcije definisane u interfejsu

# Glava 12 – Medjukodovi

## Vrste medjukodova

Međukodovi visokog nivoa:

* 1. Apstraktno sintaksno stablo
  2. Poljska inverzna notacija

Međukodovi niskog nivoa – pseudo-asemblerski jezici

* 1. Troadresni međukod

## Razlika izmedju sintaksnog stable I AST

## Nacini memorisanja AST

Apstraktno sintaksno stablo se može memorisati kao statička struktura ili kao dinamička.

## Troadresni medjukod - nacini memorisanja

• Pomoću uređenih četvorki (Quadruples)

• Pomoću uređenih trojki (Triples

• Pomoću uređenih trojki sa indirektnim ardesiranjem (Indirect triples) – umesto naredbi pamte se pokazivači na naredbe – pri promeni redosleda instrukcija menjaju se samo pokazivači

Primeri na prezentaciji 10.

# Glava 13 – Optimizacija koda

## Optimizacija toka programa

U toku generisanja međukoda (ili izlaynog koda) često se javljaju skokovi na skokove

* 1. Bezuslovni skok na bezuslovni
  2. Uslovni skok na bezuslovni,
  3. Bezuslovni skok na uslovn

Transformacija bezuslovnog skoka na bezuslovni:

go to L1 => go to L2

L1: go to L2 => L1: go to L2

Ako posle ove transformacije nema skokova na naredbu označenu sa L1: onda se ona može zameniti neoznačenom naredbom, a nekad i potpuno eliminisati.

Transformacija uslovnog skoka na bezuslovni:

if a < b go to L1 => if a < b go to L2

L1: go to L2 => L1: go to L2

Transformacija bezuslovnog skoka na uslovni:

go to L1 L1: if a< b go to L2

… go to L3

L1: if a< b go to L2 …

L3: L3:

* …

L2: L2:

Broj naredbi je ostao isti ali je drugi slučaj efikasniji:

• U prvom slučaju se uvek izvršava goto i if ,

• U drugom se uvek izvršava if, a go to samo kada navedeni uslov nije ispunjen

## Podela optimizacija

* 1. Zajednicki podizrazi
  2. Prostiranje naredbi za kopiranje
  3. Eliminisanje neaktivnog koda
  4. Optimizacija petlji
  5. Eliminacija indukcionih promenljivih
  6. Direktna redukcija

## Eliminacija reduntantnih naredbi

Eliminisanje STORE-LOAD parova:

Često se rezultat jedne operacije odmah koristi u narednoj. Kao posledica primene formalnih algoritama za prevođenje (naredba po naredba), u takvim slučajevima se generišu parovi naredbi:

Store tk

Load tk

Ovakav skup naredbi se prosto izostavlja (eliminiše).

Eliminacija se ne vrši ako je druga (Load) instrukcija obeležena (postoji skok na tu instrukciju iz drugih delova koda.

Store tk

Lab1: Load tk

Primer prezentacija 11.

## Koriscenje masinskih idioma,nabrojati ih

• Ovo je mašinski zavisna transformacija u kojoj se koriste prednosti i specifičnosti asemblerskog jezika ciljne mašine.

• Npr. neki asemblerski jezici imaju mogućnost izvršavanja auto-inkrementa ili auto-dekrementa nekog operanda pre ili posle izvršenja neke instrukcije.

• U 8086 asembleru loop instrukcija dekrementira sadržaj registra cx

1. samoindeksiranja,
2. samoinkrementiranja,
3. specifičnih adresiranja,
4. specifičnih registara procesora i
5. korišćenje steka.

## Nabrojati osnovne optimizacije

* 1. Elininisanje redundantnih naredbi.
  2. Optimizacija toka programa
  3. Algebarska uprošćenja
  4. Korišćenje mašinskih idioma

Blokovi za primere u pitanjima:

B1 B2 B3 B4

i := m -1 i:= i +1 j := j -1 if i >= j go to B6

j := n t2 := 4\*I t4 := 4 \* j

t1 := 4 \* n t3 := a [t2] t5 := a [t4]

v := a [t1] if t3 < v go to B2 if t5 > v go to B3

B5 B6

t6 := 4 \* I t11 := 4 \* I

x := a [t6] x := a [t11]

t7 := 4 \* i t12 := 4 \* i

t8 := 4 \* j t13 := 4 \* n

t9 := a [t8] t14 := a [t13]

a [t7] := t9 a [t12] := t14

t10 := 4 \* j t15 := 4 \* n

a [t10] := x t15 := 4 \* n

go to B2

## Zajednicki podizrazi

Izraz E naziva se zajedničkim ako je pre toga već bio izračunat. Npr. u bloku B5 izračunato je t6 kao 4\*i, a posle toga se ista vrednost pamti i kao t7. Isto važi i za t8 i t10. Na svim mestima gde se koristi t7 može da stoji t6, a umesto t10 može da stoji t8

Zajednički podizrazi mogu da se pronalaze i na globalnom nivou ako promenljive koje u njima učestvuju ne menjaju vrednost u međuvremenu

## Direktna redukcija u okviru petlje

Transformacija kojom se složenije (skuplje) naredbe zamenjuju jednostavnijim (jeftinijim) naredbama

• U našem primeru, direktna redukcija je primenjena u sprezi sa eliminisanjem indukcionih promenljivih.

• U okviru petlji u blokovima B2 i B3, naredbe množenja t2 := 4 \* i i t4 := 4 \* j zamenjene su jednostavnijim naredbama oduzimanja t4 := t4 - 4 i sabiranja t2 := t2 + 4.

## Indukcione promenljive

• Indukcione promenljive su promenljive čija promena izaziva promenu drugih promenljivih. Vrednost jedne promenljive direktno zavisi od druge. Ove zavisnosti se mogu iskoristiti za direktnu redukciju koda.

Npr. u bloku B3 imamo j = j-1, a zatim t4 :=4\*j. Ova naredba može da bude zamenjena naredbom: t4 := t4 – 4

B3 pre transformacije

j := j -1

t4 := 4 \* j

t5 := a [t4]

if t5 > v go to B3

B3 posle transformacije

t4 := 4 \* j

|

t4 := t4 - 4

t5 := a [t4]

if t5 > v go to B3

# Glava 14 – Generisanje koda

## Komponenta za upravljanje memorijom

1. Organizacija memorije u toku izvršenja programa:
   1. odredjivanje gde će se u memoriji smestiti kod programa,
   2. kako će se memorisati podaci različitih tipova,
   3. gde će se u memoriji smestiti podaci različitih klasa memorije
2. Pristup podacima smeštenim u različitim zonama memorije
3. Promena sadržaja memorije u trenutku poziva potprograma i povratka na pozivajući modul

3. Prevođenje naredbe poziva i naredbe povratka

## 2. Strategije za organizovanje memorije u toku izvrsenja.

Staička alokacija memorije

• Sadržaj memorije se ne menja u toku izvršenja programa – i kod programa i svi podaci su poznati u fazi prevodjenja

* + Za svaki potprogram postoji samo jedan aktivacioni slog što znači da ova strategija ne podržava rekurzivne potprograme

Dinamička alokacija memorije

• Sadržaj memorije se stalno menja u toku izvršenja programa

* + mogu se kreirati novi podaci, brisati
  + mogu se kreirati novi delovi koda, brisati
  + mogu se kreirati novi aktivacioni slogovi, brisati

• Povremeno se akriviraju posebni pomoćni programi (garbage collectori) koji vode računa o korišćenju memorijskog prostora

* + kad se uoče delovi memorije na koje ne ukazuje ni jedna referenca, oni se uklanjaju iz memorije, tj. taj deo memorije se oslobađa

Polu-statička ili stek alokacija memorije

• Operativna memorija se deli na 2 zone:

1. Statičku – čuva kod programa i globalne podatke

2.Dinamičku koja se deli na:

* + Stack – čuva aktivacione slogove potprograma
  + Heap – čuva podatke koji se po potrebi kreiraju u toku izvršenja programa i uklanjaju kada više nisu potrebni.

## Aktivacioni slogovi

|  |
| --- |
| Rezultat potprograma |
| Stvarni parametri |
| Link prema aktivacionom slogu pozivajućeg modula |
| Link prema podacima pozivajućeg modula |
| Stanje procesora u trenutku poziva |
| Lokalni podaci |
| Privrameni podaci |

Za svaki poziv potprograma kompilator obično generiše jedan aktivacioni slog u kome čuva sve informacije potrebne tom pozivu potprograma. U opštem slučaju aktivacioni slog ima strukturu. Međutim, struktura aktivacionog sloga nije ista kod svih jezika niti kod svih kompilatora za jedan jezik. Ona se obično prilagođava potrebama kompilatora

1. Na vrhu sloga, smešta se vrednost koju potprogram vraća glavnom programu. Ova vrednost se zbog efikasnosti vraća u neki od registra procesora.
2. Polje namenjeno stvarnim parametrima se koristi od strane glavnog programa da prenese parametre potprogramu koji odgovaraju konkretnom pozivu potprograma.
3. Opciono polje sa upravljačkim linkovima se koristi da se obezbedi veza sa aktivacionim slogom modula iz kojeg je pozvan potprogram.
4. Opciono polje sa linkovima za pristup podacima postoji kod jezika koji dozvoljavaju da se koriste nelokalni podaci, podaci iz aktivacionih slogova drugih potprograma. Ovo polje kod Fortrana nije potrebno zato što se svi nelokalni podaci čuvaju na jednom mestu.
5. Deo aktivacionog sloga namenjen čuvanju statusa procesora služi da se u njemu memorišu vrednosti registara pocesora koje su zatečene u trenutku poziva potprograma. Ovde se obično smešta vrednost Brojača neredbi i registara procesora opšte namene.
6. Polje namenjenu lokalnim podacima služi za smeštaj podataka koji pripadaju samo potprogram, definisani su u potprogramu.
7. Polje namenjeno privremenim podacima služi za memorisanje podataka koje generiše kompilator kao pomoćne lokacije u koje smešta međurezultate.

## Objasti pojam aktivacionog stable

Ako programski jezik dozvoljava rekurzivne pozive potprograma onda nije moguće koristiti statičku alokaciju memorije. Naime u ovom slučaju u jednom trenutku može da postoji više aktiviranih instanci potprograma i za svaki takav poziv potreban je poseban aktivacioni slog.

Jedna od tehnika koju je u ovom slučaju moguće koristiti je alokacija memorije pomoću steka, u okviru koje se prilikom svakog poziva potprograma kreira novi aktivacioni slog i smešta na stek. Kada se izvršavanje određene istance potprograma zavši odgovarajči aktivacioni slog se skida sa steka. U jednom trenutku u steku može da se nalazi više istanci aktivacionog sloga jednog potprograma, kao i više tazličitih aktivacionih slogova.

U vreme kompiliranja programa zna se samo veličina aktivacionog sloga ali ne i dubina rekurzije (koliko će aktivacionih slogova jednog potprograma u nekom trenutku biti u steku).

Ovo se resava aktivacionim stablom.

Svaki čvor u aktivacionom stablu sa prestavlja jedan poziv potprograma. Pozivi potprograma koji se izvrše iz jedne instance potprograma predstavljeni su čvorovima do kojih vode potezi iz čvora koji odgovara toj istanci. Pri tome redosled poziva odgovara redosledu čvorova poslatrano sleva u desno.

## Prevodjenje prilikom poziva

Prilikom poziva potprograma generise se aktivacioni slog i smesta na stek.

Koristi se bazni pokazivac koji uvek pozakuje na neku lokaciju aktivacionog sloga poslednjeg pozvanog potprograma,tj trenutni potprogram.

Kada se izvrsenje potprograma zavrsi bazni pokazivac treba da se vrati da ukazuje na aktivacioni slog pozivajuceg modula, dakle stara vrednost BP-a mora da se cuva I ta lokacija gde se cuva se zove kontrolna veza.

Na tu lokaciju BP ukazuje sve vreme.

|  |
| --- |
| Rezultat potprograma |
| Stvarni parametri |
| Adresa povratka |
| Staro BP  kontrolna veza |
| Prethodni sadrzaj radnih registara |
| Lokalne promenljive |
| Privrameni promenljive |

U trenutku poziva izvrsava se generisanje sekvence poziva a u trenutku povratka generisanje sekvence povratka.(generisanje I smestanje/skidanje aktivacionog sloga sa steka).

Sekvenca poziva:

Deo instrukcija koje pripadaju pozivajućem modulu

• Rezervacija prostora za rezultat potprograma

SUB SP, size

• Smeštanje u stek stvarnih parametara

PUSH arg\_n ... PUSH arg\_1

• Poziv potprograma CALL name

Deo instrukcija koje pripadaju pozvanom modulu

• Smeštanje u stek prethodne vrednosti baznog pokazivača

PUSH BP

• Promena sadržaja baznog pokazivača

MOV BP, SP

• Smeštanje u stek konteksta procesora

(sadržaja radnih registara)

PUSH AX PUSH BX PUSH CX PUSH DX PUSH SI PUSH DI

• Rezervacija prostora za lokalne promenljive

SUB SP, size

Sekvenca povratka:

Deo instrukcija koje pripadaju pozvanom modulu

• Upis rezultata u predviđenu lokaciju

MOV [BP+x], rez

• Izbacivanje lokalnih promenljivih ADD SP, size

• Vraćanje starog konteksta procesora

POP DI POP SI POP DX POP CX POP BX POP AX

• Preuzimanje stare vrednosti baznog pokazivača

POP BP

• Povratak u pozivajući modul

RET

Deo instrukcija koje pripadaju pozivajućem modulu

• Izbacivanje stvarnih parametara is steak

ADD SP, size

• Preuzivanje rezultata POP reg

## Graf toka (278 str.)

Pitanja u dokumentu su pitanja koja su se padala u januarskom i aprilskom ispitnom roku 2019. godine kod prof. Suzane Stojkovic. Da li su to sva pitanja? Sumnjam. Ali mogu sluziti kao vodic kroz ucenje i eventualno obracanje paznje na stvari koje vam se mozda ne cine tako bitnim.