Zdravko
DOVEDAN HAN

FORMALNI JEZICI I PREVODIOCI

• prevođenje i primjene

Predgovor

Ovo je treća knjiga od tri koje obrađuju teme iz formalnih jezika i prevodilaca. Posvećena je problemima teorije prevođenja i primjenama, posebno u dizajnu interpretatora i predprocesora. Svojim sadržajem u potpunosti pokriva sadržaj kolegija *Teorija prevođenja i primjene* predmeta *Formalni jezici i prevodioci* koji se predaje na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta Sveučilišta u Zagrebu. Sve teme su obrađene u devet poglavlja koja predstavljaju određene logičke cjeline.

0. *Osnove* je uvodno poglavlje u kojem je dan sažeti pregled definicija i temeljnih pojmova iz prve dvije knjige.

1. *Uvod u teoriju prevođenja* sadrži dva temeljna postupka ("formalizma") prevođenja. Prvi je "shema prevođenja", poznata još i kao sintaksno-upravljano prevođenje. Primijenjena je na beskontekstne gramatike. Drugi je postupak "pretvarač". Opisani su konačni i stogovni pretvarači.

2. *Jezici za programiranje* poglavlje je posvećeno jezicima za programiranje. Važna primjena teorije formalnih jezika je u definiranju i prevođenju jezika za programiranje. Temeljna mu je namjena za pisanje programa koji će biti izvršeni na računalu. Najprije su dane generacije jezika za programiranje i njihovu podjelu, potom općeniti opis ili specifikaciju jezika za programiranje. Na kraju je dan primjer definicije jezika **Exp**, jezika realnih izraza.

3. *Prevodioci* je poglavlje u kojem se uvode osnovni pojmovi i definicije teorije prevođenja. Evolucija jezika za programiranje, od asemblerskog jezika do jezika visoke razine i jezika četvrte generacije, uvela je potrebu za posebnim programima – prevodiocima. Opisane su vrste prevodilaca, faze prevođenja i jednoprolazno prevođenje.

U sljedeća su tri poglavlja opisane faze prevođenja jezika za programiranje: leksička analiza, sintaksna analiza i generiranje koda.

4. *Leksička analiza* je prva faza prevođenja. Opisane su dvije vrste leksičke analize: višeprolazna (neizravna), koja se primjenjivala na početku razvoja jezika za programiranje, i jednoprolazna (izravna) koja se koristi u današnjim jezicima za programiranje.

5. U poglavlju *Sintaksna analiza* opisali smo dva postupka sintaksne analize jezika za programiranje: rekurzivni spust, koji pripada klasi jednoprolaznih postupaka parsiranja s vrha, i prepoznavač jezika sa svojstvima upravljan tablicom prijelaza i akcija.

6. *Generiranje koda* je poglavlje posvećeno posljednjoj fazi prevođenja. Opisane su dvije vrste generiranja koda: interpretiranje i predprocesiranje.

Preostala tri poglavlja posvećena su primjeni teorije prevođenja u konstrukciji interpretatora, definiranju jezika za programiranje i projektiranju predprocesora.

7. *Interpretator jezika PL/0* poglavlje je koje sadrži definiciju jezika PL/0 (Wirth) i primjer realizacije njegova interpretatora.

8. *Definicija jezika DDH* uvodno je poglavlje u kojem je opisan jezik DDH koji će u devetom poglavlju biti primjer izvornog jezika u projektiranju predprocesora. Jezik DDH (Dijkstra, 1976) karakteriziran je strogom definicijom semantike i relativno jednostavnom sintaksom uz zadovoljenje svih poznatih principa strukturnog programiranja.

9. *Predprocesor jezika DDH* završno je poglavlje u kojem je realiziran predprocesor jezika DDH. U potpunosti je opisan postupak njegove leksičke analize, sintaksne analize i prevođenja. Sintaksna analiza realizirana je kao prepoznavač jezika sa svojstvima, a to je prepoznavač upravljan tablicom prijelaza i akcija. Predprocesor je realiziran u Pythonu. Ciljni jezik je Python.

U svim je poglavljima dano puno primjera koji upotpunjuju teorijska razmatranja, posebno pojedine definicije i algoritme. Na kraju poglavlja su zadaci.

Za potpuno razumijevanje pojedinih tema obrađenih u ovoj knjizi neophodno je predznanje iz prethodne dvije knjige koje obuhvaća: definiciju jezika, generatore jezika (regularne izraze, gramatike i automate) i postupke sintaksne analize beskontekstnih jezika i jezika sa svojstvima. Osim toga, primjena teorije prevođenja podrazumijeva solidno znanje: jezika za programiranje (dali smo prednost jeziku Python), algoritama i struktura podataka, tehnika programiranja i paradigmi programiranja (objektno-orijentirano, funkcionalno i logičko programiranje). Mi smo se odlučili (kao i u prethodne dvije knjige) za jezik Python koji sve to objedinjava.

Kao što je već rečeno, knjiga je namijenjena studentima studija društveno-humanističke informatike na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta Sveučilišta u Zagrebu, ali sam siguran da će knjiga s ovakvim sadržajem zainteresirati studente sličnih usmjerenja, inženjere informatike i računarskih znanosti, napredne srednjoškolce i mnoge druge samouke informatičare. Svojim sadržajem knjiga može biti pravi izazov za one koji bi htjeli dizajnirati vlastiti jezik i konstruirati mu odgovarajući prevodilac (interpretator ili predprocesor).

Posebno ću biti zahvalan svima onima koji budu pažljivo pročitali ovu knjigu, prihvatili barem jedan njezin djelić i primijenili u svojoj praksi. Sve primjedbe i pitanja možete mi poslati na e-mail adresu:

zdovedan@hotmail.com

U Zagrebu, rujna 2013. godine

Autor

Sadržaj

Mojoj dragoj Ines

Koja mi je obojila život
prekrasnim jezikom cvijeća...

mojoj vjernoj pratilji
na stazama zemaljskim...

i na putovima duhovnosti!

0.
OSNOVE

— Joseph

Tako to biva Josipe, stari moj,
Kad se osvojilo najljepšu
Među djevojkama Galileje,
Onu koja se zvaše Marija.

Mogao si Josipe, stari moj,
Uzeti Saru ili Deboru
I ništa se ne bi dogodilo,
Ali, odlučio si se za Mariju.

Mogao si Josipe, stari moj,
Ostati doma, tesati svoje drvo
Bolje nego otići u progonstvo
I skrivati se s Marijom.

Mogao si Josipe, stari moj,
Stvoriti djecu s Marijom
I učiti ih svome zanatu
Kao što je tebe tvoj otac učio.

Zašto je trebalo, Josipe,
Da tvoje dijete, to nevinašce,
Prihvaća te tuđe ideje
Koje su izazvale toliko plača kod Marije?

Katkad pomislim na tebe, Josipe,
Prijatelju moj siromašni, kad te se ismijava
Tebe koji se nisi upitao
Koliko je život s Marijom sretan bio.

Joseph

(Gorges MOUSTAKI/
(Zdravko DOVEDAN HAN)

U ovom su poglavlju sumirane definicije formalnih jezika dane u prethodne dvije knjige: definicija jezika, regularnih izraza, gramatika i automata i sintaksne analize. Potom su dani temeljni pojmovi o kompjuterima neophodni za potpunije razumijevanje tema obrađenih u ovoj knjizi, a koje se odnose na jezike za programiranje i njihove prevodioce.

0.1 JEZIK

Znak je jedinstven, nedjeljiv element. Alfabet je konačan skup znakova. Najčešće ćemo ga označivati sa $\mathcal{A}$. Ako se znakovi alfabeta $\mathcal{A}$ poredaju jedan do drugog dobije se niz znakova (engl. *string*) ili "povorka", ili "nizanica". Operacija dopisivanja znaka iza znaka, ili niza iza niza, naziva se nadovezivanje ili konkatenacija nizova.

Duljina niza znakova jest broj znakova sadržanih u njemu. Često se duljina niza znakova x označuje s $d(x)$ ili $|x|$. Niz znakova $aaaa\ldots a$, sačinjen od n jednakih znakova, piše se kao a^n.

Neka su x i y nizovi znakova nad alfabetom $\mathcal{A}$. Kaže se da je x prefiks ("početak"), a y sufiks ("dočetak") niza xy i da je y podniz niza xyz (x kao prefiks i y kao sufiks niza xy istodobno su i njegovi podnizovi). Ako je $x \neq y$ i x je prefiks (sufiks) niza y, kaže se da je x svojstveni prefiks (sufiks) od y.

Definira se i niz znakova koji ne sadrži nijedan znak. Naziva se prazan niz. Označivat ćemo ga s ε ili λ. Za bilo koji niz w vrijedi $\varepsilon w = w\varepsilon = w$. Duljina praznog niza jednaka je 0, $|\varepsilon| = 0$, odnosno, ako je a bilo koji znak, vrijedi $a^0 = \varepsilon$. Ako je $x = a_1 \ldots a_n$ niz, obrnuti niz (ili "reverzni" niz) jest x^R, $x^R = a_n \ldots a_1$. Umjesto x^R može se pisati x^{-1}. Vrijedi $x = (x^{-1})^{-1}$.

Ako je $\mathcal{A} = \{a_1, \ldots, a_n\}$ alfabet i $x \in \mathcal{A}^+$ s $N_{a_i}(x)$ označit ćemo broj pojavljivanja znaka a_i u nizu x. Ako su $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ i $\mathcal{B} = \{b_1, b_2, \ldots, b_n\}$ dva alfabeta, definira se njihov produkt:

$$\mathcal{AB} = \{a_i b_j : a_i \in \mathcal{A}, b_j \in \mathcal{B}\}$$

a to je skup svih nizova znakova duljine 2 u kojima je prvi znak iz alfabeta $\mathcal{A}$, a drugi iz skupa $\mathcal{B}$. Ako je $\mathcal{A} \neq \mathcal{B}$ tada je i $\mathcal{AB} \neq \mathcal{BA}$ (produkt dvaju alfabeta nije komutativan). Operacija potenciranja alfabeta, $\mathcal{A}^n$, $n \geq 0$, definirana je rekurzivno:

1) $\mathcal{A}^0 = \{\varepsilon\}$
2) $\mathcal{A}^n = \mathcal{A}\mathcal{A}^{n-1}$ za $n > 0$

Dakle, zaključujemo da će $\mathcal{A}^n$ biti skup svih nizova znakova nad alfabetom duljine n.

Skup svih nizova znakova koji se mogu izgraditi nad alfabetom $\mathcal{A}$, uključujući i prazan niz ε i sam alfabet, označivat ćemo sa $\mathcal{A}^*$. Vrijedi:

$$\mathcal{A}^* = \bigcup_{n=0}^{\infty} \mathcal{A}^n$$

Sa $\mathcal{A}^+$ označivat ćemo skup $\mathcal{A}^* \setminus \{\varepsilon\}$. Primijetiti da su $\mathcal{A}^*$ i $\mathcal{A}^+$ beskonačni ali prebrojivi skupovi! Sa $\mathcal{A}^{*k}$ označivat ćemo konačan skup (podskup od $\mathcal{A}^*$) svih nizova znakova nad $\mathcal{A}$ duljine od 0 do k, a sa $\mathcal{A}^{+k}$ označivat ćemo konačan skup (podskup od $\mathcal{A}^+$) svih nizova znakova nad $\mathcal{A}$ duljine od 1 do k. Unarna operacija * poznata je i pod nazivom Kleeneova zvjezdica, jer ju je prvi put definirao Stephen Kleene. Operacija $^+$ je Kleeneov plus.

Ako je $\mathcal{A}$ alfabet i $\mathcal{A}^*$ skup svih nizova znakova nad $\mathcal{A}$, jezik $\mathcal{L}$ nad alfabetom $\mathcal{A}$ jest bilo koji podskup od $\mathcal{A}^*$, tj.

$$\mathcal{L} \subseteq \mathcal{A}^*$$

Često se piše $\mathcal{L}(\mathcal{A})$ da se naznači definiranost nekog jezika $\mathcal{L}$ nad alfabetom $\mathcal{A}$. Nizove znakova koji čine elemente jezika nazivamo rečenice. Za jezik $\mathcal{L}$ u kojem za sve njegove rečenice w vrijedi da nisu svojstveni prefiks (sufiks) ni jednoj rečenici x, $x \in \mathcal{L}$ i $x \neq w$, kaže se da ima svojstvo prefiksa (sufiksa).

Operacije nad jezicima

S obzirom na to da je jezik skup, primjenom poznatih operacija nad skupovima mogu se iz definiranih graditi novi jezici. Elementi jezika su nizovi znakova pa se može definirati i operacija nadovezivanja.

Ako su $\mathcal{L}_1$ i $\mathcal{L}_2$ jezici, $\mathcal{L}_1 \subseteq \mathcal{A}_1^*$ i $\mathcal{L}_2 \subseteq \mathcal{A}_2^*$, tada je $\mathcal{L}_1\mathcal{L}_2$ nadovezivanje (ulančavanje ili konkatenacija) ili produkt jezika $\mathcal{L}_1$ i $\mathcal{L}_2$:

$$\mathcal{L}_1\mathcal{L}_2 = \{xy : x \in \mathcal{L}_1,\ y \in \mathcal{L}_2\}$$

Simboli i nizovi simbola

Često se promatraju nizovi znakova konačne duljine koji se mogu smatrati jedinstvenom, nedjeljivom cjelinom. Takvi nizovi znakova nazivaju se simboli ili riječi.

Skup svih simbola definiran nad alfabetom $\mathcal{A}$ označivat ćemo s $\mathcal{V}$ i nazivati rječnik. To je uvijek konačan skup. Budući je $\mathcal{V} \subseteq \mathcal{A}^*$, zaključujemo da je $\mathcal{V}$ jezik. Definira se i jezik nad rječnikom, tj. $\mathcal{L}_\mathcal{V} \subseteq \mathcal{V}^*$. Rečenice takvog jezika i dalje su nizovi znakova iz $\mathcal{A}^*$, ali se mogu promatrati i kao nizovi simbola rječnika $\mathcal{V}$.

Klasifikacija jezika

Prema hijerarhiji Chomskog jezike klasificiramo u četiri skupine (ili tipa), kao što je prikazano u sljedećoj tablici:

tip	naziv
0	bez ograničenja
1	kontekstni
2	beskontekstan
3	linearan

Da bismo odredili kojoj klasi jezika pripada neka rečenica, korisno je znati svojstvo napuhavanja ("pumping lemma") koje kaže da svaki jezik dane klase može biti "napuhan" i pri-tom još uvijek pripadati danoj klasi. Jezik može biti napuhan ukoliko se dovoljno dugi niz znakova jezika može rastaviti na podnizove, od kojih neki mogu biti ponavljani proizvoljan broj puta u svrhu stvaranja novog, duljeg niza znakova koji je još uvijek u istom jeziku. Stoga, ukoliko vrijedi svojstvo napuhavanja za danu klasu jezika, bilo koji neprazni jezik u klasi će sadržavati beskonačan skup konačnih nizova znakova izgrađenih jednostavnim pravilom koje daje svojstvo napuhavanja.

Regularni skupovi

Neka je $\mathcal{A}$ alfabet. Regularni skup (regularni jezik) nad $\mathcal{A}$ definiran je rekurzivno na sljedeći način:

1) $\varnothing$ (prazan skup) je regularni skup nad $\mathcal{A}$.
2) $\{\varepsilon\}$ je regularni skup nad $\mathcal{A}$.
3) $\{\mathtt{a}\}$ je regularni skup nad $\mathcal{A}$, za sve $\mathtt{a}$ iz $\mathcal{A}$.
4) Ako su P i Q regularni skupovi nad $\mathcal{A}$, regularni skupovi su i:

   a) $P \cup Q$ b) PQ c) P^* d) (P)

Dakle, podskup od $\mathcal{A}^*$ jest regularan ako i samo ako je $\varnothing$, $\{\varepsilon\}$ ili $\{\mathtt{a}\}$, za neki $\mathtt{a}$ u $\mathcal{A}$, ili se može dobiti iz njih konačnim brojem primjena operacija unije, produkta i (Kleeneove) operacije *. Izraz s regularnim skupovima može imati i zagrade da bi se naznačio prioritet izvršavanja ove tri operacije, pa najviši prioritet ima podizraz unutar zagrada, potom Kleenova operacija (potenciranje), produkt i, na kraju, unija.

SVOJSTVA REGULARNIH SKUPOVA

Sada ćemo dati jedno svojstvo regularnih skupova čime će biti određeno je li dani skup regularni. To je "svojstvo napuhavanja", a definirano je u slijedećoj lemi napuhavanja regularnih skupova: Neka je $\mathcal{R}$ regularni skup. Tada postoji konstanta k takva da ako je $w \in \mathcal{R}$ i $|w| \geq k$, tada se w može napisati kao xyz, gdje je $0 < |y| \leq k$ i $xy^i z \in \mathcal{R}$ za sve $i \geq 0$.

Lema napuhavanja definira osnovno svojstvo nizova regularnog skupa da svi proizvoljno dugi nizovi (rečenice) regularnog jezika mogu biti "napuhane", tj. postoji središnji dio niza koji se ponavlja proizvoljan broj puta da bi proizveo novi niz koji je u istom jeziku. U praksi se lema napuhavanja često koristi da bi se dokazalo da dani jezik nije regularan.

0.2 REGULARNI IZRAZI

Regularni izrazi (još i "pravilni izrazi") na alfabetu $\mathcal{A}$ označuju (generiraju) određene regularne skupove. Regularni izraz definira se rekurzivno, na sljedeći način:

(1) $\varnothing$ je regularni izraz koji označuje regularni skup $\varnothing$.
(2) ε je regularni izraz koji označuje regularni skup $\{\varepsilon\}$.
(3) $\mathtt{a}$ iz $\mathcal{A}$ je regularni izraz koji označuje regularni skup $\{\mathtt{a}\}$.
(4) Ako su p i q regularni izrazi koji označuju regularne skupove P i Q, redom, tada su:

(a) $(p+q)$ ili $(p|q)$ regularni izraz koji označuje regularni skup $P \cup Q$.
(b) (pq) regularni izraz koji označuje regularni skup PQ.
(c) $(p)^*$ regularni izraz koji označuje regularni skup P^*.

Operaciju "+" ili "|" čitamo "ili". Za regularni izraz p^* koristit ćemo i notaciju $\{p\}$ (što ne treba poistovjećivati sa skupom). Ako je pp^* regularni izraz, može se napisati kao p^+ ili $p\{p\}$. Također ćemo koristiti i notaciju $\{p\}_m^n$ koja ima značenje dopisivanje izraza p najmanje m, najviše n puta, $m \leq n$. Izostavljeno m ima značenje 0, a izostavljeno n ima značenje *. Ako ne postoji dvoznačnost u nekom regularnom izrazu, suvišne se zagrade mogu izbaciti. Može se zamisliti da operacija * (i $^+$) ima najviši proritet, potom operacija nadovezivanja i, na kraju, operacija + (ili |).

Algebarska svojstva regularnih izraza

Reći ćemo da su dva regularna izraza jednaka (=) ako označuju isti regularni skup. Ako su α, β i γ regularni izrazi, tada vrijede sljedeća algebarska svojstva:

1) $\alpha+\beta = \beta+\alpha$	2) $\alpha+(\beta+\gamma) = (\alpha+\beta)+\gamma$	3) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
4) $\alpha(\beta+\gamma) = \alpha\beta+\alpha\gamma$	5) $(\alpha+\beta)\gamma = \alpha\gamma+\beta\gamma$	6) $\alpha\varepsilon = \varepsilon\alpha = \alpha$
7) $\alpha^* = \alpha+\alpha^*$	8) $(\alpha^*)^* = \alpha^*$	9) $\alpha+\alpha = \alpha$

0.3 GRAMATIKE

Gramatika je četvorka $G=(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$, gdje su:

$\mathcal{N}$ konačan skup neterminalnih znakova,

$\mathcal{T}$ konačan skup terminalnih znakova (alfabet) uz uvjet da je

$$\mathcal{T} \cap \mathcal{N} = \varnothing$$

$\mathcal{P}$ konačan skup parova nizova:

$$\{(\alpha, \beta): \alpha=\alpha_1\gamma\alpha_2;\ \alpha_1, \alpha_2, \beta \in (\mathcal{N} \cup \mathcal{T})^*,\ \gamma \in \mathcal{N}\}$$

(niz α je iz $(\mathcal{N} \cup \mathcal{T})^+$ i mora sadržati bar jedan znak iz skupa $\mathcal{N}$),

S poseban znak iz $\mathcal{N}$, $S \in \mathcal{N}$, nazvan početni znak (ili *početni simbol*).

Element (α, β) iz $\mathcal{P}$ piše se $\alpha \rightarrow \beta$ i naziva produkcija. Simbol "$\rightarrow$" čita se "producira", "može biti zamijenjeno s" ili "preobličuje se u". Ako $\mathcal{P}$ u nekoj gramatici sadrži produkcije:

$$\alpha \rightarrow \beta_1 \quad \ldots \quad \alpha \rightarrow \beta_n$$

piše se

$$\alpha \rightarrow \beta_1 | \ \beta_2 | \ldots | \beta_n$$

Znak "|" čita se "ili". β_i su alternative za α. Ako u $\mathcal{P}$ postoji produkcija oblika

$$\alpha \rightarrow \varepsilon | \ \beta | \ \beta\beta | \ \beta\beta\beta | \ \ldots$$

piše se $\alpha \to \{\beta\}$. Vitičaste zagrade omeđuju niz koji može biti izostavljen ili napisan jedanput, dvaput, triput, itd. Produkcija oblika:

$$\alpha \to \varepsilon \mid \beta$$

piše se $\alpha \to [\beta]$. Dakle, uglate zagrade omeđuju niz koji može biti izostavljen ili napisan jedanput. U daljnjem ćemo tekstu neterminale označivati velikim slovima engl. abecede. Terminali će biti mala slova engl. abecede i ostali znakovi (brojke, +, -, *, /, (,), itd.). Neterminal na početku prve produkcije bit će početni simbol.

Gramatika kao generator jezika

Rečenična forma gramatike $G=(N, T, P, S)$ definirana je rekurzivno:

1) Početni znak je rečenična forma.
2) Ako je $\alpha\delta\gamma$, gdje su $\alpha,\gamma\in(N\cup T)^*$, rečenična forma i $\delta\to\beta$ produkcija u P, tada je $\alpha\beta\gamma$ također rečenična forma.

Rečenična forma koja ne sadrži nijedan neterminal naziva se rečenica. Nad skupom $(N\cup T)^*$ gramatike $G=(N, T, P, S)$ definira se relacija $\Rightarrow$, čita se izravno izvodi, na sljedeći način: Ako je $\alpha\delta\gamma$ niz iz $(N\cup T)^*$ i $\delta\to\beta$ produkcija iz P, tada

$$\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$$

Ako za $\alpha_0, \alpha_1, \ldots, \alpha_n, \alpha_i\in(N\cup T)^*$, $n\geq 1$, vrijedi

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$$

tada je $\alpha_0 \overset{n}{\Rightarrow} \alpha_n$ niz izvođenja duljine n. Općenito se piše

$$\alpha_0 \overset{*}{\Rightarrow} \alpha_n,\ n\geq 0,\quad \alpha_0 \overset{+}{\Rightarrow} \alpha_n,\ n>0$$

i kaže da α_0 izvodi α_n.

Shodno dvjema prethodnim definicijama jezik generiran gramatikom G može se napisati kao

$$L(G) = \{\omega\in T^* : S^* \Rightarrow \omega\}$$

što čitamo: "Jezik L generiran gramatikom G jest skup rečenica dobivenih nizom svih mogućih izvođenja krenuvši od početnog simbola S".

Klasifikacija gramatika

Gramatike se mogu klasificirati prema obliku svojih produkcija. Za gramatiku $G=(N, T, P, S)$ kaže se da je:

1) Tipa 3 ili linearna zdesna ako je svaka produkcija iz P oblika

$$A \to xB \text{ ili } A \to x \qquad A, B\in N,\ x\in T^*$$

ili linearna slijeva ako je svaka produkcija iz $\mathcal{P}$ oblika

$A \to Bx$ ili $A \to x$ $\quad A, B \in \mathcal{N},\ x \in \mathcal{T}^*$

Gramatika linearna zdesna naziva se regularna gramatika ako je svaka produkcija oblika

$A \to aB$ ili $A \to a$ $\quad A, B \in \mathcal{N},\ a \in \mathcal{T}$

i jedino je dopuštena produkcija $\mathcal{S} \to \varepsilon$, ali se tada $\mathcal{S}$ ne smije pojavljivati niti u jednoj alternativi ostalih produkcija.

2) Tipa 2 ili beskontekstna ako je svaka produkcija iz $\mathcal{P}$ oblika:

$A \to \alpha$ $\quad A \in \mathcal{N},\ \alpha \in (\mathcal{N} \cup \mathcal{T})^*$

3) Tipa 1 ili kontekstna ako je svaka produkcija iz $\mathcal{P}$ oblika

$\alpha \to \beta$ uz uvjet da je $|\alpha| \leq |\beta|$

4) Bez ograničenja ili tipa 0 ako produkcije ne zadovoljavaju nijedno od navedenih ograničenja.

Sada možemo reći da je jezik bez ograničenja ako je generiran gramatikom tipa 0, kontekstan ako je generiran gramatikom tipa 1, beskontekstan ako je generiran gramatikom tipa 2 i linearan (ili regularan) ako je generiran gramatikom tipa 3 (ili regularnom gramatikom). Četiri tipa gramatika i jezika uvedenih prethodnom definicijom nazivaju se hijerarhija Chomskog.

Svaka regularna gramatika istodobno je beskontekstna, beskontekstna bez ε–produkcija je kontekstna i, konačno, kontekstna gramatika je istodobno gramatika bez ograničenja. Ako s $\mathcal{L}_i$ označimo jezik tipa i, vrijedi $\mathcal{L}_{i+1} \subseteq \mathcal{L}_i$, $0 \leq i < 3$. Regularne gramatike generiraju najjednostavnije jezike koji mogu biti generirani regularnim izrazima.

Prikaz gramatika

U prethodnim definicijama i primjerima razlikovali smo neterminalne i terminalne simbole prema vrsti znakova: velika slova bila su rezervirana za neterminale, a mala slova i ostali znakovi za terminale. Takvim dogovorom nije bilo neophodno uvijek posebno navoditi skupove neterminala i terminala. Bilo je dovoljno napisati produkcije i zadati početni simbol. Na taj način zadana je sintaksa jezika. Dva su najčešća načina prikaza gramatika: Backus-Naurovom formom i sintaksnim dijagramima.

BACKUS-NAUROVA FORMA (BNF)

Formalizam pisanja produkcija (ili "pravila zamjenjivanja") kojim smo dosad zadavali produkcije gramatike modifikacija je formalizma poznatog kao Backus-Naurova forma ili BNF. Prvi je put bio primijenjen u definiciji jezika ALGOL 60, 1963. godine i još uvijek je prisutan u praksi. Piše se prema sljedećim pravilima:

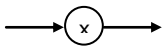
1) Neterminalni simboli pišu se između znakova "<" i ">".
2) Umjesto "→" koristi se simbol "::=" i čita "definirano je kao".

Pišući neterminale između znakova "<" i ">" moguće je izborom njihovih imena uvesti "značenje" u produkcije, jer će nas imena podsjećati na vrstu rečenica koja će se generirati u nekom podjeziku.

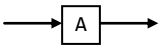
SINTAKSNI DIJAGRAMI

Produkcije gramatike $\mathcal{G}$ mogu biti prikazane i u obliku koji se naziva sintaksni dijagram. Sve je veća prisutnost sintaksnih dijagrama u novijoj literaturi, prije svega što se njihovom uporabom bolje uočava struktura jezika. Pravila konstruiranja sintaksnih dijagrama su sljedeća:

1) Terminalni simbol x prikazan je kao



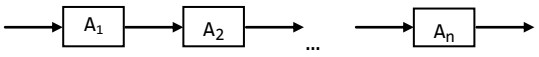
2) Neterminalni simbol A prikazan je kao



3) Produkcija oblika

$A \rightarrow A_1A_2 \ \ldots \ A_n \qquad A_i \in (\mathcal{N} \cup \mathcal{T})$

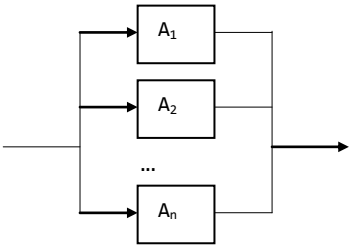
predstavlja se dijagramom



4) Produkcija oblika

$A \rightarrow A_1 | \ A_2 | \ \ldots | \ A_n \qquad A_i \in (\mathcal{N} \cup \mathcal{T})$

prikazuje se dijagramom

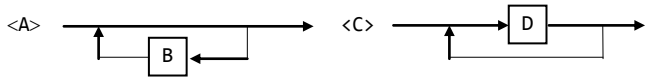


gdje je svaki A_i prikazan prema pravilima od (1) do (4). Ako je $A_i=\varepsilon$, ta se alternativa prikazuje punom crtom.

5) Produkcije oblika

$A \rightarrow \{B\} \quad \text{i} \quad C \rightarrow [D] \qquad B, D \in (\mathcal{N} \cup \mathcal{T})$

prikazuju se dijagramima:



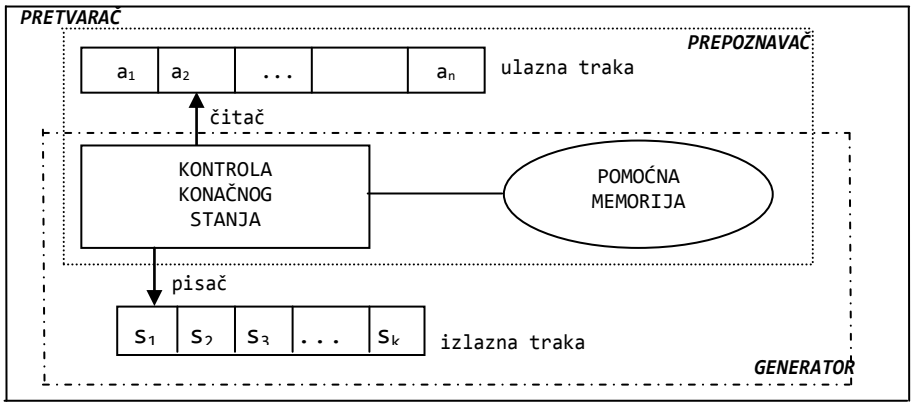
gdje su B i D prikazani dijagramima prema pravilima (1) do (4).

Često se u praksi pojednostavljuje pisanje sintaksnih dijagrama, posebno ako je nedvojbena razlika u pisanju terminala i neterminala. Na primjer, neterminali su riječi napisane malim slovima, a terminali su riječi napisane velikim slovima ili su to brojevi i ostali znakovi.

0.4 AUTOMATI

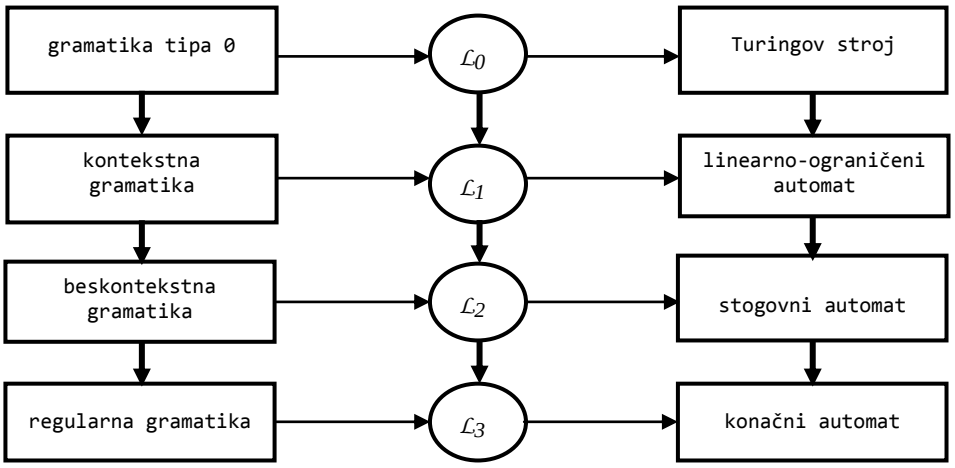
Osim gramatika, uvodimo automate kao važnu klasu generatora, prepoznavača i prevodilaca jezika, posebno pogodnih za implementaciju na računalima. Teorija je konačnih automata koristan instrument za razvoj sustava s konačnim brojem stanja čije mnogobrojne primjene nalazimo i u informatici. Programi, kao što je npr. tekstovni editor, često su načinjeni kao sustavi s konačnim brojem stanja. Na primjer, računalo se zasebno također može promatrati kao sustav s konačno mnogo stanja. Upravljačka jedinica, memorija i vanjska memorija nalaze se teoretski u svakom trenutku u jednom od vrlo velikog broja stanja, ali još uvijek u konačnom skupu stanja. Iz svakodnevnog je života upravljački mehanizam dizala još jedan dobar primjer sustava s konačno mnogo stanja. Prirodnost koncepta sustava s konačno mnogo stanja je razlog primjene tih sustava u različitim područjima, pa je i to vjerojatno najvažniji razlog njihova proučavanja.

Opći model automata dan je na sl. 0.1. Automat koji sadrži sve navedene "dijelove" naziva se pretvarač, automat bez ulazne trake je generator, a automat bez izlazne trake je prepoznavač.



Sl. 0.1 – *Opći model automata.*

Automati najčešće imaju ulogu prepoznavača. Ovisno o jeziku koji se prepoznaje, odnosno o tipu gramatike koja generira takav jezik, postoje i vrste prepoznavača dane na sljedećem crtežu.



Sl. 0.2 - *Chomskyjeva hijerarhija gramatika, njihovi odgovarajući jezici i prepoznavači.*

Konačni automat

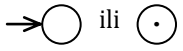
Konačni automat nema pomoćne memorije. To je matematički model sustava koji se nalazi u jednom od mnogih konačnih stanja. Stanje sustava sadrži obavijesti koje su dobivene na temelju dotadašnjih podataka i koje su potrebne da bi se odredila reakcija sustava iz idućih podataka. Drugim riječima, radi se o prijelaznim stanjima koje izvodi dani ulazni znak iz danog alfabeta Σ pod utjecajem funkcije prijelaza. Svaki se ulazni znak može nalaziti samo u jednom prijelaznom stanju, pri čemu je dopušten povratak na prethodno stanje.

Konačni automat je uređena petorka $\mathcal{M}=(Q, \Sigma, \delta, q_0, F)$, gdje su:

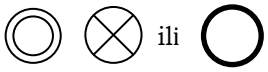
- Q konačni skup stanja
- Σ alfabet
- δ funkcija prijelaza, definirana kao $\delta: Q\times\Sigma\rightarrow\mathcal{P}(Q)$ gdje je $\mathcal{P}(Q)$ particija od Q
- q_0 početno stanje, $q_0\in Q$
- F skup završnih stanja, $F\subseteq Q$

DIJAGRAM PRIJELAZA

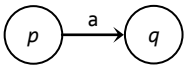
Iz definicije funkcije prijelaza zaključujemo da je to označeni, usmjereni graf čiji čvorovi odgovaraju stanjima automata, a grane su označene znakovima iz alfabeta. Ako, dakle, funkciju prijelaza prikažemo kao graf, dobivamo dijagram prijelaza. Označimo li u tom dijagramu početno stanje i skup završnih stanja, dobivamo konačni automat prikazan grafički. Početno ćemo stanje označivati s:



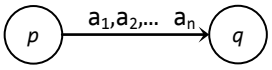
a završno s:



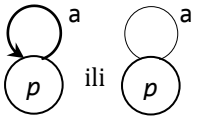
Ako je $p=\delta(q,a)$, tada postoji prijelaz iz stanja q u stanje p pa će grana u dijagramu prijelaza koja spaja q i p, s početkom u q i završetkom u p, biti označena s a:



Ako postoji više grana s početkom u q i završetkom u p, tj. ako je $p=\delta(q,a_1)=...=\delta(q,a_n)$, to će biti prikazano s:



Povratak nekim prijelazom a u isto stanje, tj. ako je $p=\delta(p,a)$, dijagram prijelaza je:



TABLICA PRIJELAZA

Funkcija prijelaza može se prikazati tablično. Tada se kaže da je to tablica prijelaza. Redovi tablice predstavljaju stanja, a stupci su označeni znakovima iz alfabeta i predstavljaju prijelaze. Na mjestu u redu označenom s q, $q\in Q$, i stupcu označenom s x, $x\in\Sigma$, upisan je skup narednih stanja (bez vitičastih zagrada) ako je funkcija $\delta(q, x)$ definirana, odnosno nije ništa upisano ako $\delta(q, x)$ nije definirano. Početno ćemo stanje označiti s → ili >, a konačno s ⊗ ili *.

DETERMINISTIČKI I NEDETERMINISTIČKI AUTOMAT

Iz definicije funkcije prijelaza vidimo da je moguć prijelaz iz tekućeg stanja u više narednih stanja s istim prijelazom a, $a\in\Sigma$, tj. da je ponašanje automata općenito nedeterminističko. Neka je $\mathcal{M}=(Q, \Sigma, \delta, q_0, F)$ konačni automat. Kažemo da je automat deterministički ako za sve $a\in\Sigma$ i sva stanja q, q' i q'' iz F i

$$\delta(q, a) = \{q', q''\}$$

slijedi da je $q'=q''$. Ako postoji barem jedan $a\in\Sigma$ tako da je $q'\neq q''$, automat je nedeterministički.

Stogovni automat

Stogovni automat PDA (Push Down Automaton) jest uređena sedmorka, $\mathcal{P}=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Γ alfabet znakova stoga (potisne liste)
- δ funkcija prijelaza, definirana kao $\delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma \rightarrow Q\times\Gamma^*$

q_0 početno stanje, $q_0 \in Q$
Z_0 početni znak stoga (potisne liste), $Z_0 \in \Gamma$
F skup završnih stanja, $F \subseteq Q$

Dvostruko-stogovni automat

Dvostruko-stogovni automat definiran je kao uređena sedmorka

$$Pt = (Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, s, F)$$

gdje su:

Q konačan skup stanja (kontrole konačnog stanja)
Σ ulazni alfabet
Γ_1, Γ_2 alfabet prvog i drugog stoga
Δ funkcija prijelaza, definirana kao $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma_1 \times \Gamma_2 \rightarrow Q \times \Gamma_1^* \times \Gamma_2^*$
s početno stanje, $s \in Q$
F skup konačnih stanja, $F \subseteq Q$

Vidimo da se ovaj automat razlikuje od stogovnog automata u definiciji funkcije prijelaza Δ koja koristi dva stoga kao pomoćnu memoriju. Kaže se da je Pt linearno-ograničen jer je duljina oba stoga linearno proporcionalna duljini generiranog niza.

0.5 PARSIRANJE

Jezik generiran gramatikom jest

$$L(G) = \{w \in T^*: S \Rightarrow^* w\}$$

a to je skup rečenica w dobivenih svim mogućim izvođenjima iz S. U praksi, poslije izvođenja gramatike nekog jezika i njezine verifikacije, dalje će njezina uporaba biti u provjeri je li dani niz w rečenica jezika $L(G)$. Tada se problem svodi na nalaženje niza izvođenja, počevši od S, koji bi rezultirao tim nizom (rečenicom). Takav se postupak sintaksne analize naziva parsiranje. Ovdje treba napomenuti da je termin "parsiranje" izravni kalk iz engleskog "parsing". Primjerenije bi bilo "posuditi" ga iz latinskog, pa je to parsanje, kako je prije nekoliko godina predložio prof. dr. sc. Marko Tadić. No, s obzirom na to da smo u prethodnoj knjizi rabili termin "parsiranje", zadržat ćemo ga i ovdje. Ustrojbu postupka parsiranja na računalu (program u izabranom jeziku za programiranje) nazivat ćemo parser. Stablo izvođenja dobiveno iz niza izvođenja $S \Rightarrow^* w$ sada ćemo zvati stablo sintaksne analize.

Lijevo i desno parsiranje

Neka je $G=(N,T,P,S)$ beskontekstna gramatika u kojoj su produkcije iz P označene (numerirane) s 1,2,...,p i neka je $\alpha \in (N \cup T)^*$. Tada je

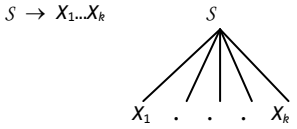
(1) lijevo parsiranje za α niz produkcija rabljenih u krajnjem izvođenju slijeva krenuvši od S: $S \Rightarrow_{lm}^* \alpha$
(2) desno parsiranje za α reverzni niz produkcija rabljenih u krajnjem izvođenju zdesna krenuvši od S: $S \Rightarrow_{rm}^* \alpha$

Ako i predstavlja i-tu produkciju, pisat ćemo $\alpha \Rightarrow_i \beta$ ako je $\alpha \Rightarrow_{lm} \beta$, odnosno $\alpha \Rightarrow_i \beta$ ako je $\alpha \Rightarrow_{rm} \beta$. Također ćemo pisati $\alpha \Rightarrow_{i_1 i_2} \gamma$ ako je $\alpha \Rightarrow_{i_1} \beta$ i $\beta \Rightarrow_{i_2} \gamma$, odnosno $\alpha \Rightarrow_{i_1 i_2} \gamma$ ako je $\alpha \Rightarrow_{i_1} \beta$ i $\beta \Rightarrow_{i_2} \gamma$.

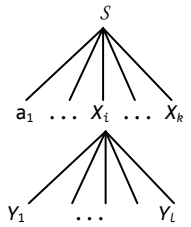
Lijevo i desno parsiranje primjenljivi su samo nad beskontekstnim (i linearnim) jezicima. U lijevom parsiranju stablo sintaksne analize izvodi se s vrha ili silazno, od korijena prema listovima (top-down). U desnom parsiranju stablo sintaksne analize izvodi se odozdo prema vrhu ili uzlazno, od listova prema korijenu (bottom-up).

Silazna sintaksna analiza

Znajući lijevo parsiranje $\pi = i_1 i_2 \ldots i_n$ rečenice $w = a_1 a_2 \ldots a_n$ iz $L(G)$ može se izvesti stablo parsiranja (sintaksne analize) u značenju "s vrha" ili silazno, pa kažemo da je to silazna sintaksna analiza. Počinje se s korijenom stabla, označenim sa S. Parsiranje i_1 daje produkciju koja je upotrijebljena u ekspanziji stabla. Pretpostavimo da i_1 označuje produkciju $S \rightarrow X_1 \ldots X_k$ pa se može kreirati k sljedbenika iz čvora S koji su označeni s $X_1, X_2, \ldots, X_k$



Ako su $X_1, X_2, \ldots, X_{i-1}$ terminali, tada prvih $i-1$ simbola (znakova) ulaznog niza w moraju biti $X_1 X_2 \ldots X_{i-1}$. Produkcija i_2 mora biti oblika $X_i \rightarrow Y_1 \ldots Y_l$ pa se nastavlja ekspanzija stabla iz čvora X_i:



Postupak se nastavlja ekspanzijom na opisani način sve dok se ne izgradi stablo čiji će li-stovi biti znakovi niza w. Na primjer, neka je G_E gramatika s numeriranim produkcijama

(1) $E \rightarrow E+T$ (2) $E \rightarrow T$ (3) $T \rightarrow T*F$ (4) $T \rightarrow F$ (5) $F \rightarrow (E)$ (6) $F \rightarrow a$

Lijevo parsiranje rečenice a*(a+a) je 23465124646, što je dobiveno iz niza izvođenja slijeva:

$E \Rightarrow_2 T \Rightarrow_3 T*F \Rightarrow_4 F*F \Rightarrow_6 a*F \Rightarrow_5 a*(E) \Rightarrow_1 a*(E+T) \Rightarrow_2 a*(T+T) \Rightarrow_4 a*(F+T) \Rightarrow_6 a*(a+T) \Rightarrow_4 a*(a+F) \Rightarrow_6 a*(a+a)$

što se može napisati kao

$E \Rightarrow_{23465124646} a*(a+a)$

Uzlazna sintaksna analiza

Analizirajmo sada desno parsiranje. Ako pogledamo krajnje desno izvođenje rečenice a*(a+a) krenuvši od startnog simbola E, gramatike G_E iz prethodnog primjera, imamo:

$$E \Rightarrow_2 T \Rightarrow_3 T*F \Rightarrow_5 T*(E) \Rightarrow_1 T*(E+T) \Rightarrow_4 T*(E+F) \Rightarrow_6 T*(E+a) \Rightarrow_2 T*(T+a)$$
$$\Rightarrow_4 T*(F+a) \Rightarrow_6 T*(a+a) \Rightarrow_4 F*(a+a) \Rightarrow_6 a*(a+a)$$

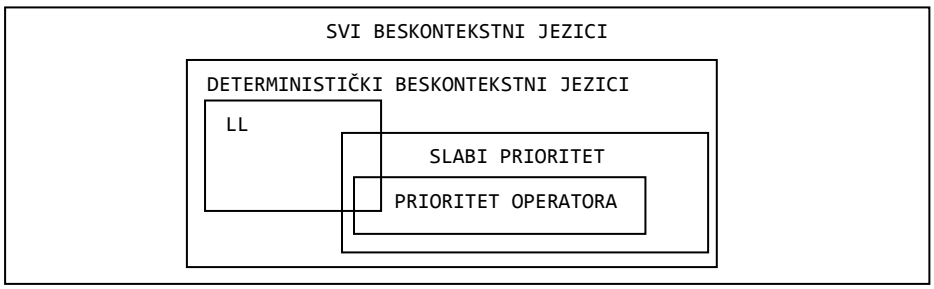
Pišući niz upotrijebljenih produkcija u rečeničnim formama, u obrnutom redoslijedu, imamo desno parsiranje 64642641532, pa zaključujemo da je općenito desno parsiranje niza w u gramatici $G=(N,T,P,S)$ niz produkcija koje su bile upotrijebljene za reduciranje (pretvorbu) rečenice w u startni simbol, u našem primjeru E. Ako to promatramo kao stablo sintaksne analize, zaključujemo da se desno parsiranje rečenice $w=a_1...a_n$ može predočiti kao izvođenje stabla od listova, znakova $a_1,...,a_n$, prema korijenu, startnom simbolu S. Otud i naziv uzlazna sintaksna analiza ili uzlazno parsiranje (eng. "bottom-up" – odozdo prema vrhu, uzlazno).

Hijerarhija beskontekstnih jezika

Osim toga što su postupci sintaksne analize silazni ili uzlazni, općenito se mogu podijeliti na: višeprolazne (nedeterminističke) i jednoprolazne (determinističke).

Višeprolazni postupci sintaksne analize, gdje se "višeprolaznost" odnosi na višestruko pretraživanje ulaznog niza, kao što ćemo pokazati, općeniti su, univerzalni postupci sintaksne analize beskontekstnih jezika, primjenljivi nad cijelom klasom beskontekstnih jezika, ali su najčešće prilično neefikasni. Glavni im je nedostatak vrijeme trajanja (ili broj koraka), posebno ako ulazni niz nije u jeziku.

Jednoprolazni su postupci definirani samo za određene klase beskontekstnih jezika, sl. 0.3. To su jezici (gramatike) tipa LL(k) i LR(k), za koje je moguće konstruirati jednoprolazni postupak sintaksne analize slijeva (za LL) ili zdesna (za LR), koji će raditi deterministički ako im se dopusti da "pogledaju" najviše k ulaznih znakova slijeva nadesno (prvo slovo L u LL i LR to naznačuje) od neke tekuće pozicije, te gramatike s prioritetom operatora za koje je moguće napisati deterministički postupak sintaksne analize upravljan tablicom prioriteta relacije. Osnovni nedostatak jednoprolaznih postupaka sintaksne analize jest ograničena primjenljivost, nad relativno malom klasom beskontekstnih jezika.



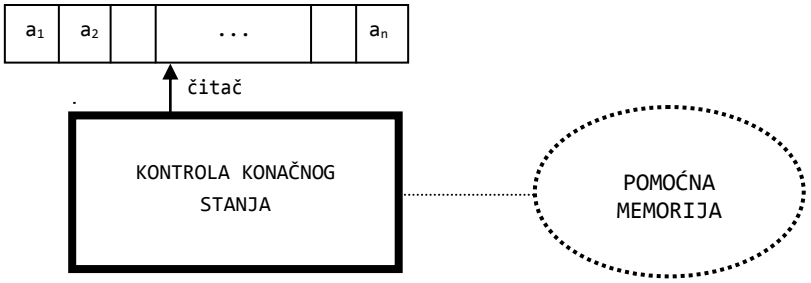
Sl. 0.3 – *Hijerarhija beskontekstnih jezika.*

U [Dov2012b], u poglavlju "Višeprolazna sintaksna analiza" opisane su dvije povratne ("backtrack") metode nedeterminističke sintaksne analize slijeva i zdesna koje se, uz određene

transformacije gramatika, mogu primijeniti na cijeloj klasi beskontekstnih jezika. U poglavlju "Tablični postupci sintaksne analize" iste knjige opisana su još dva općenita postupka sintaksne analize. Jednoprolazna sintaksna analiza opisana je u poglavljima "LL(k) jezici i sintaksna analiza" i "LR(k) jezici i sintaksna analiza".

0.6 PREPOZNAVANJE

Automati su drugi formalizam za generiranje jezika. Time smo se bavili u prvoj knjizi, [Dov2012a]. No, kao i gramatike, tako su i automati češće u ulozi prepoznavanja rečenica danog jezika. Kažemo da je to prepoznavač jezika. Čine ga ulazna traka, čitač, kontrola konačnog stanja i pomoćna memorija, sl. 0.4.



Sl. 0.4 – *Opći model prepoznavača.*

Ulazna traka se može promatrati kao da se sastoji od linearnog niza ćelija koje sadrže po jedan simbol (znak) alfabeta jezika koji se prepoznaje. Početak i kraj mogu biti označeni posebnim znakovima (markerima) koji nisu u alfabetu. Najčešće se marker nalazi na kraju ulaznog niza.

Čitač (ili "ulazna glava") čita jedan znak s pozicije na kojoj se nalazi. Može se pomicati lijevo, desno za jedno mjesto, ili ostati stacioniran. Prepoznavač koji nikad ne može pomaknuti čitač ulijevo naziva se *jednosmjerni* prepoznavač.

Uobičajeno je da se ulazna traka promatra kao traka u kojoj se ne mijenja sadržaj (*read-only*). Međutim, postoje prepoznavači (Turingov stroj) u kojima je dopuštena promjena sadržaja ulazne trake pa se u tom slučaju kaže da je to ulazno-izlazna traka, a ulazna glava je čitač - pisač.

Kontrola konačnog stanja, ili kraće samo kontrola, glavni je dio ili srce prepoznavača. Općenito se sastoji od pet komponenti $(Q, \Sigma, \delta, q_0, F)$, gdje su:

- Q konačni skup stanja
- Σ alfabet
- δ funkcija prijelaza
- q_0 početno stanje, $q_0 \in Q$
- F skup završnih stanja, $F \subseteq Q$

Pomoćna memorija, ili kraće samo memorija, prepoznavača može sadržavati podatke bilo kojeg tipa. Pretpostavlja se da postoji konačni *alfabet memorije*, označen s Γ, i da podaci memorije sadrže znakove alfabeta organiziranih u nekoj strukturi podataka, na primjer mogu biti sa strukturom stoga, $Z_1Z_2...Z_n$, gdje je svaki $Z_i \in \Gamma$ i Z_1 je na vrhu.

Za rad s memorijom u većini prepoznavača koriste se dvije funkcije – *funkcija pohranjivanja* i *funkcija dohvata* podataka. Pretpostavlja se da je funkcija za dohvat podataka preslikavanje iz skupa svih mogućih konfiguracija memorije u konačni skup informacijskih simbola, koji mogu biti jednaki simbolima alfabeta memorije.

Funkcija pohranjivanja jest preslikavanje koje opisuje način promjene sadržaja memorije, tj. preslikava tekući sadržaj memorije i *kontrolni niz* u memoriju. Ako pretpostavimo da memorija ima strukturu stoga, funkcija pohranjivanja g u tom slučaju može biti definirana tako da zamjenjuje simbol Z_1 na vrhu stoga kontrolnim nizom $Y_1...Y_k$, pa se funkcija g može definirati kao

$$g: \Gamma^+ \times \Gamma^* \rightarrow \Gamma^*$$

tako da je

$$g(Z_1Z_2...Z_n, Y_1...Y_k) = Y_1...Y_kZ_2...Z_n$$

Ako je kontrolni niz prazan, vrh će stoga Z_1 biti zamijenjen s ε, što je ekvivalentno operaciji *pop* ("pucanje") nad stogom. Poslije toga je simbol Z_2 na vrhu stoga.

Kontrola se može zamisliti kao program koji opisuje ponašanje prepoznavača. Predstavljena je konačnim skupom stanja zajedno s preslikavanjem (funkcijom prijelaza) koje opisuje kako se mijenjaju stanja ovisno o tekućem stanju, tekućem simbolu (znaku) ulaznog niza i tekućoj informaciji dohvaćenoj iz pomoćne memorije. Kontrola također odlučuje u kojem će se slučaju ulazna glava pomaknuti i koja će informacija biti pohranjena u memoriju.

Prepoznavač analizira (prepoznaje) ulazni niz čineći niz *pomaka*, mijenjajući svoje konfiguracije, od početnog stanja i početne konfiguracije do dosezanja konačnog stanja i konačne konfiguracije. Konfiguracija prepoznavača jest njegova "slika" koja se općenito sastoji od

(1) stanja kontrole,
(2) sadržaja ulazne trake, zajedno s pozicijom čitača i
(3) sadržaja memorije.

Početna konfiguracija prepoznavača jest ona u kojoj je kontrola u početnom stanju, čitač je pozicioniran na prvi znak ulaznog niza i memorija ima početni sadržaj (može biti prazna ili, na primjer, sadržavati oznaku dna stoga).

Konačna konfiguracija prepoznavača jest ona u kojoj je kontrola u jednom od konačnih stanja, čitač je pozicioniran iza posljednjeg znaka ulaznog niza (na markeru kraja, ako ga ima). Često sadržaj memorije dosezanjem konačne konfiguracije također mora zadovoljavati određene uvjete.

Reći ćemo da prepoznavač prihvaća ulazni niz *w* ako je, krenuvši od početne konfiguracije s *w* na ulaznoj traci, konačnim nizom pomaka dosegnuo jednu od konačnih konfiguracija. Pritom, općenito, prepoznavač može biti deterministički ili nedeterministički, što će ovisiti o definiciji njegove funkcije prijelaza.

Jezik definiran prepoznavačem jest skup prihvatljivih nizova. Prema Chomskyjevoj hijerarhiji za svaku klasu gramatika i jezika koje generiraju postoje ekvivalentni automati koji generiraju, a sada ćemo reći prepoznaju istu klasu jezika. To su konačni, stogovni i linearno ograničeni prepoznavač, te Turingov stroj. Dakle, vrijedi slijedeća klasifikacija prepoznavača ovisno o jeziku kojeg prepoznaju, pa sada možemo reći da je jezik:

(1) desno-linearan (tipa 3) ako i samo ako se može definirati jednosmjernim (determinističkim) konačnim automatom,
(2) beskontekstan (tipa 2) ako i samo ako se može definirati stogovnim (jednosmjerno nedeterminističkim) automatom,
(3) kontekstan (tipa 1) ako i samo ako se može definirati linearno ograničenim (dvosmjerno nedeterminističkim) automatom, i
(4) rekurzivno prebrojiv (tipa 0) ako i samo ako se može definirati Turingovim strojem.

Svaki od navedenih automata (prepoznavača) detaljno je obrađen u [Dov2012b].

0.7 KOMPJUTERI

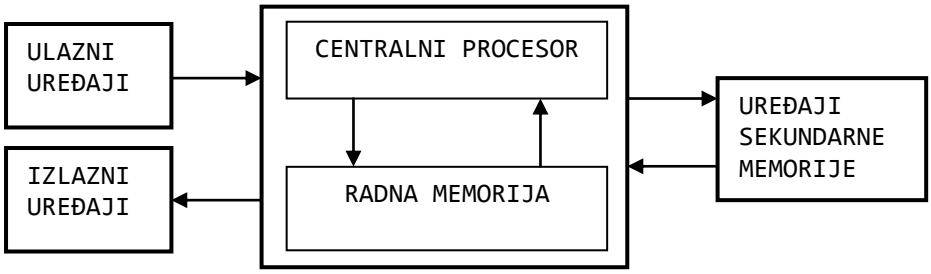
Kompjuter (engl. „computer“) je stroj koji može prihvatiti i upamtiti podatke i upute (instrukcije) kojima može izvršiti dug, složen i ponavljajući niz operacija s podacima, a rezultate računanja neposredno prikazati ili pohraniti za buduću uporabu. Svaki kompjuter sastoji se od dva bitna dijela: tehničkog i programskog.

Tehnički dio kompjutera ili hardver („hardware“) obuhvaća materijalne komponente. Sastoji se od niza povezanih uređaja čiji su rad i funkcije međusobno usklađeni. Programski dio kompjutera ili softver („software“) obuhvaća skupove instrukcija - programe za upravljanje i nadzor uređaja i za specificiranje operacija koje treba izvršiti s podacima.

Hardver

Kompjuter je stroj složene tehničke strukture, s brojem komponenata ovisnim o njegovoj namjeni. Osnovne komponente od kojih se sastoji svaki suvremeni kompjuter, što je pojednostavljeno prikazano na sljedećoj slici, jesu:

- centralni procesor
- radna memorija
- uređaji sekundarne memorije
- ulazno-izlazni uređaji



Sl. 0.5 – *Struktura kompjutera*.

Centralni procesor izvršava upravljačke zadatke i operacije s podacima. Sastoji se od:

- upravljačke jedinice
- aritmetičko-logičke jedinice

Upravljačka ili *kontrolna jedinica* nadgleda i usklađuje funkcioniranje svih komponenti i kompjutera kao cjeline. Pri izvršenju instrukcija programa ona uključuje u rad ostale jedinice i uređaje, u skladu s izraženim zahtjevima.

Aritmetičko-logička jedinica izvršava osnovne aritmetičke i logičke operacije s podacima: zbrajanje, oduzimanje, množenje, dijeljenje i usporedbu vrijednosti. Složenije operacije izražene u programu izvršavaju se prethodnim razlaganjem u fazi prevođenja programa na skup osnovnih aritmetičkih i logičkih operacija.

Centralni procesor sadrži još i nekoliko memorijskih registara neophodnih za smještaj podataka s kojima procesor trenutno operira.

U memoriju se pohranjuju podaci, instrukcije i tekstovi programa za tekuću ili buduću uporabu. Kompjuteri imaju dvije vrste memorija: radnu i sekundarnu.

Radna (glavna ili primarna) *memorija* dio je centralnog dijela kompjutera koji služi za pohranjivanje programskih instrukcija i podataka za tekuću uporabu.

Glede tehničke izvedbe radna se memorija sastoji od velikog broja ćelija – bistabilnih elektroničkih sklopova, čija je karakteristika da mogu biti u jednom od dva moguća stanja (+/- ili ima/nema, na primjer), koja označavamo brojevima 0 i 1. Prema tome, svaka ćelija predstavlja binarnu znamenku koju nazivamo bit (od riječi binary digit - binarna znamenka).

Bitovi se, radi predočavanja ostalih znamenaka, slova i drugih znakova, grupiraju u bajtove (byte - bajt). Jedan bajt sastoji se najčešće od 8 bitova, koji u kombinacijama nula i jedinica mogu predstaviti 2^8 (256) znakova.

Bajtovima se obično izražava kapacitet memorije i to kao potencija broja 2. Na primjer, $M=2^{14}$ bajtova, ili u obliku $M=Nx2^{10}$. Jedinica mjere memorijskog kapaciteta 1 KB (1 kilobajt), jednaka je upravo 2^{10}, odnosno 1024 bajtova. Bitna karakteristika radne memorije jest velika brzina pristupa svakoj adresi i velika brzina pisanja i čitanja podataka i in-

strukcija. Osim toga, svi podaci s kojima program radi pohranjeni su u radnoj memoriji i gube se nepovratno okončanjem programa (kao i sam izvršni program).

Želimo li sačuvati podatke i programe za neku buduću upotrebu, možemo ih pohraniti u *sekundarnoj* (vanjskoj, perifernoj ili pomoćnoj) *memoriji*. Sekundarnu memoriju sačinjavaju magnetski mediji (diskovi, USB stickovi, diskete, magnetske vrpce, kasete, itd), te posebni nosači podataka kao što su CD-i i DVD-i. Bitne karakteristike tih memorija su veliki kapacitet, desetine tisuća do nekoliko milijuna puta veći od kapaciteta primarne memorije, kao i trajno pamćenje pohranjenih podataka.

Ulazni i izlazni uređaji (neki od njih mogu biti i jedno i drugo) omogućuju komuniciranje s kompjuterom: unošenje, čitanje, upisivanje, ispisivanje i prikaz podataka i programa. Drugim riječima, oni omogućuju prijenos i razmjenu podataka i programa između vanjske okoline kompjutera. Ovdje se pod pojmom "ulaz" podrazumijeva unošenje ili čitanje podataka s ulaznog uređaja u primarnu ili sekundarnu memoriju, a "izlaz" znači iznošenje - ispis rezultata obrade iz memorije na izlazni uređaj i prikaz tih rezultata na nekom izlaznom mediju (ekran, papir itd).

Karakteristike današnjih kompjutera nadmašuju višestruko kompjutere od prije tridesetak godina. Na primjer, početkom 80-tih godina prošloga stoljeća radilo se na kompjuterima sa sljedećim karakteristikama:

- RAM 392 KB
- tvrdi disk, 4x32MB 128 MB
- multimedija nema
- procesor < 10 MHz, dimenzije: 1m x 2m x 1m, masa > 500 kg
- cijena > $ 30,000,000
- snaga pogona 185 KW
- ulazni uređaji čitač kartica, čitač magnetnih vrpci
- izlazni uređaji linijski printer, ploter, snimač magnetnih vrpci
- posada 12 ljudi

I da se malo našalimo: dani kompjuter nije bio na vodu (niti paru), ali nije radio ako bi nestalo vode! Razlog: dvorana u kojoj je bio smješten kompjuter (10x20m) bila je klimatizirana sa strogo kontroliranom temperaturom i vlažnošću za što je trebala voda!

Nećemo pokušati ni usporediti današnje kompjutere (ili mobitele) niti po karakteristikama, niti po cijeni. Pokušajte samo izračunati, na primjer, koliko bi vam trebalo tvrdih diskova od 32 MB, u obliku valjka promjera oko 0.5m i visine oko 0.2m, umjesto vašeg diska ili sticka!? Možda ćete vi pričati, za dvadesetak godina, o primitivnim današnjim kompjuterima, jer su imali samo 500 GB RAM-a itd.

Softver

Hardverski dio kompjutera može izvršiti ulazne, računske i izlazne operacije, ali koje i kojim redoslijedom određuje se softverskim dijelom. Na kompjuteru razlikujemo dvije vrste softvera:

- sistemski
- aplikacijski

Sistemski softver je skup programa za upravljanje i kontrolu rada uređaja, za opsluživanje korisnika i izvršenje njihovih programa. U gruboj podjeli sačinjavaju ga programi operacijskog sustava, uslužni programi i prevodioci.

Bitna komponenta sistemskog softvera je *operacijski sustav*. To je skup programskih modula pomoću kojih se, jednostavno rečeno: nadziru hardverski resursi (procesor, primarna i sekundarna memorija i ulazno-izlazni uređaji), rješavaju konflikti u zahtjevima za hardverskim resursima izraženi korisničkim programima, optimizira i usuglašava rad hardverskih uređaja u skladu s korisničkim zahtjevima i upravlja komuniciranjem korisnika s kompjuterom i hardverskim uređajima pri izvršenju korisničkih programa. Operacijski se sustav, dakle, ponaša kao posrednik između korisničkih programa i hardvera. Od mnogih operacijskih sustava danas su u široj upotrebi Microsoft Windowsi, UNIX i LINUX.

Uslužni programi omogućuju sortiranje podataka, prijenos podataka i programa s jednog medija na drugi, upisivanje podataka i programa, vođenje statistike i evidencije o radu kompjutera i sl. Osnovne su im karakteristike otvorenost (moguće je proširenje novim programima), relativna neovisnost o operativnom sustavu i smještaj na sekundarnoj memoriji.

Prevodioci su programi koji "razumiju" jezike za programiranje i korisničke programe, napisane u tim jezicima, prevode u programe strojnog ili nekog drugog jezika za programiranje.

Sistemski softver na suvremenim kompjuterima obuhvaća i sustave za upravljanje bazama podataka, koji omogućuju definiranje složenih struktura podataka i upravljanje pristupom podacima ovisno o potrebama korisnika.

Aplikacijski softver je skup programa kojima korisnici izražavaju zahtjeve za informacijama ili podacima ili rješavaju različite vrste problema. Neke od tih programa napisali su sami korisnici, drugi postoje kao gotovi programski paketi. Spomenimo samo neke vrste aplikacijskih programskih paketa:

- poslovni informacijski sustavi
- programski paketi za upravljanje komunikacijskim mrežama
- sustavi za projektiranje i proizvodnju podržani kompjuterom, CAD/CAM
- znalački (ekspertni) sustavi specifične namjene
- paketi za interaktivnu grafiku
- generatori aplikacija
- biblioteke znanstvenih programa itd.

1.
UVOD U TEORIJU PREVOĐENJA

— Marinette

Kad sam pojurio otpjevati
svoju malu pjesmu Marineti,
Ljepotica, izdajnica jedna,
već je otišla u Operu!

Sa svojom malom pjesmom,
izgledao sam k'o glupan,
majko moja,
Sa svojom malom pjesmom,
izgledao sam k'o glupan!

Kad sam požurio odnijeti
svoju čašu sa senfom Marineti,
Ljepotica, izdajnica jedna,
već je večerala!

Sa svojom malom čašom,
izgledao sam k'o glupan,
majko moja,
Sa svojom malom čašom,
izgledao sam k'o glupan!

Kad sam poklonio bicikl
za novogodišnji dar Marineti,
Ljepotica, izdajnica jedna,
već je kupila auto!

Sa svojim malim biciklom,
izgledao sam k'o glupan,
majko moja,
Sa svojim malim biciklom,
izgledao sam k'o glupan!

Kad sam pojurio sav uspaljen
na spoj s Marinetom,
Ljepotica je rekla: "Obožavam te",
jednom ljigavom tipu
koji ju je ljubio!

Sa svojim buketom cvijeća,
izgledao sam k'o glupan,
majko moja,
Sa svojim buketom cvijeća,
izgledao sam k'o glupan!

Kad sam pojurio
ustrijeliti Marinetu,
Ljepotica, već je bila mrtva
od posljedica prehlade!

Sa svojim revolverom,
izgledao sam k'o glupan,
majko moja,
Sa svojim revolverom,
izgledao sam k'o glupan!

Kad sam pojurio zloslutan
na Marinetin pogreb,
Ljepotica, izdajnica jedna,
već je bila uskrsla!

Sa svojim malim vijencem,
izgledao sam k'o glupan,
majko moja,
Sa svojim malim vijencem,
izgledao sam k'o glupan!

Marinette

(Georges Brassens/
Zdravko DOVEDAN HAN)

Prevođenje je skup parova nizova. U ovom su poglavlju dana dva temeljna postupka ("formalizma") prevođenja. Prvi je "shema prevođenja", a to je gramatika s mehanizmom za produciranje izlaza za svaku generiranu rečenicu jezika. Shema prevođenja primijenjena je na beskontekstne gramatike. Drugi je postupak "pretvarač", odnosno prepoznavač (automat) koji može emitirati niz izlaznih simbola konačne duljine u svakom svom pomaku. Opisani su konačni i stogovni pretvarači.

1.1 PREVOĐENJE I SEMANTIKA

U prvoj smo knjizi promatrali sintaksni aspekt (beskontekstnog) jezika. Dali smo nekoliko postupaka provjere korektno napisanih rečenica jezika (sintaksna analiza, upravljana gramatikom jezika, ili prepoznavač). Ovdje će biti pokazana tehnika pridruživanja drugog niza svakoj rečenici jezika koji će biti "izlaz" te rečenice. Izraz "semantika" katkad ćemo koristiti da bismo naznačili vezu izlazne i ulazne rečenice u kojoj izlazna rečenica definira "značenje" ulazne rečenice.

♦ Prevođenje

Neka je Σ ulazni alfabet, a Δ izlazni alfabet. Prevođenje iz jezika $L_1 \subseteq \Sigma^*$ u jezik $L_2 \subseteq \Delta^*$ je relacija T iz $\Sigma^* \times \Delta^*$ tako da je L_1 domena, L_2 kodomena od T. Rečenica y, tako da je (x, y) u T naziva se izlaz za x.

Općenito, jedan ulazni niz može biti preveden u nekoliko izlaznih nizova. Dakle, prevođenje nije uvijek funkcija. Postoji mnogo primjera prevođenja.

♣ Primjer 1.1

Pretpostavimo da ulazni alfabet Σ sadrži brojke `0` do `9`, te mala i velika slova engleskog alfabeta, a izlazni alfabet Δ brojke i samo velika slova engleskog alfabeta. Možemo definirati prevodjenje $p(x)=x$, ako je x brojka ili veliko slovo engleskog alfabeta; $p(x)$ je veliko slovo engleskog alfabeta jednako x, ako je x malo slovo engleskog alfabeta. Na primjer, ako je x = `1a2b3C`, $p(x)$ = `1A2B3C`.

♣ Primjer 1.2

Pretpostavimo da želimo zamijeniti svako pojavljivanje brojki u rečenicama iz Σ^* ispisujući ih hrvatski. Možemo definirati prevođenje $t(x)=x$, ako x nije brojka; t(`0`)=`nula`, t(`1`)=`jedan`, ..., t(`9`)=`devet`. Na primjer, ako je x = `1 plus 2 = 3`, $t(x)$ = `jedan plus dva = tri`.

Jedan vrlo važan primjer prevođenja jest preslikavanje aritmetičkih izraza napisanih u uobičajenoj ("infiksnoj") notaciji u ekvivalentan izraz napisan u poljskoj notaciji. Najprije dajemo definiciju poljske notacije.

♦ Poljska notacija

Neka je Θ (theta) skup binarnih operatora, na primjer {+, *}, i neka je Σ skup operanada. Dva oblika poljske notacije, *prefiksna poljska notacija* i *postfiksna poljska notacija* (naziv "poljska" dolazi od poljskog matematičara Lukasiewicza koji ju je prvi opisao) definirana su rekurzivno na sljedeći način:

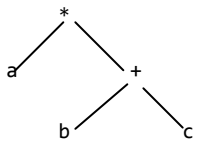
1) Ako infiksni izraz E sadrži samo jedan operand a, $a \in \Sigma$, tada su i prefiksna i postfiksna poljska notacija od E jednake a.
2) Ako je $E_1 \theta E_2$ infiksni izraz, gdje je θ operator, a E_1 i E_2 infiksni izrazi, operandi od θ, tada je:
   a) $\theta E'_1 E'_2$ prefiksna poljska notacija od $E_1 \theta E_2$, gdje su E'_1 i E'_2 prefiksne poljske notacije od E_1 i E_2, respektivno, i
   b) $E''_1 E''_2 \theta$ postfiksna poljska notacija od $E_1 \theta E_2$, gdje su E''_1 i E''_2 postfiksne poljske notacije od E_1 i E_2, respektivno.
3) Ako je (E) infiksni izraz, tada je i prefiksna i postfiksna poljska notacija jednaka E.

♣ Primjer 1.3

Dan je infiksni izraz a*(b+c). To je izraz oblika E_1*E_2, gdje je E_1=a, E_2=(b+c). Prefiksna i postfiksna notacija od E1 jednaka je a. Prefiksna notacija od E_2 jednaka je kao i za b+c, a to je +bc. Dakle, prefiksna notacija izraza a*(b+c) jednaka je *a+bc. Slično, postfiksna notacija od b+c, jednaka je bc+, pa je postfiksna notacija izraza a*(b+c) jednaka abc+*.

Primijetimo da će općenito prevođenje izraza napisanog u infiksnoj notaciji u prefiksnu ili postfiksnu biti nejednoznačno. Na primjer, izraz a+b*c može biti shvaćen kao E_1+E_2, gdje je E_1=a, a E_2=b*c, ili kao E_1*E_2, gdje je sada E_1=a+b, a E_2=c. U prvom bi slučaju dani izraz u prefiksnoj notaciji bio *+abc, a u drugom +a*bc. I u posfiksnoj notaciji imali bismo dva izlazna niza: ab+c* i abc*+. U praksi, ako je + operacija zbrajanja, a * operacija množenja, s obzirom na to da operacija množenja ima prioritet izvršavanja u odnosu na zbrajanje, prihvatljiva je druga interpretacija, u oba slučaja: +a*bc u prefiksnoj i abc*+ u postfiksnoj notaciji.

Aritmetički se izraz može prikazati i kao stablo. Čvorovi stabla bit će označeni operatorima iz Θ, a listovi operandima iz Σ. Na primjer, izraz a*(b+c) može se predstaviti stablom:



S obzirom na važnost prefiksne i postfiksne notacije u prevođenju, u nastavku dajemo algoritme za prevođenje izraza iz infiksne u prefiksnu i postfiksnu notaciju vodeći računa o prioritetu operatora.

♥ Algoritam 1.1 *Prevođenje izraza iz infiksne u prefiksnu notaciju*

1) Ulaz: Niz simbola izraza u infiksnoj notaciji, $x = s_1 s_2 \ldots s_n$, i dvije potisne liste, L_1 i L_2, s prefiksima kao vrhovima.
2) Izlaz: Niz simbola izraza x u prefiksnoj notaciji, $y = p_1 p_2 \ldots p_k$.
3) Postupak:
   a) Analizirati ulazni niz x od posljednjeg prema prvom znaku. Tekući znak jest s_i, $i=n, n-1, \ldots, 1$.

b) Ako je s_i

1) *operand*, upisati ga u L_2

2) *zatvorena zagrada*, upisati je u L_1

3) *operator*

a) potisna lista L_1 je prazna ili je na njezinom vrhu zatvorena zagrada, upisati s_i u listu L_1

b) na vrhu liste je operator većeg ili jednakog prioriteta, prenijeti simbol s vrha liste L_1 na vrh liste L_2, potom upisati s_i u listu L_1

4) *otvorena zagrada*, prenijeti sve simbole s vrha liste L_1 do zatvorene zagrade u listu L_2, potom brisati zatvorenu zagradu s vrha liste L_1

c) `i=i-1`. Ako je `i>0`, idi na korak b). Inače, prekinuti postupak i simbole s liste L_1 prenijeti u listu L_2. Potisna lista L_2 sadrži izraz u prefiksnoj notaciji.

♣ Primjer 1.4

Neka je dan izraz:

(	1	+	2	)	*	3
1	2	3	4	5	6	7

Postupak njegova prevođenja u prefiksnu notaciju dan je u sljedećoj tablici:

i	s_i	L_1	L_2
7	3		3
6	*	*	3
5	)	) *	3
4	2	) *	2 3
3	+	+) *	2 3
2	1	+) *	1 2 3
1	(	*	+ 1 2 3
0			* + 1 2 3

Dakle, izraz `(1+2)*3` u infiksnoj notaciji može se predstaviti kao `*+123` u prefiksnoj notaciji.

♣ Primjer 1.5

Implementacija algoritma 1.1 programom napisanim u Pythonu dana je u dijelu *PROGRAMI* ovoga poglavlja. Evo nekoliko primjera ulaznih nizova (izraza u infiksnoj notaciji) i njihovih prijevoda u prefiksnu notaciju dobivenih izvršenjem programa:

Infiks	*Prefiks*
1*2+3*4+5*6+7*8	+++*12*34*56*78
(1+2)*(3+4)	*+12+34
(1+2)*3+(4+5)*6	+*+123*+456
1+2*(3+4)*5	+1**2+345
(((1+2)*3+(4+5)*6+(7+8)*9)*2+3)*2	*+*++*+123*+456*+789232
(((1+2)*3+4)*5+6)*7	*+*+*+1234567
(5-6)*3+(1-9)*9	+*-563*-199
(1+2+3)*4+(5+6*7)*8	+*++1234*+5*678
(1+2+3+4)*5	*+++12345
1*(2+3)+4*(5+6)+7*(8+9)	++*1+23*4+56*7+89

♥ Algoritam 1.2 *Prevođenje izraza iz infiksne u postfiksnu notaciju*

1) Ulaz: Niz simbola izraza u infiksnoj notaciji, $x = s_1 s_2 \ldots s_n$, i dvije potisne liste, L_1 i L_2, sa sufiksima kao vrhovima.
2) Izlaz: Niz simbola izraza x u postfiksnoj notaciji, $y = p_1 p_2 .. p_k$.
3) Postupak:
   a) Analizirati ulazni niz x od prvog prema posljednjem znaku. Tekući znak jest s_i, i=1, ..., n-1, n.
   b) Ako je s_i
      1) *operand*, upisati ga u L_2
      2) *zatvorena zagrada*, upisati je u L_1
      3) *operator*
         a) potisna lista L_1 je prazna ili je na njezinom vrhu zatvorena zagrada, upisati s_i u listu L_1
         b) na vrhu liste je operator većeg ili jednakog prioriteta, prenijeti simbol s vrha liste L_1 na vrh liste L_2, potom upisati s_i u listu L_1
      4) *otvorena zagrada*, prenijeti sve simbole s vrha liste L_1 do zatvorene zagrade u listu L_2, potom brisati zatvorenu zagradu s vrha liste L_1
   c) $i=i+1$. Ako je $i<n+1$, idi na korak b). Inače, prekinuti postupak i simbole s liste L_1 prenijeti u listu L_2. Potisna lista L_2 sadrži izraz u postfiksnoj notaciji.

♣ Primjer 1.6

Postupak prevođenja izraza iz primjera 1.4:

(	1	+	2	)	*	3
1	2	3	4	5	6	7

u postfiksnu notaciju dan je u sljedećoj tablici:

i	s_i	L_1	L_2
1	(	(	
2	1	(	1
3	+	(+	1
4	2	(+	1 2
5	)	*	1 2 +
6	*	*	1 2 +
7	3	*	1 2 + 3
8			1 2 + 3 *

Dakle, izraz (1+2)*3 u infiksnoj notaciji može se predstaviti kao 12+3* u postfiksnoj notaciji.

♣ Primjer 1.7

Implementacija algoritma 1.2 programom napisanim u Pythonu dana je u dijelu *PROGRAMI* ovoga poglavlja. Evo nekoliko primjera ulaznih nizova (izraza u infiksnoj notaciji) i njihovih prijevoda u postfiksnu notaciju dobivenih izvršenjem programa:

Infiks	*Postfiks*
1*2+3*4+5*6+7*8	12*34*56*78*+++
(1+2)*(3+4)	12+34+*

```
(1+2)*3+(4+5)*6                          12+3*45+6*+
1+2*(3+4)*5                              1234+*5*+
(((1+2)*3+(4+5)*6+(7+8)*9)*2+3)*2        12+3*45+6*78+9*++2*3+2*
(((1+2)*3+4)*5+6)*7                      12+3*4+5*6+7*
(5-6)*3+(1-9)*9                          56-3*19-9*+
(1+2+3)*4+(5+6*7)*8                      12+3+4*567*+8*+
(1+2+3+4)*5                              12+3+4+5*
1*(2+3)+4*(5+6)+7*(8+9)                  123+*456+*789+*++
```

1.2 SINTAKSNO-UPRAVLJANO PREVOĐENJE

Dva važna primjera prevođenja jesu skupovi parova (x, y) gdje je x izraz u infiksnoj notaciji, a y izraz u prefiksnoj (ili postfiksnoj) poljskoj notaciji. Radi toga trebamo formalizam za njihovu potpunu specifikaciju. Problem konačnog opisa beskonačnog prevođenja sličan je problemu opisa beskonačnog jezika. Postoji nekoliko mogućih pristupa u rješavanju tog problema.

Analogno generatorima jezika, kao što je gramatika, možemo imati sustav koji generira parove u prevođenju. Također se može upotrijebiti prepoznavač s dvije trake za prepoznavanje takvih parova u prevođenju. Ili, može se definirati automat koji prihvaća niz x na ulazu i odašilje (nedeterministički, ako je potrebno) sve nizove y tako da je y prijevod od x.

"Uređaj" koji za dani ulazni niz x izračunava izlazni niz y tako da je (x,y) u zadanom prijevodu T, nazovimo prevodilac za T. Postoji nekoliko osobina poželjnih u definiciji prevođenja. Dvije od tih osobina su:

a. Definicija prevođenja treba biti čitljiva. Drugim riječima, treba biti lako uočljivo koji su parovi u prevođenju.
b. Treba biti moguće konstuirati efikasni prevodilac za dano prevođenje izravno iz definicije.

Poželjne osobine prevodilaca jesu:

1) Efikasna operacija. Za ulazni niz w duljine n potrebno vrijeme za obradu treba biti linearno proporcionalno s n.
2) Mala veličina.
3) Korektnost. Treba imati mali konačni test tako da ako prevodilac prođe taj test, to bi trebalo biti jamstvo da će korektno raditi za sve moguće ulaze.

Jedan od formalizama za definiranje prevođenja jest shema sintaksno-upravljanog prevođenja. Intuitivno, shema sintaksno-upravljanog prevođenja jest gramatika u kojoj su elementi prevođenja pridruženi svakoj produkciji. Kad bi neka produkcija bila upotrijebljena u izvođenju ulazne rečenice, element prevođenja se koristi kao pomoć u izračunavanju dijela izlazne rečenice pridružene dijelu ulazne rečenice generirane tom produkcijom.

♦ Shema sintaksno-upravljanog prevođenja

Shema sintaksno-upravljanog prevođenja (SDTS – *syntax-directed translation schema*) jest uređena petorka $T = (N, \Sigma, \Delta, R, S)$

gdje su:

- N - konačni skup *neterminalnih simbola*
- Σ - *ulazni alfabet*
- Δ - *izlazni alfabet*
- R - konačni skup *pravila* oblika $A \rightarrow \alpha, \beta$, gdje je $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$, i neterminali u β permutacije su neterminala iz α
- S - *startni simbol*, $S \in N$

Neka je $A \rightarrow \alpha, \beta$ pravilo. Za svaki neterminal iz α postoji identični *pridruženi* neterminal u β. Ako se neterminal B pojavljuje samo jedanput u α i β, pridruženje je očigledno. Međutim, ako se B pojavljuje više puta, koristit ćemo cijeli broj kao potenciju u zagradi da bismo naznačili pridruženje.

♣ Primjer 1.8

U pravilu

$$A \rightarrow B^{(1)}CB^{(2)}, B^{(2)}B^{(1)}C$$

tri su pozicije u $B^{(1)}CB^{(2)}$ pridružene pozicijama 2, 3 i 1, redom, u $B^{(2)}B^{(1)}C$.

♦ Translacijska forma

Translacijska forma od T definira se na sljedeći način:

1) (S, S) je translacijska forma i prvi S pridružen je drugom S.
2) Ako je $(\alpha A \beta, \alpha' A \beta')$ translacijska forma, u kojoj su dvije eksplicitne instance od A udružene i ako je $A \rightarrow \gamma, \gamma'$ pravilo u R, tada je $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ translacijska forma. Neterminali iz γ *i* γ' egzaktno su udruženi kao što su udruženi i u pravilu. Neterminali iz α i β udruženi su s takvim neterminalima iz α' i β' u novoj translacijskoj formi egzaktno kao i u staroj. Udruživanje će biti označeno potencijama, ako bude bilo potrebno.

Ako je translacijska forma $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ izvedena iz translacijske forme $(\alpha A\beta, \alpha' A \beta')$, piše se:

$$(\alpha A\beta, \alpha' A \beta') \overset{T}{\rightarrow} (\alpha\gamma\beta, \alpha'\gamma'\beta')$$

što čitamo "izravno izvodi". Znak T može biti izostavljen ako se podrazumijeva izvođenje translacijskih formi. Slično kao i kod izvođenja rečeničnih formi, s $+\rightarrow$ će biti označen niz od k izvođenja translacijskih formi, za $k \geq 1$, a s $*\rightarrow$ ako je $k \geq 0$, pa je prevođenje definirano s T, označeno s $\tau(T)$, skup parova:

$$\{(x, y) \mid (S, S) \overset{*}{\rightarrow} (x, y),\ x \in \Sigma^*,\ y \in \Delta^*\}$$

♣ Primjer 1.9

Evo primjera sheme prevođenja kojom je definirano prevođenje $\{(x, x^R) \mid x \in \{0,1\}^*\}$. Dakle, za svaki ulaz x izlaz je reverzni niz od x. Pravila koja definiraju takvo prevođenje su:

Produkcija	Element prevođenja
(1) $S \rightarrow 0S$	$S = S0$
(2) $S \rightarrow 1S$	$S = S1$
(1) $S \rightarrow \varepsilon$	$S = \varepsilon$

Ulazno-izlazni par u prevođenju definiranom ovom shemom može se dobiti generiranjem sekvence parova nizova (α,β) nazvanih translacijske forme, gdje je α ulazna rečenična forma, a β izlazna rečenična forma. Počinje se s translacijskom formom (S,S). Ako se sada upotrijebi prvo pravilo, β, zamjenjujemo S u izlaznoj rečeničnoj formi s S0, na temelju pravila S=S0. Nova translacijska forma je (0S,S0). Ponovo izvodimo ulazni niz prema pravilu (1), pa je translacijska forma (00S,S00). Sada koristimo pravilo (2), pa će nova translacijska forma biti (001S,S100). Ako sada upotrijebimo produkciju (3), dobije se (001,100). Izvođenje dalje nije definirano, pa je (001,100) prijevod definiran danom shemom prevođenja, odnosno, izlazna rečenica 100 jest prijevod ulazne rečenice 001.

Shema prevođenja T definira neko prevođenje $\tau(T)$. Može se konstruirati prevodilac za $\tau(T)$ iz sheme prevođenja tako da radi na sljedeći način. Za dani ulazni niz x prevodilac pronađe (ako je moguće) neko izvođenje od x iz S upotrebljavajući produkcije iz sheme prevođenja. Pretpostavimo da je $S = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n = x$ upravo to izvođenje. Tada prevodilac kreira izvođenje translacijskih formi:

$$(\alpha_0,\beta_0) \Rightarrow (\alpha_1,\beta_1) \Rightarrow (\alpha_2,\beta_2) \Rightarrow \ldots \Rightarrow (\alpha_n,\beta_n)$$

tako da je $(\alpha_0,\beta_0)=(S,S)$, $(\alpha_n,\beta_n)=(x,y)$ i svaki je dobiven upotrebljavajući za β_{i-1} element prevođenja koji je usklađen s produkcijom upotrijebljenom u prijelazu iz α_{i-1} u α_i na "odgovarajućem" mjestu. Niz y je izlaz za x. Često se izlazna rečenična forma može kreirati u vrijeme sintaksne analize ulaza.

♣ Primjer 1.10

Evo primjera sheme prevođenja infiksnih aritmetičkih izraza u prefiksnu poljsku notaciju:

Produkcija	*Element prevođenja*
E → E + T	E = + ET
E → T	E = T
T → T * F	T = * TF
T → F	T = F
F → (E)	F = E
F → a	F = a
F → b	F = b
F → c	F = c

Produkciji E→E+T pridružen je element prevođenja E=+ET, itd. Treba, na primjer, odrediti izlaz za ulaz a+b*c. Da bismo to učinili, treba najprije naći lijevu rečeničnu formu od a+b*c krenuvši od startnog simbola, koristeći zadane produkcije. Tada računamo odgovarajuće sekvence translacijskih formi, kako slijedi:

```
(E, E)  ➔ (    E + T, + ET)        ➔ (    T + T, + TT)
        ➔ (    F + T, + FT)        ➔ (    a + T, + aT)
        ➔ (a + T * F, + a * TF)    ➔ (a + F * F, + a * FF)
        ➔ (a + b * F, + a * bF)    ➔ (a + b * c, + a * bc)
```

Svaka izlazna rečenična forma dobivena je zamjenom odgovarajućeg neterminala (prvog s lijeva) u prethodnoj izlaznoj rečeničnoj formi desnom stranom pravila prevođenja pridruženog produkciji koja je bila upotrijebljena u izvođenju odgovarajuće ulazne rečenične forme. Evo još jednog primjera prevođenja rečenice (a+b)*c:

```
(E, E)  ➔ (          T,       T)    ➔ (      T * F,    * TF)
        ➔ (      F * T,    * FF)    ➔ (     (E)* F,    * EF)
        ➔ ((E + T) * F, * + ETF)    ➔ ((T + T) * F, * + TTF)
        ➔ ((F + T) * F, * + FTF)    ➔ ((a + T) * F, * + aTF)
        ➔ ((a + F) * F, * + aFF)    ➔ ((a + b) * F, * + abF)
        ➔ ((a + b) * c, * + abc)
```

♣ Primjer 1.11

U sljedećem primjeru dano je prevođenje logičkih izraza u prefiksnu poljsku notaciju. Ulazna gramatika je:

L → ¬L| L∨L| L∧L| L⇒L| (L)| f| t

Produkcija	*Element prevođenja*
L → ¬L	L = ¬L
L → L∨L	L = ∨LL
L → L∧L	L = ∧LL
L → L⇒L	L = ⇒LL
L → (L)	L = L
L → f	L = f
L → t	L = t

Na primjer, za ulazni niz t∧f⇒t imamo niz prevođenja:

(L, L) ➔ (L ∧ L, ∧ L L) ➔ (L ∧ L ⇒ L, ∧ L ⇒ L L)
➔ (t ∧ L ⇒ L, ∧ t ⇒ L L) ➔ (t ∧ f ⇒ L, ∧ t ⇒ f L)
➔ (t ∧ f ⇒ t, ∧ t ⇒ f t)

a za ulazni niz (t∧f)⇒t imamo niz prevođenja:

(L, L) ➔ ((L) ⇒ L, ⇒ L L) ➔ ((L ∧ L) ⇒ L, ⇒ ∧ L L L)
➔ ((t ∧ L) ⇒ L, ⇒ ∧ t L L) ➔ ((t ∧ f) ⇒ L, ⇒ ∧ t f L)
➔ ((t ∧ f) ⇒ t, ⇒ ∧ t f t)

Izlazna gramatika ima produkcije:

L → ¬L| ∨LL| ∧LL| ⇒LL| f| t

pa se može definirati sintaksno-upravljano prevođenje

Produkcija	*Element prevođenja*
L → ¬fL	L = tL
L → ¬tL	L = fL
L → ∨ffL	L = fL
L → ∨ftL	L = tL
L → ∨tfL	L = tL
L → ∨ttL	L = tL
L → ∧ffL	L = fL
L → ∧ftL	L = fL
L → ∧tfL	L = fL
L → ∧ttL	L = tL
L → ⇒ffL	L = tL
L → ⇒ftL	L = tL
L → ⇒tfL	L = fL
L → ⇒ttL	L = tL
L → ε	L = ε

Ako sada primijenimo tu shemu na ulazni niz ⇒∧tft koji je dobiven na izlazu prethodnog prevođenja, imamo:

(L, L) ➔ (⇒LL, L) ➔ (⇒∧tfL, fL)
➔ (⇒ L, ⇒ ∧ t L L) ➔ ((t ∧ f) ⇒ L, ⇒ ∧ t f L)
➔ ((t ∧ f) ⇒ t, ⇒ ∧ t f t)

♣ Primjer 1.12

Prevođenje rimskih brojeva u arapske:

R → A	R = A	H → ε	H = 00
R → B	R = B	H → C	H = C
R → C	R = C	H → D	H = 0D
R → D	R = D		
		C → IJ	C = IJ
A → EF	A = EF	I → x	I = 1
E → m	E = 1		
		I → xx	I = 2
E → mm	E = 2	I → xxx	I = 3
E → mmm	E = 3	I → xl	I = 4
F → ε	F = 000	I → l	I = 5
F → B	F = B	I → lx	I = 6
F → C	F = 0C	I → lxx	I = 7
F → D	F = 00D	I → lxxx	I = 8
		I → xc	I = 9
B → GH	B = GH	J → ε	J = 0
G → c	G = 1	J → D	J = D
G → cc	G = 2		
G → ccc	G = 3	D → i	D = 1
G → cd	G = 4	D → ii	D = 2
G → d	G = 5	D → iii	D = 3
G → dc	G = 6	D → iv	D = 4
G → dcc	G = 7	D → v	D = 5
G → dccc	G = 8	D → vi	D = 6
G → cm	G = 9	D → vii	D = 7
		D → viii	D = 8
		D → ix	D = 9

Gramatika ulaznog jezika je:

R → A| B| C| D
E → m| mm| mmm
G → c| cc| ccc| cd| d| dc| dcc| dccc| cm
I → x| xx| xxx| xl| l| lx| lxx| lxxx| xc
D → i| ii| iii| iv| v| vi| vii| viii| ix

A → EF B → GH C → IJ
F → ε| B| C| D
H → ε| C| D
J → ε| D

a gramatika izlaznog jezika može se dobiti iz translacijske sheme:

R → A| B| C| D A → EF B → GH C → IJ D → 1| 2| 3| 4| 5| 6| 7| 8| 9
E → 1| 2| 3 F → 000| B| 0C| 00D G → 1| 2| 3| 4| 5| 6| 7| 8| 9
H → 00| C| 0D I → 1| 2| 3| 4| 5| 6| 7| 8| 9 J → 0| D

Evo i tri primjera prevođenja:

mmmcmxcix
(R, R) ➔ (A, A) ➔ (EF, EF) ➔ (mmmF, 3F)
➔ (mmmB, 3B) ➔ (mmmGH, 3GH) ➔ (mmmcmH, 39H)
➔ (mmmcmC, 39C) ➔ (mmmcmIJ, 39IJ) ➔ (mmmcmxcJ, 399J)
➔ (mmmcmxcD, 399D) ➔ (mmmcmxcix, 3999)

mi
(R, R) ➔ (A, A) ➔ (EF, EF) ➔ (mF, 1F) ➔ (mD, 100D) ➔ (mi, 1001)

cxi
(R, R) ➔ (B, B) ➔ (GH, GH) ➔ (cH, 1H) ➔ (cC, 1C) ➔ (cIJ, 1IJ)
➔ (cxJ, 11J) ➔ (cxD, 11D) ➔ (cxi, 111)

♣ Primjer 1.13

Promjena "tonaliteta" za pola tona:

```
R → AB          R = AB          T → g           T = g#
B → ,R          B = ,R          T → g#          T = a
B → ε           B = ε           T → a           T = a#
A → TD          A = TD          T → a#          T = h

T → c           T = c#          T → h           T = c
T → c#          T = d
T → d           T = d#          D → -dur        D = -dur
T → d#          T = e           D → -mol        D = -mol
                                D → -7          D = -7
T → e           T = f           D → -dim        D = -dim
T → f           T = f#          D → -maj        D = -maj
T → f#          T = g
```

Na primjer, ulazni niz:

```
a-mol,d-mol,a-mol,e-7,a-mol,d-mol,a-mol,f-dur,g-dur,f-dur,a-mol,f-dur,g-dur,e-7
```

imat će izlazni niz:

```
a#-mol,d#-mol,a#-mol,f-7,a#-mol,d#-mol,a#-mol,f#-dur,g#-dur,f#-dur,a#-mol,
f#-dur,g#-dur,f-7
```

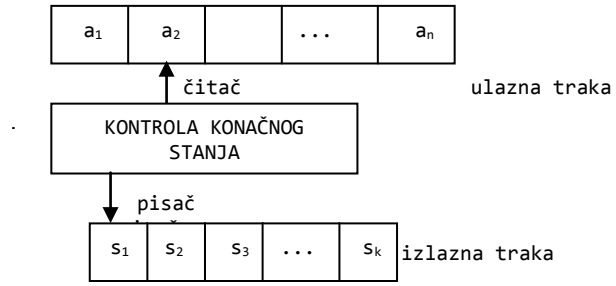
Primijetiti da je izlazni niz također rečenica ulaznog jezika. Ako sada taj niz shvatimo kao novi ulazni, dobili bismo sljedeći izlazni niz:

```
h-mol,e-mol,h-mol,f#-7,h-mol,e-mol,h-mol,g-dur,a-dur,g-dur,h-mol,g-dur,a-dur,f#-7
```

a to je prijevod prvog ulaznog niza za "cijeli ton".

1.3 KONAČNI PRETVARAČ

Drugi postupak prevođenja utemeljen je na primjeni konačnog pretvarača. Konačni pretvarač jest automat koji emitira izlazni niz, koji može biti i prazan, poslije svakog pomaka, sl. 1.1.



Sl. 1.1 – *Konačni pretvarač.*

♦ Konačni pretvarač

Konačni pretvarač, T, jest uređena šestorka

$$T=(Q,\Sigma,\Delta,\delta,q_0,F)$$

gdje su:

- Q konačni skup stanja
- Σ konačni ulazni alfabet
- Δ konačni izlazni alfabet
- δ funkcija prijelaza i generiranja, definirana kao

  $\delta: Q\times(\Sigma\cup\{\varepsilon\})\rightarrow Q\times\Delta^*$

- q_0 početno stanje, $q_0\in Q$
- F skup završnih stanja, $F\subseteq Q$

♦ Konfiguracija konačnog pretvarača

Konfiguracija konačnoga pretvarača T jest uređena trojka (q,x,y), gdje su:

- $q\in Q$ tekuće stanje kontrole konačnog stanja
- $x\in\Sigma^*$ preostali dio ulaznog niza; čitač je ispod prvog znaka od x
- $y\in\Delta^*$ tekući generirani izlazni niz

(q_0,x,ε) je početna konfiguracija, a $(\varepsilon,\varepsilon,y)$ završna (prihvatljiva ili konačna) konfiguracija. Pomak u T prikazan je binarnom relacijom $\vdash$ na konfiguracijama. Ako $\delta(q,a)$ sadrži (q',z), tada vrijedi:

$$(q,ax,y) \vdash (q',x,yz) \quad \text{za sve } x\in\Sigma^*$$

Ako postoje konfiguracije $C_0, C_1, C_2, \ldots, C_k$, tako da je

$$C_i \vdash C_{i+1} \quad 0\le i<k,$$

tada je

$$C_0 \vdash^k C_k$$

niz pomaka duljine k. Ako nije bitan broj pomaka, pišemo:

$$C \vdash^* C'$$

što ima značenje

$$C \vdash^k C' \quad k\ge 0$$

a ako je postojao najmanje jedan pomak

$$C \vdash^+ C' \quad k>0$$

♣ Primjer 1.14

Prevođenje rimskih brojeva u arapske, primjer 1.12, može se realizirati konačnim pretvaračem u kojem je funkcija (tablica) prijelaza i generiranja definirana na sljedeći način:

	q	**m**	**d**	**c**	**l**	**x**	**v**	**i**	**@**
	0	1,ε	8,ε	4,ε	17,ε	13, ε	26,ε	22,ε	
m	1	2,ε	8,1	4,1	17,10	13,10	26,100	22,100	ε,1000
mm	2	3,ε	8,2	4,2	17,20	13,20	26,200	22,200	ε,2000
mmm	3		8,3	4,3	17,30	13,30	26,300	22,300	ε,3000

	q	m	d	c	l	x	v	i	@
cc	5			6,ε	17,2	13,2	26,20	22,20	ε,200
ccc	6				17,3	13,3	26,30	22,30	ε,300
cd	7				17,4	13,4	26,40	22,40	ε,400
d	8			9,ε	17,5	13,5	26,50	22,50	ε,500
dc	9			10,ε	17,6	13,6	26,60	22,60	ε,600
dcc	10			11,ε	17,7	13,7	26,70	22,70	ε,700
dccc	11				17,8	13,8	26,80	22,80	ε,800
cm	12				17,9	13,9	26,90	22,90	ε,900
x	13			21,ε	16,ε	14,ε	26,1	22,1	ε,10
xx	14					15,ε	26,2	22,2	ε,20
xxx	15						26,3	23,3	ε,30
xl	16						26,4	24,4	ε,40
l	17						26,5	25,5	ε,50
lx	18						26,6	26,6	ε,60
lxx	19						26,7	27,7	ε,70
lxxx	20						26,8	28,8	ε,80
xc	21						26,9	22,ε	ε,90
i	22					30,ε	25,ε	23,ε	ε,1
ii	23							24,ε	ε,2
iii	24								ε,3
iv	25								ε,4
v	26							27,ε	ε,5
vi	27							28,ε	ε,6
vii	28							29,ε	ε,7
viii	29								ε,8
ix	30								ε,9

```
(0, mmmcmxcix, ε)
   ⊢ ( 1, mmcmxcix,   ε)  ⊢ ( 2, mcmxcix, ε)  ⊢ ( 3, cmxcix,   ε)
   ⊢ ( 4, mxcix,     3)  ⊢ (12, xcix,  39)  ⊢ (13, cix,    399)
   ⊢ (21, ix,      399)  ⊢ (22, x,     399)  ⊢ (30, ε,      399) ⊢ ( ε,ε, 3999)
```

1.4 STOGOVNI PRETVARAČ

Primjena konačnih pretvarača ograničena je na prevođenje linearnih jezika (ili jezika tipa 3). Ovdje ćemo definirati stogovni (ili potisni) pretvarač koji je primjenljiv u prevođenju beskontekstnih jezika (jezika tipa 2).

♦ Stogovni pretvarač

Stogovni pretvarač je uređena osmorka $\mathcal{T}=(Q,\Sigma,\Delta,\Gamma,\delta,q_0,Z_0,\mathcal{F})$, gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Δ izlazni alfabet
- Γ alfabet znakova stoga (potisne liste)
- δ funkcija prijelaza, definirana kao $\delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma \rightarrow Q\times\Gamma^*\times(\Delta\cup\{\varepsilon\})$
- q_0 početno stanje, $q_0\in Q$
- Z_0 početni znak potisne liste, $Z_0\in\Gamma$
- $\mathcal{F}$ skup završnih stanja, $\mathcal{F}\subseteq Q$

♦ Konfiguracija stogovnog pretvarača

Konfiguracija stogovnog pretvarača $\mathcal{T}$ jest (q, w, α, β) iz $Q\times\Sigma^*\times\Gamma^*\times\Delta^*$, gdje su:

- q tekuće stanje
- w preostali dio ulaznog niza
- α niz znakova koji predstavlja sadržaj potisne liste; vrh je prvi znak niza
- β izlazni niz znakova (prijevod)

Početna konfiguracija je $(q_0, w, Z_0, \varepsilon)$, a završna konfiguracija $(q, \varepsilon, \varepsilon, \beta)$, $q\in F$, $\beta\in\Delta^*$.

♦ Pomak stogovnog pretvarača

Pomak stogovnog pretvarača $\mathcal{T}$ jest relacija $\vdash_{\mathcal{T}}$ (ili samo $\vdash$ ako se $\mathcal{T}$ podrazumijeva):

$$(q, aw, Z\alpha, \beta) \vdash (q', w, \gamma\alpha, \beta\tau)$$

ako $\delta(q, a, Z)$ sadrži (q', γ, τ) za $q\in Q$, $a\in\Sigma\cup\{\varepsilon\}$, $w\in\Sigma^*$, $Z\in\Gamma$. Kaže se da je ulazni niz w prihvatljiv s $\mathcal{T}$ i da je β prijevod od w ako

$$(q_0, w, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \alpha, \beta)$$

♣ Primjer 1.15

Definirajmo stogovni pretvarač za prevođenje infiksnih izraza u prefiksne izraze, prema definiciji ulaznog i izlaznog jezika, te pravila prevođenja kao u primjeru 1.10. Bit će to pretvarač:

$$\mathcal{T} = (\{q\}, \Sigma, \{a, +, *, (,)\}, \Gamma, \delta, q, Z, \varnothing)$$

gdje je skup završnih stanja prazan, a funkcija prijelaza δ definirana je sa:

```
(1) δ(q,ε,E)  = {(q,E+T,+),(q,T,ε)}  (2) δ(q,ε,T)  = {(q,T*F,*),(q,F,ε)}
(3) δ(q,ε,F)  = {(q,(E),ε),(q,a,a)}  (4) δ(q,a,a)  = {(q,ε,a)}
(5) δ(q,+,+)  = {(q,ε,ε)}            (6) δ(q,*,*)  = {(q,ε,ε)}
(7) δ(q,(,()  = {(q,ε,ε)}            (8) δ(q,),))  = {(q,ε,ε)}
```

Na primjer, za ulazni niz a+a*a $\mathcal{T}$ će načiniti sljedeće pomake:

```
(q,a+a*a,E,ε)
   ⊢ (q, a+a*a, E+T,     +) ⊢ (q, a+a*a, T+T,     +) ⊢ (q, a+a*a, F+T,     +)
   ⊢ (q, a+a*a, a+T,    +a) ⊢ (q, +a*a,   +T,    +a) ⊢ (q, a*a,     T,    +a)
   ⊢ (q, a*a,   T*F,   +a*) ⊢ (q, a*a,   F*F,   +a*) ⊢ (q, a*a,   a*F,  +a*a)
   ⊢ (q, *a,     *F,  +a*a) ⊢ (q, a,       F,  +a*a) ⊢ (q, a,       a, +a*aa)
   ⊢ (q, ε,        ε, +a*aa)
```

A za ulazni niz (a+a)*a, imamo:

```
(q,(a+a)*a,E,ε)
   ⊢ (q, (a+a)*a,       T,      ε)    ⊢ (q, (a+a)*a,     F*T,      *)
   ⊢ (q, (a+a)*a,   (E)*T,      *)    ⊢ (q, a+a)*a,   E+T)*T,     *+)
   ⊢ (q, a+a)*a,   T+T)*T,     *+)    ⊢ (q, a+a)*a,   F+T)*T,     *+)
   ⊢ (q, a+a)*a,   a+T)*T,     *+)    ⊢ (q, +a)*a,     +T)*T,    *+a)
   ⊢ (q, a)*a,       T)*T,    *+a)    ⊢ (q, a)*a,       F)*T,    *+a)
   ⊢ (q, a)*a,       a)*T,    *+a)    ⊢ (q, )*a,         )*T,   *+aa)
   ⊢ (q, *a,           *T,   *+aa)    ⊢ (q, a,             T,   *+aa)
   ⊢ (q, a,            F,   *+aa)    ⊢ (q, a,             a,   *+aa)
   ⊢ (q, ε,            ε,  *+aaa)
```

P R O G R A M I

PREVOĐENJE IZRAZA U PREFIKSNU I POSTFIKSNU NOTACIJU

Prevođenje jednostavnih cjelobrojnih izraza koji sadrže brojke kao operande u prefiksnu i postfiksnu notaciju, prema algoritmima 1.1 i 1.2, objedinili smo u jednom programu. Sintaksna analiza izvodi se uz pomoć prepoznavača jezika sa svojstvima.

Prefiks-postfiks.py

```
# -*- coding: cp1250 -*-
# PROGRAM Infix_prefix_postfix
#         Prevođenje izraza iz infiksne u prefiksnu i postfiksnu notaciju

from fun import *

Operandi = ['+', '-', '*', '/']
Zn       = 'O()B@'

def Ok (N) :
  """ Sintaksna analiza infiksnih izraza
            O   (   )   B  @
            0   1   2   3  4  """
  Tp  = ( (-1,  0, -1,  1, -1),
          ( 0, -1,  1, -1,  0) )
  Ta  = ( ( 0,  1,  0,  0,  0),
          ( 0,  0,  2,  0,  3) )

  X = komp (N)
  N, X = X, X +'@';  S = 0;  b = 0
  Pogr = False;  Kraj = False;  i = 0
  while i < len (X) and not Pogr and not Kraj:
    C = X[i]
    if   C in Operandi : K = 0
    elif C in B        : K = 3
    else               : K = pos (C, Zn)
    Pogr = K == -1
    if not Pogr:
      A = Ta[S][K];  S = Tp[S][K]; Pogr = S == -1
      if not Pogr and A != 0:
        if   A == 1 : b += 1
        elif A == 2 :
          if b > 0  : b -= 1
          else      : Pogr = True
        elif A == 3 : Pogr = b > 0;  Kraj = not Pogr
    if Pogr : print 'POGREŠKA'
    i += 1

  return not Pogr

def PrefixSufix (PS, S):

  def push (Ch, L)     : L = Ch*T +L +Ch*(1-T);  return L
  def pop  (L)         : return L[T-1], L1[T: len(L1)-1 +T]
  def PopPush (L1, L2) :  c, L1 = pop (L1);  L2 = push (c, L2);  return L1, L2
```

```
    L1 = '';  L2 = '';
    T  = 1*(PS == 'P'); T1 = T-1
    Z = ')'*T + '('*(1-T); Z1 = '('*T + ')'*(1-T);
    i1 = T*len(S)-T; i2 = -T1*len(S) -T;  i3 = -T-T1

    for i in range (i1, i2, i3):
      C = S[i]
      if   C in B  : L2 = push (C, L2)
      elif C in ['+', '-', '*', '/', Z] :
        if L1 == '' or C == Z: L1 = push (C, L1)
        else :
          if L1[T1] in ['+', '-'] and C in ['+', '-'] or L1[T1] in ['*', '/'] :
            L1, L2 = PopPush (L1, L2)
          L1 = push (C, L1)
      elif C == Z1 :
        while L1 <> '' and L1[T1] <> Z:  L1, L2 = PopPush (L1, L2)
        L1 = L1[T: len(L1)-1 +T]

    while L1 <> '': L1, L2 = PopPush (L1, L2)
    return L2

N = raw_input ('Upiši izraz ')
while len (N) > 0:
  if Ok (N):
    P = PrefixSufix ('P', N); print P
    P = PrefixSufix ('S', N); print P
  N = raw_input ('Upiši izraz ')
```

Infiks	Prefiks	Postfiks
(1+2)*((3+4)*5+6)*7	**+12+*+34567	12+34+5*6+*7*
(((((1+2)))))	+12	12+
(((((1+2)*3+4)*5+6)*7+8)*9+1)*6	*+*+*+*+*+12345678916	12+3*4+5*6+7*8+9*1+6*
1*2+3*4+5*6+7*8	+++*12*34*56*78	12*34*56*78*+++
(1+2)*(3+4)	*+12+34	12+34+*
(1+2)*3+(4+5)*6	+*+123*+456	12+3*45+6*+
1+2*(3+4)*5	+1**2+345	1234+5**+
(((1+2)*3+(4+5)*6+(7+8)*9)*2+3)*2	*+*++*+123*+456*+789232	12+3*45+6*78+9*++2*3+2*
(((1+2)*3+4)*5+6)*7	*+*+*+1234567	12+3*4+5*6+7*
(5-6)*3+(1-9)*9	+*-563*-199	56-3*19-9*+
(1+2+3)*4+(5+6*7)*8	+*+1+234*+5*678	12+3+4*567*+8*+
(1+2+3+4)*5	*+1+2+345	12+3+4+5*
1*(2+3)+4*(5+6)+7*(8+9)	++*1+23*4+56*7+89	123+*456+*789+*++

Zadaci

1) *Napišite program koji će izraz napisan u prefiksnoj notaciji prevesti u kompoziciju funkcija za izvršavanje (izračunavanje) izraza.* Funkcije su: add(*x,y*) – *zbrajanje;* sub(*x,y*) – *oduzimanje;* mpy(x,y) – množenje i div(x,y) – dijeljenje. *Na primjer:*

   2 * (3 + 4) -> * 2 + 3 4 -> mpy (2, add (3, 4))

2) *Napišite program (pretvarač) koji će prevoditi rimske brojeve u arapske.*

3) *Napišite program za prevođenje logičkih izraza iz primjera 1.11 u prefiksnu notaciju.*

2.
JEZICI ZA PROGRAMIRANJE

— Cupidon s'en fout

Za pretvoriti u ljubav ljubakanje naše,
Trebalo je učiniti samo malo,
Ali, tad je Venera bila volje loše,
A Kupidonu ni do čega nije stalo!

Dani su to kad on beskorisno korača,
Kad očišćene su mahovine na vrhu,
Uglađeni šiljci strelica koje na nas baca,
Ali za Kupidona to nema nikakvu svrhu!

Posvetivši se drugim tupavcima,
Nije imao vremena mariti za nas,
Sa svojim lukom i svim svojim strelicama,
To su dani kad Kupidon ne želi naš spas!

Pokušalo se bez njega otvoriti slavu,
Na travi mekoj se valjati,
Ali izgubili ste moć, izgubili ste glavu,
Jer Kupidon se ne želi javljati!

Premda ste mi i dali pune ovlasti,
Srce, nažalost, u igri bilo nije,
Sveta je vatra sjajila unatoč njegovoj odsutnosti,
Tih se dana Kupidon samo smije!

Kidao sam dvadeset puta latice ivančice,
Dvadeset puta je ispalo «nema šanse».
I naša sirota idila ostala je bez iskrice,
Jel' moguće da Kupidonu nije do naše romanse?

Kad budete išli u šumu po cvijeće,
Nebo je s vama, a vi ljupki i mladi,
Nisam imao, nažalost, te sreće,
Baš briga Kupidona - ne radi!

Baš briga Kupidona
Cupidon s'en fout

(Georges BRASSENS/
Zdravko DOVEDAN HAN)

Važna primjena teorije formalnih jezika je u definiranju i prevođenju jezika za programiranje. Temeljna mu je namjena za pisanje programa koji će biti izvršeni na računalu.

Najprije dajemo generacije jezika za programiranje i njihovu podjelu, potom općeniti opis ili specifikaciju jezika za programiranje. Na kraju je dan primjer definicije jezika Exp, jezika realnih izraza.

2.1 UVOD

Snažan poticaj razvoju teorije formalnih jezika dala je masovnija upotreba računala pedesetih i šezdesetih godina prošloga stoljeća.

Računalo (kompjuter) je, kao što sigurno svi znamo, stroj koji može uraditi samo ono za što mu je netko dao instrukcije (program) – niz logičkih i aritmetičkih operacija napisanih jezikom kompjutera. Kompjuter takve instrukcije izvršava brzo i gotovo nepogrešivo, upravo onako kako su zadane.

Kompjuter može izvršiti samo mali broj veoma jednostavnih operacija. Na primjer, oduzimanje, množenje i dijeljenje svodi na operacije zbrajanja i pomicanja znamenki. Kompjuter "razumije" i može izvršiti samo instrukcije strojnog jezika. S razvojem kompjutera usavršavao se i jezik za komuniciranje (programiranje), pa razlikujemo četiri generacije:

1) Prva generacija - strojni jezici
2) Druga generacija - simbolički (asemblerski) jezici
3) Treća generacija - jezici za programiranje visoke razine
4) Četvrta generacija - jezici četvrte generacije (jezici krajnjih korisnika)

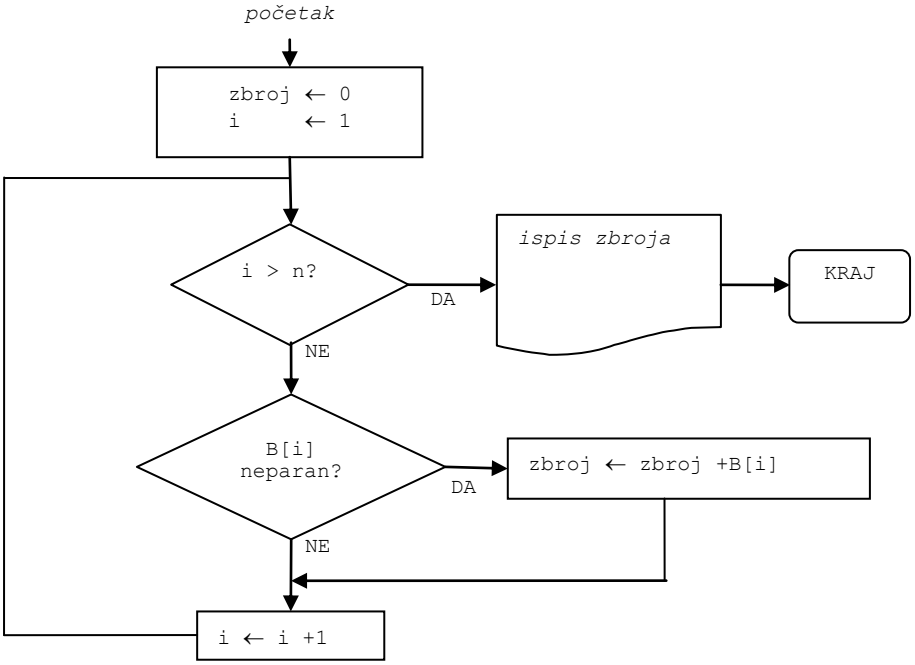
Generacije jezika za programiranje ne treba poistovjećivati s generacijama kompjutera. Na primjer, danas su u upotrebi kompjuteri četvrte generacije, na kojima nalazimo sve četiri generacije jezika za programiranje. No, prije nego što detaljnije opišemo generacije jezika, neka nam sljedeći problem i dani postupak - rješenje u "pseudojeziku" - posluži kao primjer koji će dovoljno zorno odražavati temeljne karakteristike pojedinih generacija jezika.

Problem

Izračunati zbroj neparnih brojeva komponenti cjelobrojnog niza brojeva (polja) duljine n.

Rješenje

Dajemo rješenje (algoritam) u notaciji nekad popularnog "dijagrama toka".



Strojni jezik

Pisanje programa za prve kompjutere obavljalo se isključivo strojnim jezikom. To je bilo dosta mukotrpno jer je strojni jezik niz nula i jedinica određene konačne duljine. Trebalo je pamtiti što koji od tih nizova znači. Na primjer, rješenje postavljenog problema u strojnom jeziku, napisanom za mikroprocesor Motorola 68000 jest:

00100100	01011111				
00100010	01011111				
00110010	01000000				
01001110	11111010	00000000	00000110		
00001000	00101001	00000000	00000000	00000000	00000001
01100111	00000010				
11010100	01010001				
01010100	01001001				
01010001	11001001	11111111	11110010		
00111110	10000010				
01001110	11010010				

Pisanje programa strojnim jezikom ne samo da je bilo otežano nerazumljivim kodovima pojedinih instrukcija (je li dani program napisan bez pogreške?!), već nije postojao ni jedinstveni strojni jezik, pa onaj tko je poznavao jezik jednog stroja nije mogao tim jezikom programirati na drugome.

Simbolički (asemblerski) jezik

Prvi korak prema tzv. jezicima više razine bilo je uvođenje simboličkog (asemblerskog ili mnemoničkog) jezika. U tom se jeziku programer koristi mnemoničkim imenima i za operacije i za operande. Tako se simboličkim jezikom mikroprocesora Motorola 68000 može napisati `SUMODDS MOVE.L (A7)+,A2` umjesto `00100100 01011111` u strojnom jeziku.

Program napisan simboličkim jezikom lakši je za pisanje i razumijevanje od programa pisanog strojnim jezikom. Prije svega, numerički kodovi za operacije i adrese zamijenjeni su prihvatljivim simboličkim kodovima koji već svojim nazivom podsjećaju korisnika na svoju namjenu. Pogledajmo ustroj danog algoritma na asemblerskom jeziku, takoder za mikroprocesor Motorola 68000:

```
SUMMODDS    MOVE.L    (A7)+,A2
            MOVE.L    (A7)+,A1
            MOVE.W    (A7)+,D2
            CLR.W     D2
            JMP       COUNT
LOOP        BTST      0,1(A1)
            BEQ.S     NEXT
            ADD.W     (A1),D2
NEXT        ADDQ.W    #2,A1
COUNT       DBF       D1,LOOP
            MOVE.W    D2,-(A7)
            JMP       (A2)
```

Jezici visoke razine

Uvođenjem asemblerskih jezika znatno je olakšano pisanje i razumijevanje programa, ali je još ostalo određenih teškoća. Naime, programer je morao detaljno poznavati način na koji određeni kompjuter izvršava operacije. Također je morao računanjem napamet prevoditi kompleksne operacije i strukture podataka u niz operacija niske razine, koje rabe samo primitivne tipove podataka strojnog jezika. Odnosno, morao je paziti kako su i gdje podaci postavljeni u radnoj memoriji kompjutera.

Radi izbjegavanja takvih i sličnih problema, razvijeni su jezici visoke razine. U osnovi, jezici visoke razine omogućuju programeru da piše algoritme u prirodnijoj notaciji u kojoj se ne treba baviti mnogim detaljima vezanim za neki specifični kompjuter. Na primjer, neusporedivo je ugodnije pisati `A=B+C` nego niz asemblerskih instrukcija.

Danas je u široj upotrebi dvadesetak jezika visoke razine. To su: Ada, APL, Assembly, (Visual) BASIC, C, C++, C#, Delphi/Object Pascal, FORTRAN, Java, JavaScript, LISP, LOGO, Lua, MATLAB, Objective-C, Pascal, Perl, PHP, PL/SQL, Python, Ruby, Transact-SQL itd. Razlikuju se po svom stupnju bliskosti matematičkome ili prirodnim jezicima, s jedne strane, i strojnom jeziku, s druge strane. Također se razlikuju po vrsti problema čijem su rješavanju najbolje prilagođeni. Suvremeni jezici za programiranje visoke razine sličnih

su osnovnih mogućnosti: sadrže neproceduralne dijelove, omogućuju objektno i vizualno programiranje, rad s bazama podataka, web programiranje itd.

Jezici za programiranje visoke razine podvrgnuti su standardima s propisanom *sintaksom* (pravilima pisanja) i *semantikom* (značenjem), čime je uvelike postignuta neovisnost o karakteristikama kompjutera i operacijskog sustava na kojemu su instalirani. Od velikog broja jezika visoke razine, nekih već i zaboravljenih, izabrali smo BASIC, Pascal, COBOL, APL, LISP i Python da bismo na primjeru programa za ustroj našeg algoritma predočili koliko su bliski i istodobno toliko različiti.

BASIC

```
100 DIM B(100)
110 READ N
120 FOR I = 1 TO N
130   READ B(I)
140   NEXT I
150 GOSUB 220
160 PRINT S
170 END
180 '
190 DATA 4
200 DATA 23, 34, 7, 9
210 '
220 S = 0
230 FOR I = 1 TO N
240   IF B(I) MOD 2 <> 0 THEN S = S +B(I)
250   NEXT I
260 RETURN
```

PASCAL

```
PROGRAM Zbroj;
  CONST n = 4;
  TYPE  p = array [1.. n] of integer;
  CONST B : p = (23, 34, 7, 9);

FUNCTION zbrojNep (X : p) : integer;  VAR i, S : integer; BEGIN
  S := 0;
  FOR i := 1 TO n DO IF odd(X[i]) THEN S := S  +X[i];
  zbrojNep := S
  END;
BEGIN
  Write (zbrojNep (B)); ReadLn
END.
```

COBOL

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUMERIC-VARIABLES USAGE IS COMPUTATIONAL.
      02 B PICTURE 9999 OCCURS 100 TIMES INDEXED BY 1.
      02 N PICTURE 999.
      02 S PICTURE 999999.
      02 B-POLA PICTURE 9999.
```

```
      02 OSTATAK PICTURE 9.


PROCEDURE DIVISION.
PRIMJER.
      MOVE 23 TO B(1).
      MOVE 34 TO B(2).
      MOVE 7 TO B(3).
      MOVE 9 TO B(4).
      MOVE 4 TO N.
      PERFORM ZBROJ-NEP.
ZBROJ-NEP.
      MOVE 0 TO S.
      PERFORM PRIBROJ VARYING I FROM 1 BY 1 UNTIL I N.
PRIBROJ.
      DIVIDE 2 INTO B(I) GIVING POLA-B REMAINDER OSTATAK.
      IF OSTATAK IS EQUAL TO 1; ADD B(I) TO S.
```

APL

```
      V  S  <-- ZBROJNEP   B
[1]      S  +/(2 | B)/B
      V
ZBROJNEP  23 34  7  9
                B  <--     23    34     7     9
            (2 | B)    <--   1    0         1     1
            (2 | B)/B  <--  23              7     9
           +/(2 | B)/B  <--  23    +         7  +  9
                    S   <--  39
```

LISP

```
(DEFUN ZBROJNEP
 (LAMBDA (B)
  (COND
  ((NULL B) 0)
    ((ODD (CAR B)) (PLUS (CAR B) (ZBROJNEP (CDR B))))
(ZBROJNEP (23 34 7 9))
(ZBROJNEP (23 34 7 9))
  = (PLUS 23 (ZBROJNEP (34 7 9)))
     = (PLUS 23 (ZBROJNEP (7 9)))
        = (PLUS 23 (PLUS 7 (ZBROJNEP (9)))
           = (PLUS 23 (PLUS 7 (PLUS 9 (ZBROJNEP ( )))
              = (PLUS 23 (PLUS 7 (PLUS 9 0)))
           = (PLUS 23 (PLUS 7 9))
        = (PLUS 23 16)
     = 39
```

PYTHON

```
L = [23, 34, 7, 9];  S = 0
for x in L:
  if x % 2 != 0: S += x
print S
```

Dakako, s obzirom na to da je problem „algoritamske prirode“, Python i Pascal daju najčitljiviji (ili najprihvatljiviji) kôd.

Jezici četvrte generacije

Bez obzira na to što je svaka nova generacija jezika bila bliža korisniku, još je uvijek programiranje s tim jezicima posebna disciplina. Tek je s pojavom jezika četvrte generacije stvorena alternativa profesionalnom programiranju. Karakteristika tih jezika je potpuna prilagođenost krajnjim korisnicima – najčešće neprogramerima, a primjenjuju ih veoma uspješno i informatičari (programeri i analitičari sustava) radi ubrzanja procesa programiranja.

Ne postoje opći standardi za sintaksu i semantiku jezika četvrte generacije, osim u jednom dijelu - za upitne jezike za baze podataka. Ovisno o njihovoj namjeni, možemo ih razvrstati u nekoliko kategorija:

- upitni jezici za bazu podataka,
- generatori izvještaja,
- jezici za podršku odlučivanju,
- generatori programa,
- jezici za obradu teksta,
- jezici za crtanje itd.

Neki su od tih jezika višenamjenski (ali ne i općenamjenski), pa se mogu svrstati u nekoliko kategorija. Obično rade s bazom podataka, pa krajnjim korisnicima omogućuju definiranje i kreiranje vlastite baze podataka i pristup tim podacima.

Često se neki jezici za programiranje visoke razine pogrešno svrstavaju u jezike četvrte generacije, na primjer, APL, C#, Delphi i Prolog. Točno je da dijelovi nekih novijih jezika pripadaju klasi jezika četvrte generacije, ali sam jezik pripada trećoj generaciji.

2.2 DEFINIRANJE JEZIKA ZA PROGRAMIRANJE

Ako nad alfabetom $\mathcal{A}$ definiramo jezik - rječnik $\mathcal{V}$, $\mathcal{V} \subseteq \mathcal{A}^*$, jezik za programiranje jest:

$$\mathcal{L} \subseteq \mathcal{V}^*$$

Dakle, reći ćemo da je jezik za programiranje skup rečenica (koji se tada nazivaju "programi") iz skupa svih nizova riječi (ili simbola) nad rječnikom $\mathcal{V}$. I definicija gramatike imat će šire značenje. Umjesto nad alfabetom, bit će definirana nad rječnikom:

$$\mathcal{G} = (\mathcal{N}, \mathcal{V}, \mathcal{P}, \mathcal{S})$$

Skup $\mathcal{N}$ je skup neterminalnih simbola (riječi), a startni znak sad ima značenje "startnog simbola", jer je $\mathcal{S}$ element skupa $\mathcal{N}$.

Neformalno, kaže se da je jezik za programiranje notaciona tehnika, pismo, kojom se na kompaktan, nedvosmislen i konačan način specificira skup naredbi da bi se izvršila neka aktivnost, riješio određeni problem.

Jezici za programiranje daleko su jednostavniji od prirodnih (npr. hrvatskoga, engleskoga, francuskoga, talijanskoga, itd). Osim toga, "rečenice" jezika za programiranje mogu se opisati strogim pravilima, bez izuzetaka i s jedinstvenim značenjem. Na žalost, u mnogim knjigama o jezicima za programiranje i školama programiranja jezik se nastoji prikazati isključivo primjerima, a onome tko ga uči preostaje da sam zaključi i izvede opća pravila za pisanje naredbi. S druge strane, još uvijek ne postoji "recept" koji bi upućivao na prave puteve definiranja i učenja jezika za programiranje. Međutim, iskustvo pokazuje da se najveći učinci postižu ako se pri učenju jezika istaknu tri stvari: leksička struktura, sintaksna struktura i semantika jezika.

Leksička struktura

Definirati leksičku strukturu nekog jezika znači definirati alfabet i rječnik. Alfabet je skup svih znakova koji se koriste u pisanju. To su slova, znamenke, operacije, te drugi znakovi.

Rječnik je skup riječi (simbola) definiranih nad alfabetom. Riječ (ili simbol) je niz znakova iz alfabeta koji se može promatrati kao jedinstvena, nedjeljiva cjelina. Na primjer, neka je dan niz `A-B*C`. Sastoji se od pet znakova koji mogu biti grupirani na nekoliko načina. Može se smatrati da niz `A-B` čini jednu riječ (u jeziku `COBOL` to bi bilo ime varijable). Ili se `A-B` može promatrati kao varijabla `A` minus varijabla `B` (kao što je u `FORTRAN`-u i nekim drugim jezicima). Što je od ovog korektno, ovisi o definiciji leksičke strukture jezika. Njome će biti propisano koje riječi treba tretirati kao imena, a koje kao operatore.

Dakle, rječnik je regularni jezik i moguće ga je definirati regularnim izrazom, regularnom gramatikom ili konačnim automatom. Radi boljega sagledavanja leksičke strukture nekoga jezika uobičajeno je rječnik podijeliti u klase riječi (simbola) koje imaju zajednička svojstva: brojeve, imena, rezervirane riječi, imena funkcija, ostale (posebne) simbole itd.

Sintaksna struktura

Sintaksna (ili sintaktička) struktura jezika utvrđuje grupiranje leksičkih konstrukata u šire strukture, nazvane *sintaksne kategorije*. Pravila koja određuju pripada li niz simbola jeziku ili ne nazivaju se sintaksa jezika. Danas je u uporabi nekoliko notacija ili načina prikazivanja sintakse nastalih iz Backus-Naurove forme (BNF), prvi puta objavljen 1963. godine u opisu jezika ALGOL 60. Upravo od pojave Pascala popularni su *sintaksni dijagrami*. Oba ova formalizma opisali smo u prvoj knjizi. Ovdje ćemo prikazati njihovu prilagodbu opisu sintakse jezika za programiranje. Osim toga dajemo i mogućnosti uporabe regularnih izraza u prikazu osnovne sintaksne strukture jezika za programiranje.

PROŠIRENA BACKUS-NAUROVA FORMA

Za jednostavniji i potpuniji opis osnovne sintaksne strukture jezika za programiranje prilagodit ćemo i proširiti pravila pisanja Backus-Naurove forme i uvesti sljedeća pravila:

1) Neterminalni simboli pišu se malim nakošenim slovima ili brojkama, bez razmaka. Ako se simbol sastoji od više riječi, riječi su povezane podcrtom (donjom crtom) "_".
2) Umjesto "→" ili "`::=`" pisat će se "`:`" (čita se kao i prije, "... definirano je kao...").
3) Terminalni simboli su nizovi sačinjeni od svih ostalih znakova i velikih slova. Pišu se uspravno.
4) $\alpha\,\beta$ slijed. α i β moraju biti razdvojeni s jednim ili više razmaka.
5) $\alpha \mid \beta$ ili $(\alpha \mid \beta)$ alternativa (α ili β).
6) $[\alpha]$ α izostavljeno ili napisano jedanput.
7) $\{\alpha\}$ α izostavljeno ili napisano jedanput, dvaput, ...

To su pravila proširene Backus-Naurove forme (EBNF). Na primjer, gramatika s produkcijama:

```
prirodni_broj : nenula { brojka }
  nenula      : 1 | 2 | 3 | 4 |5 |6 |7 |8 |9
  brojka      : nenula |0
```

generira jezik prirodnih brojeva {1,2,3,4, ...,1001,...}.

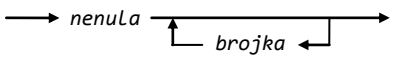
SINTAKSNI DIJAGRAMI

U prvoj smo knjizi, [Dov2012a], definirali sintaksne dijagrame. Notacija sintaksnih dijagrama primijenjena u opisu jezika za programiranje može biti pojednostavljena:

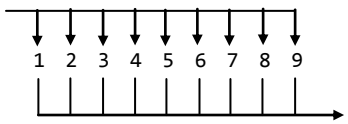
- sintaksne kategorije (neterminali) bit će pisane nakošenim malim slovima
- ako je ime sintaksne kategorije sačinjeno od više riječi, riječi će biti razdvojene donjom crtom (podcrtom)
- simboli jezika bit će pisani velikim slovima ili nizovima znakova različitim od malih slova

Na primjer, definirajmo primjenom sintaksnih dijagrama pravilo pisanja prirodnih brojeva:

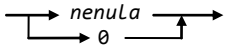
prirodni_broj:



nenula:



brojka:



Pravilo pisanja prirodnih brojeva sadrži dva simbola koji nisu u jeziku. Sve tri definicije u potpunosti opisuju tvorbu prirodnih brojeva. Na primjer, krenuvši od definicije prirodnog broja prvo nailazimo na *nenula*. Poslije toga može se završiti ili krenuti putem preko *brojka*. Pretpostavimo da smo završili. Međutim, *nenula* nije element jezika, pa ga moramo zamijeniti njegovom definicijom. Ako pogledamo definiciju sintaksne kategorije *nenula*, zaključit

ćemo da moramo izabrati jednu brojku od 1 do 9. Na primjer, izaberimo brojku 8, pa umjesto *nenula* možemo napisati 8. To je ujedno i prirodan broj. Evo još jednog primjera primjene danih pravila:

```
nenula   brojka   brojka
   9     brojka   brojka
   9        0     brojka
   9        0        1
```

Uočimo da dijagram kojim je definiran *prirodni_broj* sadrži petlju koja osigurava da generiramo potpuni (beskonačni) skup zapisa beskonačnog skupa prirodnih brojeva. Također primijetimo da je bilo neophodno uvesti strukturu *nenula* koja je osigurala da prirodni broj ne smije biti 0, niti smije početi nulom.

Već iz ovog jednostavnog primjera mogu se uočiti prednosti primjene sintaksnih dijagrama, odnosno definiranje pravila pisanja jezika, u njegovom učenju. Navedimo samo najbitnije:

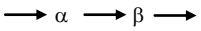
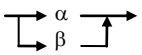
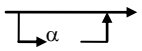
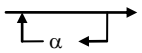
1) Kompaktnim konačnim pravilom - sintaksnim dijagramom - moguće je opisati veliki skup kombinacija u generiranju dane naredbe.
2) Pravila predočena dijagramima brže se pamte i uče, jer većina ljudi bolje pamti vizualno.
3) Promatranjem svih naredbi jezika moguće je uočiti dijelove koji su jednaki. Uvođenjem posebnih imena za takve dijelove znatno se pojednostavljuje učenje jezika.

Međutim, često ćemo se, da bismo izbjegli crtanje, odlučiti sintaksnu strukturu jezika prikazati u EBNF-u. Ali, ni sintaksni dijagrami ni EBNF neće uvijek biti dovoljni za potpuni opis svih naredbi jezika za programiranje. Naime, pojedine dodatne (kontekstne) uvjete koji moraju biti ispunjeni prilikom pisanja nekih naredbi nije moguće opisati jer su oba prikaza beskontekstne gramatike. Na primjer, u Pascalu izraz A+B napisan je korektno ako su A i B varijable ili konstante istog primitivnog tipa, nizovnog ili skupovnog tipa.

Osim toga, postojat će situacije gdje je moguć opis sintaksnim dijagramom, ali bi bio prekompliciran, kao na primjer da ime može sadržavati od jednog do 127 znakova. I u jednom i u drugom slučaju davat ćemo dodatna pravila riječima. Na primjer, ako želimo definirati pravilo za tvorbu prirodnih brojeva, od 1 do 9999999999, onda ćemo uz dano pravilo reći da ono vrijedi uz uvjet da se *brojka* smije upotrijebiti do 10 puta ili ćemo u sintaksnom dijagramu označiti maksimalni broj prolazaka određenim putem.

Ponekad ćemo osnovnu sintaksnu strukturu jezika kojeg želimo definirati prikazati kombinirajući ta dva formalizma. U sljedećoj smo tablici prikazali kako se pravila napisana u EBNF-u mogu prevesti u ekvivalentne sintaksne dijagrame:

Pravilo	Opis	Značenje

1)	$\alpha\,\beta$	slijed	
2)	$\alpha \mid \beta$	alternativa (α ili β)	
3)	$[\alpha]$	α izostavljeno ili napisano jedanput	
4)	$\{\alpha\}$	α izostavljeno ili napisano jedanput, dvaput, triput, ...	

Na primjer, pravilo pisanja prirodnih brojeva u ovoj notaciji prikazano je sa:

```
prirodni_broj: nenula { brojka }
nenula:        1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
brojka:        nenula | 0
```

REGULARNI IZRAZI I PRIKAZ OSNOVNE SINTAKSNE STRUKTURE

Poznato nam je značenje regularnih izraza u generiranju i pretraživanju regularnih jezika. Osim toga, u posljednje se vrijeme proširena notacija pisanja regularnih izraza koristi u prikazu osnovne sintaksne strukture nekih jezika za programiranje, posebno Pythona. Takva je notacija slična EBNF-u. Regularni podizrazi su označeni imenima („neterminalima“). Pišu se malim uspravnim slovima s podcrtom između riječi. Riječi rječnika jezika koji se definira pišu se između navodnika. Značenje metasimbola je kao i u proširenoj notaciji za definiranje regularnih izraza. Na primjer, dio sintakse Pythona može se prikazati kao

```
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
stmt: simple_stmt | compound_stmt
simple_stmt:  small_stmt (';' small_stmt)* [';'] NEWLINE
compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | ...
small_stmt:   (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
              import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
testlist: test (',' test)* [',']
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |...
...
```

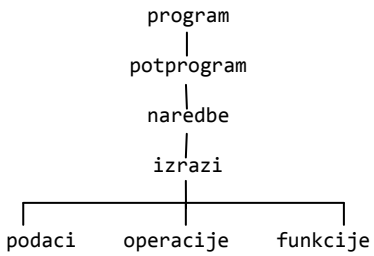
```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
...
```

Primijetimo da su dopuštene rekurzije što se moglo očekivati s obzirom na to da se opisuje osnovna sintaksna struktura beskontekstnog jezika.

Hijerarhijska struktura jezika

Jezik za programiranje smo definirali kao notacijsku tehniku (pismo) kojom se na kompaktan, nedvosmislen i konačan način specificira niz operacija koje će biti izvršene nad nekim objektima - podacima. Određeni niz tih operacija napisan u nekom jeziku naziva se program.

Općenito se operacije i podaci u većini jezika za programiranje mogu grupirati hijerarhijski, kao što je prikazano na sljedećoj slici:



Sl.2.1 - Hijerarhija elemenata programa.

Korištenjem ovakve hijerarhijske strukture znatno se olakšava i učenje jezika. Naime, premda smo rekli da su jezici za programiranje daleko jednostavniji od prirodnih, bilo bi ih teško učiti bez neke sistematizacije. Sagledavajući dijelove jezika postupno, kroz hijerarhijsku strukturu i grupe naredbi, jezik se može naučiti daleko brže i potpunije. U daljnjem detaljnijem opisu elemenata programa pođimo redom od onih koji su na najnižoj razini prema vrhu.

Tipovi i strukture podataka

Podaci su na razini strojnog jezika predočeni znakovima binarnog alfabeta. U jezicima za programiranje visoke razine podaci ne pripadaju samo jednom skupu vrijednosti, niti su nad različitim skupovima dopuštene sve operacije. Zato se u tim jezicima uvodi pojam tipa podataka.

Tip podataka je skup vrijednosti koje imaju izvjesne zajedničke osobine. Najznačajnija od njih je skup operacija koje su definirane nad vrijednostima tog tipa. U programiranju općenito, a posebno u jezicima za programiranje visoke razine, pojam tipa od posebne je važnosti. Tipom se određuje iz kojega se skupa vrijednosti varijablama u programu mogu dodjeljivati vrijednosti, i koje su operacije dopuštene. U većini jezika za programiranje, kao

što je to na primjer u Pascalu, zahtijeva se eksplicitno deklariranje varijabli po tipu prije njihove prve upotrebe u programu.

Jedan od osnovnih razloga za uvođenje tipova bio je omogućivanje kontrole korektnosti upotrebe vrijednosti različitog tipa i operacija s njima u izrazima programa. Jednako važan razlog u implementaciji jezika jest što različiti tipovi vrijednosti zahtijevaju različit broj ćelija za memoriranje i različit prikaz u memoriji. U većini jezika za programiranje susreću se sljedeći standardni tipovi podataka:

- brojčani (cjelobrojni i realni)
- logički
- znakovni

Ova četiri tipa podataka nazivaju se još *primitivni tipovi*, zato što u jezicima za programiranje visoke razine predstavljaju nedjeljive cjeline i imaju izravan prikaz u memoriji kompjutera.

CJELOBROJNI TIP

Cjelobrojni tip („integer“) podskup je skupa cijelih brojeva, odnosno skup cjelobrojnih vrijednosti iz intervala

$$-2^{n-1} \div 2^{n-1}-1$$

gdje je n duljina memorijske ćelije (u bitovima) za pamćenje cjelobrojnih vrijednosti, uključujući predznak. Koji je to točno podskup, ovisi o realizaciji jezika za programiranje i verzije kompjutera. Nad ovim tipom, u većini jezika za programiranje, definirane su standardne operacije cjelobrojne aritmetike: zbrajanje, oduzimanje, množenje, cjelobrojno dijeljenje i potenciranje. Pri izvršavanju tih operacija vrijede svi zakoni aritmetike pod uvjetom da nije došlo do prekoračenja najmanje i najveće moguće vrijednosti cijelog broja (što je određeno duljinom pridružene memorijske ćelije).

REALNI TIP

Realni tip (real) je podskup realnih brojeva, odnosno skup brojeva oblika

$$\pm 0.m \times 2^{\pm e}$$

gdje je m mantisa, a e eksponent. Mantisa je binarni broj duljine n, a eksponent binarni broj duljine k. Preciznost zapisa realnog broja ovisna je o n i k. U nekim jezicima postoji samo jedan prikaz realnih brojeva.

Nad realnim tipom definirane su standardne operacije realne aritmetike (zbrajanje, oduzimanje, množenje, dijeljenje i potenciranje). Zbog ograničene i konačne duljine memorijskih ćelija za pamćenje realnih vrijednosti u memoriji kompjutera, praktički je nemoguće prikazati i u memoriji upamtiti proizvoljnu realnu vrijednost, već samo podskup realnih vrijednosti. Drugim riječima, realni tip na kompjuteru je konačan skup realnih

vrijednosti. Zbog toga se računske operacije ne obavljaju s točnim već s približnim vrijednostima. Posljedica je toga da su rezultati operacija aproksimacija točnih rezultata.

LOGIČKI TIP

Skup vrijednosti logičkoga tipa („logical“) sastoji se od dvije logičke konstante: *true* (istina) i *false* (neistina, laž). U većini jezika za programiranje, u kojima je uveden logički tip podataka, definirane su Booleove operacije – negacija, konjunkcija i disjunkcija, a u nekim jezicima i implikacija i još neke druge operacije.

ZNAKOVNI TIP

Skup vrijednosti znakovnoga tipa („character“, odnosno „char“) određen je alfabetom jezika za programiranje (skupom svih znakova dopuštenih u jeziku za programiranje) uz dodatak svih preostalih znakova dostupnih u operacijskom sustavu u kojem je jezik implementiran, što se definira tablicom `ASCII` kodova. Nad znakovnim tipom najčešće je definirana samo operacija nastavljanja, ali rezultat te operacije nije znak već niz znakova.

IMENA

Na apstraktnoj razini svaki jezik za programiranje koristi objekte: cjelobrojne, realne, logičke, itd. Kao što je već opisano u prethodnom potpoglavlju, memorija kompjutera sastoji se od ćelija koje mogu pamtiti vrijednost bilo kojega tipa. Ćelije su različite duljine, od jednog bajta do nekoliko memorijskih riječi, ovisno o tipu podataka. Svaka ćelija ima ime koje je obično varijabla jezika za programiranje. Ime je riječ koja se tvori prema pravilima leksičke strukture jezika. U većini jezika ime je najmanje jedno slovo ili slovo kojega slijedi niz sastavljen od znamenki i slova.

VARIJABLE I KONSTANTE

Općenito, podaci bilo kojega tipa mogu biti varijable i konstante. Varijabla u matematici ne predstavlja nijednu posebnu vrijednost. Tako je, na primjer, u tvrdnji da za svaki prirodni broj `n` vrijedi:

$n^2>0$

U jezicima za programiranje pojam "varijable" koristi se za nešto što postoji s vremenom i koje u svakom trenutku ima izvjesnu vrijednost. Varijabla je određena svojim imenom i tipom. Tip varijable određuje iz kojeg će se skupa vrijednosti dodjeljivati (pridruživati) varijabli. U nekim jezicima tip varijable može biti definiran početnim slovom, u drugima određenim sufiksom, a u nekima tek eksplicitnom deklaracijom tipa.

Konstante imaju određene vrijednosti koje se ne mijenjaju tokom izvršavanja programa. Zadaju se eksplicitno, pišući konkretnu vrijednost iz skupa svih vrijednosti nekog tipa ili im se pridružuje simboličko ime. Na primjer, `-123` je cjelobrojna konstanta, a `0.505` realna. Važno je primijetiti da u jezicima za programiranje konstante nisu vrijednosti. Bolje ih je zamisliti kao posebnu vrstu varijabli koje svoje vrijednosti steknu prije nego što se program počne odvijati i nikad ih ne mijenjaju.

STRUKTURE PODATAKA

Vrijednosti bez komponenti, koje predstavljaju same sebe i dalje se ne dijele, nazivaju se primitivni tipovi. Nosioci primitivnih tipova su konstante ili varijable koje se mogu nazvati "primitivnim" varijablama. Polazeći od primitivnih tipova moguće je definirati strukturirane tipove - skupove vrijednosti čija struktura ima određeni smisao. Nosioci strukturiranih podataka bit će strukturirane varijable. One se mogu koristiti tako da se odnose na strukturiranu vrijednost u cjelini ili na pojedine komponente. Da bi se definirao strukturirani tip, neophodno je definirati metodu strukturiranja i tipove komponenti. U većini jezika za programiranje susreću se sljedeći strukturirani tipovi podataka:

- polje
- niz
- slog
- datoteka

Pascal ima, pored navedenih, skup i objekte (klase) kao strukturirani tip podataka, a Python ima tzv. „sekvencijalne tipove podataka“ kao što su liste i n-torke.

Polje („array“) je kolekcija elemenata istog tipa (na primjer realnog ili znakovnog) objedinjenih u k-dimenzionalnoj strukturi. Elementi polja imaju isto ime ali k različitih indeksa:

$A\ (i_1, i_2, \ldots, i_k)$

Ovdje je A ime polja, a $i_1, i_2, \ldots, i_k$ su indeksi elemenata polja. Broj k, $k \geq 1$, je dimenzija polja. Na primjer, jednodimenzionalno polje $B(n)$ od n elemenata može se prikazati kao uređena n-torka:

$\{ B(1), B(2), \ldots , B(n) \}$

a dvodimenzionalno polje $C(m,n)$, od $m \times n$ elemenata, kao uređena tablica:

C(1,1)	C(1,2)	...	C(1,n)
C(2,1)	C(2,2)	...	C(2,n)
...	...	...	...
C(m,1)	C(m,2)	...	C(m,n)

Jednodimenzionalno polje naziva se *vektor*, a dvodimenzionalno *matrica*. Polje iz prethodnoga primjera je vektor, a polje C matrica. Svi poznati jezici za programiranje imaju strukturu polja, ali rijetki su oni u kojima su definirane operacije s poljima. Sintakse polja, a djelomično i semantike, razlikuju se od jezika do jezika.

Niz („string“) je kolekcija znakova koja se na razini jezika programiranje tretira kao nedjeljiva cjelina. Zbog toga se niz znakova često promatra kao primitivni tip podataka, nad kojim je najčešće definirana samo operacija nastavljanja. U nekim jezicima, na primjer u Pascalu, niz je znakova poseban slučaj jednodimenzionalnog polja čiji su elementi znakovi. Tako realiziran, niz znakova predstavlja strukturu podataka znakovnog tipa, koja omogućava definiranje operacija nad cjelinom, ali i nad njezinim dijelovima.

Slog („record“) je struktura podataka koju čini uređena kolekcija općenito različitih primitivnih ili strukturiranih tipova podataka. Svaka komponenta sloga ima jedinstveno ime kojim se na nju upućuje. Nad komponentama sloga dopuštene su operacije suglasno njihovom tipu. Slog se, međutim, posebno ulazno-izlaznim operacijama, često tretira kao kompaktna cjelina, što olakšava rad i programiranje te ubrzava izvršenje programa.

Datoteka („file“) je organizirana kolekcija zapisa, obično pohranjena u sekundarnoj memoriji kompjutera. *Zapis* je podatak primitivnog ili strukturiranog tipa. U većini jezika za programiranje razlikuju se dvije vrste datoteka: one u kojima je pristup zapisima sekvencijalan (u nizu) i datoteke s direktnim pristupom zapisima. Tekstualne datoteteke su sekvencijalne. U datotekama se ne čuvaju samo podaci već i softver potreban za rad na kompjuteru kao i korisnički programi.

Skup („set“) je uređena kolekcija podataka istog primitivnog tipa. Na razini jezika za programiranje promatra se kao nedjeljiva cjelina, odnosno kao skup u matematičkom smislu, pa su nad njim definirane skupovne operacije (unija, presjek i razlika).

IZRAZI

Izrazi nisu naredbe već sintaktičke strukture koje, općenito, sadrže operande (varijable, konstante i funkcije) i operacije. Prema tipu vrijednosti izraza, u većini jezika za programiranje razlikujemo aritmetičke (brojčane), znakovne i logičke izraze. Aritmetički izrazi sadrže cjelobrojne i realne operande, te poznate i o tipu ovisne aritmetičke operacije. Znakovni izrazi sadrže nizove znakova kao operande i samo jednu operaciju - nastavljanje. Logički izrazi sadrže relacijske podizraze, logičke konstante i varijable, te logičke operacije.

NAREDBE

Elementarne akcije izračunavanja, dodjeljivanja i kontrole redoslijeda izračunavanja specificiraju se naredbama jezika za programiranje. Naredbe mogu imati različite oblike i značenje. Uobičajena je podjela na:

- primitivne (jednostavne)
- strukturirane (složene) naredbe

Ova podjela odraz je sintakse naredbi. Osim nje, ponekad se naredbe, na temelju svoga značenja, dijele na:

- naredbe za izračunavanje
- naredbe za kontrolu toka izvršavanja
- deklarativne naredbe
- ulazno/izlazne naredbe, itd.

Primitivne naredbe su one koje ne sadrže druge naredbe. To su, na primjer, naredba za dodjeljivanje, naredba za ispisivanje, naredba za čitanje podataka, itd. U nekim jezicima program se može promatrati kao niz primitivnih naredbi (na primjer u jezicima SNOBOL i APL). Strukturirane naredbe (složene ili komponirane) sadrže jednu ili više naredbi. Strukturirane naredbe su, na primjer, naredbe WHILE i REPEAT petlje u Pascalu.

POTPROGRAMI

Potprogrami su također strukturirane naredbe koje sadrže grupu primitivnih i strukturiranih naredbi – cjeline po svojoj funkciji i operacijama koje obavljaju. Dijelimo ih na procedure i funkcijske potprograme.

PROGRAMI

Konačno, stigli smo do najviše hijerarhijske strukture elemenata programa. Program sada možemo definirati kao niz naredbi, primitivnih i složenih, kojim se opisuje postupak ulaza, izračunavanja i izlaza podataka, te rezultat izračunavanja.

Semantika jezika

Kad se zna da je naredba sintaktički korektna, postavlja se pitanje: što je njezino značenje? Pravila koja daju odgovor na to pitanje nazivaju se semantikom jezika za programiranje. Semantiku jezika za programiranje daleko je teže specificirati od njegove sintakse.

Jasna razlika između sintakse i semantike jezika za programiranje bila je prvi put izražena u izvještaju jezika ALGOL 60, 1963. godine. Otad, pa sve do pojave rada Floyda, [Flo1967], bilo je intuitivno prihvaćeno da je semantika naredbe nekog jezika ono što stroj napravi tijekom njezinog izvršavanja. To je tzv. operaciona ili interpretativna semantika. Vrhunac takvog pristupa predstavlja "The Vienna Definition Language". Na primjer, semantika naredbe

```
N := 55
```

u Pascalu, ili

```
N = 55
```

u Pythonu, jest dodjeljivanje (pridruživanje) vrijednosti (broja) 55 varijabli s imenom N. To možemo prikazati sa N←55, a čitat ćemo "N stječe vrijednost 55" ili "Broj 55 dodijeljen je varijabli s imenom N". Ili, semantika naredbe

```
Write (N)
```

u Pascalu, ili

```
print N
```

u Pythonu, bit će ispis vrijednosti varijable N.

Još uvijek ne postoji pogodna notacija za opis interpretativne semantike naredbi jezika za programiranje, pa se to najčešće čini riječima.

Od 1967. godine i spomenutog Floydovog rada semantika se definira kolekcijom aksioma i pravila zaključivanja koji osiguravaju dokaz svojstava programa, posebno da je dani program korektan. To je tzv. aksiomatska semantika. Objektivnost aksiomatske semantike jest formalni sustav koji osigurava da se utvrdi korektnost programa koristeći samo njegov neinterpretirani tekst, bez referiranja na vezu s konkretnim strojem (računalom) i izvršavanjem programa na njemu.

Osnovni Floydovi koncepti razvili su se u dva temeljna pravca čiji su predstavnici Hoare, [Hoa1969], i Manna, [Man1968]. Poseban doprinos definiranju semantike dao je i Dijkstra, [Dij1976], u definiranju svog "mini" algoritamskog jezika. S obzirom na to da smo u ovoj knjizi, u devetom poglavlju, realizirali predprocesor toga jezika, u osmom smo poglavlju dali kompletni opis njegove sintakse i semantike. Nazvali smo ga jezik DDH.

P R I M J E N E

Osim jezika DDH, u sedmom poglavlju ove knjige definiran je i jezik PL/0, poznati "školski" primjer jezika, "malog Pascala", koji sadrži jedanaest rezerviranih riječi, ima samo cjelobrojni tip podataka, cjelobrojne i relacijske izraze, primitivne naredbe: naredbu za pridruživanje i ispis, te složene naredbe selekciju i WHILE petlju.

DEFINICIJA JEZIKA Exp

Ovdje ćemo definirati jezik **Exp**, jezik realnih izraza koji će poslužiti kao primjer mini jezika koji sadrži mali broj riječi, ali dovoljno za potpuno razumijevanje svega onoga što smo iznijeli u ovom poglavlju. Osim toga, jezik **Exp** će nam pomoći i za bolje razumijevanje preostalih poglavlja ove knjige.

Alfabet

Alfabet jezika **Exp** čine sljedeće skupine znakova:

1) brojke: { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
2) slova: { A, B, C, D, E, F, I, L, N, O, P, Q, R, S, T, U, X }
3) operacije: { +, -, *, /, %, ^ }
4) ostali znakovi: { (,) }

Rječnik

Nad alfabetom su definirane sljedeće skupine riječi:

1) Brojevi, napisani prema pravilu:

   broj : ***brojka*** { ***brojka*** } [. ***brojka*** { ***brojka*** }]

   brojka : **0| 1| 2| 3| 4| 5| 6| 7| 8| 9**

2) Imena funkcija: ABS SIN COS EXP LN FRAC INT SQR SQRT ROUND
3) Operacije: **+ - * / % ^**
4) Ostale riječi: **( )**

Leksička pravila

Leksička pravila su sljedeća:

1) Sve se riječi pišu bez razmaka
2) Mala i velika slova imaju jednako značenje
3) Riječi mogu biti razdvojene razmacima

Sintaksa

Osnovna sintaksna struktura jezika **Exp** može se prikazati sljedećim pravilima napisanim u EBNF-u:

```
izraz        : term { op1 term }
term         : faktor { op2 faktor }
faktor       : [ + | - ] operand | ^faktor | funkcija | ( izraz )
funkcija     : ime_funkcije ( izraz )
op1          : + | -
op2          : * | / | %
ime_funkcije : ABS | SIN | COS | EXP | LN | FRAC | INT | SQR | SQRT | ROUND
operand      : broj | funkcija |  (izraz)
```

Evo nekoliko primjera pravilno napisanih rečenica u jeziku **Exp**:

```
1 +2*3  (10 *25.6)*3   2 *6.6 *3.14   exp(ln(2.5))  sin(1) ^2 + cos(1) ^2
```

Semantika

Značenje rečenica jezika **Exp** jest uobičajeno značenje realnih izraza u većini jezika za programiranje: realni broj dobiven izračunavanjem uz slijedeće značenje operacija:

`+`	zbrajanje (ili predznak)	`/`	dijeljenje
`-`	oduzimanje (ili predznak)	`%`	`x % y` ima značenje x posto od y, odnosno `(x /100) *y`
`*`	množenje	`^`	potenciranje, `x^y, x>0`

Sve funkcije imaju jedan argument. Ako ga označimo s x, značenje je:

`ABS`	apsolutna vrijednost od x, $\|x\|$	`FRAC`	decimalni dio realnog broja
`SIN`	$sin(x)$, x je u radijanima	`INT`	cijeli dio realnoga broja
`COS`	$cos(x)$, x je u radijanima	`SQR`	x^2
`EXP`	e^x	`SQRT`	$\sqrt{x}$ (drugi korijen iz x), $x \geq 0$
`LN`	$log_e x$, $x>0$	`ROUND`	zaokružena vrijednost od x (`0.5` se zaokružuje na `1`)

Zadaci

1) Definirajte sintaksu jezika Exp uz pomoć:

a) sintaksnih dijagrama
b) regularnih izraza

3.
PREVODIOCI

— dans mon hamac

sunce se popelo visoko
lagano k'o na jutro uskrsno
a ja ležim ispružen
u svojoj mreži za spavanje

to traje iz godine u godinu
takav je moj znak u zodijaku
možda sam stvarno rođen
u svojoj mreži za spavanje

ponekad zaželim raditi
ali tu je lijenčina u meni što u
protunapad kreće
i u mrežu za spavanje
jastuk mi podmeće

inače vidjevši druge kako rade
dobro shvaćam da ih to uništava
a ja čelična sam zdravlja
u svojoj mreži za spavanje

niti mi je hladno niti vruće
niti sam gladan niti žedan
vjetar mi lagano mrsi kosu
i miluje kožu

novca ipak treba naći
no ja sam strašno snalažljiv
i pustim da mi plate za patent
moje mreže za spavanje

to je izmišljena mreža za uživanje
udobna kao Cadillac
gotovo jedan dom jedno ljubavno gnijezdo
ta moja mreža za spavanje

čim se u zraku osjeti
slatki afrodizijački miris
može se vidjeti potrgano lišće
u mojoj mreži za spavanje

ali iako ima mjesta za jednog
kada smo dvoje sve se mijenja i kida
kad sve uzmem u obzir
jednako nam je dobro
na travi

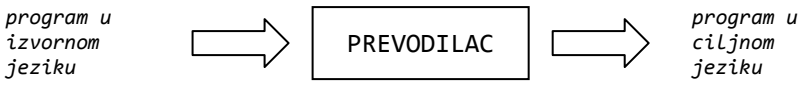
u svojoj mreži za spavanje
dans mon hamac

(georges moustaki/
sanja seljan)

Evolucija jezika za programiranje, od asemblerskog jezika do jezika visoke razine i jezika četvrte generacije, uvela je potrebu za posebnim programima – prevodiocima. S obzirom na to da računalo neposredno izvršava instrukcije u svom, strojnom jeziku, neophodno je prevesti programe u taj jezik, a to radi prevodilac.

3.1 VRSTE PREVODILACA

Prevodilac je program koji instrukcije napisane u obliku programa u izvornom jeziku prevodi u niz semantički ekvivalentnih instrukcija - program izražen u ciljnom ili objektnom jeziku, sl. 3.1.



Sl. 3.1 – *Prevodilac.*

Izvršavanje programa pisanog u jeziku visoke razine u biti je proces koji se izvodi u dva koraka. Prvo se izvorni program "kompilira", tj. prevede u objektni program, potonji se zatim smjesti u memoriju i izvrši. S obzirom da postoji nekoliko vrsta izvornih i objektnih jezika, razlikuje se i nekoliko vrsta prevodilaca.

Ako izvorni jezik pripada klasi jezika visoke razine, kao što su to Pascal i C, a ciljni je asemblerski ili strojni jezik, prevodilac se naziva kompilator.

Asembler je prevodilac (a ne jezik, kako se najčešće misli!) koji prevodi program pisan u asemblerskom jeziku u strojni jezik.

Neki prevodioci transformiraju program pisan u izvornom jeziku u pojednostavljeni jezik, nazvan "međukod", koji se može direktno izvršiti koristeći program zvan interpretator.

Termin predprocesor upotrebljava se za prevodioce koji prihvaćaju programe pisane u jednom jeziku visoke razine i prevode ga u ekvivalentan program u drugom jeziku također visoke razine.

Ovisno o vrsti prevodioca, imat ćemo izvedene riječi koje opisuju njihov postupak prevođenja: kompiliranje, asembliranje, interpretiranje i predprocesiranje. Od pojave PC-a i MS-DOS-a često se govori i o disasemblerima i dekompilatorima.

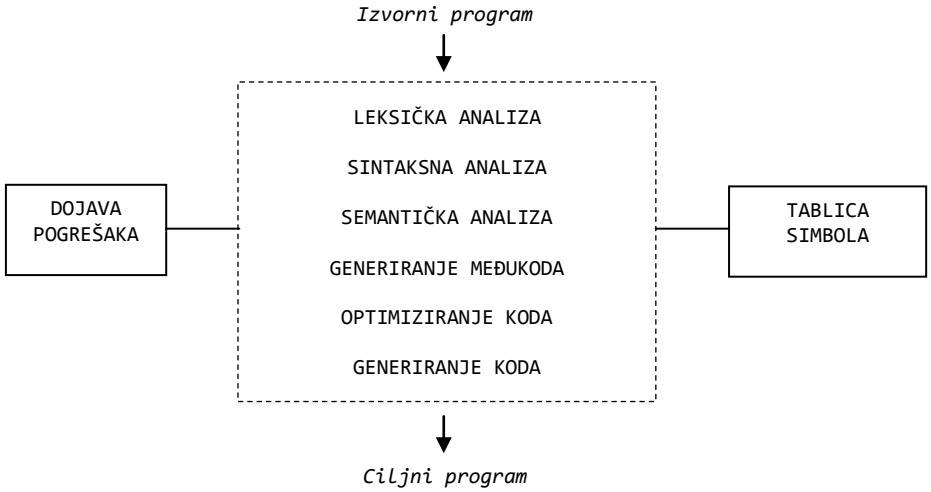
Disasembler je prevodilac koji prevodi program iz strojnog u asemblerski jezik. Dakle, to je inverzna operacija asembliranju. Dekompilator je program (prevodilac) koji prevodi program napisan u strojnom ili asemblerskom jeziku u program jezika visoke razine, odnosno, dekompiliranje je inverzna operacija kompiliranju.

Prvi su jezici za programiranje najčešće bili realizirani kao kompilatori. Kompilator je bio ovisan o stroju na kojem je bio implementiran. Svaki je stroj imao vlastiti operacijski sustav, strojni i asemblerski jezik u koje se generirao kôd, pa nije postojala mogućnost da se kompilatori prenose s jednog na drugo računalo. Na primjer, jezik ALGOL 60 imao je šest kompilatora: Atlas, KDF9, B5500, 1108, CDC 6600 i 1900. Jezik FORTRAN imao je kompilatore na sustavima IBM, UNIVAC, CDC, DECC itd.

Pojavom prvih PC računala početkom osamdesetih godina prošloga stoljeća polako se uvode određeni standardi, a s obzirom na sve veću popularnost PC računala (uvjetovanu niskom cijenom), jezici za programiranje i njihovi prevodioci postali su dostupni velikom broju korisnika. Zajedno s tim dizajniraju se razni „derivati" poznatih jezika za programiranje, koji su svi bili izvedivi na PC računalima, u MS-DOS operacijskom sustavu, UNIX-u ili MS-Windowsima.

3.2 FAZE PREVOĐENJA

Općenito je proces prevođenja ili "kompilacije", ako je riječ o kompilatorima koji su najčešći u realizaciji prevodilaca većine jezika visoke razine i jezika četvrte generacije, isuviše kompleksan, i s logičkog i s implementacijskog aspekta, da bi se mogao razmatrati u jednom koraku. Zbog toga je mnogo podesnije podijeliti ga na niz procesa, nazvanih faze, i promatrati ih odvojeno, kao što je prikazano na sl. 3.2. Svaka je od tih faza prevodilac za sebe.



Sl. 3.2 – *Faze prevođenja.*

Leksička analiza jest prva faza kompilacije. Njezin je zadatak čitanje izvornog programa, znak po znak, i grupiranje nizova znakova u šire leksičke strukture – riječi (simbole). Na primjer, ako je na ulazu niz znakova

```
IF(A<K)and(B<K)THEN MAX:=K
```

onda se u leksičkoj analizi Pascala ovaj niz znakova grupira u niz simbola (odvojenih razmacima):

```
IF ( A < K ) and ( B < K ) THEN MAX := K
```

Sintaksna analiza druga je faza kompilacije. To je proces utvrđivanja može li dani niz simbola (izlaz iz leksičke analize) tvoriti dio programa napisanog u izvornom jeziku. Ako može, simboli će na izlazu iz sintaktičke analize biti grupirani u šire, sintaktičke strukture. Često će to biti predstavljeno u obliku stabla.

Semantička analiza faza je prevođenja povezana sa sintaksnom analizom. U njoj se provjeravaju kontekstni aspekti programa napisanog u izvornom jeziku. Na primjer, ako je ime `Max` deklarirano kao konstanta, naredba

```
Max := K
```

ne bi bila semantički korektna, jer konstanta ne može biti na početku naredbe za dodjeljivanje (ne smije joj se mijenjati vrijednost). U semantičkoj analizi izraza provjeravaju se, na primjer, tipovi operatora, tipovi indeksa polja `i`, ako je to moguće, domena indeksa polja. Na primjer, ako je polje `X` u Pascalu deklarirano kao

```
VAR X : ARRAY [1..10] OF real;
```

i u dijelu programa je napisano

```
X[0] := -1.25
```

bila bi dojavljena pogreška jer je indeks `0` izvan domene indeksa polja `X`.

Generator međukoda koristi strukturu dobivenu na izlazu sintaksne analize da bi kreirao niz primitivnih instrukcija, u nekom međujeziku – apstraktnom jeziku sličnom asemblerskom.

Optimiziranje koda je faza prevođenja u kojoj se međukod "popravlja". Time se dobiva konačni objektni kod koji radi brže i/ili zauzima manje memorijskog prostora. Na primjer, bili su poznati kompilatori FORTRAN-a na nekim strojevima koji su pronalazili invarijantu iteracije (naredba koja se uvijek izvršavala unutar neke petlje koja je uvijek davala isti rezultat neovisno o broju izvršavanja). Na primjer, ako je u dijelu programa napisanom u FORTRAN-u bio niz naredbi:

```
      DO 10 I = 1, 10000
        A = 5.5
        WRITE 20, 1/I
10    CONTINUE
```

Naredba `A=5.5` izvršit će se `10000` puta i svaki put će varijabli `A` biti pridružena vrijednost `5.5`. Kompilatori koji su to mogli prepoznati preuredili bi taj dio programa u fazi optimiziranja u:

```
      A = 5.5
      DO 10 I = 1, 10000
        WRITE 20, 1/I
10    CONTINUE
```

Generiranje koda posljednja je faza prevođenja koja proizvodi objektni kôd uz određivanje memorijskih lokacija za podatke, odvaja kôd za pristup svakom podatku i bira registre za obavljanje pojedinih izračunavanja.

Tablica simbola dio je prevodioca u kojem se vode imena upotrijebljena u programu, sa svim potrebnim dodatnim informacijama. Dostupna je u svim fazama prevođenja. U fazi leksičke analize svako novo ime i pridružena svojstva unose se u tablicu. Na primjer, u leksičkoj analizi deklaracija dijela programa napisanog u Pascalu:

```
VAR A, B, k, M : integer; CONST Max = 999;
```

imena A, B, K i M bila bi upisana u tablicu simbola kao cjelobrojne varijable, a MAX kao cjelobrojna konstanta (velika i mala slova u Pascalu imaju isto značenje, pa smo sva imena upisali velikim slovima). U preostalim fazama prevođenja tablica se simbola koristi u provjeri zadovoljenja određenih svojstava.

U bilo kojoj fazi prevođenja može se otkriti pogreška. Prevodilac na jednom mjestu vodi popis pogrešaka. Pogreške mogu biti leksičke, sintaksne ili semantičke. Leksičke pogreške nastaju prekršenjem leksičkih pravila. Na primjer, ako izvorni program sadrži znak koji nije u alfabetu jezika, ili pri pokušaju ponovnog deklariranja postojećeg imena (u Pascalu) ili pri pojavi nedeklariranog imena u glavnom dijelu programa (u Pascalu). Sintaksne pogreške nastaju prekršenjem sintaksnih pravila (sintakse) jezika. Na primjer, ako realni izraz ne sadrži jednak broj otvorenih i zatvorenih zagrada, ili nedostaje simbol END (u Pascalu), ili kad se pojavi simbol koji ne pripada nizu simbola naredbe izvornoga jezika, kao u nizu simbola:

```
FOR i := 1 TO 10 THEN
```

Ako je to program u Pascalu, onda umjesto THEN treba stajati riječ DO. U Pascalu će, također, biti dojavljena pogreška ako je u selekciji ispred simbola ELSE napisana točka-zarez, itd. Semantičke pogreške bit će dojavljene ako je prekršeno neko od semantičkih (kontekstnih) pravila u programu izvornog jezika. Na primjer, ako je P cjelobrojna, a Q realna varijabla u Pascalu i napisano je P DIV Q bila bi dojavljena pogreška jer je operacija cjelobrojnog dijeljenja definirana samo nad cjelobrojnim operandima.

Ako je prevodilac realiziran kao predprocesor, generiranje međukoda i optimiziranje koda mogu se izostaviti.

3.3 JEDNOPROLAZNO PREVOĐENJE

Prikazane faze prevođenja bile su strogo odvojene i vrijedile su za rane jezike visoke razine, krajem pedesetih pa do početka osamdesetih godina prošloga stoljeća. Svaka je faza prevođenja bila jedan "prolazak", pa se tada koristio pojam "višeprolazno prevođenje". Često su prevodioci imali mogućnost biranja želi li se optimizirati kôd. Tada je to možda bilo važno jer se analizom međukoda i, na primjer otkrivanjem invarijanti u iteracijama, moglo znatno ubrzati izvršavanje programa. Ponekad je to značilo i uštedu od nekoliko sati! A danas? Sigurno nam nije važno izvršava li se neki program par milisekundi ili sekundu niti nam je važno zauzima li memorijski prostor od, na primjer, 400KB ili 100MB!

I popis pogrešaka je stvar prošlosti. Nekad ste, na primjer programirajući u FORTRAN-u, poslije kompilacije svoga programa mogli dobiti par stranica popisa pogrešaka. Čak je među programerima bilo uvriježeno mišljenje da ako je kompilator ispisao više pogrešaka, bio je "savršeniji" ili "pametniji". Na primjer, ako ste u FORTRANu zaboravili deklarirati varijablu `A` sa strukturom polja, a napisali ste je na deset mjesta, imali biste barem deset pogrešaka. Bilo je dovoljno samo napisati:

```
DIMENSION A (100)
```

i više ne bi bilo nijedne pogreške!

Velika većina današnjih prevodilaca dojavi prvu pogrešku na koju naiđe i prekida se daljnji postupak prevođenja.

Dakle, danas govorimo o "jednoprolaznom prevođenju". Faze prevođenja nisu strogo odvojene. U jednom prolasku istovremeno se odvija leksička, sintaksna i semantička analiza, te generiranje koda, a postupak se prekida nailaskom na prvu pogrešku, bilo koje vrste ili dosezanjem kraja izvornog programa. U iduća ćemo tri poglavlja detaljnije opisati te tri faze prevođenja i ukazati na moguće primjene.

P R I M J E N E

Teorija se prevođenja može smatrati dijelom discipline programiranja. Mnogi se problemi teorije algoritama i programiranja mogu "elegantno" riješiti ako se dobro vlada teorijom prevođenja. Ovdje ćemo na primjeru prevođenja rimskih brojeva u arapske prikazati sve faze prevođenja.

PREVOĐENJE RIMSKIH BROJEVA U ARAPSKE

Dosad smo jezik rimskih brojeva definirali gramatikom ili generatorom. U prvom smo poglavlju, primjer 1.14, pokazali kako možemo definirati konačni pretvarač koji bi rimske brojeve prevodio u arapske. Ali, ako rimske brojeve promatramo kao "jezik za programiranje" onda ćemo definirati njegov alfabet, rječnik, pravila pisanja (sintaksu) i njegovo značenje (semantiku).

Alfabet i rječnik

Alfabet rimskih brojeva čini skup znakova:

```
{ I, V, X, L, C, D, M }
```

a rječnik je skup riječi:

```
{ I, IV, V, IX, X, XL, L, XC, C, CD, D, CM, M }
```

Prepoznavač

Promatrajući sve nizove koji tvore rimske brojeve od I do MMMCMXCIX lako možemo izvesti tablicu prijelaza prepoznavača rimskih brojeva u kojoj su prijelazi riječi rječnika rimskih brojeva:

	I	IV	V	IX	X	XL	L	XC	C	CD	D	CM	M
0	15	17	14	17	11	9	10	9	6	4	5	4	1
1	15	17	14	17	11	9	10	9	6	4	5	4	2
2	15	17	14	17	11	9	10	9	6	4	5	4	3
3	15	17	14	17	11	9	10	9	6	4	5	4	
4	15	17	14	17	11	9	10	9					
5	15	17	14	17	11	9	10	9	6				
6	15	17	14	17	11	9	10	9	7				
7	15	17	14	17	11	9	10	9	8				
8	15	17	14	17	11	9	10	9					
9	15	17	14	17									
10	15	17	14	17	11								
11	15	17	14	17	12								
12	15	17	14	17	13								
13	15	17	14	17									
14	15												
15	16												
16	17												
17													

Generiranje koda

U fazi generiranja koda tekuću "znamenku" ćemo prevesti u ekvivaletni arapski broj koji će predstavljati operand cjelobrojnog izraza. Na primjer, rimski broj

```
CMIV
```

bit će preveden u

```
CM IV -> 900 + 4
```

Prevodilac

Sada možemo definirati prevodilac rimskih brojeva koji će učitati niz znakova, prevesti ih u niz simbola, provjeriti jesu li napisani prema danim pravilima i, ako jesu, generirati cjelobrojni izraz sačinjen od "znamenki" rimskih brojeva prevedenih u arapske i operacije zbrajanja. Izračunavanjem generiranog cjelobrojnog izraza dobit će se ekvivalentni cijeli broj – prijevod rimskog broja u arapski. Sve se to obavlja u jednom prolasku. Prevodilac je realiziran u Pythonu.

🖫 Rimski_brojevi.py

```
# -*- coding: cp1250 -*-

# PREVODILAC RIMSKIH BROJEVA U ARAPSKE

# Tablica simbola

V  = { 'I': (0,1),   'IV': (1,4),   'V': (2,5),    'IX': (3,9),
       'X': (4,10),  'XL': (5,40),  'L': (6,50),   'XC': (7,90),
       'C': (8,100), 'CD': (9,400), 'D': (10,500), 'CM': (11,900),
       'M': (12,1000) }
```

```
# Tablica prijelaza

'        I   IV   V  IX   X  XL   L  XC   C  CD   D  CM   M '
Tpa = ( (15, 17, 14, 17, 11,  9, 10,  9,  6,  4,  5,  4,  1 ), # 0
        (15, 17, 14, 17, 11,  9, 10,  9,  6,  4,  5,  4,  2 ), # 1
        (15, 17, 14, 17, 11,  9, 10,  9,  6,  4,  5,  4,  3 ), # 2
        (15, 17, 14, 17, 11,  9, 10,  9,  6,  4,  5,  4, -1 ), # 3
        (15, 17, 14, 17, 11,  9, 10,  9, -1, -1, -1, -1, -1 ), # 4
        (15, 17, 14, 17, 11,  9, 10,  9,  6, -1, -1, -1, -1 ), # 5
        (15, 17, 14, 17, 11,  9, 10,  9,  7, -1, -1, -1, -1 ), # 6
        (15, 17, 14, 17, 11,  9, 10,  9,  8, -1, -1, -1, -1 ), # 7
        (15, 17, 14, 17, 11,  9, 10,  9, -1, -1, -1, -1, -1 ), # 8
        (15, 17, 14, 17, -1, -1, -1, -1, -1, -1, -1, -1, -1 ), # 9
        (15, 17, 14, 17, 11, -1, -1, -1, -1, -1, -1, -1, -1 ), #10
        (15, 17, 14, 17, 12, -1, -1, -1, -1, -1, -1, -1, -1 ), #11
        (15, 17, 14, 17, 13, -1, -1, -1, -1, -1, -1, -1, -1 ), #12
        (15, 17, 14, 17, -1, -1, -1, -1, -1, -1, -1, -1, -1 ), #13
        (15, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ), #14
        (16, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ), #15
        (17, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ), #16
        (-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ) )#17

print 'PREVOĐENJE RIMSKIH BROJEVA U ARAPSKE'

def Poruka (t): global i; print i*' '+chr(24) +' ' +t

while 1 :
  r  = raw_input ()
  if r == '' : break
  r = r.upper() +'@';  Gen = '';  Ok = True;  i = 0;  q = 0
  while Ok and i < len(r)-1:
    # Leksička analiza
    C = r[i]; C2 = r[i+1]
    if C+C2 in V : K, B = V[C+C2]; i += 1
    elif C in V  : K, B = V[C]
    else         : Poruka ('Leksička pogreška');  Ok = False;  break

    # Sintaksna analiza i generiranje koda
    Q = Tpa[q][K]
    if Q <> -1 : Gen += (Gen <> '') *'+' +str(B);  q = Q
    else       : Poruka ('Sintaksna pogreška');  Ok = False

    i += 1

  if Ok : print r[:-1] +' -> ' +Gen, '\n', eval (Gen)
```

Evo nekoliko primjera prevođenja rimskih brojeva u arapske:

```
PREVOĐENJE RIMSKIH BROJEVA U ARAPSKE
i1i
 Leksička pogreška
mmmcM
MMMCM -> 1000+1000+1000+900 = 3900
MmMcMccc
     Sintaksna pogreška
MMMM
   Sintaksna pogreška
```

```
Mili
  Sintaksna pogreška
Mi
MI -> 1000+1
1001
MMMDCCCLXXXVIII
MMMDCCCLXXXVIII -> 1000+1000+1000+500+100+100+100+50+10+10+10+5+1+1+1
3888

>>>
```

Zadaci

1) Jezik rimskih brojeva sadrži konačan broj rečenica (rimskih brojeva), od I *do* MMMCMXCIX*, odnosno od* 1 *do* 3999*. Koristeći sljedeći program*

```
# -*- coding: cp1250 -*-
# PREVOĐENJE RIMSKIH BROJEVA U ARAPSKE ILI ARAPSKE U RIMSKE

M = ['', 'M', 'MM', 'MMM']
C = ['', 'C', 'CC', 'CCC', 'CD', 'D', 'DC', 'DCC', 'DCCC', 'CM']
X = ['', 'X', 'XX', 'XXX', 'XL', 'L', 'LX', 'LXX', 'LXXX', 'XC']
I = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX']

Ra = {}; Ar = {}
for a in range (1, 4000) :
  m = a / 1000; c = (a % 1000) /100;  x = (a % 100) /10;  i = (a % 10)
  R = M[m] +C[c] +X[x] +I[i]
  Ra.update ( { R: a } );  Ar.update ( { a: R } )

while 1:
  R = raw_input ('Upiši rimski ili arapski broj ')
  if R == '' : break

  R = R.upper()
  if R.isalpha() :
    if R in Ra : print R, '->', Ra[R]
    else       : print R, 'nije rimski broj!'
  if R.isdigit() :
    R = int(R)
    if R in Ar : print R, '->', Ar[R]
    else       : print R, 'ne postoji rimski broj!'
```

koji prevodi rimski broj u arapski ili arapski broj u rimski, definirajte prevodilac izraza s rimskim brojevima. Definirane su samo dvije operacije: + – zbrajanje i - – oduzimanje. Rezultat se prikazuje kao rimski broj ili se dojavljuje semantička pogreška ako je rezultat izračunavanja izraza izvan kodomene (od I *do* MMMCMXCIX*).*

4.
LEKSIČKA ANALIZA

– la philosophie

to je zgodno društvo veseljaka
koji liježu u zoru i kasno ustaju
misleći samo na ljubav i sviranje gitare

jedina im je životna filozofija
"imamo čitav život za zabavljanje
imamo cijelu smrt za odmaranje"

ne rade ništa drugo do slavljenja svakoga
trenutka, pozdrava punom mjesecu i
proslave proljeća

jedina im je životna filozofija
"imamo čitav život za zabavljanje
imamo cijelu smrt za odmaranje"

i često se prepoznam u njima
kao i oni tratio sam život na svim vjetrovima
i govorim si da su mi to braća ili djeca

jedina im je životna filozofija
"imamo čitav život za zabavljanje
imamo cijelu smrt za odmaranje

ako prođu kraj vas dobro ih pogledajte
i kao oni budite ludi i kao oni budite pijani
jer njihova jedina ludost je želja da budu
slobodni

jedina im je životna filozofija
"imamo čitav život za zabavljanje
imamo cijelu smrt za odmaranje"

i oni će ostariti, neka ostanu to što jesu
sa starim utopijama na čudne načine
ljubavnika, poeta i onih što spjevaju
pjesme

jedina im je životna filozofija
"imamo čitav život za zabavljanje
imamo cijelu smrt za odmaranje"

filozofija
la philosophie

(georges moustaki/
dubravka stojnić)

Leksička analiza je prva faza prevođenja. U njoj se znakovi ulaznog niza znakova prevode u šire leksičke strukture, a to su riječi ili simboli iz rječnika izvornoga jezika. Program koji obavlja leksičku analizu naziva se leksički analizator, lekser (lexer) ili skener (scanner).

Leksička analiza je važna za proces prevođenja iz više razloga. Pridružujući imenima programa (konstantama, varijablama, itd.) svojstva i jedinstvene oznake znatno se olakša-vaju sljedeće faze prevođenja. U ranim jezicima za programiranje, kao na primjer u FORTRAN-u, u leksičkoj se analizi znatno reducirao izvorni program izbacujući razmake i komentare. Osim toga, niz ulaznih znakova nije se mogao jednoznačno prevesti u niz riječi. Kažemo da je tada leksička analiza bila neizravna ili višeprolazna. Noviji jezici za programiranje imaju izravnu ili jednoprolaznu leksičku analizu. U ovom smo poglavlju opisali te dvije vrste leksičke analize.

4.1 NEIZRAVNA LEKSIČKA ANALIZA

Višeprolaznost neizravne analize uvjetovana je nemogućnošću jednoznačnog prepoznavanja riječi iz rječnika analizirajući ulazni niz znakova. Leksička analiza se izvodi zajedno sa sintaktičkom, koja je također višeprolazna (backtrack). Na primjer, analizirajući ulazni niz u FORTRAN-u:

```
DO 10 I = 1, 100
```

u prvom će prolasku biti izbačeni razmaci, pa imamo niz:

```
DO10I=1,100
```

Dalje će se u leksičkoj analizi, nailaskom na znak "=", pretpostaviti da je `DO10I` realna varijabla koja je dio naredbe za dodjeljivanje. Analizirajući dalje niz, poslije nailaska na zarez, zaključit će se da `1,100` ne može biti prihvaćeno kao realni broj, pa će se vratiti na početak niza. Tada će se analizirati podniz `DO10I` i zaključit će se da je sastavljen od riječi `DO`, početak petlje, broja `10`, labele koja označuje posljednju naredbu petlje, imena `I`, kontrolna varijabla petlje, `1`, početna vrijednost kontrolne varijable i `100` konačna vrijednost kontrolne varijable. Dakle, tek se u drugom prolasku zaključilo da ulazni niz ima strukturu:

```
DO 10 I = 1 , 100
```

gdje smo razmacima razdvojili pojedine riječi.

Iz svega navedenog proizlazi da je neizravna leksička analiza bila prilično neefikasna. Nastojanje da se programerima "olakša" pisanje programa u kojima ne mora pojedine riječi razdvajati razmacima dovela je do složenijeg postupka leksičke analize i povećanja vremena prevođenja što je na tadašnjem stupnju razvoja kompjuterske tehnologije bio veliki nedostatak. Nije se mogao definirati ni općeniti postupak (algoritam) leksičke analize jer je svaki jezik imao određene specifičnosti (svoju sintaksu i semantiku) koje su izravno utjecale na postupak leksičke analize.

4.2 IZRAVNA LEKSIČKA ANALIZA

Može se slobodno reći da je neizravna leksička analiza stvar prošlosti. I pisanje jedne naredbe u jednom redu također je stvar prošlosti. Već pojavom nekih verzija jezika BASIC-a, ALGOL-a 68 krajem šezdesetih, te publiciranjem Pascala, početkom sedamdesetih godina prošlog stoljeća, strogo su odvojene klase riječi u rječniku i uvedeni su delimiteri u tekstu programa.

Drugim riječima, analizirajući tekst programa u samo jednom prolasku moglo se jednoznačno prepoznati sve riječi. Razmak ili blank bio je najčešći delimiter. I kraj je reda bio delimiter pa je bilo dopušteno pisati naredbe programa u više redova. Na primjer, u Pascalu su sljedeća dva niza znakova:

```
if X>=0THEN Write(sqrt(X):0:4)

IF X >= 0 then
  Write (sqrt(X) :0:4)
```

bila interpretirana na jednak način:

```
IF X >= 0 THEN Write ( sqrt ( X ) : 0 : 4 )
```

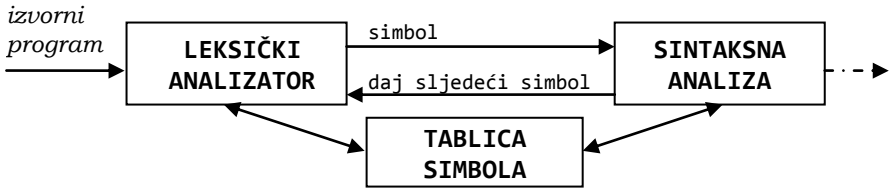
gdje smo riječi odvojili razmacima.

Postupak izravne leksičke analize

Općenito se rječnik jezika za programiranje visoke razine sastoji od sljedećih skupina riječi:

- rezervirane riječi (BEGIN, CALL, CONST, DO, END, IF, ODD, PROCEDURE, THEN, VAR, WHILE, itd.)
- standardna imena (imena tipova podataka, standardnih funkcija i procedura, itd.)
- imena (konstani, varijabli, funkcija, procedura itd.)
- brojevi (cijeli ili realni)
- posebni simboli (:=, >=, <>, ==, != itd.)
- ostali znakovi ((,), +, -, *, /, =, <, >, :, ;, [,] itd.)

Rezervirane riječi, standardna imena, posebni simboli i ostali znakovi definiraju se u posebnoj tablici ili tablicama, dok se imena i brojevi definiraju pravilima tvorbe. Svakom se simbolu iz skupa rezerviranih riječi, standardnih imena, posebnih simbola i ostalih znakova te brojevima pridružuje jedinstveni kôd. Imena se vode u posebnoj tablici koju nazivamo tablica simbola, sl. 4.1.



Sl.4.1 – *Interakcija leksičke i sintaksne analize.*

♣ Primjer 4.1

Analizom niza:

```
VAR A, B, C, A : integer;
```

u jeziku Pascal, bez istovremene semantičke analize, ne bi bila dojavljena pogreška, a ako je semantička analiza uključena, pogreška bi bila dojavljena nailaskom na drugo ime A, jer nije dopušteno pisati ga dvaput u istoj naredbi za deklariranje varijabli. Poslije ispravka, niz:

```
VAR A, B, C : integer;
```

bi bio prihvaćen. U tablicu simbola bila bi upisana imena A, B i C s pridruženim svojstvom da su primitivne varijable cjelobrojnog tipa. Ako poslije toga napišemo:

```
VAR X, Y, A: real;
```

bila bi dojavljena pogreška jer je ime A već bilo definirano.

Ako zanemarimo izvjesna kontekstna svojstva riječi iz rječnika, može se reći da je rječnik jezika za programiranje jezik tipa 3 ili, preciznije, regularni jezik. No, u tablici simbola općenito će se pojedinim imenima pridruživati određena svojstva, kao što su tip, struktura, vrijednost, operacije, funkcije itd. Pritom leksička analiza mora biti u interakciji sa sintaksnom analizom, sl. 4.1.

P R O G R A M I

Konstruiranje leksera ovisno je o jeziku kojeg treba prevoditi. Ovdje dajemo tri jednostavna primjera.

LEKSIČKA ANALIZA RIMSKIH BROJEVA

Alfabet rimskih brojeva čini skup znakova:

```
{ I, V, X, L, C, D, M }
```

a rječnik je skup riječi:

```
{ I, IV, V, IX, X, XL, L, XC, C, CD, D, CM, M }
```

🖫 RIM_Lex.py

```
## -*- coding: cp1250 -*-

# Tablica simbola (rječnik)

V  = { 'I':  1, 'IV':   4, 'V':    5, 'IX':   9, 'X':   10, 'XL':  40, 'L': 50,
       'XC': 90, 'C' : 100, 'CD': 400, 'D':  500, 'CM': 900, 'M': 1000          }

def LA () : # Leksička analiza
  global i
  C = r[i]; C2 = r[i+1]
  if C+C2 in V : i += 1; return True, C+C2;
  elif C in V  : return True, C
  else         : Poruka ('Leksička pogreška');  return False, ''

while 1 : # Testiranje
  r  = raw_input ('Upiši rimski broj ')
  if r == '' : break
  r = r.upper() +'@';  i = 0;
```

```
    while i < len(r)-1:
      Ok, Sym = LA ()
      if Ok : print Sym,
      else  : break
      i += 1
  print
```

Primjeri:

```
   Upiši rimski broj mmmi
   M M M I
   Upiši rimski broj xix
   X IX
   Upiši rimski broj mmmcmxcix
   M M M CM XC IX
   Upiši rimski broj IVIXXLXCCDCM
   IV IX XL XC CD CM
```

LEKSIČKA ANALIZA JEZIKA KEMIJSKIH FORMULA

JKM_Lex.py

```
# -*- coding: cp1250 -*-
import string

VS     = string.ascii_uppercase  # niz velikih slova eng. alfabeta
MS     = string.ascii_lowercase  # niz malih   slova eng. alfabeta
Brojke = string.digits           # niz brojki
Brojke = Brojke [1:]

# Izabrani skup kemijskih elementa
V = {'H', 'He', 'C', 'N', 'O', 'F', 'Na', 'Mg', 'Al', 'Si', 'S', 'Cl', 'Ca', 'Cr',
    'Fe', 'Co', 'Ni', 'Cu', 'Zn', 'Ag', 'I', 'Au', 'Hg', 'Pb' }

def Get_Sym () : # LEKSIČKA ANALIZA
  global i
  i += 1;  C = 0;  S = F[i]
  if S in VS :
    if F[i+1] in MS : i += 1; S += F[i]
    if S in V : C = 1
  elif S in Brojke :
    C = 2
    while F [i+1] in Brojke +'0' : i += 1;  S += F[i]
  elif S in '()' : C = '()'.find (S) +3
  return S, C

F = raw_input ('Upiši kemijsku formulu ') +'#'
i = -1;
while True :
  Sym, Code = Get_Sym()
  if Code == 0 :
    Gr = True; print 'LEKSIČKA POGREŠKA!'
    break
  print Sym,
  if F[i+1] == '#' : break
```

```
Upiši kemijsku formulu C12H22O11
C 12 H 22 O 11

Upiši kemijsku formulu (CH3)(CH2)(OH)
( C H 3 ) ( C H 2 ) ( O H )

Upiši kemijsku formulu H2SO4
H 2 S O 4

Upiši kemijsku formulu NaCl
Na Cl

Upiši kemijsku formulu H2O3
H 2 O 3

Upiši kemijsku formulu (H2O)10(H2SO4)
( H 2 O ) 10 ( H 2 S O 4 )

Upiši kemijsku formulu Ha2O
LEKSIČKA POGREŠKA!
```

LEKSIČKA ANALIZA JEZIKA Exp

Program leksičke analize jezika Exp može se napisati u Pythonu:

Exp_Lex.py

```
# -*- coding: cp1250 -*-
# PROGRAM Leks_ANALIZA
from math import *
import string

Brojke   = string.digits
Slova    = string.ascii_lowercase
Simboli  = ['+','-','*','/','^','(',')']
Funkcije = ['abs', 'sin', 'cos', 'tan', 'exp', 'log', 'sqrt', 'trunc', 'floor']

global Niz, Z, S, Sim, i, Stop, Izl

def Znak() :
  global Niz, i
  Z = Niz[i]
  while Z == ' ':  i += 1;  Z = Niz[i]
  i += 1
  return Z

def Ucitaj()        : return raw_input ('Upiši izraz ')
```

```
def Greska (Poruka, p) :
  print Niz [:-1]
  print (p-1)*' ' + chr(24), Poruka

def Broj () :
  global Sim, i, Z
  while Z in Brojke : Sim += Z;  Z = Znak ()

def Leks_an (Niz) :
  global Z, Sim, i
  i = 0;  Izl = [];  Z = Znak()

  while Z != '#':
    Sim = ''
    if not (Z in Brojke +Slova or Z in Simboli) :
      Greska ('* ilegalan znak', i);
      return False, []
    if Z in Simboli :
      Izl.append (Z); Z = Znak()
    else :
      if Z in Brojke :
        Broj ()
        if Z == '.' :
          Sim += Z; Z = Znak ()
          if Z in Brojke :
            Broj ()
          else :
            Greska ('* Leks.'+ ' pogreška! (nije brojka)', i); return False, []
        Izl.append (Sim)
      else :
        while Z in Slova   : Sim += Z;  Z = Znak ()
        if Sim in Funkcije : Izl.append (Sim)
        else               :
          Greska ('* Leks.'+ ' pogreška! (nepoznato ime "' +Sim +'")',
                  i-len(Sim))
          return False, []
  return True, Izl

def Ispis (L):
  for Sim in L : print Sim,
  print

Niz = Ucitaj ();  Stop = Niz == ''

while not Stop :
  Niz += '#'
  Ok, L = Leks_an (Niz)
  if Ok : Ispis (L)
  Niz = Ucitaj ();  Stop = Niz == ''
```

Evo nekoliko primjera:

```
1)  sin(2)^2+cos(2)^2            sin ( 2 ) ^ 2 + cos ( 2 ) ^ 2

2)  10+-*/33.3)*25               10 + - * / 33.3 ) * 25
```

```
3)  SIN(3.14/2)                 SIN(3.14/2)
                                * ilegalan znak

4)  sin*cos()/789               sin * cos ( ) / 789
```

Primijetimo da ulazni nizovi u primjerima (2) i (4) sadrže simbole jezika Exp, premda ne mogu biti prihvaćeni kao njegove rečenice, na što je program leksičke analize potpuno "ravnodušan". Znamo da je to dio sljedeće faze prevođenja – sintaksne analize.

P R I M J E N E

U teoriji je formalnih jezika gotovo 40 godina poznat računalni program pod nazivom Lex - generator programa za leksičku analizu. Originalni Lex, pod naslovom „Lex – A Lexical Analyzer", napisali su Mike Lesk i Eric Schmidt i objavili 21. srpnja 1975. godine. Radio je u operacijskom sustavu UNIX.

Lex je učitavao pravila koja su definirala leksičku strukturu jezika koji se analizira i na izlazu generirao program leksičke analize u programskom jeziku C.

LEX U PYTHONU

Danas su u uporabi mnoge inačice Lexa. Za nas je posebno interesantna njegova inačica realizirana u Pythonu koju ćemo ovdje ukratko opisati.

```
# ------------------------------------------------------------
# calclex.py
#
# tokenizer for a simple expression evaluator for
# numbers and +,-,*,/
# ------------------------------------------------------------
import lex
T = '\t'

# List of token names.  This is always required
tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN')

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers

def t_newline(t): r'\n+' ; t.lexer.lineno += len(t.value)
```

```
# A string containing ignored characters (spaces and tabs)

t_ignore  = ' \t'

# Error handling rule

def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Test it out
data = '''
3 + 4 * 10
 + -20 *2**5
 + 5/ 66
'''

# Give the lexer some input
lexer.input(data)
print data

# Tokenize

print 'tip', T, 'simbol', T, 'red', T, 'pozicija'
print '-' *36

while True:
    tok = lexer.token()
    if not tok: break      # No more input
    print tok.type, T, tok.value, T, tok.lineno, T, tok.lexpos
```

```
3 + 4 * 10
 + -20 *2**5
 + 5% 66

tip     simbol  red     pozicija
-----------------------------------
NUMBER  3       2       1
PLUS    +       2       3
NUMBER  4       2       5
TIMES   *       2       7
NUMBER  10      2       9
PLUS    +       3       12
MINUS   -       3       14
NUMBER  20      3       15
TIMES   *       3       18
NUMBER  2       3       19
TIMES   *       3       20
TIMES   *       3       21
NUMBER  5       3       22
PLUS    +       4       24
NUMBER  5       4       26
Illegal character '%'
NUMBER  66      4       29
```

```
3 + 4 * 10
 + -20 *2**5
 + 5/ 66

tip     simbol  red     pozicija
-----------------------------------
NUMBER  3       2       1
PLUS    +       2       3
NUMBER  4       2       5
TIMES   *       2       7
NUMBER  10      2       9
PLUS    +       3       12
MINUS   -       3       14
NUMBER  20      3       15
TIMES   *       3       18
NUMBER  2       3       19
TIMES   *       3       20
TIMES   *       3       21
NUMBER  5       3       22
PLUS    +       4       24
NUMBER  5       4       26
DIVIDE  /       4       27
NUMBER  66      4       29
```

PYTHON, REGULARNI IZRAZI I LEKSIČKA ANALIZA

Tokenizaciju (*tokenisation*) ili opojavničenje moglo bi se definirati kao dovođenje korpusa u stanje u kojem su sve riječi-pojavnice identificirane i eksplicitno obilježene, gdje se razlikuju XML/SGML oznake interpunkcije i znamenaka od riječi-pojavnica. U jednostavnom pristupu pojavnice je lako identificirati jer pojavnica je sve ono što se nalazi između dva znaka za obilježavanje razmaka, što najčešće odgovara dvjema bjelinama. Međutim, tokenizacija je u složenijem slučaju mnogo zahtjevnija, jer se pojavnicama mogu smatrati i jedinice koje se sastoje od više riječi (*multi-word units (MWU)*), a povezane su sintaktički ili semantički. Na primjer, datum *20. svibanj* ili *20. 5.* mogao bi biti obrađivan kao jedna jedinica, pa ga u ranijem smislu određenja pojavnice nije moguće tokenizirati. U ovakvom pristupu tokenizacija bi uključivala **prepoznavanje naziva** (*named entity recognition*). Prepoznavanje naziva uključuje obradu teksta pri kojoj se identificiraju izrazi koji su nazivi za npr. ljude, organizacije, datume i sl.

Za jednostavni pristup u ovom trenutku već postoji gotovo rješenje tj. alat 2XML. Osim tokenizacije ovaj se alat pokazao učinkovitim i kod pretvaranja (*conversion*) HTML i RTF datoteka u XML format.

U nastavku dajemo program u Pythonu koji koristi modul `collections` za tokeniziranje teksta. Vrste riječi (pojavnica) zadane su u

```
Token = collections.namedtuple('Token', ['typ', 'value', 'line', 'column'])
```

Regularni izrazi koji ih uparuju dani su u listi n-torki `token_specification`.

```
import collections
import re

Token = collections.namedtuple('Token', ['typ', 'value', 'line', 'column'])

def tokenize(s):
  keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',  r'\d+(\.\d*)?'), # Integer or decimal number
        ('ASSIGN',  r':='),          # Assignment operator
        ('END',     r';'),           # Statement terminator
        ('ID',      r'[A-Za-z]+'),   # Identifiers
        ('OP',      r'[+*\/\-]'),    # Arithmetic operators
        ('NEWLINE', r'\n'),          # Line endings
        ('SKIP',    r'[ \t]'),       # Skip over spaces and tabs
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    get_token = re.compile(tok_regex).match
    line = 1;  pos = line_start = 0;  mo = get_token(s)
    while mo is not None:
        typ = mo.lastgroup
        if typ == 'NEWLINE':
            line_start = pos;  line += 1
        elif typ != 'SKIP':
            val = mo.group(typ)
            if typ == 'ID' and val in keywords: typ = val
            yield Token(typ, val, line, mo.start()-line_start)
        pos = mo.end();  mo = get_token(s, pos)
```

```
        if pos != len(s):
            raise RuntimeError('Unexpected character %r on line %d' %(s[pos], line))

    statements = '''
        IF quantity THEN
            total := total + price * quantity;
            tax := price * 0.05;
        ENDIF;
    '''
    for token in tokenize(statements):
        print(token)
```

Izvršenjem ovog programa bilo bi ispisano:

```
Token(typ='IF',     value='IF',       line=2, column=5)
Token(typ='ID',     value='quantity', line=2, column=8)
Token(typ='THEN',   value='THEN',     line=2, column=17)
Token(typ='ID',     value='total',    line=3, column=9)
Token(typ='ASSIGN', value=':=',       line=3, column=15)
Token(typ='ID',     value='total',    line=3, column=18)
Token(typ='OP',     value='+',        line=3, column=24)
Token(typ='ID',     value='price',    line=3, column=26)
Token(typ='OP',     value='*',        line=3, column=32)
Token(typ='ID',     value='quantity', line=3, column=34)
Token(typ='END',    value=';',        line=3, column=42)
Token(typ='ID',     value='tax',      line=4, column=9)
Token(typ='ASSIGN', value=':=',       line=4, column=13)
Token(typ='ID',     value='price',    line=4, column=16)
Token(typ='OP',     value='*',        line=4, column=22)
Token(typ='NUMBER', value='0.05',     line=4, column=24)
Token(typ='END',    value=';',        line=4, column=28)
Token(typ='ENDIF',  value='ENDIF',    line=5, column=5)
Token(typ='END',    value=';',        line=5, column=10)
```

Zadaci

1) Proširite rječnik jezika kemijskih formula i tablicu simbola na potpuni skup riječi periodnog sustava elemenata.

5.
SINTAKSNA ANALIZA

– le temps de vivre

prepustit ćemo se životu
da bismo bili slobodni
bez planova i navika
o njemu ćemo moći sniti

priđi tu sam
samo tebe čekam
sve je moguće
sve je dopušteno

dođi slušaj ove riječi
što trepere
na zidovima svibnja
što svjedoče nam o istini
da se sve jednoga dana
može izmijeniti

priđi tu sam
samo tebe čekam
sve je moguće
sve je dopušteno

vrijeme življenja
le temps de vivre

(georges moustaki/
zdravko dovedan han)

Sintaksna analiza je druga faza prevođenja. Primarni joj je zadatak da analizira, odnosno prepoznaje rečenice (programe) i rečenične strukture jezika za programiranje. Ne postoji općeniti postupak sintaksne analize jezika za programiranje. U prethodnoj smo knjizi, [Dov2012b], opisali desetak postupaka parsiranja beskontekstnih jezika i nekoliko postupaka prepoznavanja jezika svih razina. Samo bi se jedan dio tih postupaka, uz određenu nadogradnju, mogao primijeniti u sintaksnoj analizi jezika za programiranje.

U ovom smo poglavlju opisali samo dva postupka sintaksne analize jezika za programiranje: rekurzivni spust, koji pripada klasi jednoprolaznih silaznih postupaka parsiranja, i prepoznavač jezika sa svojstvima upravljan tablicom prijelaza i akcija.

5.1 REKURZIVNI SPUST

Često se koristi sintaksna analiza "s rekurzivnim spustom", koju je publicirao Wirth u sintaksnoj analizi jezika **PL/0**, [Wir1976]. Moglo bi se reći da je to pravi "školski primjer" realizacije problema SA *LL*(1) jezika.

Definicija *LL*(1) jezika dana je u [Dov2012b]. Tamo su definirane i funkcije $\mathtt{FIRST}_1$ i $\mathtt{FOLLOW}_1$, pa je gramatika $\mathcal{G}$ tipa *LL*(1) ako i samo ako za svaki par A-produkcija $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$ vrijede sljedeći uvjeti:

1) $\mathtt{FIRST}_1(\alpha_1), \mathtt{FIRST}_1(\alpha_2), \ldots, \mathtt{FIRST}_1(\alpha_n)$ su disjunktni po parovima.
2) Ako je $\alpha_i * \Rightarrow \varepsilon$, tada je $\mathtt{FIRST}_1(\alpha_j) \cap \mathtt{FOLLOW}_1(A) = \emptyset$, za $1 \le j \le n$, $i \ne j$.

Isto će vrijediti i u definiciji osnovne sintaksne stukture nekog jezika za programiranje samo će se ovdje "1" odnositi na jedan simbol umjesto jedan znak.

Nacrt parsera

Ako je jezik kojeg treba analizirati tipa *LL*(1), moguće je konstruirati parser koji će izvoditi sintaksnu analizu postupkom rekurzivnog spusta. Ako je prvi uvjet ispunjen, program sintaksne analize pratit će sintaksnu strukturu jezika prikazanu uz pomoć produkcija u BNF-u ili sintaksnih dijagrama.

Wirth u svojoj knjizi, [Wir1976], pokazuje kako se iz dane gramatike beskontekstnog jezika predočene u BNF-u mogu izvesti ekvivalentni sintaksni dijagrami, a potom iz njih konstruirati odgovarajući postupak SA – parser koji će analizirati ulazni niz simbola (ili riječi) dobivenih na izlazu leksičke analize. Također pokazuje kako se takav parser može dopuniti (proširiti) kontekstnim aspektima jezika za programiranje i primijeniti u postupku prevođenja. Ako pretpostavimo da su produkcije neterminala A, $A \to \alpha_1|\ldots|\alpha_n$, i ako je Sym tekući simbol (za razliku od beskontekstnih jezika gdje je to bio znak), dio postupka sintaksne analize s rekurzivnim spustom koji se odnosi na A možemo općenito prikazati ovako, [Dov2012b]:

```
def A():
  if   Sym in FIRST(alfa1 +FOLLOW(A())):
    " Kod za alfa1 "
  elif Sym in FIRST(alfa2 +FOLLOW(A())):
    " Kod za alfa2 "
    ...
  elif Sym in FIRST(alfaN +FOLLOW(A())):
    " Kod za alfaN "
```

To znači da će se sparivanjem tekućeg simbola s jednim od očekujućih pozvati odgovarajuća procedura. Ako to nije moguće, bit će dojavljena pogreška.

Postupak sintaksne analize rekurzivnim spustom, kao što slijedi iz njegova imena, sadrži rekurzije. Otuda i zaključak da će se najbolji učinci postići njegovom ustrojbom u nekom jeziku za programiranje koji dopušta rekurzije. Tada je moguće iz sintaksnih dijagrama izravno izvesti program sintaksne analize danog jezika. Ako se postupak ustroji u jeziku koji ne dopušta rekurzije, treba uložiti dodatni napor za eliminiranje rekurzija, za što je potrebno uvesti pomoćne varijable i strukture podataka. Dakako, taj problem je davno nadiđen jer su više od četrdeset godina u upotrebi jezici za programiranje koji sadrže rekurzivne pozive procedura i funkcija (HP BASIC, Pascal, C, Python itd).

Stablo sintaksne analize

Postupak će sintaksne analize prevesti ulazni niz simbola, ako ne bude otkrivena nijedna pogreška, u neku strukturu podataka na izlazu. Najčešće je to stablo koje se u ovom slučaju naziva stablo sintaksne analize. Na primjer, ako su naredbe u nekom jeziku za programiranje definirane produkcijama:

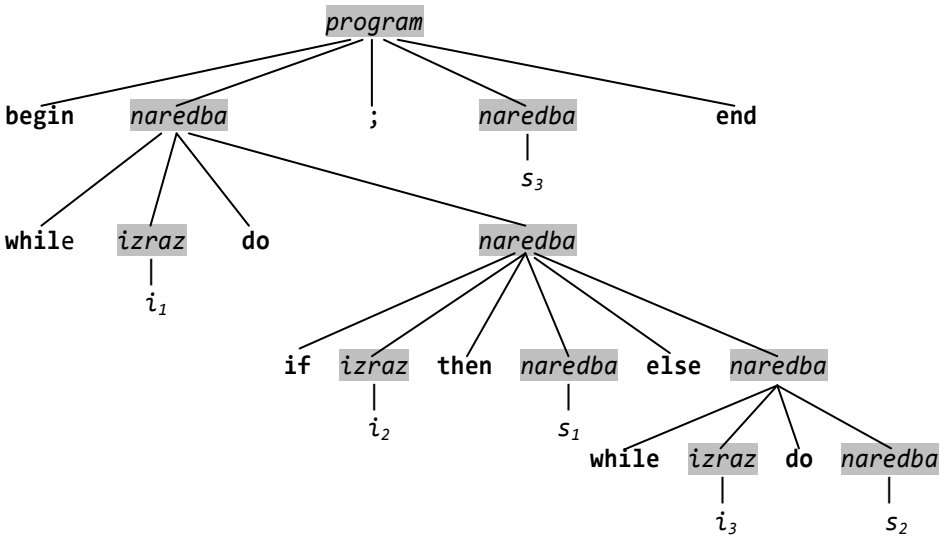
program : **begin** *naredba* { **;** *naredba* } **end**

naredba : *pridruživanje* | *ispis* |
if *izraz* **then** *naredba* [**else** *naredba*] |
while *izraz* **do** *naredba* |

stablo sintaksne analize ulaznog niza simbola

```
begin
  while i1 do
    if i2 then s1
    else
      while i3 do s2;
  s3
  end
```

može biti prikazano kao na sl. 5.1.



Sl. 5.1 – *Stablo sintaksne analize.*

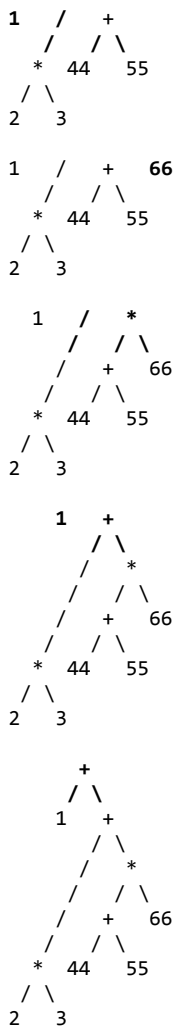
IZRAZI

Izrazi su sintaksne kategorije koje se najčešće pojavlju kao dio mnogih primitivnih i složenih naredbi jezika za programiranje. Uobičajeno je da se na izlazu sintaksne analize generira stablo sintaksne analize u kojem će čvorovi biti označeni operacijama i funkcijama, a listovi operandima. Na primjer, ako imamo izraz i njegov prijevod u postfiksni izraz:

```
1+2*3+(44+55)*66 ➔ 1, 2, 3, *, 44, 55, +, 66, *, +, +
```

gdje smo operande i operacije razdvojili zarezom, isti se izraz može prikazati stablom sintaksne analize kojeg smo izgradili uzlazno:

```
1

1   2

1   2   3

  1   *
     / \
    2   3

  1   *  44
     / \
    2   3

  1   *  44   55
     / \
    2   3
```



Krenuli smo slijeva generiranog postfiksnog izraza i u svakom smo koraku dopisivali sljedeći operand na istoj razini, a ako je simbol bio operacija, povećali smo razinu za jedan i prikazali je između prethodna dva operanda.

Rezultirajuće stablo sintaksne analize "čitamo" od korijena prema listovima, slijeva nadesno, zapisujući operacije i u zagradi operande:

```
+(1, +(*(2,3), *(+(44, 55), 66)))
```

Time se dobiva prefiksni zapis originalnog izraza:

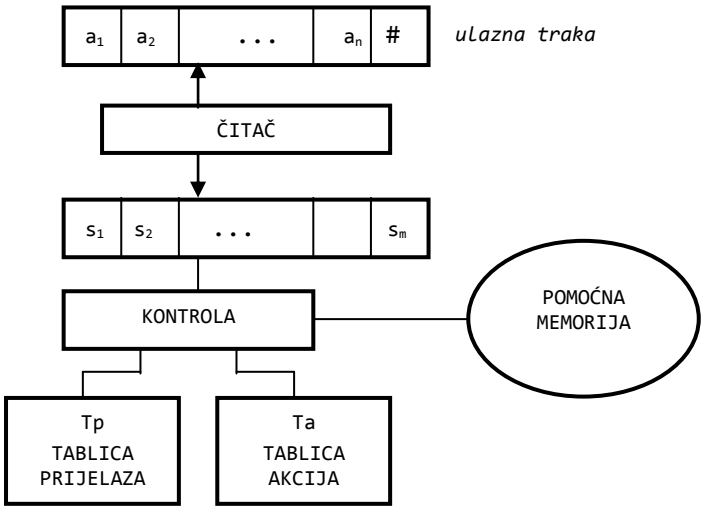
```
add(1,add(mpy(2,3),mpy(add(44,55),66)))
```

gdje smo operacije + i * zamijenili pozivima funkcija `add` i `mpy`.

Primjena sintaksne analize s rekurzivnim spustom dana je u sedmom poglavlju kao dio interpretatora jezika **PL/0**. U dijelu *PROGRAMI* ovoga poglavlja dali smo primjenu sintaksne analize s rekurzivnim spustom u sintaksnoj analizi jezika **Exp**.

5.2 PREPOZNAVAČ JEZIKA SA SVOJSTVIMA

Definiciju jezika sa svojstvima i njegova prepoznavača dali smo u prethodnoj knjizi [Dov2012b, str. 155].



Sl. 5.2 - *Model prepoznavača jezika sa svojstvima.*

Jezik sa svojstvima jest jezik čiji je generator zadan kao uređena osmorka:

$\mathcal{J} = (Q, \Sigma, \mathcal{V}, \delta, q_0, \mathcal{F}, \alpha, \mathcal{M})$

gdje su:

- Q konačan *skup stanja* (kontrole završnog stanja)
- Σ *alfabet*
- $\mathcal{V}$ *rječnik*, $\mathcal{V} \subseteq \Sigma^+$
- δ *funkcija prijelaza*, definirana kao $\delta: Q \times \mathcal{V} \cup \{@\} \to P(Q)$ gdje je $P(Q)$ particija od Q; $@$ je oznaka kraja ulaznog niza
- q_0 *početno stanje*, $q_0 \in Q$
- $\mathcal{F}$ *skup završnih stanja*, $\mathcal{F} \subseteq Q$
- α *skup akcija* pridruženih svakom paru (q_i, s_j), $q_i \in Q$, $s_j \in \mathcal{V}$, za koji je definirana funkcija prijelaza
- $\mathcal{M}$ *pomoćna memorija*

Funkcijom prijelaza δ (primijetimo da je definirana nad rječnikom) zadaju se sve moguće promjene stanja. Akcija je, općenito, postupak koji, ovisno o tekućem stanju q_i i informacijama dobivenih od pomoćne memorije, reducira broj mogućih prijelaza iz stanja q_i, zadanih funkcijom δ, u trenutačno ostvarive prijelaze. I ne samo to, akcija može promijeniti vri-

jednosti funkcije prijelaza. Ako je prijelaz iz stanja q_i u stanje q_j prijelazom s_k uvijek ostvariv, smatrat ćemo da je takvom prijelazu pridružena prazna akcija. Generator regularnih jezika primjer je u kojem su svi prijelazi uvijek ostvarivi. Stoga generator regularnih jezika nema potrebe za pomoćnom memorijom.

U ustrojbi prepoznavača jezika sa svojstvima kontrolu konačnog stanja i dalje ćemo prikazivati dijagramom (tablicom) prijelaza, ali ćemo svakom prijelazu dodati kôd njemu pridružene akcije sa značenjem:

`0` (ili izostavljeno)	- prazna akcija
`1, 2, ...`	- neprazna akcija

Model prepoznavača jezika sa svojstvima (jezika za programiranje) prikazan je na sl. 5.1.

POMOĆNA MEMORIJA

Zamislit ćemo da pomoćnu memoriju čine primitivne i strukturirane varijable svih tipova, kao što je to, na primjer, u Turbo Pascalu ili Pythonu. Inicijalno će te varijable sadržavati određene vrijednosti.

ČITAČ

Čitač je jednostavni pretvarač (program leksičke analize) koji prihvaća niz ulaznih znakova, kojima je na kraju dodan znak ¨@¨, i prevodi ih u niz simbola. Svakom simbolu pridužen je jedinstveni cjelobrojni kôd, od `1` do `k`, ako rječnik sadrži `k` simbola, te `k+1` za znak ¨@¨.

SINTAKSNA ANALIZA

Ako je `C` kôd učitanog simbola, `S` tekuće stanje kontrole konačnog stanja i `Tp` tablica prijelaza, mogući prijelaz `Ss` zadan je s:

```
Ss := Tp [S][C]
```

a akcija kodom na mjestu `Ta[S][C]`, gdje je `Ta` tablica akcija. Tada se kontrola završnog stanja prepoznavača jezika sa svojstvima (postupak sintaksne analize) može prikazati sljedećim dijelom programa:

```
S = 1;  Kraj = False;  Pogreska = False
while not Kraj and not Pogreska:
  Ucitaj_Sim; Ss = Tp [S-1][C]
  if C == 0 or Ss == 0:
    print '* Sintaksna pogreška'; Pogreska = True
  else:
    a = Ta [S-1][C]
    if   a == 0 :
      '* prazna akcija *'
      S = Ss
    elif a == 1:
      '* Akcija broj 1 *'
      _1 ()
    ...
    elif a == n:
      '* Akcija broj n *'
      _n ()
```

Najprije će u posebnoj proceduri biti inicijalizirane varijable programa, tablica prijelaza i tablica akcija. Procedura `Ucitaj_Sim` učitat će tekući simbol i vratiti njegov kôd, `C`. Ako je kôd različit od `0` i ako je definirano naredno stanje, izvršit će se odgovarajuća akcija. Akcija može biti prazna, tada se bezuvjetno prelazi u naredno stanje. Neprazna se akcija izvodi u posebnoj proceduri.

Postupak se sintaksne analize nastavlja sve do dosezanja kraja ulaznog niza i njegova prihvaćanja ili se prekida ako je učitan simbol koji nije u rječniku ili je pronađena sintaksna pogreška.

Funkcija prijelaza bilo kojeg automata jest statična, nepromjenjiva. S obzirom na to da je akcija prema našoj definiciji općenito naredba ili niz naredaba u jeziku implementacije prepoznavača jezika sa svojstvima, moguće je definirati takve akcije koje će mijenjati inicijalnu definiciju tablice prijelaza i/ili tablice akcija. Kažemo da je takva tablica prijelaza i akcija dinamička. Uporabom dinamičkih tablica prijelaza i akcija, kao što ćemo kasnije pokazati, može se znatno smanjiti kôd programa prepoznavača uz povećanje njegove čitljivosti.

P R O G R A M I

SINTAKSNA ANALIZA JEZIKA Exp

Dajemo dva rješenja problema sintaksne analize jezika Exp: parser i prepoznavač.

Parser

Parser predstavlja realizaciju jednoprolaznog postupka sintaksne analize kojeg smo koristili u sintaksnoj analizi beskontekstnih jezika, rekurzivnog spusta. S obzirom na to da je Exp jezik sa svojstvima, bilo je neophodno proširiti značenje rekurzivnog spusta, dodajući mu, prije svega, leksičku analizu.

Exp-RS.py

```
# -*- coding: cp1250 -*-
# PROGRAM SINTAKSNA_ANALIZA
"""
   Sintaksna analiza (rekurzivni spust - recursive descent) i prevođenje u
   posfiksnu notaciju realnih izraza napisanih prema sintaksi:

   izraz    : term { [+|-] term }
   term     : faktor { [ * | / ] faktor }
   faktor   : [+|-] broj | ^faktor | funkcija | ( izraz )
   funkcija : ime ( izraz )
   ime      : abs | sin | cos | tan  | exp | log | sqr | sqrt | trunc
   broj     : brojka { brojka } [ . brojka { brojka } ]
   brojka   : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
"""

from math import *
import string
```

```
def sqr(x): return x**2

Brojke   = list (string.digits)
Slova    = list (string.ascii_lowercase)
Simboli  = ['+','-','*','/','^','(',')']
Alfabet  = Brojke +Slova +Simboli
Funkcije = ['abs', 'sin', 'cos', 'tan', 'exp', 'log', 'sqr', 'sqrt', 'trunc']

def Znak() :
  global Niz, i, Z
  while Niz[i] == ' ' : i += 1
  Z = Niz[i]; i += 1
  return

def Greska (Poruka, p) :
  global Gr
  print Niz [:-1];  print (p-1)*' ' + chr(24), Poruka
  Gr = True
  return

def Broj () :
  global Sim, i, Z
  while Z in Brojke :  Sim += Z;  Znak ()

def Leks_an () :
  global Niz, Z, S, Sim, i, R, Gr

  if Z == '#' : S = Z;  return

  S = ' ';  Sim = ''
  if Z not in Alfabet : Greska ('* ( 1) Ilegalan znak', i);  return

  if Z in Simboli : S = Z; Znak()
  else :
    if Z in Brojke :
      Broj ()
      if Z == '.' :
        Sim += Z; Znak ()
        if Z in Brojke : Broj ()
        else           :
          Greska ('* ( 2) Leksička pogreška! (nije brojka)', i)
          return
      S = 'B'
    else :
      while Z in Slova : Sim += Z;  Znak ()
      if Sim in Funkcije : S = 'F'
      else               :
        Greska ('* (3) Leksička pogreška! (nepoznato ime "' +Sim +'")',
                i-len(Sim))
        return
  return

def Izraz () :
  global Niz, Z, S, Sim, i, R, Gr, F, M, Q

  def Term () :
    global Niz, Z, S, Sim, i, R, Gr
```

```
    def Faktor () :
      global Niz, Z, S, Sim, i, R, Gr

      '<faktor> : [+|-] <broj> | ^<faktor> | <funkcija> | ( <izraz> )'
      if S == 'B' :
        R.append (float(Sim)); Leks_an()
      elif S == '(' :
        Leks_an (); Izraz ();
        if S <> ')' : Greska('* ( 5) Sintaksna pogreška', i); return
        else        : Leks_an ()
      elif S == 'F' :
        F = Funkcije.index (Sim); Leks_an ()
        if S == '(' :
          Leks_an (); Izraz()
          if S == ')' :
            R.append (Funkcije[F])
            Leks_an ()
          else :
            Greska ('* ( 6) Sintaksna pogreška', i)
            return
        else :
          Greska ('* ( 7) Nepotpuni izraz', i)
          return
      elif S in ['+','-'] :
        Pr = S; Leks_an (); Faktor ()
        if Pr == '-' : R.append ('#')
        else         : Greska ('* ( 8) Nepotpuni izraz', i);     return
      else           : Greska ('* ( 9) Sintaksna pogreška', i);  return

      while S == '^' :
        Leks_an ();  Faktor ()
        R.append ('**')
      return

    "<Term> : <faktor> { [ * | / ] <faktor> }"
    Faktor ();
    while S in ['*','/'] :
      M = S; Leks_an (); Faktor()
      if not Gr : R.append (M)
      else      : break
    return

  "<Izraz> : <term> { [+|-] <term> }"
  Term ()
  while S in ['+', '-'] :
    Q = S;  Leks_an ();  Term ()
    R.append (Q)
  return

def Ispis () : print ' ->', R

print 'Upiši izraz:'
while True :
  print
  Niz = raw_input ()
  if Niz == '' : break
```

```
  Niz += '#';  i = 0;  Gr = False;  R = [];  Znak()
  Leks_an (); Izraz ()
  if not Gr and S == '#': Ispis ()
  elif not Gr           :
    j = len(Sim);  j += (j == 0)*1;  Greska ('* (11) Sintaksna pogreška', i -j)

Upiši izraz ili slovo P za testne primjere: p

1+2*(30
1+2*(30
       * ( 5) Sintaksna pogreška

1*2+3*4+5*6
 -> [1.0, 2.0, '*', 3.0, 4.0, '*', '+', 5.0, 6.0, '*', '+']

 (((11+22)*3 -44)*5 -66)*7 -88
 -> [11.0, 22.0, '+', 3.0, '*', 44.0, '-', 5.0, '*', 66.0, '-', 7.0, '*', 88.0, '-']

111 + 11^3.5
 -> [111.0, 11.0, 3.5, '**', '+']

sin(2)^2 + cos(2)^2
 -> [2.0, 'sin', 2.0, '**', 2.0, 'cos', 2.0, '**', '+']

sqrt(144)
 -> [144.0, 'sqrt']

1 / log(1)
 -> [1.0, 1.0, 'log', '/']

sqrt(2) + sqrt(-2)
 -> [2.0, 'sqrt', 2.0, '#', 'sqrt', '+']

log (-10)
 -> [10.0, '#', 'log']

4*sqrt(1+2+3+4+5)
 -> [4.0, 1.0, 2.0, '+', 3.0, '+', 4.0, '+', 5.0, '+', 'sqrt', '*']

trunc (abs(sin(3.33)*5) /4 + exp(4)
trunc (abs(sin(3.33)*5) /4 + exp(4)
                                  * ( 6) Sintaksna pogreška

trunc (abs(sin(3.33)*5) /4 + exp(4))
 -> [3.33, 'sin', 5.0, '*', 'abs', 4.0, '/', 4.0, 'exp', '+', 'trunc']

sin(60 * 3.14159265/180)
 -> [60.0, 3.14159265, '*', 180.0, '/', 'sin']

sqr(sin(5)) +sqr(cos(5))
 -> [5.0, 'sin', 'sqr', 5.0, 'cos', 'sqr', '+']
>>>
```

Prepoznavač

S obzirom na to da je **Exp**, prema našoj definiciji, jezik sa svojstvima, moguće je konstruirati njegov prepoznavač. Procedura `Tree()` "crta" stablo sintaksne analize.

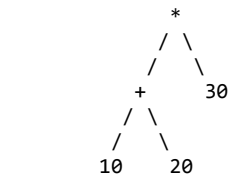
Exp_post.py

```
# -*- coding: cp1250 -*-
# PROGRAM Infix_postfix
#         SA izraza, prevođenje u postfiks notaciju i generiranje stabla SA

from fun import *

Operator = ['+', '-', '*', '/', '^'];  Zn = 'O()BF#@'

""" Tablice prijelaza i akcija
            O   (   )   B   F   -   @
            0   1   2   3   4   5   5  """
Tp  = ( (-1,  0, -1,  1,  2,  0, -1),
        ( 0, -1,  1, -1, -1, -1,  0),
        (-1,  0, -1, -1, -1, -1, -1) )

Ta  = ( ( 0,  1,  0,  0,  0,  4,  0),
        ( 0,  0,  2,  0,  0,  0,  3),
        ( 0,  1,  0,  0,  0,  0,  0) )

Fun = ['abs', 'sin', 'cos', 'tan', 'exp', 'log', 'sqr', 'sqrt', 'trunc']

def SA_Tpa (X):

  PO  = { '-' : 0, '+' : 0,  '*' : 1, '/' : 1,  '^' : 2, '#' : 4 } # prioritet
  for x in Fun : PO.update ( { x : 3 } )

  def push (Sym, L)    : L.append (Sym);  return L
  def pop  (L)         : x = L.pop(); return x, L
  def PopPush (L1, L2) : sym, L1 = pop(L1);  L2 = push(sym, L2); return L1, L2

  X = komp (X).lower()
  N, X = X, X +'@';  S = 0;  b = 0
  Pogr = False;
  L1 = [];  L2 = [];
  Z = '(';  Z1 = ')';  i = 0
  while i < len (X):
    C = X[i];  K = -1
    if S == 0 and C == '-' : C = '#'

    if   C in Operator : K = 0
    elif C in B        :
      while i+1 < len(X) :
        c = X[i+1]
        if c in B: i += 1; C += c
        else     : break
      if X[i+1] == '.' :
        C += X[i+1]; Pogr = True
        i += 1
        while i+1 < len(X) and X[i+1] in B : Pogr = False; i += 1; C += X[i];
      if not Pogr : K = 3
    elif C in a :
      while i+1 < len(X) and X[i+1] in a: i += 1; C += X[i]
      if C in Fun : K = 4
    elif C in Zn  : K = pos (C, Zn)
    Pogr = K == -1
```

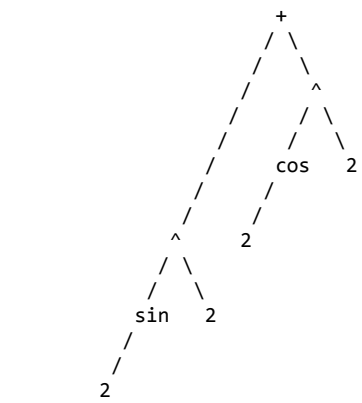
```
      if not Pogr:
        A = Ta[S][K]; S = Tp[S][K];  Pogr = S == -1
        if not Pogr and A != 0:
          if   A == 1 : b += 1
          elif A == 2 :
            if b > 0  : b -= 1
            else      : Pogr = True
          elif A == 3 : Pogr = b > 0
      if Pogr : print i*' ' +chr(24), 'POGREŠKA'; return False, []

      if K == 3 :  L2 = push (C, L2)
      elif C in PO or C == Z:
        if L1 == [] or C == Z: L1 = push (C, L1)
        else        :
          L = L1[-1]
          if L in PO and PO [L1[-1]] >= PO[C] : L1, L2 = PopPush (L1, L2)
          L1 = push (C, L1)
      elif C == Z1 :
        while L1 <> [] and L1[-1] <> Z:  L1, L2 = PopPush (L1, L2)
        L1 = L1[: len(L1)-1]
      i += 1

    while L1 <> []: L1, L2 = PopPush (L1, L2)
    return True, L2

  def Tree (PF) : # Stablo SA
    n = len (PF); T = ['']
    j = -1
    for x in PF : # Izgradi stablo SA s desnim granama
      j += 1
      if x in Operator :
        k     = ' '*(len(T[0])-5-len(PF[j-1]) -len(x)/2)
        T     = [k+'  '] +[k +' / \\'] +[k +'/   \\'] +T
        T[0] += x
      elif x == '#' or x in Fun :
        k     = ' '*(len(T[0])+1-len(PF[j-1]) )
        T     = [k+'  '] +[k +' /'] +[k +'/'] +T
        T[0] += x
      else :
        k = 5
        # if j > 0 and PF[j-1] in Fun : k = k -len (PF[j-1]) +1
        if j > 0 : k = k -len (PF[j-1]) +1
        T[0] += ' '*k +x

    for i in range (0, len(T) -2, 3) : # dopuni stablo SA slijevim granama
      k = 0;  t = T[i]
      while t[k] in [' ', '/'] :
        if t[k] == '/' and T[i+1][k+1] == '\\' : break
        k += 1
      k -= 3; j = i+3
      while j < len(T) -1 :
        x = T[j]
        if x[k] <> ' ' : break
        x = x[:k] +'/' +x[k+1:]
        T[j] = x
        j += 1;  k -= 1
    return T
```

Evo i glavnog programa za poziv prepoznavača jezika Exp :

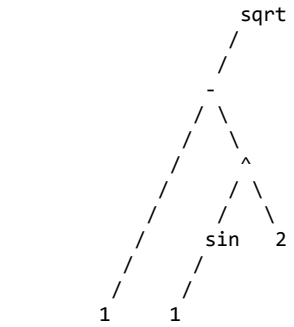
```
# -*- coding: cp1250 -*-

from Exp_post import *

print '\n\nUpiši izraz (Enter za prekid):'

while 1 :
  N = raw_input ()
  if len(N) == 0 : break

  Ok, P = SA_Tpa (N)
  if Ok :
    print ' ->', P, '\n'

    T = Tree (P)
    for x in T : print x
  print


Upiši izraz (Enter za prekid):
10+20+30
 -> ['10', '20', '+', '30', '+']

          +
         / \
        /   \
       +     30
      / \
     /   \
    10    20

10*20+30
 -> ['10', '20', '*', '30', '+']

          +
         / \
        /   \
       *     30
      / \
     /   \
    10    20

10+20*30
 -> ['10', '20', '30', '*', '+']

          +
         / \
        /   \
       /     *
      /     / \
     /     /   \
    10    20    30

(10+20)*30
 -> ['10', '20', '+', '30', '*']
```




```
sin(2)^2 + cos(2)^2
 -> ['2', 'sin', '2', '^', '2', 'cos', '2', '^', '+']
```




```
sqrt(1-sin(1)^2)
 -> ['1', '1', 'sin', '2', '^', '-', 'sqrt']
```



Zadaci

1) Definirajte parser i prepoznavač jezika logičkih izraza.

6.
GENERIRANJE KODA

- sans la nommer

Želio bih vam govoriti o njoj
bez spominjanja imena njezina
kao o jednoj ljubljenoj
jednoj nevjernoj
djevojci prilično živahnoj
koja se budi za sutrašnjice
koje pjevaju pod suncem

To je ona koju se pendrekom mlati
koju se progoni, na koju se hajka diže
to je ona koja se pridiže
koja trpi i koja štrajka
to je ona koju se u zatvor baca
koja ne odaje i ne napušta
koja nam daje želju za životom
koja nam potiče volju da je slijedimo
do kraja, do kraja

Želio bih vam govoriti o njoj
odati joj počast
prekrasnom cvijetu svibnja
ili divljem plodu
jednoj biljci dobro posađenoj
na svoje dvije noge
i koja je potka slobode
ili dobra što čini

To je ona koju se pendrekom mlati
koju se progoni, na koju se hajka diže
to je ona koja se pridiže
koja trpi i koja štrajka
to je ona koju se u zatvor baca
koja ne odaje i ne napušta
koja nam daje želju za životom
koja nam potiče volju da je slijedimo
do kraja, do kraja

Želio bih vam govoriti o njoj
bez spominjanja imena njezina
puno ili malo ljubljenoj
ona je vjerna
i ako baš želite
da vam je predstavim
ona se zove
revolucija koja teče!

bezimena
sans la nommer

(georges moustaki/
zdravko dovedan han)

Generiranje koda posljednja je faza prevođenja. U toj se fazi generira kôd u ciljnom jeziku. Znamo da ima više vrsta ciljnih jezika, od strojnog jezika do jezika visoke razine. Za naše primjene posebno su važne dvije vrste generiranja koda:

- *interpretiranje, u kojem je ciljni jezik virtualni jezik više razine (sličan asemblerskom jeziku) i može se interpretirati, i*
- *predprocesiranje, u kojem je ciljni jezik neki postojeći jezik za programiranje visoke razine.*

U ovom ćemo poglavlju detaljnije opisati ta dva postupka.

6.1 INTERPRETIRANJE

Prvi jezici za programiranje bili su realizirani kao kompilatori. Sredinom šezdesetih godina prošloga stoljeća pojavljuju se prvi interpretatori i uvodi se naziv **P-kôd** za hipotetski jezik kojeg interpretiraju: prvi put 1966. godine u realizaciji jezika BCPL (kao O-kôd), potom s imenom P-kôd u realizaciji jezika Euler. Otad se i interpretator ili virtualni stroj, koji interpretira P-kôd (asemblerski jezik), naziva **P-stroj**.

Termin P-kôd puno se koristio u implementacijama prevodilaca Pascala (Pascal-P, Nori, Ammann, Jensen, Hageli, and Jacobi, 1973. i Pascal-S kompilator, Wirth 1975. godine).

Programi napisani u jezicima koji se prevode u P-kôd kao međujezik interpretiraju se uz pomoć softvera (programa) koji oponaša hipotetski CPU. Nekad LISP i BASIC, a danas poznati jezici, na primjer Java i Python, realizirani su kao interpretatori. Na primjer, ako P-stroj sadrži instrukcije (mnemoničke naredbe):

LIT *broj* - umetanje broja ("literala") na vrh stoga
LOD *adr* - umetanje sadržaja varijable (sa adrese *adr*) na vrh stoga
STO *adr* - instrukcija pohranjivanja sadržaja na vrhu stoga na adresu *adr*
OPR *op* - skup aritmetičkih operatora *op*

i ako su A i B konstante:

```
CONST  A = 10,  B = 20;
```

a X, Y i Z varijable:

```
VAR X, Y, Z;
```

s adresama 3, 4 i 5, redom, tada će niz naredbi:

```
X := A+1;  Y := 2*B;  Z := X+Y
```

biti interpretiran kao:

```
LIT  10
LIT  1
OPR  2       (+)
STO  3       (X)

LIT  2
LIT  20
OPR  4       (*)
STO  4       (Y)

LOD  3
LOD  4
OPR  2       (+)
STO  5       (Z)
```

Pri izvršenju tih instrukcija imali bismo:

```
LIT  10
LIT  1
OPR  2       (+)
STO  3       (X <- 11)

LIT  2
LIT  20
OPR  4       (2*20)
STO  4       (Y <- 40)

LOD  3       (X)
LOD  4       (Y)
OPR  2       (11+40)
STO  5       (Z <- 51)
```

Općenito P-kod sadrži instrukcije koje osiguravaju potpuno prevođenje programa napisanog u izvornom jeziku.

U sljedećem je poglavlju detaljno opisan interpretator mini jezika **PL/0**.

6.2 PREDPROCESIRANJE

Kompilatori nisu pogodni za implementiranje prevodilaca koji su bliži našim primjenama prevođenja. Ostavimo to onima koji se bave sistemskim softverom i onima koji razvijaju jezike za široku uporabu.

Za naše primjene najbliži su prevodioci koji će generirati međukod u jeziku visoke razine. Takve smo prevodioce nazvali predprocesori.

Osim intepretiranja jezika za programiranje koje nam omogućuje da sami možemo dizajnirati svoj prevodilac, predprocesiranje se također može iskoristiti za to. Štoviše, ako solidno vladamo nekim od jezika za programiranje, isti jezik možemo upotrijebiti i za dizajn predprocesora i za ciljni jezik. Na primjer, ako bismo dizajnirali predprocesor jezika Pascal u kojem bi se za ciljni jezik izabrao Python, izvorni program za simuliranje igre stolni tenis napisan u Pascalu mogao bi biti preveden u Python kao što slijedi:

Pascal

```
PROGRAM Stolni_Tenis;
USES    Crt;
VAR     A, B, K, X, Y, S, M : integer;
BEGIN
  ClrScr;
  REPEAT
    Write ('Igra se do 11 ili 21? ');
    ReadLn (K)
    UNTIL K in [11, 21];

  M := 4; IF k = 21 THEN m := 3;

  A := 0;  B := 0; S := 0;
  Randomize;

  REPEAT { Meč }
    X := 0; Y := 0; S := S+1;
    REPEAT { Set }
      IF random (2) = 1 THEN
        X := x+1
      ELSE Y := y+1

      UNTIL ((X >= K) OR (Y >= K))
            AND (abs (X-Y) > 1);

    WriteLN (S, '. set A : B = ', X :2,
             ' : ', Y :2);

    IF X > Y THEN
      A := A+1
    ELSE
      B := B+1
    UNTIL (A = M) OR (B = M);

  WriteLN;
  IF A > B THEN
    WriteLN ('Pobjeda igrača A  ', A,
      ' : ', B)
  ELSE
    WriteLN ('Pobjeda igrača B  ', B,
      ' : ', A);
  Readln
END.
```

Python

```
# -*- coding: cp1250 -*-
# PROGRAM Stolni_Tenis;



while True : # REPEAT
  print 'Igra se do 11 ili 21? ';
  K = input ()
  if K in [11, 21] : break # UNTIL

M = 4
if K == 21 : M = 3
A = 0; B = 0; S = 0;
from random import *

while True : # Meč
  X = 0; Y = 0; S = S+1;
  while True : # Set
    if int (1 +round(random () )) == 1 :
      X = X+1
    else :
      Y = Y+1
    if ((X >= K) or (Y >= K))
       and (abs (X-Y) > 1) : break

  print S, '. set A : B = ', X, ' : ', Y


  if X > Y :
    A = A+1
  else :
    B = B+1
  if (A == M) or (B == M) : break

print
if A > B :
  print 'Pobjeda igrača A  ', A, ':', B

else     :
  print 'Pobjeda igrača B  ', B, ':', A
```

Pravila prevođenja u ovom primjeru su sljedeća:

1) Deklaracije i definicije Pacala i poziv standardne procedure (`ClrScr`) bili bi ignorirani. Poziv procedure `Randomize` prevodi se u `from random import *`
2) Sva slova imena varijabli bila bi u prijevodu velika (da bismo izbjegli imena varijabli koja su legalna u Pascalu, a u Pythonu bi bila rezervirana)
3) Simbol pridruživanja `:=` prevodi se u `=`
4) Operacija `+` ostaje nepromijenjena
5) Sve relacije, osim `=`, ostaju nepromijenjene, a `=` se prevodi u `==`

6) Logičke operacije `not`, `and` i `or` pišu se samo malim slovima
7) Rezervirane riječi `BEGIN` na početku i `END` na kraju glavnog programa kao i točka na kraju programa se ne prevode (prevode se u prazan niz)
8) Složene se naredbe prevode kao što slijedi:

```
REPEAT                while True :

UNTIL uvjet             if uvjet : break

IF uvjet THEN          if uvjet :
  naredba                naredba_Py
ELSE                   else :
  naredba                naredba_Py
```

9) Naredba za ispis:

```
WriteLn ( lista )      print lista
```

10) Poziv funkcije `random(c)+b` prevodi se u `int(c*random())+b`.

Dakako, pri prevođenju u Python moraju se poštovati pravila pisanja programa (uvlačenje redova unutar složenih naredbi).

U osmom smo poglavlju opisali jedan jezik visoke razine pod nazivom **DDH**, a u devetom poglavlju detaljno opisali njegov predprocesor napisan u Pythonu s generiranjem koda također u Pythonu.

P R O G R A M I

PREVOĐENJE JEZIKA Exp

Evo nas na posljednjoj fazi prevođenja našeg jezika **Exp** – generiranju koda. Naravno, pokazat ćemo kako se može definirati njegov interpretator i predprocesor.

Interpretator

Interpretator jezika **Exp** koristit će stablo sintaksne analize (izraz u postfiksnoj notaciji) dobiveno na izlazu prepoznavača danog u prethodnom poglavlju, program `Exp-Tpa.py`.

Predprocesor

Sada možemo dati program za prevođenje (predprocesor) jezika **Exp**. Sintaksna analiza je realizirana postupkom rekurzivnog spusta. Izrazi koji ne sadrže leksičku niti sintaksnu pogrešku prevode se izravno (tj. ostaju nepromijenjeni) u Python i izvršavaju pozivom funkcije `eval`. Eventualne semantičke pogreške pri izračunavanju izraza, kao što je na primjer dijeljenje s nulom ili poziv funkcije izvan njezine domene, bit će dojavljene.

Exp.py

```
# -*- coding: cp1250 -*-
"""
   Sintaksna analiza (rekurzivni spust - recursive descent),
   prevođenje i izračunavanje realnih izraza napisanih prema sintaksi:

   izraz    : term { [+|-] term }
   term     : faktor { [ * | / ] faktor }
   faktor   : [+|-] broj | ^faktor | funkcija | ( izraz )
   funkcija : ime ( izraz )
   ime      : abs | sin | cos | tan  | exp | log | sqr | sqrt | trunc
   broj     : brojka { brojka } [ . brojka { brojka } ]
   brojka   : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
"""

from math import *
import string

def sqr(x): return x**2

Brojke   = list (string.digits);           Slova   = list (string.ascii_lowercase)
Simboli  = ['+','-','*','/','^','(',')'];  Alfabet = Brojke +Slova +Simboli
Funkcije = ['abs', 'sin', 'cos', 'tan', 'exp', 'log', 'sqr', 'sqrt', 'trunc']

def Znak() :
  global Niz, i, Z
  while Niz[i] == ' ' : i += 1
  Z = Niz[i]; i += 1

def Ucitaj() : return raw_input ('Upiši izraz ')

def Greska (Poruka, p) :
  global Gr
  print Niz [:-1];  print (p-1)*' ' + chr(24), Poruka
  Gr = True

def Broj () :
  global Sim, i, Z
  while Z in Brojke :  Sim += Z;  Znak ()

def Leks_an () :
  global Niz, Z, S, Sim, i, R, Gr

  if Z == '#' : S = Z;  return
  S = ' ';  Sim = ''
  if Z not in Alfabet : Greska ('* ( 1) Ilegalan znak', i);  return
  if Z in Simboli : S = Z; Znak()
  else :
    if Z in Brojke :
      Broj ()
      if Z == '.' :
        Sim += Z; Znak ()
        if Z in Brojke : Broj ()
        else           :
          Greska ('* ( 2) Leksička pogreška! (nije brojka)', i)
          return
      S = 'B'
```

```
      else :
        while Z in Slova : Sim += Z;  Znak ()
        if Sim in Funkcije : S = 'F'
        else               :
          Greska ('* (3) Leksička pogreška! (nepoznato ime "' +Sim +'")',
                  i-len(Sim))

  def Izraz () :
    global Niz, Z, S, Sim, i, R, Gr, F, M, Q

    def Izr_fun (F):
      try    : R[-1] = eval (Funkcije [F] +'(R[-1])')
      except :
        Greska ('* ( 4) Funkcija ' +Sim +'nije definirana za dani argument', i)

    def Term () :
      global Niz, Z, S, Sim, i, R, Gr

      def Faktor () :
        global Niz, Z, S, Sim, i, R, Gr

        '<faktor> : [+|-] <broj> | ^<faktor> | <funkcija> | ( <izraz> )'
        if S == 'B' :
          R.append (float(Sim)); Leks_an()
        elif S == '(' :
          Leks_an (); Izraz ();
          if S <> ')' : Greska('* ( 5) Sintaksna pogreška', i); return
          else        : Leks_an ()
        elif S == 'F' :
          F = Funkcije.index (Sim); Leks_an ()
          if S == '(' :
            Leks_an (); Izraz()
            if S == ')' :
              Izr_fun (F)
              if Gr : return
              Leks_an ()
            else :
              Greska ('* ( 6) Sintaksna pogreška', i)
              return

          else :
            Greska ('* ( 7) Nepotpuni izraz', i)
            return

        elif S in ['+','-'] :
            Pr = S; Leks_an (); Faktor ()
            if Pr =='-' : R[-1] = -R[-1];
            else        : Greska ('* ( 8) Nepotpuni izraz', i);     return
        else            : Greska ('* ( 9) Sintaksna pogreška', i);  return

        while S == '^' :
          Leks_an ();  Faktor ()
          try    : R[-2] = eval ('R[-2]**R[-1]')
          except :
            Greska ('* ( 9) Stupnjevanje negativnog broja!', i)
            return
          R = R[:-1]
```

```
    "<Term> : <faktor> { [ * | / ] <faktor> } "
    Faktor ();
    while S in ['*','/'] :
      M = S; Leks_an (); Faktor()
      if not Gr :
        try    : R[-2] = eval ('R[-2]' +M +'R[-1]')
        except :
          Greska ('* (10) Nije dopušteno dijeliti s nulom!', i-len(Sim)-1)
          return
        R = R[:-1]
      else : break

  "<Izraz> : <term> { [+|-] <term> } "
  Term ()
  while S in ['+', '-'] :
    Q = S;  Leks_an ();  Term ()
    R[-2] = eval ('R[-2]' +Q +'R[-1]')
    R = R[:-1]
  return

def Ispis () :
  print Niz[:-1] +' =', R[0]

while True :
  print
  Niz = Ucitaj ()
  if Niz == '' : break

  Niz += '#';  i = 0;  Gr = False;  R = [];  Znak()
  Leks_an (); Izraz ()
  if not Gr and S == '#': Ispis ()
  elif not Gr           :
    j = len(Sim);  j += (j == 0)*1;  Greska ('* (11) Sintaksna pogreška', i -j)


Upiši izraz 1*2+3*4+5*6
1*2+3*4+5*6 = 44.0

(10+20)*30 = 900.0

(((11+22)*33 -44)*55 -66)*77 -88 = 4420405.0

111 + 11^3.5 = 4525.42759596

sin(2)^2 + cos(2)^2 = 1.0

11.5^3.5 = 5157.53805654

sqrt(144) = 12.0

1+2*(30
       * ( 5) Sintaksna pogreška

1 / log(1)
         * (10) Nije dopušteno dijeliti s nulom!

sqrt(2) = 1.4142135623730951
```

```
sqrt (-1)
        * ( 4) Funkcija nije definirana za dani argument

log (-10)
        * ( 4) Funkcija nije definirana za dani argument

4*sqrt(1+2+3+4+5) = 15.4919333848

trunc (abs(sin(3.33)*5) /4 + exp(4)) = 54

sin(60 * 3.14159265/180) = 0.866025403186

sqr(sin(5)) +sqr(cos(5)) = 1.0
```

P R I M J E N E

U sljedeća dva poglavlja opisat ćemo dva mini jezika za programiranje, **PL/0** i **DDH**, i prikazati realizaciju interpretatora prvog jezika, odnosno realizaciju predprocesora drugog jezika. Ovdje dajemo realizaciju interpretatora jezika **Exp**.

KEMIJSKE FORMULE

Kemijske formule su jezik! To znači da možemo definirati njihov alfabet, rječnik i pravila pisanja. Alfabet i rječnik dobit ćemo iz periodnog sustava elemenata:

```
  1 H       1.008 vodik               hydrogen
  2 He      4.003 helij               helium
  3 Li      6.941 litij               lithium
    ...
 50 Sn    118.710 kositar             tin
    ...
111 Rg    272.000 roentgenij          roentgenium
112 Cn    277.000 ununbij             ununbium
```

Alfabet je:

```
{ A, B, C, D, E, F, G, H, I, K, L, M, N, O, P, R, S, T, U, V, W, X, Y, Z,
  a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, r, s, t, u, y } ∪
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (, ) }
```

a rječnik se sastoji od:

- simbola kemijskih elemenata, od H, simbola vodika, do Cn, simbola elementa ununbija, ukupno 112 simbola:

```
{  H, He, Li, Be,  B,  C,  N,  O,  F, Ne, Na, Mg, Al, Si,  P,  S, Cl, Ar,  K,
  Ca, Sc, Ti,  V, Cr, Mn, Fe, Co, Ni, Cu, Zn, Ga, Ge, Ka, Se, Fr, Kr, Rb, Sr,
   Y, Zr, Nb, Mo, Tc, Ru, Rh, Pd, Ag, Cd, In, Sn, Sb, Te,  I, Xe, Cs, Ba, La,
  Ce, Pr, Nd, Pm, Sm, Eu, Gd, Tb, Dy, Ho, Er, Tm, Yb, Lu, Hf, Ta,  W, Re, Os,
  Ir, Pt, Au, Hg, Tl, Pb, Bi, Po, At, Rn, Fr, Ra, Ac, Th, Pa,  U, Np, Pu, Am,
  Cm, Bk, Cf, Es, Fm, Md, No, Lr, Rf, Db, Sg, Bh, Hs, Mt, Ds, Rg, Cn }
```

- prirodnih brojeva i
- zagrada (i)

Jezik kemijskih formula nad danim rječnikom definirat ćemo regularnim izrazom:

(((*simbol* [*broj*])$^{+}$) [*broj*] | *simbol* [*broj*])$^{+}$

Na primjer, kemijske formule su:

```
H2O  HHO  H2SO4  NaCl  C12H22O11  (CH3)(CH2)(OH)  HoHo  (SeNiLaN)55
```

Značenje (semantika) je zbroj relativnih atomskih masa elemenata formule, bez obzira postoji li ili ne takav spoj.

Prevodilac

Sada možemo definirati prevodilac kemijskih formula koji će učitati niz znakova, prevesti ih u niz simbola, provjeriti jesu li napisani prema danim pravilima i, ako jesu, generirati realni izraz koji će sadržavati zagrade, operaciju zbrajanja i množenja, a umjesto simbola kemijskih elemenata njihove relativne atomske mase. Sve se to obavlja u jednom prolasku. Prevodilac je realiziran u Pythonu. Slijedi njegov opis.

TABLICA SIMBOLA

Tablica simbola sadržavat će svih `112` kemijskih elemenata. Svakom elementu (simbolu) bit će pridružena svojstva: redni broj i relativna atomska masa, uz dodatak hrvatskog i latinskog naziva koje nećemo koristiti u ovom programu, već smo ih ostavili za eventualne nadgradnje.

Za realizaciju tablice simbola najprikladnije je koristiti Pythonovu standardnu strukturu podataka, klasu *`dict`*, koja predstavlja rječnik u koji se podaci unose s jedinstvenim ključem (simbol kemijskog elementa u našem primjeru) i pridruženim vrijednostima bilo kojeg sekvencijalnog tipa Pythona (`n`-torka sa svojstvima kemijskog elementa u našem primjeru). Podaci su unutar rječnika tako uređeni da se mogu jednostavno pretraživati po ključu, a to nam je vrlo važno.

Da bismo izgradili tablicu simbola, prvo smo iz teksualne datoteke periodnog sustava elemenata izgradili listu `PS0`, mjesto `#2` u programu, gdje je svaki element liste jedan red tekstualne datoteke, potom smo listu `PS0` podijelili po stupcima i dobili listu `PS`, mjesto `#3` u programu, u kojoj svaki element sadrži simbol kemijskog elementa i njegova svojstva. Na kraju smo definirali klasu `kem_element` i objekt s imenom `Psus` u koji smo prenijeli podatke iz liste `PS`.

LEKSIČKA ANALIZA

U leksičkoj analizi treba iz ulaznog niza ekstrahirati pojedine riječi. Ako skupinama riječi rječnika pridružimo kodove:

1 simbol kemijskog elementa
2 broj
3 (
4)

i dodamo znak "`#`" s kodom `5` kao kraj ulaznog niza, procedura `Get_Sym()` vraća tekući simbol ulaznog niza (odnosno, relativnu masu ako je tekući simbol kemijski element) i njegov kôd, ili dojavljuje pogrešku ako učitani simbol nije u rječniku.

SINTAKSNA ANALIZA

Jezik kemijskih formula je prema našoj definiciji jezik sa svojstvima, pa ćemo u sintaksnoj analizi koristiti naš postupak sintaksne analize upravljan tablicom prijelaza i akcija. S obzirom na to da u formulama nije dopušteno ugnježđenje zagrada, tj. jezik kemijskih formula definirali smo kao regularni, nije potrebno brojati zagrade, pa smo definirali dinamičku tablicu prijelaza i akcija. Tablica `T1` predstavlja inicijalnu tablicu u kojoj je dopušteno napisati otvorenu zagradu. Pisanjem otvorene zagrade `T2` postaje nova (aktualna) tablica prijelaza i akcija. U njoj se očekuje pojava zatvorene zagrade. Kad se to dogodi, ponovo će `T1` postati aktualna tablica prijelaza i akcija. Dosezanjem kraja ulaznog niza akcijom `7` provjerava se je li aktualna tablica prijelaza i akcija jednaka `T1`. Ako nije, dojavljuje se pogreška.

`T1`

	simbol 1	*broj* 2	(3	) 4	# 5
0	1, 1		0, 4		
1	1, 2	1, 3	0, 5		0, 7

`T2`

	simbol 1	*broj* 2	(3	) 4	# 5
0	1, 1				
1	1, 2	1, 3		1, 6	0, 7

GENERIRANJE KODA

U svakom se koraku postupka prevođenja, poslije leksičke i sintaksne analize, generira kôd prema sljedećim pravilima:

riječ jezika	pozicija u ulaznom nizu	kôd (prijevod)
simbol	početak ili iza (	*relativna_atomska_masa*
	iza simbola, broja ili)	*+ relativna_atomska_masa*
broj		** broj*
(	početak	(
	iza bilo kojeg simbola	+ (
)		)

Generiranje koda realizirano je značenjem pojedinih akcija. Kodovi akcija pridruženi su funkciji prijelaza u tablici prijelaza i akcija.

IZVRŠAVANJE PROGRAMA

Na kraju će prevođenja, pod uvjetom da nije bilo leksičkih niti sintaksnih pogrešaka, varijabla `Gen` sadržavati prijevod - brojčani izraz čijim će se izvršenjem dobiti relativna atomska masa "spoja" danog formulom. Kao što smo već rekli, ne provjeravamo može li zadana formula predstavljati kemijski spoj.

Brojčani niz napisan kao znakovni niz bit će izvršen naredbom `eval`. Naravno, Python ne smije otkriti pogrešku u izrazu! Evo kompletnog programa prevodioca jezika kemijskih formula i nekoliko primjera:

Kem_form.py

```
# -*- coding: cp1250 -*-

import os, string

VS     = string.ascii_uppercase;  MS     = string.ascii_lowercase
Brojke = string.digits;           Brojke = Brojke [1:]
NL     = '\n'


#1                      Tablica prijelaza i akcija
'        S             B             (             )             #          '
'        1             2             3             4             5          '
'-------------------------------------------------------------------------------'
T1 = { (0,1): (1,1),               (0,3): (0,4),
       (1,1): (1,2), (1,2): (1,3), (1,3): (0,5),              (1,5): (0,7) }
T2 = { (0,1): (1,1),
       (1,1): (1,2), (1,2): (1,3),              (1,4): (1,6), (1,5): (0,7) }
'-------------------------------------------------------------------------------'

#2 UČITAJ PERIODNI SUSTAV
Dat = 'Per-SUS.TXT'
if os.path.exists(Dat):
  PS0 = []
  for line in open (Dat, 'r') : PS0.append (line[:-1])
else : print 'NE POSTOJI DATOTEKA', Dat

#3 'RAZBIJ' PO STUPCIMA
PS = []
for x in PS0 : PS.append (x.split())

#4 TABLICA SIMBOLA
class kem_element :
  def __init__ (self): self.v = { }
  def insert (self, Simbol, Red_br, Rel_masa, Hrv_naz, Lat_naz) :
    self.v.update ( { Simbol : (Red_br, Rel_masa, Hrv_naz, Lat_naz) } )
  def rb (self, i) : return self.v[i][0]  # RB
  def m  (self, i) : return self.v[i][1]  # relativna masa
  def hi (self, i) : return self.v[i][2]  # hrvatski naziv
  def li (self, i) : return self.v[i][3]  # latinski

Psus = kem_element ()
for i in range (len(PS)):
  Y  = PS[i]
  Psus.insert ( Y[1], int (Y[0]), Y[2], Y[3], Y[4] )

#5 PREVODILAC
rm = Psus.m;  El = Psus.v

def Get_Sym () : # LEKSIČKA ANALIZA
  global i
  i += 1;  C = 0
  S  = F[i]
  if S in VS :
    if F[i+1] in MS : i += 1; S += F[i]
    if S in El : C = 1;  S = rm(S)
```

```
    elif S in Brojke :
      C = 2
      while F [i+1] in Brojke +'0' : i += 1;  S += F[i]
    elif S in '()#' : C = '()#'.find (S) +3
    return S, C

  while 1 :
    F = raw_input ('Upiši kemijsku formulu ') +'#'
    if F == '#' : break
    Gen = '';  i = -1;  Gr = False; Kraj = False
    q = 0; T = T1

  while not Gr and not Kraj:
      " Leksička analiza "
      Sym, Code = Get_Sym()
      if Code == 0 : Gr = True; break

      " Sintaksna analiza "
      Qt = (q, Code)

      if Qt in T :
        " Generiranje koda (prevođenje) "
        q, A = T [Qt]
        if   A == 1 : Gen += Sym
        elif A == 2 : Gen += ' +' +Sym
        elif A == 3 : Gen += ' *' +Sym
        elif A == 4 : Gen += Sym; T = T2
        elif A == 5 : Gen += ' +' +Sym; T = T2
        elif A == 6 : Gen += Sym; T = T1
        elif A == 7 : Kraj = T == T1; Gr = not Kraj
      else : Gr = True

    " Izvršavanje generiranog koda "
    if not Gr : print F[:-1] +' =', Gen, '=', eval (Gen), NL
    else      : print 'POGREŠKA!', NL

  " Primjeri: C12H22O11 šećer (saharoza), CH3CH2OH  etanol "

  Upiši kemijsku formulu H2O
  H2O = 1.008 *2 +16.000 = 18.016

  Upiši kemijsku formulu H2so4
  POGREŠKA!

  Upiši kemijsku formulu H2SO4
  H2SO4 = 1.008 *2 +32.070 +16.000 *4 = 98.086

  Upiši kemijsku formulu C12H22O11
  C12H22O11 = 12.010 *12 +1.008 *22 +16.000 *11 = 342.296

  Upiši kemijsku formulu (CH3)(CH2)(OH)
  (CH3)(CH2)(OH) = (12.010 +1.008 *3) +(12.010 +1.008 *2) +(16.000 +1.008) = 46.068
```

Program `Kem_form.py` osim što predstavlja pojednostavljeni prikaz faza prevođenja i izvršavanja "programa" napisanog u jeziku kemijskih formula može poslužiti i kao dobar primjer kako se teorija formalnih jezika može primijeniti kao posebna tehnika programiranja.

NAJMANJI „PREDPROCESOR“ JEZIKA Exp

Koristeći proceduru `eval(RI)` u Pythonu koja izračunava vrijednost realnog izraza *RI* napisanog kao niz znakova, slijedi primjer ”predprocesora” jezika **Exp** napisan, dakako, u Pythonu.

Eval.py

```
# -*- coding: cp1250 -*-
# PROGRAM Eval - Evaluira realni izraz (sintaksna analiza i izvršavanje)

from math import *

def sqr(x): return x**2
Pi = pi

Primjer = """
" Primjer "
x = 12
izraz = '(x+10)*20'
print
print 'x =', x
print izraz, '=', eval (izraz)
print
"""

print Primjer
exec (Primjer)

while True:
  print
  izraz = raw_input ('Upiši izraz > ')
  if izraz == '' : break
  try    : print izraz +' =', eval (izraz)
  except : print 'SINTAKSNA POGREŠKA!'
```

Izvršenjem programa prvo će biti ispisano:

```
" Primjer "
x = 12
izraz = '(x+10)*20'
print
print 'x =', x
print izraz, '=', eval (izraz)
print

x = 12
(x+10)*20 = 440
```

što je rezultat još jedne procedure Pythona korisne za primjene u teoriji formalnih jezika – procedure `exec`, koja nije ništa drugo do poziv interpretatora jezika Python iz programa napisanog u Pythonu da bi izvršio (interpretirao) ponuđeni mu niz naredbi sadržanih u više linija koda. (Osim te procedure postoji i `execfile ()` koja za argument ima datoteku koja će biti interpretirana.) Evo dva primjera prevođenja :

```
Upiši izraz > sqr (sin(Pi)) +sqr (cos(Pi))
sqr (sin(Pi)) +sqr (cos(Pi)) = 1.0
```

```
Upiši izraz > (1+2
(1+2 = SINTAKSNA POGREŠKA!
```

Ako zadani niz sadrži leksičke pogreške ili ako sadrži poziv standardne funkcije s argumentom s vrijednošću izvan domene funkcije, također će biti dojavljena "sintaksna pogreška", što je mali nedostatak ovog programa. Na primjer:

```
Upiši izraz > sqrt (-1)
sqrt (-1) = SINTAKSNA POGREŠKA!

Upiši izraz > 1/0
1/0 = SINTAKSNA POGREŠKA!

Upiši izraz > (1+2
(1+2 = SINTAKSNA POGREŠKA!

Upiši izraz > abc(12)
abc(12) = SINTAKSNA POGREŠKA!
```

PISANJE INTERPRETATORA UZ POMOĆ LEXa i YACCa

Čitatelje koji bi željeli proširiti svoja znanja o projektiranju interpretatora upućujemo na LEX i YACC o kojima se mnogo toga nalazi na WEB-u.

S obzirom na to da su u ovoj knjizi svi algoritmi napisani u Pythonu, upućujemo vas na programski paket PLY - implementaciju LEXa i YACCA u Pythonu. Sve se nalazi na stranici:

http://www.dabeaz.com/ply/

Zadaci

1) Proširite jezik Exp tako da se mogu pamtiti vrijednosti izraza pridružujući ih imenima, velikim slovima od A do Z naredbom za dodjeljivanje:

```
ime_varijable = izraz
```

Proširite i pravila pisanja izraza tako da na mjestima operanda može stajati i ime (realne) varijable uz uvjet da je prije toga morala biti inicijalizirana. Definirajte prevodilac novonastalog jezika.

7.
INTERPRETATOR JEZIKA PL/0

— la pierre

pred kamenom napuštenim
okićenim s nekoliko uvelih cvjetova
samo križ je što para vjetar
i moje uspomene jedine preživjele

koliko trebat će molitve
pred kamenom sa srcem kamenim
da probudi se duša ušutjela
u vječnoj tišini statua

a ništa ne može više oživjeti
mrtvi pepeo zatvoren
pod kamenom golim kao smrt
nježne ljubavi teže od grižnje savjesti

pred kamenom napuštenim
okićenim s nekoliko uvelih cvjetova
samo križ je što para vjetar
i moje uspomene jedine preživjele

kamen
la pierre

(georges moustaki/
nepoznati student)

Kao "školski primjer" nacrta interpretatora izabrali smo mini jezik PL/0 kojeg je prof. Wirth prikazao u svojoj knjizi "Algorithms + Data Structures = Programs", [Wir1976]. Najprije dajemo opis jezika PL/0 koji može poslužiti i kao dobar primjer definicije jednog jezika za programiranje. Potom detaljno opisujemo njegov interpretator realiziran u Pythonu.

7.1 JEZIK PL/0

Jezik **PL/0** je relativno potpun jezik za programiranje. Sadrži samo jedan primitivan tip podataka – cjelobrojni, definiranje konstanti, deklariranje varijabli, naredbu za dodjeljivanje, `WHILE` petlju, `IF-THEN` naredbu, procedure (potprograme), poziv procedure i niz naredbi. Od sintaksnih stuktura definirani su cjelobrojni i relacijski izrazi. U nastavku dajemo njegovu leksičku i sintaksnu strukturu, te semantiku.

Leksička struktura

Definirati leksičku strukturu jezika za programiranje znači definirati njegov alfabet, rječnik i leksička pravila.

ALFABET

Alfabet jezika **PL/0** čine sljedeći skupovi znakova:

- velika slova engleskog alfabeta: `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
- mala slova engleskog alfabeta: `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- brojke: `0 1 2 3 4 5 6 7 8 9`
- posebni znakovi: `+ - * / = < > ( ) . : , ; $ # %`

Skupu posebnih znakova pripada i praznina (razmak ili "blank"). U daljnjem tekstu skupove znakova jezika **PL/0** označivat ćemo neterminalima ***slovo*** i ***brojka***.

RJEČNIK

Nad alfabetom su definirani sljedeći skupovi riječi: rezervirane riječi, imena, brojevi i posebni simboli.

Rezervirane riječi

Rezervirane riječi, kao što im i ime kaže, imaju unaprijed definirano ili "rezervirano" značenje. Pojavljuju se kao dijelovi naredbi jezika i nije ih dopušteno rabiti u druge svrhe. Evo potpunog skupa rezerviranih riječi:

```
BEGIN CALL CONST DO END IF  ODD PROCEDURE THEN VAR WHILE
```

Imena

Imena su klase riječi koje se pišu prema sljedećem pravilu:

```
ime:     slovo { slovo | brojka }
slovo:   A| B| C| D| E| F| G| H| I| J| K| L| M| N| O| P| Q| R| S| T| U| V| W| X| Y| Z|
         a| b| c| d| e| f| g| h| i| j| k| l| m| n| o| p| q| r| s| t| u| v| w| x| y| z
brojka:  0| 1| 2| 3| 4| 5| 6| 7| 8| 9
```

Dakle, ime je sačinjeno od samo jednog slova, odnosno, to su riječi koje počinju slovom, a potom slijedi slovo ili brojka. Maksimalna duljina imena je 10. Imena sastavljena samo od slova ne smiju biti iz skupa rezerviranih riječi.

Brojevi

Brojevi se pišu prema pravilu:

```
broj:   brojka { brojka }
```

Dakle, to su nepredznačeni cijeli brojevi.

Posebni simboli

Jezik **PL/0** ima samo jedan poseban simbol (niz znakova sačinjen od posebnih znakova):

```
posebni_simbol: :=
```

Leksička pravila

Postoje samo dva leksička pravila: sve se riječi pišu kompaktno, bez razmaka, i značenje velikih i istih takvih malih slova u rezerviranim riječima ili imenima je jednako.

Osnovna sintaksna struktura

Osnovna sintaksa struktura jezika PL/0 dana je sljedećim sintaksnim pravilima:

```
program        : blok .
blok           : [ definiranje_konstanti ]
                 [ deklariranje_varijabli ]
                 { procedura }
                 niz_naredbi
niz_naredbi    : BEGIN naredba { ; naredba } END
naredba        : naredba_za_dodjeljivanje | poziv_procedure |
                 niz_naredbi              |selekcija        |
                 petlja                   | prazna_naredba
prazna_naredba : ε
```

Vidimo da će "minimalni program" biti samo:

```
BEGIN END.
```

U nastavku dajemo detaljniju sintaksu pojednih sintaksnih kategorija i njihovo značenje (semantiku).

DEFINIRANJE KONSTANTI

Pravilo definiranja konstanti dano je sa:

```
definiranje_konstanti : CONST ime_konstante = broj
                            { , ime_konstante = broj } ;
ime_konstante         : ime
```

Značenje je pridruživanje brojčane vrijednosti izabranom imenu koje će poslije toga primiti svojstvo "cjelobrojna konstanta" čime je "fiksirana" njezina vrijednost u programu i nije je moguće promijeniti (smije se pojaviti samo u izrazima).

DEKLARIRANJE VARIJABLI

Sve varijable programa moraju biti deklarirane, prema sintaksi:

```
deklariranje_varijabli :VAR ime_varijable { , ime_varijable } ;
ime_varijable          :ime
```

Navedena imena imat će svojstvo da su cjelobrojne varijable s dosegom svoga značenja unutar bloka u kojem su deklarirane. Početna vrijednost im nije poznata.

PROCEDURA

Pisanje procedura uvodi module u programe pisane u jeziku **PL/0**. Dopušteno je ugniježđenje procedura i rekurzivni poziv.

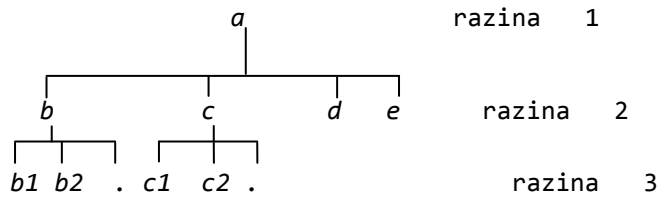
```
procedura : PROCEDURE ime_procedure ; blok ;
```

Poziv procedure dan je pravilom:

```
poziv_procedure : CALL ime_procedure
```

Hijerarhijska struktura programa

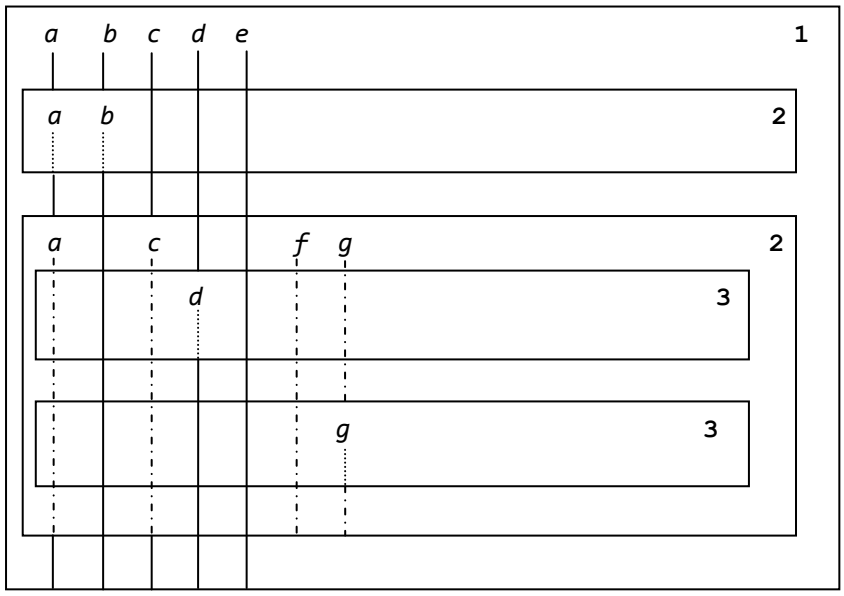
Iz dane sintakse slijedi da procedura ima strukturu sličnu programu: sastoji se od zaglavlja i bloka. S obzirom da blok sadrži definiciju konstanti, deklaraciju varijabli, proceduru i niz naredbi, općenito će procedura imati svoju definiciju konstanti, deklaraciju varijabli, svoju proceduru i niz naredbi itd. Može se zamisliti da glavni program i procedure čine hijerarhijsku strukturu u kojoj je glavni program na vrhu (razina 1), a svaka je procedura na razini uvećanoj za jedan u odnosu na razinu programa (ili procedure) u kojoj je ugniježđena:



Globalna i lokalna imena

Dana hijerarhijska struktura definira dosege značenja imena, koje se proteže od mjesta definicije (odgovarajućega čvora) i obuhvaća sve sljednike. Na primjer, ime definirano u glavnom programu ima značenje u cijelom programu, a ime definirano u proceduri *c* ima značenje u svim procedurama sadržanim u proceduri *c*. U bilo kojoj proceduri dopuštena je promjena definicije imena. Doseg je novoga značenja do kraja bloka u kojem je definicija uvedena. Poslije toga vrijedi prethodno značenje.

Na danom crtežu, sl. 7.1, okviri predstavljaju procedure. Glavni program je na razini *1*, a procedure na razini *2* i *3*. Pune i isprekidane crte predstavljaju dosege značenja imena *a*, *b*, *c*, *d*, *e*, *f* i *g*.



Sl. 7.1 – *Globalna i lokalna imena.*

NAREDBA ZA DODJELJIVANJE

Općenito pravilo pisanja naredbe za dodjeljivanje dano je sa:

```
naredba_za_dodjeljivanje : ime_varijable := izraz
```

Značenje je uobičajeno: izračunavanje (cjelobrojnog) izraza i pridruživanje dobivene vrijednosti navedenoj varijabli. Dalje je:

```
izraz      : [ + | - ] term { operacija1 term }
operacija1 : + | -
term       : faktor { operacija2 faktor }
operacija2 : * | /
faktor     : broj |ime_varijable | ime_konstante | ( izraz )
```

Sve su operacije cjelobrojne, sa sljedećim značenjem:

- `+` zbrajanje
- `-` oduzimanje
- `*` množenje
- `/` dijeljenje

Dakle, može se reći da su izrazi u jeziku **PL/0** jednostavni brojčani izrazi s četiri osnovne operacije i zagradama, bez ijedne fukcije. Prioritet izvršavanja je uobičajen:

1) izraz u zagradi
2) predznak
3) množenje i dijeljenje
4) zbrajanje i oduzimanje

SELEKCIJA

Selekcija je složena naredba definirana sintaksom:

```
selekcija : IF uvjet THEN naredba
uvjet     : ODD izraz | izraz relacija izraz
relacija  : = | <> | < | > | <= | >=
```

Značenje relacija je:

- `=` jednako
- `<>` različito
- `<` manje
- `>` veće
- `<=` manje ili jednako
- `>=` veće ili jednako

Rezervirana riječ **ODD** predstavlja logičku funkciju koja vraća vrijednost `True` ako je vrijednost izraza neparna.

Dakle, selekcija u jeziku **PL/0** ima samo **THEN** granu. Značenje je: ako je postavljeni uvjet istinit, izvrši naredbu iza rezervirane riječi **THEN**. Inače, prijeđi na slijedeću naredbu.

WHILE PETLJA

Druga složena naredba je `WHILE` petlja koja se piše prema pravilu:

```
petlja : WHILE uvjet DO naredba
```

Značenje `WHILE` petlje je kao u Pascalu: Ponavlja se naredba unutar petlje sve dok je postavljeni uvjet istinit. Ako treba ponoviti izvršavanje više naredbi, koristi se (kao i u Pascalu) niz naredbi. Podrazumijeva se da, ako se naredbe petlje izvrše barem jedanput, mora se osigurati promjena vrijednosti varijabli sadržanih u uvjetu tako da u određenom času uvjet postane neistinit. U suprotnom bismo imali beskonačnu petlju.

Primjeri programa

Da rekapituliramo: PL/0 ima samo jedan tip podataka – cjelobrojni. Mogu se definirati cjelobrojne konstante i deklarirati cjelobrojne varijable. Procedure omogućuju uvođenje lokalnih varijabli i konstanti. Prijenos vrijednosti iz jedne procedure u drugu, odnosno u glavni program, isključivo je preko globalnih varijabli. Dopušten je rekurzivni poziv procedure. Na kraju ovoga kratkog opisa jezika PL/0 evo i nekoliko primjera programa.

♣ Primjer 7.1

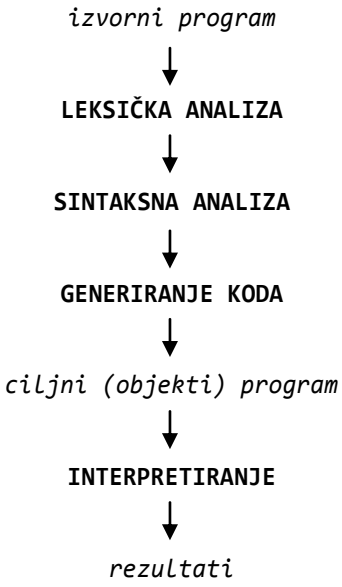
Evo primjera programa napisanog u jeziku **PL/0**:

```
CONST M = 7, N = 85;  VAR X, Y, Z, Q, R;

PROCEDURE MULTIPLY;
    VAR A, B;
  BEGIN
    A := X; B := Y; Z := 0;
    WHILE B > 0 DO BEGIN
      IF ODD B THEN Z := Z+A;
      A := 2*A; B := B/2;
      END
  END;

PROCEDURE DIVIDE;
    VAR W;
  BEGIN R := X; Q := 0; W := Y;
    WHILE W <= R DO W := 2*W;
    WHILE W > Y DO BEGIN
      Q := 2*Q; W := W/2;
      IF W <= R THEN BEGIN
        R := R-W; Q := Q+1
        END
      END
  END;

PROCEDURE GCD;
    VAR F, G;

  BEGIN
    F := X; G := Y;
    WHILE F <> G DO BEGIN
      IF F < G THEN G := G-F;
      IF G < F THEN F := F-G;
      Z := F
      END
  END;

BEGIN
  X := M;  Y := N;  CALL MULTIPLY;
  X := 25; Y := 3;  CALL DIVIDE;
  X := 84; Y := 36; CALL GCD
END.
```

Primjećujemo da PL/0 neodoljivo podsjeća na Pascal, pa ćemo, primjenjujući semantiku naredbi Pascala, brzo zaključiti da ovaj program sadrži poznate algoritme za množenje, dijeljenje i nalaženje najvećeg zajedničkog djelitelja dvaju prirodnih brojeva.

7.2 PREVOĐENJE

Proces prevođenja programa pisanih u jeziku **PL/0** izvodi se u četiri faze:



Poslije treće faze izvorni je program preveden u ciljni ili objektni program kojeg prihvaća i izvršava interpretator.

Leksička analiza

U prvoj se fazi prevođenja, leksičkoj analizi, učitava ulazni niz znakova i grupira u riječi (ili simbole). Iz definicije sintakse jezika **PL/0** slijedi da postoje slijedeće vrste riječi:

- rezervirane riječi (`BEGIN`, `CALL`, `CONST`, `DO`, `END`, `IF`, `ODD`, `PROCEDURE`, `THEN`, `VAR`, `WHILE`)
- imena (konstanti, varijabli ili procedura)
- brojevi (cjelobrojnog tipa)
- posebni simboli: `:=`
- ostali znakovi: `( ) , ; . + - * /  = ≠ < > # $ %`

Imena se tvore prema danim pravilima uz uvjet da moraju biti različita od rezerviranih riječi. Sve se riječi pišu bez razmaka. Ta dva uvjeta su dovoljna za primjenu izravne leksičke analize koja je realizirana potprogramom `Getsym`.

Sintaksna analiza

Za rješenje problema sintaksne analize, druge faze prevođenja, izabrana je dobro nam poznata metoda *rekurzivni spust*. U realizaciji tog postupka u Pascalu dobiva se niz općenito rekurzivnih procedura koje prate sintaksnu strukturu jezika. Izbor odgovarajuće procedure jednoznačno je određen prepoznavanjem prve riječi naredbe, a izlazak iz proce-

dure je nailaskom na jedan od simbola koji jednoznačno označuju kraj naredbe. Na primjer, ako je `CONST` tekući simbol ulaznog niza (programa), poziva se procedura `Blok`, a ako je tekući simbol `CALL`, `BEGIN`, `IF`, `WHILE` ili ime (varijable), poziva se procedura `Naredba`. U priloženom ćemo programu lako prepoznati pojedine procedure po njihovom nazivu. Na primjer, procedura `Vardeclaration` odnosi se na deklariranje varijabli, a procedura `Izraz` na sintaksnu kategoriju ***izraz***.

Generiranje koda (1)

Poslije leksičke i sintaksne analize slijedi generiranje koda. Da bi se zadržala što veća jednostavnost i prihvatljivost postupka sintaksne analize i osigurala jednoprolaznost u generiranju koda i njegovoj interpretaciji, uveden je apstraktni (hipotetički) procesor prilagođen semantici jezika **PL/0** i nazvan **PL/0** stroj. Taj se stroj sastoji iz dva temeljna dijela: instrukcijskog registra i tri adresna registra.

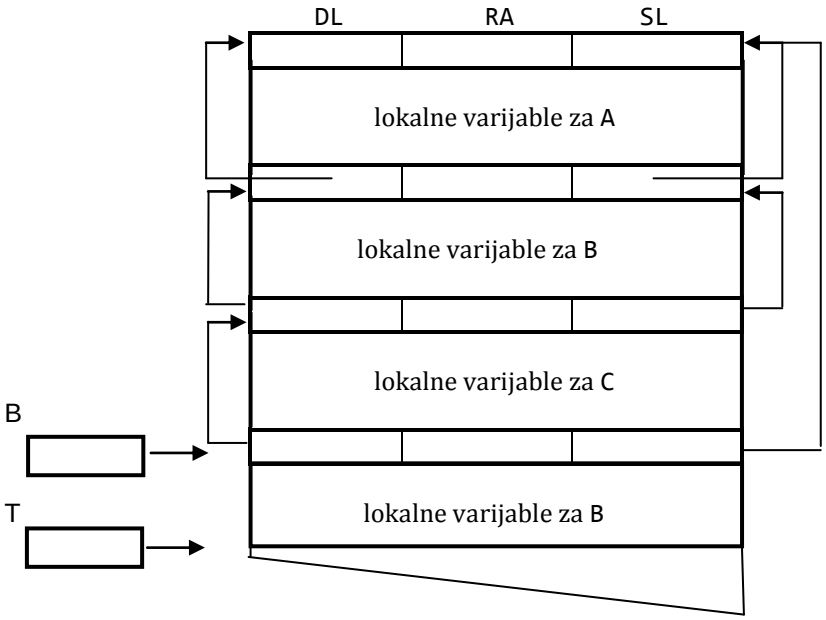
Instrukcijski registar ***I*** sadrži instrukciju koja se trenutačno izvršava. "Top-stack" registar ***T*** adresira element na vrhu potisne liste (stoga) ***S*** za smještaj inicijalnih podataka, međurezultata i konačnih rezultata. Programski adresni registar ***P*** označuje sljedeću instrukciju koja će biti interpretirana. Bazni adresni registar ***B*** sadrži adresu memorijskog segmenta koji je posljednji alociran. Programsku memoriju, nazvanu "code", puni prevodilac i ona ostaje nepromijenjena tijekom interpretacije mnemoničkog koda.

Svaka procedura u jeziku **PL/0** može sadržavati lokalne varijable. Procedure se mogu pozivati rekurzivno, pa se ne smije alocirati memorijski prostor za lokalne varijable prije poziva procedure. Tako se segmenti podataka za pojedine procedure pohranjuju u potisnu listu ***S*** na takav način da je prilikom njihove interpretacije u potpunosti zadovoljeno načelo "First-In – Last-Out" (`FILO`). Svaka procedura ima neke vlastite interne informacije, kao što su:

`RA` - povratna adresa, `DL` - dinamička veza i `SL` - statička veza

`RA` predstavlja adresu naredbe za poziv procedure (CALL naredbe) ili tzv. povratnu adresu. `DL` predstavlja adresu memorijskog segmenta "pozivnika", tzv. dinamičku vezu. Drugim riječima, dinamička veza pokazuje na adresu posljednje alociranog memorijskog segmenta i ta je adresa upamćena u baznom registru ***B***. S obzirom na to da se alokacija memorijskog prostora obavlja tijekom interpretacije koda, prevodilac ne može generiranom kodu pridružiti apsolutnu adresu. On može odrediti lokaciju varijabli samo unutar memorijskog segmenta i iz toga slijedi da omogućuje samo relativno adresiranje. Interpretator dodaje tzv. odstojanje aktualnoj baznoj adresi. Ako je ta varijabla lokalna za proceduru koja se trenutačno interpretira, bazna je adresa dana u ***B*** registru. U slučaju da varijabla nije lokalna, bazna se adresa pribavlja spuštanjem po nizu memorijskih segmenata. Naime, prevodilac može znati samo statičku dubinu memorijskog segmenta, a niz adresa dinamičkih veza podržava povijest aktiviranih procedura.

Na primjer, neka procedura `A` poziva proceduru `B`, procedura `B` poziva proceduru `C` koja je deklarirana kao lokalna proceduri `B` i `C` poziva `B` rekurzivno. Na sljedećoj slici prikazana je potisna lista **PL/0** stroja.



Kažemo da je procedura A deklarirana na razini 1, B na razini 2 i C na razini 3. Ako se varijabli X, deklariranoj u A, pristupa u proceduri B, prevodilac "zna" da između A i B postoji razlika razina jednaka 1. Spuštanjem za jedan korak po korak u nizu dinamičkih veza omogućuje se pristup varijabli deklariranoj u C. Dakle, potrebno je osigurati pravilno povezivanje memorijskih segmenata. To je omogućeno sa SL. Adrese su generirane kao parovi brojeva koji predstavljaju razliku razine odstojanja unutar memorijskog segmenta. Svaka lokacija sadrži adresu ili cijeli broj.

Jezik PL/0 stroja

Skup instrukcija **PL/0** stroja određen je zahtjevima **PL/0** jezika. To je simbolički ("asemblerski") jezik koji sadrži slijedeće mnemoničke naredbe:

- **LIT** - umetanje broja na vrh stoga
- **LOD** - umetanje varijable na vrh stoga
- **STO** - instrukcija pohranjivanja
- **CAL** - poziv potprograma (procedure)
- **INT** - alociranje memorije u stogu s inkrementiranjem pokazivača stoga T
- **JMP**, **JPC** - instrukcija za bezuvjetni i uvjetni prijenos kontrole upotrijebljenih u **IF** i **WHILE** naredbi
- **OPR** - skup aritmetičkih i relacijskih operatora

Format instrukcija određen je potrebom da se uvedu tri parametra: operacijski kod **f**, razina **l** i parametar **a**. Ako je operacijski kôd jednak OPR, parametar **a** predstavlja identifikaciju operatora. U drugim slučajevima parametar **a** može biti broj (**LIT** ili **INT**), programska adresa (**JMP**, **JPC** ili **CAL**), ili adresa varijable (**LOD**, **STO**). Detaljan opis **PL/0** stroja dan je u proceduri `Interpreter()`.

Generiranje koda (2)

Da bi mogao sastaviti instrukciju **PL/0** stroja, prevodilac mora znati njezin kôd i parametre, koji mogu biti ili broj ili adresa. Te vrijednosti prevodilac pridružuje tijekom definiranja konstanti, deklariranja varijabli i procedura. Radi toga tablica imena pored samih imena sadrži i njihove atribute. Ako ime predstavlja konstantu, njegov atribut je vrijednost te konstante (cjelobrojna vrijednost), a ako je to varijabla, njezin atribut je adresa koja se sastoji od "odstojanja" i razine. Konačno, ako ime određuje ime procedure, njegovi su atributi ulazna adresa procedure i njezina razina.

Svakom se deklaracijom varijable inkrementira alokacijski indeks za 1. Alokacijski je indeks **dx** inicijaliziran prije početka prevođenja procedure, čime memorijski segment postaje prazan (preciznije, inicijalno mu je stanje jednako 3 jer svaki memorijski segment sadrži najmanje tri varijable: RA, DL i SL). Procedura `Enter` određuje atribute imena i unosi ih u tablicu imena.

S informacijama o operandima generiranje aktualnog koda prilično je pojednostavljeno. Sa stogovnom organizacijom **PL/0** stroja korespodencija se između operanada svodi samo na dva elementa, primjenom tzv. "postfiksne" notacije, u kojoj operacije stoje iza operanada i u kojoj je suvišna upotreba zagrada. Konvencionalna, ili "infiksna" notacija, prevodi se u posfiksnu notaciju u procedurama `Expression` i `Term`. Jednostavna mogućnost interpretiranja tako prevedenih izraza glavni je razlog uporabe postfiksne notacije.

Nešto je manje jednostavno prevođenje selekcije i petlje. U tom je slučaju potrebno generiranje tzv. "jump" instrukcija za koje nisu poznate adrese. Jedno od rješenja je drugi prolaz u prevođenju, što nije cilj jer se uz jednoprolaznu leksičku i sintaksnu analizi želi i jednoprolazno prevođenje. Taj problem je riješen primjenom tzv. "fixup" metode koja dopušta dodavanje, u početku nepoznatih adresa u trenutku kad budu poznate. Detaljnije se može naći u dijelu programa koji se odnosi na sintaktičku analizu i prevođenje selekcije (IF-THEN naredbe) i WHILE petlje.

Za generiranje mnemoničkih instrukcija (tri komponente, odnosno parametra) uvedena je procedura `Gen`. Ona automatski inkrementira indeks **cx** koji određuje lokaciju slijedeće instrukcije.

♣ Primjer 7.2

Kao primjer generiranja koda pogledajmo prevođenje definicije globalnih konstanti i varijabli, te proceduru MULTIPLAY iz danog primjera programa u jeziku **PL/0**. Desno su dani potrebni komentari.

```
CONST M = 7, N = 85;
VAR   X, Y, Z, Q, R;

PROCEDURE MULTIPLY;
    VAR A, B;
  BEGIN
    A := X; B := Y; Z := 0;
    WHILE B > 0 DO BEGIN
      IF ODD B THEN Z := Z+A;
      A := 2*A; B := B/2;
      END
  END;
```

⇨

```
 2 INT  0, 5   al. prost.   15 JPC  0, 20
 3 LOD  1, 3   X            16 LOD  1, 5   Z
 4 STO  0, 3   A            17 LOD  0, 3   A
 5 LOD  1, 4   Y            18 OPR  0, 2   +
 6 STO  0, 4   B            19 STO  1, 5   Z
 7 LIT  0, 0   0            20 LIT  0, 2   2
 8 STO  1, 5   Z            21 LOD  0, 3   A
 9 LOD  0, 4   B            22 OPR  0, 4   *
10 LIT  0, 0   0            23 STO  0, 3   A
11 OPR  0, 12  >            24 LOD  0, 4   B
12 JPC  0, 29               25 LIT  0, 2   2
13 LOD  0, 4   B            27 STO  0, 4   B
14 OPR  0, 6   ODD          28 JMP  0, 9
15 JPC  0, 20               29 OPR  0, 0   return
```

7.3 INTERPRETATOR

Na kraju prevođenja, pod uvjetom da nije bilo leksičkih niti sintaksnih pogrešaka u programu napisanom u jeziku **PL/0**, poziva se procedura **Interpret** koja izvršava naredbe objektnog jezika. Preostalo je još da priložimo kompletan program interpretatora jezika **PL/0** napisan u Pythonu. U modulu `PL0_init.py` dane su konstante i definirane osnovne strukture podataka interpretatora.

PL0_init.py

```
Word   = { 'BEGIN' : 'beginsym', 'CALL'      : 'callsym',
           'CONST' : 'constsym', 'DO'        : 'dosym',
           'END'   : 'endsym',   'IF'        : 'ifsym',
           'ODD'   : 'oddsym' ,  'PROCEDURE' : 'procsym',
           'THEN'  : 'thensym',  'VAR'       : 'varsym',
           'WHILE' : 'whilesym'                          }

Ssym   = { '+' : 'plus',    '-' : 'minus',   '*' : 'times',
           '/' : 'slash',   '(' : 'lparen',  ')' : 'rparen',
           '=' : 'eql',     ',' : 'comma',   '.' : 'period',
           '#' : 'neq',     '<' : 'lss',     '>' : 'gtr',
           '$' : 'leq',     '%' : 'geq',     ';' : 'semicolon' }

symbol = [ 'nul',      'ident',   'number',    'plus',    'minus',
           'times',    'slash',   'oddsym',    'eql',     'neq',
           'lss',      'leq',     'gtr',       'geq',     'lparen',
           'rparen',   'comma',   'semicolon', 'period',  'becomes',
           'beginsym', 'endsym',  'ifsym',     'thensym', 'whilesym',
           'dosym',    'callsym', 'constsym',  'varsym',  'rocsym' ]

LIT, OPR, LOD, STO, CAL, INT, JMP, JPC = \
  'LIT', 'OPR', 'LOD', 'STO', 'CAL', 'INT', 'JMP', 'JPC'
fct    = [ 'LIT', 'OPR', 'LOD', 'STO', 'CAL', 'INT', 'JMP', 'JPC' ]

"""
   LIT 0, a  load constant a
   OPR 0, a  execute operation a
   LOD l, a  load variable l,a
   STO l, a  store varijable l,a
   CAL l, a  call procedure a at level l
   INT 0, a  increment t-register by a
   JMP 0, a  jump to a
   JPC 0, a  jump conditional to a }
"""
RO     = ['eql', 'neq', 'lss', 'gtr', 'leq', 'geq']
RelOp  = [  '=',   '#',   '<',   '>',   '$',   '%']

GenRO  = { 'eql' : ('OPR', 0,  8),  'neq' : ('OPR', 0,  9),
           'lss' : ('OPR', 0, 10),  'geq' : ('OPR', 0, 11),
           'gtr' : ('OPR', 0, 12),  'leq' : ('OPR', 0, 13) }

Mnemonic   = fct
Declbegsys = ['constsym', 'varsym',  'procsym']
Statbegsys = ['beginsym', 'callsym', 'ifsym', 'whilesym']
Facbegsys  = ['ident',    'number',  'lparen']
Object     = ['constant', 'variable', 'procedur']
```

```
class table :
  def __init__ (self):
    self.val = [ ('', '', '', '', '') ]
    self.len = 1
    self.tab = { 0: 0 }

  def insert (self, Name, Kind, Val, Level, Adr) :
    self.val += [(Name, Kind, Val, Level, Adr)]
    self.tab.update ( { Name : self.len } )
    self.len += 1

  def n (self, i) : return self.val[i][0]  # name
  def k (self, i) : return self.val[i][1]  # kind
  def v (self, i) : return self.val[i][2]  # val
  def l (self, i) : return self.val[i][3]  # level
  def a (self, i) : return self.val[i][4]  # adr
```

PL0-INT.py

```
# PROGRAM PL0_interpreter;

'''
program = block "." .

block = [ "const" ident "=" number {"," ident "=" number} ";"]
        [ "var" ident {"," ident} ";"]
        { "procedure" ident ";" block ";" } statement .

statement = [ ident ":=" expression | "call" ident |
            "begin" statement {";" statement } "end" |
            "if" condition "then" statement |
            "while" condition "do" statement ].

condition = "odd" expression |
            expression ("="|"<>"|"<"|"<="|">"|">=") expression .

expression = [ "+"|"-"] term { ("+"|"-") term}.

term = factor {("*"|"/") factor}.

factor = ident | number | "(" expression ")".
'''

from   PL0_init     import *

def Interpreter () :

  def Base (l) :
    B1 = B  # find base L levels down
    while l > 0 :
      B1 = S[B1]; l -= 1
    return B1

  # {Interpret}
  Stack = 1024
  print NL, 'START PL/0', NL
  T = 0;  B = 1;  P = 0; P0 = Code[P][2]; S = [0] *Stack;
```

```
  while True :
    F, L, A = Code[P]; P += 1
    print NL, P-1, Tb, F, Tb, L, Tb, A,
    if   F == LIT :  T += 1; S [T] = A
    elif F == OPR :
      if   A == 0 :
        if len (Code) == P : break
        else               : T = B-1; P = S [T+3]; B = S [T+2]
      elif A == 1 : S [T] = -S [T]
      elif A == 6 : S [T] = 1 *(S[T] % 2 <> 0) # ODD
      else        : # <operator>
        T -= 1
        if   A ==  2 : S [T] += S[T+1]
        elif A ==  3 : S [T] -= S[T+1]
        elif A ==  4 : S [T] *= S[T+1]
        elif A ==  5 : S [T] /= S[T+1]
        elif A ==  8 : S [T] = 1 *(S[T] == S[T+1])
        elif A ==  9 : S [T] = 1 *(S[T] <> S[T+1])
        elif A == 10 : S [T] = 1 *(S[T] <  S[T+1])
        elif A == 11 : S [T] = 1 *(S[T] >= S[T+1])
        elif A == 12 : S [T] = 1 *(S[T] >  S[T+1])
        elif A == 13 : S [T] = 1 *(S[T] <= S[T+1])
    elif F == LOD : T += 1;  S [T] = S [Base (L) + A]
    elif F == STO : S [Base (L) + A] = S[T];  print ' -->', S[T];  T -= 1
    elif F == CAL :
      print NL, '-'*27
      ' generate new block mark '
      S [T+1] = Base (L)
      S [T+2] = B
      S [T+3] = P;
      B = T +1; P = A
    elif F == INT : T += A
    elif F == JMP : P = A
    elif F == JPC :
      if S [T] == 0 : P = A
      T -= 1
    if P == 0 : break
  print NL, 'END PL/0'
  # END {Interpret}

def Ucitaj_PROG (Poruka, Tip): # Učitavanje programa
  G = fileopenbox(Poruka, None, Tip, '*')
  if G == None: G = ''
  if os.path.exists(G):
    P = []
    for line in open (G, 'r') : P.append (line[:-1])
    Ok = True
    if G.rfind('\\') > 0: G = G[G.rfind('\\')+1:]
  else:
    Ok = False
  return G, Ok, P

def Ucitaj_L (i) :
  red = P[i] + ' '
  print "%02d" %(i+1), P[i]
  return red
```

```
def Error (N) :
  global Err
  print ' ****', ' '* (Cc-1)+chr (24), N;  Err += 1

def Nadji_id () :
  global A, Err, End, Cc, Line, Ch, LN, Ident, Sym
  if Ident == {}  : Ident.update({ A : 0 });  Sym = A
  elif A in Ident : Sym = A
  else            : Sym = 'ident'

def Get_Ch () :
  global Line, Cc, P, Ii, Ch, End
  if Cc == len (Line) -1 and Ii < len (P)-1 :
    Ii   += 1;
    Line = Ucitaj_L (Ii); Cc = -1
    Line.upper()
  Cc   += 1;  Ch = Line[Cc]
  End = Ii == len(P) -1 and Cc == len (Line) -1

def Get_Sym () :
  global Err, End, Cc, Line, Ch, LN, Ident, A, Sym, Id
  A = '';  Get_Ch ();
  while ord (Ch) <= 32 and not End : Get_Ch()

  if Ch in Slova :
    ' Identifikator ili rezervirana rijec '
    while Ch in Slova +Brojke: A += Ch;  Get_Ch ()
    if   A in Word : Sym = Word[A]   # rezervirana riječ
    else           : Sym = 'ident'; Id = A
    Cc -= 1
  else :
    if Ch in Brojke : # broj
      while Ch in Brojke: A += Ch;  Get_Ch ()
      Sym = 'number';  Cc -=  1
    elif Ch == ':' :
      Get_Ch ()
      if Ch == '=' : Sym = 'becomes';  Get_Ch ()
      else         : Sym = 'nul'
    elif Ch in ['<', '=', '>'] :
      Ch0 = Ch
      Get_Ch ();  SymRO = Ch0 +Ch
      if SymRO not in ['<>', '>=',  '<='] : SymRO = Ch0;  Cc -= 1
      Sym = RelOp2 [SymRO]
    elif Ch in Ssym : Sym = Ssym [Ch]
    else :
      print '*** LEKSIČKA POGREŠKA U REDU', Ii+1
      print 'Sym =', '*'+Sym+'*';  print 'Ch  =', '*'+Ch+'*'
      Gr = True;
      exit ()

def Gen (X, Y, Z) :
  global Code, Cx
  Code.append ((X, Y, Z))
  Cx += 1

def Test (S1, S2, N) :
  if Sym not in S1 :
    Error (N); S1.append (S2)
    while Sym not in S1 : Get_Sym()
```

```
def Block (Lev, Tx, Fsys) :
  global Cc, Line, Ch, Sym, A, Err, Code, Id, Dx

  def ChangeCode (i, adr) :
    x0, x1, x2 = Code[i]; x2 = adr; Code[i] = (x0, x1, x2)

  def Enter (K) :
    global Id, Dx

    if   K == 'constant' : Table.insert (Id, K, int(A), '', '')
    elif K == 'variable' : Table.insert (Id, K, 0, Lev, Dx); Dx += 1
    elif K == 'procedur' : Table.insert (Id, K, '', Lev, len(Code) +1)

  def position (Id) :
    Ok = False; i = len (Table.val) -1
    while not Ok and i >= 0 :
      Ok = Table.n(i) == Id
      if not Ok : i -= 1
    if not Ok : Error ('pogreška: ne postoji identifikator '+Id); return 0
    return i

  def Constdeclaration () :
    global Id, Dx
    if Sym == 'ident' :
      Get_Sym ()
      if Sym in ['eql', 'becomes'] :
        if Sym == 'becomes' : Error (1)
        Get_Sym ()
        if Sym == 'number'  : Enter ('constant');  Get_Sym ()
        else                : Error (2)
      else : Error (3)
    else :   Error (4)


  def Vardeclaration () :
    global Dx
    if Sym == 'ident' : Enter ('variable'); Get_Sym ()
    else              : Error (4)

  def Listcode () : # generirana lista kodova tekućeg bloka
    for i in range (Cx0, Cx) :
      com, lev, adr = Code[i]
      print i, com, lev, adr

  def MATCH (Sym, Word, ErCode) :
    if Sym == Word : Get_Sym ()
    else           : Error (ErCode)


  def Statement (Fsys)    : # VAR I, Cx1, Cx2: integer;

    def Expression (Fsys) : # VAR Addop: symbol;

      def Term (Fsys)     : # VAR Mulop: symbol;

        def Factor (Fsys) : # VAR I: integer;
          Test (Facbegsys, Fsys, 24)
```

```
        while Sym in Facbegsys :
          if Sym == 'ident' :
            i = position (Id)
            if   i == 0 : Error (11)
            elif Table.k (i) == 'constant' : Gen (LIT, 0, Table.v(i))
            elif Table.k (i) == 'variable' : Gen (LOD, Lev-Table.l(i), Table.a(i))
            elif Table.k (i) == 'procedur' : Error (21)
            Get_Sym ()
          else :
            if Sym == 'number' : Gen (LIT, 0, int(A)); Get_Sym ()
            else :
              if Sym == 'lparen' :
                Get_Sym (); Expression (['rparen']+Fsys)
                if Sym == 'rparen' : Get_Sym ()
                else               : Error (22)
            Test (Fsys, ['lparen'], 23)
          # end Factor
        # <Term> :
        Op2 = ['times', 'slash']
        Factor (Fsys +Op2)
        while Sym in Op2 :
          Mulop = Sym;  Get_Sym ();  Factor (Fsys +Op2)
          if Mulop == 'times' : Gen (OPR, 0, 4)
          else                : Gen (OPR, 0, 5)
        # end {Term}
      # <Expression> :
      Op1 = ['plus', 'minus']
      if Sym in Op1 :
        Addop = Sym;  Get_Sym;  Term (Fsys +Op1)
        if Addop =='minus' : Gen (OPR, 0, 1)
      else          :
        Term (Fsys +Op1)
        while Sym in Op1 :
          Addop = Sym;  Get_Sym ();  Term (Fsys +Op1)
          if Addop == 'plus' : Gen (OPR, 0, 2)
          else               : Gen (OPR, 0, 3)
      # end {Expression};
    def Condition (Fsys) : # VAR Relop: symbol;
      if Sym == 'oddsym' : Get_Sym ();  Expression (Fsys);  Gen (OPR, 0, 6)
      else               :
        Expression (RelOp +Fsys);
        if Sym not in RO : Error (20)
        else :
          Relop = Sym;  Get_Sym ();  Expression (Fsys)
          A, B, C = GenRO [Relop]; Gen (A, B, C)
      # end {Condition};
    # <Statement> :
    if Sym == 'ident' :
      I0 = position (Id)
      if I0 == 0 : Error (11)
      elif Table.k(I0) <> 'variable' : Error (12);  I0 = 0
      Get_Sym ()
      if Sym == 'becomes' : Get_Sym ()
      else                : Error (13)
      Expression (Fsys)
      if I0 <> 0 : Gen (STO, Lev-Table.l(I0), Table.a(I0))
```

```
  elif Sym == 'callsym' :
    Get_Sym ()
    if Sym <> 'ident' : Error (14)
    else :
      I = position (Id)
      if I == 0 : Error (11)
      else :
        if Table.k(I) == 'procedur' : Gen (CAL, Lev-Table.l(I), Table.a(I))
        else                        : Error (15)
        Get_Sym ()
  elif Sym == 'ifsym' :
     Get_Sym ();  Condition (['thensym', 'dosym'] + Fsys)
     if Sym == 'thensym' : Get_Sym ()
     else                : Error (16)
     Cx1 = Cx;  Gen (JPC, 0, 0)
     Statement (Fsys)
     ChangeCode (Cx1, Cx)
  elif Sym == 'beginsym' :
    Get_Sym ();  Statement (['semicolon', 'endsym'] +Fsys)
    while Sym in ['semicolon'] +Statbegsys :
      MATCH (Sym, 'semicolon', 10)
      Statement (['semicolon', 'endsym'] + Fsys)
    MATCH (Sym, 'endsym', 17)
  elif Sym == 'whilesym' :
    Cx1 = Cx;  Get_Sym (); Condition (['dosym'] +Fsys);
    Cx2 = Cx;  Gen (JPC, 0, 0);
    MATCH (Sym, 'dosym', 18)
    Statement (Fsys);  Gen (JMP, 0, Cx1);  ChangeCode (Cx2, Cx)
  Test (Fsys, [], 19)
  # end {Statement}


def Size (L) :
  for i in range (len (Table.val) -1, 0, -1) :
    if Table.k(i) == 'variable' and Table.l(i) == L : return Table.a(i) +1

# <Block>
Dx = 3;   Tx0 = Tx;  x0, x1, x2, x3, x4 = Table.val[Tx];
x4 = Cx;  Table.val[Tx] = (x0, x1, x2, x3, x4)

Gen (JMP, 0, 0);

while True :
  if Sym == 'constsym' :
    Get_Sym ();
    while True :
      Constdeclaration ()
      while Sym == 'comma' : Get_Sym ();  Constdeclaration ()
      MATCH (Sym, 'semicolon', 5)
      if Sym <> 'ident' : break
  if Sym == 'varsym' :
    Get_Sym ()
    while True :
      Vardeclaration ()
      while Sym == 'comma' : Get_Sym ();  Vardeclaration ()
      MATCH (Sym, 'semicolon', 5)
      if Sym <> 'ident' : break
```

```
    while Sym == 'procsym' :
      Get_Sym ()
      if Sym == 'ident' : Enter ('procedur'); Get_Sym ()
      else              : Error (4)
      MATCH (Sym, 'semicolon', 5)
      Block (Lev+1, Tx, ['semicolon'] +Fsys)
      if Sym == 'semicolon' :
        Get_Sym ();  Test (Statbegsys +['ident', 'procsym'], Fsys, 6)
      else : Error (5)
    Test (Statbegsys + ['ident'], Declbegsys, 7)
    if Sym not in Declbegsys : break

  # Code [Table [Tx0].Adr].A = Cx;
  t0, t1, t2, t3, adr = Table.val[Tx0];  ChangeCode (adr, Cx)
  Table.val[Tx0] = (t0, t1, t2, t3, Cx); ChangeCode (0, Cx)
  dx = Size(Lev) # size of data segment
  Cx0 = Cx; Gen (INT, 0, dx); Statement (['semicolon', 'endsym'] +Fsys)
  Gen (OPR, 0, 0) # {return}
  Test (Fsys, [], 8);  Listcode ()
  # end {Block}

while True:
  Ime_PROG, Ok, P = Ucitaj_PROG ('***', '*.PL0;*.TXT')
  if Ok : print Ime_PROG, NL
  else  : print '*** ne postoji program', Ime_PROG;  exit (99)
  print; Cc = -1; Cx = 0; Ch = ' '; Err = 0; Code = []; Ii = 0; Table = table ()
  Line = Ucitaj_L (Ii);  Get_Sym ()
  Block (0, 0, ['period'] +Declbegsys +Statbegsys);
  if Sym <> 'period' : Error (9)
  if Err == 0: Interpreter ()
  else       : print 'ERRORS IN PL/0 PROGRAM'
  print NL;  Ponovi = raw_input ('UČITAVAM DRUGI PROGRAM (D/N)? ')
  if Ponovi.upper()[0] <> 'D' : break
```

7.4 PRIMJERI

Dajemo nekoliko primjera interpretiranja programa napisanih u jeziku **PL/0**. Svi se programi učitavaju iz tekstualnih datoteka.

♣ Primjer 7.3

Najprije pogledajmo interpretaciju jednog jednostavnog programa:

```
CONST M = 7, N = 85;
VAR X, Y, Z;
BEGIN
  X := M; Y := N;
  Z := X +Y
END.
```

```
0 CONST M = 7, N = 85;
1 VAR X, Y, Z;
1 BEGIN
2   X := M; Y := N;
6   Z := X +Y
8 END.
```

```
1  INT  0    6
2  LIT  0    7
3  STO  0    3
4  LIT  0   85
5  STO  0    4
6  LOD  0    3
7  LOD  0    4
8  OPR  0    2
9  STO  0    5
10  OPR  0    0
START PL/0
7
85
92
END PL/0
```

♣ Primjer 7.4

Program dan u primjeru 7.1 sadrži procedure:

```
 0 CONST
 1   M = 7, N = 85;
 1 VAR
 1   X, Y, Z, Q, R;
 1
 1 PROCEDURE MULTIPLY;
 1     VAR A, B;
 2   BEGIN
 3     A := X; B := Y; Z := 0;
 9     WHILE B > 0 DO BEGIN
13           IF ODD B THEN Z := Z+A;
20           A := 2*A; B := B/2;
28           END
28   END;
```

```
   2  INT  0    5
   3  LOD  1    3
   4  STO  0    3
   5  LOD  1    4
   6  STO  0    4
   7  LIT  0    0
   8  STO  1    5
   9  LOD  0    4
  10  LIT  0    0
  11  OPR  0   12
  12  JPC  0   29
  13  LOD  0    4
  14  OPR  0    6
  15  JPC  0    0
  16  LOD  1    5
  17  LOD  0    3
  18  OPR  0    2
  19  STO  1    5
  20  LIT  0    2
  21  LOD  0    3
  22  OPR  0    4
  23  STO  0    3
```

```
   24  LOD  0    4
   25  LIT  0    2
   26  OPR  0    5
   27  STO  0    4
   28  JMP  0    9
   29  OPR  0    0

30 PROCEDURE DIVIDE;
30     VAR W;
31   BEGIN R := X; Q := 0; W := Y;
38     WHILE W <= R DO W := 2*W;
47     WHILE W > Y DO BEGIN
51           Q := 2*Q; W := W/2;
59           IF W <= R THEN BEGIN
63              R := R-W; Q := Q+1
69              END
71           END
71   END;

   31  INT  0    4
   32  LOD  1    3
   33  STO  1    7
   34  LIT  0    0
   35  STO  1    6
   36  LOD  1    4
   37  STO  0    3
   38  LOD  0    3
   39  LOD  1    7
   40  OPR  0   13
   41  JPC  0   47
   42  LIT  0    2
   43  LOD  0    3
   44  OPR  0    4
   45  STO  0    3
   46  JMP  0   38
   47  LOD  0    3
   48  LOD  1    4
   49  OPR  0   12
   50  JPC  0   72
   51  LIT  0    2
   52  LOD  1    6
   53  OPR  0    4
   54  STO  1    6
   55  LOD  0    3
   56  LIT  0    2
   57  OPR  0    5
   58  STO  0    3
   59  LOD  0    3
   60  LOD  1    7
   61  OPR  0   13
   62  JPC  0    0
   63  LOD  1    7
   64  LOD  0    3
   65  OPR  0    3
   66  STO  1    7
   67  LOD  1    6
   68  LIT  0    1
   69  OPR  0    2
```

```
    70  STO  1    6
    71  JMP  0   47
    72  OPR  0    0

 73 PROCEDURE GCD;
 73     VAR F, G;
 74   BEGIN
 75     F := X; G := Y;
 79     WHILE F # G DO BEGIN
 83           IF F < G THEN G := G-F;
 91           IF G < F THEN F := F-G;
 99           Z := F
100           END
101   END;

    74  INT  0    5
    75  LOD  1    3
    76  STO  0    3
    77  LOD  1    4
    78  STO  0    4
    79  LOD  0    3
    80  LOD  0    4
    81  OPR  0    9
    82  JPC  0  102
    83  LOD  0    3
    84  LOD  0    4
    85  OPR  0   10
    86  JPC  0    0
    87  LOD  0    4
    88  LOD  0    3
    89  OPR  0    3
    90  STO  0    4
    91  LOD  0    4
    92  LOD  0    3
    93  OPR  0   10
    94  JPC  0    0
    95  LOD  0    3
    96  LOD  0    4
    97  OPR  0    3
    98  STO  0    3
    99  LOD  0    3
    100  STO  1    5
    101  JMP  0   79
    102  OPR  0    0


103 BEGIN
104   X := M;  Y := N;  CALL MULTIPLY;
109   X := 25; Y := 3;  CALL DIVIDE;
114   X := 84; Y := 36; CALL GCD
119 END.

    103  INT  0    8
    104  LIT  0    7
    105  STO  0    3
    106  LIT  0   85
    107  STO  0    4
```

```
108  CAL  0    2
109  LIT  0   25
110  STO  0    3
111  LIT  0    3
112  STO  0    4
113  CAL  0   31
114  LIT  0   84
115  STO  0    3
116  LIT  0   36
117  STO  0    4
118  CAL  0   74
119  OPR  0    0
START PL/0
7
85
7
85
0
7
14
42
END PL/0
```

♣ Primjer 7.5

Jezik PL/0 dopušta rekurzivne pozive procedura kao što je to u sljedećem programu (faktorijel broja n):

```
VAR   f, n;
PROCEDURE Fakt;
VAR i;
BEGIN
  i := n;
  IF i <= 1 THEN f := 1;
  IF i >  1 THEN BEGIN
    n := n -1; CALL Fakt; f := i *f
    END
  END;
BEGIN
  n := 5; CALL Fakt
END.
```

```
01 VAR   f, n;
02
03 PROCEDURE Fakt;
04   VAR i;
05   BEGIN
06   i := n;
07   IF i <= 1 THEN f := 1;
08   IF i >  1 THEN BEGIN
09     n := n -1; CALL Fakt; f := i *f
10     END
11   END;
2 INT 0 4
3 LOD 1 4
4 STO 0 3
5 LOD 0 3
6 LIT 0 1
7 OPR 0 13
```

```
8 JPC 0 11
9 LIT 0 1
10 STO 1 3
11 LOD 0 3
12 LIT 0 1
13 OPR 0 12
14 JPC 0 24
15 LOD 1 4
16 LIT 0 1
17 OPR 0 3
18 STO 1 4
19 CAL 1 2
20 LOD 0 3
21 LOD 1 3
22 OPR 0 4
23 STO 1 3
24 OPR 0 0
12
13 BEGIN
14   n := 5; CALL Fakt
15 END.
25 INT 0 4
26 LIT 0 5
27 STO 0 4
28 CAL 0 2
29 OPR 0 0


START PL/0

0    JMP     0       25
25   INT     0       4
26   LIT     0       5
27   STO     0       4
*** 5
28   CAL     0       2
------------------------------
2    INT     0       25
3    LOD     1       4
4    STO     0       3
*** 5
5    LOD     0       3
6    LIT     0       1
7    OPR     0       13
8    JPC     0       11
11   LOD     0       3
12   LIT     0       1
13   OPR     0       12
14   JPC     0       24
15   LOD     1       4
16   LIT     0       1
17   OPR     0       3
18   STO     1       4
*** 4
19   CAL     1       2
------------------------------
```

```
2    INT     0       25
3    LOD     1       4
4    STO     0       3
*** 4
5    LOD     0       3
6    LIT     0       1
7    OPR     0       13
8    JPC     0       11
11   LOD     0       3
12   LIT     0       1
13   OPR     0       12
14   JPC     0       24
15   LOD     1       4
16   LIT     0       1
17   OPR     0       3
18   STO     1       4
*** 3
19   CAL     1       2
------------------------------
2    INT     0       25
3    LOD     1       4
4    STO     0       3
*** 3
5    LOD     0       3
6    LIT     0       1
7    OPR     0       13
8    JPC     0       11
11   LOD     0       3
12   LIT     0       1
13   OPR     0       12
14   JPC     0       24
15   LOD     1       4
16   LIT     0       1
17   OPR     0       3
18   STO     1       4
*** 2
19   CAL     1       2
------------------------------
2    INT     0       25
3    LOD     1       4
4    STO     0       3
*** 2
5    LOD     0       3
6    LIT     0       1
7    OPR     0       13
8    JPC     0       11
11   LOD     0       3
12   LIT     0       1
13   OPR     0       12
14   JPC     0       24
15   LOD     1       4
16   LIT     0       1
17   OPR     0       3
18   STO     1       4
*** 1
19   CAL     1       2
------------------------------
```

```
2    INT     0       25
3    LOD     1       4
4    STO     0       3
*** 1
5    LOD     0       3
6    LIT     0       1
7    OPR     0       13
8    JPC     0       11
9    LIT     0       1
10   STO     1       3
*** 1
11   LOD     0       3
12   LIT     0       1
13   OPR     0       12
14   JPC     0       24
24   OPR     0       0
20   LOD     0       3
21   LOD     1       3
22   OPR     0       4
23   STO     1       3
*** 2
24   OPR     0       0
20   LOD     0       3
21   LOD     1       3
22   OPR     0       4
23   STO     1       3
*** 6
24   OPR     0       0
20   LOD     0       3
21   LOD     1       3
22   OPR     0       4
23   STO     1       3
*** 24
24   OPR     0       0
20   LOD     0       3
21   LOD     1       3
22   OPR     0       4
23   STO     1       3
*** 120
24   OPR     0       0
29   OPR     0       0

END PL/0
```

8.
JEZIK DDH

—Chanson pour l'Auvergnat

Tebi Overnjanine pjevam ja
što si mi bez okolišanja
dao ona četiri drveta
kada mi je u životu bilo hladno.

Tebi koji si me ugrijao
kada su mi seljačine i djevojčure
svi ti ljudi "dobrih" namjera
pred nosom zalupili vrata.

Bila je to obična vatra od drveta
ali tako me je dobro ugrijala
da još plamti u duši mojoj
poput radosnog plamena.

Ti Overnjanine kad ćeš mrijet
kad će te grobar pokupit htjet
nek' te ponese preko nebesa
do Oca vječnoga

Tebi hosteso pjevam ja
koja si mi bez okolišanja
četiri okrajka od kruha dala
kad me je u životu morila glad.

Tebi koja si škrinju otvorila
kada su mi seljačine i djevojčure
svi ti ljudi "dobrih" namjera
zabavljali nad mojim patnjama

Bio je to samo običan kruh
ali tako me je tada dobro ugrijao
da još plamti u duši mojoj
kao za najsvečanijih dana

Ti hosteso kad ćeš mrijet
kad će te grobar pokupit htjet
nek' te ponese preko nebesa
do Oca vječnoga.

Tebi stranče pjevam ja
koji si bez okolišanja
lica tužnog osmjehnuo se
na me u rukama žandara.

Ti koji nisi pljeskao tada
kada su mi seljačine i djevojčure
svi ti ljudi "dobrih" namjera
smijali se do neba.

Bilo je to tek malo slatkoga meda
što mi je ugrijao bolno tijelo
ali on još plamti u duši mojoj
poput sunca golema.

Ti stranče kad ćeš mrijet
kad će te grobar pokupit htjet
nek' te ponese preko nebesa
do Oca vječnoga

Pjesma Overnjaninu
Chanson pour l'Auvergnat

(Georges BRASSENS/
Nada VRKLJAN KRIŽIĆ)

*Na kraju ćemo ove knjige primijeniti sva stečena znanja u opisu jednog posebnog jezika i realizaciji njegova predprocesora. Nazvat ćemo ga jezik **DDH**. Interesantno je da je taj jezik nastao sredinom sedamdesetih godina i publiciran je u čuvenoj knjizi "legende" teorije programiranja, prof. Dijkstre, a kao rezultat njegovih tadašnjih razmišljanja kako prikazati neke od svojih algoritama, rješenja poznatih problema. Dijsktra je zaključio da tada nije postojao odgovarajući jezik i u svojoj je monografiji „A Discipline of Programming", [Dij1976], definirao sintaksu i semantiku jednog svog jezika. Glavne odlike su mu:*

- *stroga definicija semantike,*
- *jednostavna sintaksa (mali broj naredbi),*
- *zadovoljenje svih poznatih principa strukturnog programiranja.*

S takvim karakteristikama posebno je podesan za učenje matematičkih principa programiranja, a nama je interesantan i kao primjer definicije i izvođenja jednog jezika za programiranje.

*U ovom smo poglavlju u potpunosti definirali jezik **DDH**, a to je osnovni Dijkstrin jezik dopunjen s nekoliko primitivnih naredaba.*

8.1 OSNOVNE DEFINICIJE

Prije nego prijeđemo na opis jezika **DDH**, uvodimo nekoliko definicija. Varijabla u matematici ne predstavlja nijednu posebnu vrijednost, kao na primjer u tvrdnji da za svaki prirodni broj n vrijedi $n^2>0$. U našem kontekstu pojam varijabla bit će korišten za nešto što egzistira u vremenu i čija je vrijednost u određenom trenutku konstantna.

Skup varijabli koje promjenom svojih vrijednosti u potpunosti određuje tok računanja po određenom postupku nazivaju se varijable stanja tog postupka.

Konstante su nosioci određenih vrijednosti koje se ne mijenjaju tokom računanja po određenom postupku.

Za opisivanje određenog postupka osim varijabli stanja uvodi se i pojam slobodna varijabla. Slobodne varijable nisu nosioci posebnih vrijednosti tokom računanja. Služe za iskazivanje relacija koje postoje između različitih stanja jednog računanja.

Stanje postupka računanja je preslikavanje između skupa varijabli postupka i skupa vrijednosti. Može se smatrati da svaki postupak određuje klasu proračuna koji se odvijaju u svom prostoru stanja.

Algoritam (postupak) je apstraktni mehanizam koji, pokrenut iz jednog početnog stanja, putuje kroz svoj prostor stanja dok se ne zaustavi u konačnom stanju (odnosno u stanju u kojem više nijedan takt mehanizma nije određen) koje, po pravilu, ovisi o izboru početnog stanja.

Predikatne formule u kojima se kao predikatne varijable pojavljuju varijable stanja, slobodne varijable i konstante definirane u svim stanjima računanja, čija istinitost ovisi samo o vrijednosti varijabli, nazivaju se jednim imenom predikati.

Kaže se da predikat *P* karakterizira sva stanja računanja (prostora stanja) u kojima je istinit. Dva predikata *P* i *Q* iskazuju isti uvjet, što se piše *P=Q*, ako i samo ako karakteriziraju isti skup stanja računanja.

S *T* će biti označen predikat koji je istinit u svim stanjima računanja. Predikat *T* karakterizira univerzum računanja. S *F* će biti označen predikat koji je neistinit u svim stanjima računanja. Predikat *F* karakterizira prazan skup.

Ako se algoritam $\mathcal{A}$ tokom određenog računanja nađe u stanju u kojem je predikat *P* istinit, kaže se da je $\mathcal{A}$ "utvrdio istinitost" od *P* ili da je $\mathcal{A}$ ispunio uvjet *P*.

Mehanizam računanja *S* kojem je putanja kroz prostor stanja jednoznačno određena početnim stanjem naziva se deterministički. Ako nije ispunjen taj uvjet, tj. od početnog do konačnog stanja može se doći na dva ili više načina, mehanizam je nedeterministički.

Predikat koji karakterizira ona i samo ona početna stanja mehanizma *S* iz kojih računanje uvijek okonča sa zaključkom *R* naziva se najširi preduvjet da mehanizam *S* utvrdi zaključak *R* i označava se sa *wp(S,R)*.

♣ Primjer 8.1

Neka je *S* dodjeljivanje (kao u Pascalu)

```
i := i +1
```

a zaključak

$R \equiv i \leq 1$

Tada je

$wp(\text{"i := i +1"}, i \leq 1) = (i \leq 0)$

♣ Primjer 8.2

Neka je *S* naredba

```
if x > y then z := x else z := y
```

a zaključak $R \equiv z = max(x,y)$. Izvršenjem naredbe *S* uvijek će varijabli z biti dodijeljena vrijednost $max(x,y)$, pa je $wp(S,R)=T$.

♣ Primjer 8.3

S je kao u primjeru 8.2, a *R* je z=y-1. Tada je:

$wp(S,R) = F$

tj. ne postoji početno stanje koje će dovesti da *S* okonča sa zaključkom $R \equiv z=y-1$.

♣ Primjer 8.4

S je isto kao u primjeru 8.2, a *R* je z=y+1. Tada je $wp(S,R) = (x = y+1)$.

♣ **Primjer 8.5**

Ako je S naredba, $wp(S, T)$ karakterizira skup svih onih stanja koja garantiraju da će naredba S sigurno okončati (u nekom konačnom stanju).

8.2 DEFINICIJA SEMANTIKE

Semantika mehanizma S je pravilo koje opisuje kako se za bilo koji zaključak R može izvesti odgovarajući najširi preduvjet $wp(S, R)$.

Za fiksni mehanizam S takvo pravilo koje je pridruženo predikatu R (zaključku) i zadaje $wp(S, R)$ naziva se predikatni transformator, pa se sada može reći: "semantika mehanizma S je njegov odgovarajući predikatni transformator".

Predikatni transformator u odnosu na bilo koji zaključak kojeg neki mehanizam S treba ustanoviti mora zadovoljiti slijedeća svojstva "razumnosti":

svojstvo 1.

$$wp(S, F) = F$$

To znači da ne postoji početno stanje iz kojeg mehanizam može utvrditi nemogući zaključak. Ovo svojstvo naziva se još i princip o isključenju nemogućeg.

svojstvo 2.
Za bilo koji mehanizam S i za bilo koje zaključke Q i R, takve da $Q \Rightarrow R$ za sva stanja, vrijedi:

$$wp(S, Q) \Rightarrow wp(S, R) \quad \text{za sva stanja}$$

Ovo je svojstvo monotonosti.

svojstvo 3.
Za bilo koji mehanizam S i bilo koje zaključke Q i R vrijedi:

$$wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R) \quad \text{za sva stanja}$$

Ovo je svojstvo superpozicije.

svojstvo 4.
Za bilo koji mehanizam S i zaključke Q i R vrijedi:

$$wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R) \quad \text{za sva stanja}$$

Sada smo u stanju znati moguće karakteristike mehanizma S dovoljno dobro znajući kako pridruženi predikatni transformator $wp(S,R)$ djeluje na bilo koji zaključak R.

Ako znamo da je mehanizam deterministički, poznavanje predikatnog transformatora fiksira njegovo moguće ponašanje u potpunosti. Tada svojstvo (4) prelazi u:

svojstvo 4'.
Za bilo koji deterministički mehanizam S i zaključke Q i R vrijedi:

$$wp(S, Q) \vee wp(S, R) = wp(S, Q \vee R) \quad \text{za sva stanja}$$

8.3 IZVOĐENJE NAREDBI JEZIKA DDH

Intuitivno, jezik za programiranje može se definirati kao notaciona tehnika, pismo, kojom se na kompaktan, nedvosmislen i konačan način definiraju mehanizmi računanja. Određeni mehanizam definiran tim pismom nazivamo program.

Dijsktra zadatak formalnog izvođenja jezika za programiranje postavlja na sljedeći način:

1) Svaka uvedena naredba mora zadovoljavati svojstva od 1 do 4.
2) Svako dopušteno grupiranje naredbi također mora zadovoljavati svojstva od 1 do 4.

Prazna naredba i naredba otkaza

Dva vrlo jednostavna predikatna transformatora odmah se nameću. Postoji mehanizam S takav da je za bilo koji zaključak R $wp(S, R) = R$. Poznat je kao „prazan iskaz" (npr. `CONTINUE` u `FORTRAN`-u ili `pass` u Pythonu) ili „prazna naredba" (npr. u Pascalu). Dijkstra je takvom iskazu dao ime **SKIP**. Dakle, sintaksa prazne naredbe je:

prazna_naredba : SKIP

a semantika je dana sa:

$wp(\mathbf{SKIP}, R) = R$

- **Teorem 8.1**

  Mehanizam **SKIP** zadovoljava svojstva od 1 do 4.

  Dokaz:

  svojstvo 1: $wp(\mathbf{SKIP}, \mathtt{F}) = \mathtt{F}$, što slijedi iz definicije mehanizna skip da je $wp(\mathbf{SKIP}, R) = R$ za svaki R, pa i za F.

  svojstvo 2: Iz $Q \Rightarrow R$, pišući umjesto Q $wp(\mathbf{SKIP}, Q)$ i umjesto R $wp(\mathbf{SKIP}, R)$, slijedi:

  $$wp(\mathbf{SKIP}, Q) \Rightarrow wp(\mathbf{SKIP}, R)$$

  svojstvo 3: $wp(\mathbf{SKIP}, Q) \wedge wp(\mathbf{SKIP}, R) = Q \wedge R = wp(\mathbf{SKIP}, Q \wedge R)$

  svojstvo 4: $wp(\mathbf{SKIP}, Q) \vee wp(\mathbf{SKIP}, R) = Q \vee R = wp(\mathbf{SKIP}, Q \vee R)$

Sljedeći jednostavni predikatni transformator je onaj koji vodi ka konstantnom najširem preduvjetu koji uopće ne ovisi o zaključku R. Kao konstantne predikate imamo T i F. Mehanizam S za kojeg je $wp(S, R) = \mathtt{T}$ ne može postojati, jer ne zadovoljava svojstvo 1. Mehanizam S za kojeg je $wp(S, R) = \mathtt{F}$, međutim, dolazi u obzir jer ima predikatni transformator koji zadovoljava sva neophodna svojstva. Tom ćemo mehanizmu dati ime **ABORT** i zvat ćemo ga "naredba otkaza":

naredba_otkaza : ABORT

Semantika naredbe otkaza je:

$wp(\mathbf{ABORT}, R) = \mathtt{F}$

Aktiviranje mehanizma (naredbe) **ABORT** odvodi sustav u nedefinirano stanje, odnosno, aktiviranje naredbe **ABORT** ima značenje otkaza u programu ili prekida njegova izvršavanja. Mnogi jezici imaju takvu naredbu. Na primjer, u Pythonu je to `break`, a u Pascalu `HALT`.

- **Teorem 8.2**

  Mehanizam **ABORT** zadovoljava svojstva od 1 do 4.

  Dokaz: Dokaz je trivijalan i bit će izostavljen.

Ovim su iscrpljene mogućnosti izvođenja elementarnih mehanizama.

Naredba za dodjeljivanje

Sljedeća klasa mehanizama temelji se na supstituciji. Pod supstitucijom podrazumijevamo zamjenu varijable X na svakom mjestu njezina pojavljivanja u predikatnoj formuli zaključka R određenim izrazom E. Takvu transformaciju označujemo sa $R_{E\to X}$. Za određenu varijablu X i određeni izraz E možemo razmatrati mehanizam čije je djelovanje za bilo koji zaključak R zadano semantikom:

$$wp(S, R) = R_{E\to X}$$

gdje je X koordinatna varijabla prostora stanja mehanizma S. Mehanizam S s takvim svojstvom nazivat ćemo "naredba za dodjeljivanje" ili "dodjeljivanje". Pravilo pisanja dano je:

dodjeljivanje : *varijabla = izraz*

Pisat ćemo

$$X = E$$

gdje se "=" čita kao "stječe vrijednost". Semantika naredbe za dodjeljivanje je:

$$wp("X=E", R) = R_{E\to X}$$

uz uvjet da je tip varijable jednak tipu izraza. To je uobičajeno značenje naredbe za dodjeljivanje u svim jezicima za programiranje: izračuna se izraz E i dobivena vrijednost se pridruži varijabli X.

- **Teorem 8.3**

  Predikatni transformator mehanizma zadovoljava svojstva od 1 do 4.

  Dokaz:

  svojstvo 1: $wp("X=E", F) = F_{E\to X} = F$

  svojstvo 2: Ako $Q \Rightarrow R$, onda i $Q_{E\to X} \Rightarrow R_{E\to X}$, a odavde slijedi (prema definiciji $wp("X=E", R)$):

  $$wp("X=E", Q) \Rightarrow wp("X=E", R)$$

svojstvo 3: $wp("X=E", Q) \wedge wp("X=E", R) = Q_{E\to X} \wedge R_{E\to X} = (Q \wedge R)_{E\to X} = wp("X=E", Q \wedge R)$

svojstvo 4: $wp("X=E", Q) \vee wp("X=E", R) = Q_{E\to X} \vee R_{E\to X} = (Q \vee R)_{E\to X} = wp("X=E", Q \vee R)$

iz čega zaključujemo da je mehanizam deterministički.

KONKURENTNO DODJELJIVANJE

Dodjeljivanje koje smo opisali je "primitivno dodjeljivanje". Osim njega u jeziku **DDH** postoji i "konkuretno dodjeljivanje" definirano sa:

```
konkurentno_dodjeljivanje :
  varijabla, konkurentno_dodjeljivanje, izraz | dodjeljivanje
```

Ovom definicijom proširena je sintaksa naredbe za dodjeljivanje, pa se primitivno dodjeljivanje može shvatiti kao poseban slučaj konkurentnog dodjeljivanja. Na primjer, ako su *X*1 i *X*2 varijable, a *E*1 i *E*2 izrazi, može se napisati

```
X1, X2 = E1, E2
```

To je semantički ekvivalentno sa

```
X2, X1 = E2, E1
```

pa zaključujemo da konkurentno dodjeljivanje *X*1,*X*2=*E*1,*E*2 nije ekvivalentno nizu dodjeljivanja:

```
X1 = E1; X2 = E2
```

Osim zadovoljenja osnovne sintaksne strukture, za konkurentno dodjeljivanje moraju biti ispunjeni i sljedeći kontekstni uvjeti:

1) Varijable na lijevoj strani konkurentnog dodjeljivanja moraju biti različite.
2) Prvoj varijabli odgovara prvi izraz, drugoj drugi itd. Zbog toga tip izraza mora odgovarati tipu varijable.

Pri izvršavanju naredbe za konkurentno dodjeljivanje prvo se izračunaju izrazi. Potom se dobivene vrijednosti pridruže odgovarajućim varijablama. Na primjer, razmjena vrijednosti ("swap") dviju varijabli *x* i *y* istoga tipa može se realizirati sa `x, y = y, x`.

Niz naredbi

Ako svaka od navedenih naredbi odražava jedan formalni mehanizam, postavlja se pitanje: Kako organizirati kolekciju naredbi koje će odražavati složeni mehanizam? Odgovor na to pitanje je u proširenju pojma "mehanizam" na sljedeći način:

1) Svaki primitivni mehanizam je mehanizam.
2) Ako su *S*1 i *S*2 mehanizmi, onda je njihova kompozicija *S*1;*S*2 također mehanizam.

Jedan od najpoznatijih načina izvođenja nove funkcije iz dvije dane je kompozicija funkcija, tj. osiguranje da vrijednost jedne funkcije bude argument druge. Komponirani mehanizam koji odgovara predikatnom transformatoru označuje se sa "$S1;S2$" i definira sa:

$$wp("S1;S2", R) = wp(S1, wp(S2, R))$$

a ovo je definicija semantike operatora "**;**" (nastavljanja ili "konkatenacije").

- **Teorem 8.4**

  Ako su $S1$ i $S2$ mehanizmi i ako su $wp(S1, R)$ i $wp(S2, R)$ odgovarajući najširi preduvjeti za bilo koji zaključak R, pri čemu predikatni transformatori za $S1$ i $S2$ zadovoljavaju svojstva 1 do 4, onda i $wp("S1;S2", R)$ zadovoljava svojstva 1 do 4.

  Dokaz:

  svojstvo 1: Po definiciji je: $wp(S1, F) = F$ i $wp(S2, F) = F$, a po definiciji $wp("S1;S2", R)$, vrijedi:

  $$wp("S1;S2", F) = wp(S1, wp(S2, F)) = wp(S1, F) = F$$

  svojstvo 2: Ako $Q \Rightarrow R$, po definiciji slijedi:

  $$wp(S1, Q) \Rightarrow wp(S1, R) \text{ i } wp(S2, Q) \Rightarrow wp(S2, R)$$

  a odavde slijedi:

  $$\begin{aligned} wp("S1;S2", Q) &= wp(S1, wp(S2, Q)) \\ &\Rightarrow wp(S1, wp(S2, R)) \\ &= wp("S1;S2", R) \end{aligned}$$

  svojstvo 3: Po definiciji je:

  $$wp(S1, Q) \wedge wp(S1, R) = wp(S1, Q \wedge R) \text{ i}$$
  $$wp(S2, Q) \wedge wp(S2, R) = wp(S2, Q \wedge R)$$

  pa je:

  $$\begin{aligned} &wp("S1;S2", Q) \wedge wp("S1;S2", R) \\ &\quad = wp(S1, wp(S2, Q)) \wedge wp(S1, wp(S2, R)) \\ &\quad = wp(S1, wp(S2, Q) \wedge wp(S2, R)) \\ &\quad = wp(S1, wp(S2, Q \wedge R)) \\ &\quad = wp(S1, wp(S2, Q \wedge R)) \end{aligned}$$

  svojstvo 4: Po definiciji je:

  $$wp(S1, Q) \vee wp(S1, R) = wp(S1, Q \vee R) \text{ i}$$
  $$wp(S2, Q) \vee wp(S2, R) = wp(S2, Q \vee R)$$

  pa je:

  $$\begin{aligned} &wp("S1;S2", Q) \vee wp("S1;S2", R) \\ &\quad = wp(S1, wp(S2, Q)) \vee wp(S1, wp(S2, R)) \\ &\quad = wp(S1, wp(S2, Q) \vee wp(S2, R)) \\ &\quad = wp(S1, wp(S2, Q \vee R)) \\ &\quad = wp(S1, wp(S2, Q \vee R)) \end{aligned}$$

Dakle, ako složenom mehanizmu "*S1;S2*" zadamo određeni zaključak *R*, time smo istodobno isti zaključak zadali i mehanizmu *S2*. Po definiciji je:

wp("*S1;S2*", *R*) = *wp*(*S1*, *wp*(*S2*, *R*))

Ovaj je uvjet očigledno ispunjen ako se ukupna aktivnost mehanizma "*S1;S2*" okonča okončanjem mehanizma *S2*. Pri tome odgovarajući najširi preduvjet za okončanje, *wp*(*S2*, *R*), osigurava zaključak mehanizmu *S1*, prema gornjoj definiciji, odnosno, definicija identificira konačno stanje mehanizma *S1* s početnim stanjem mehanizma *S2*. To je upravo slučaj kada okončanje od *S1* prethodi aktiviranju mehanizma *S2* (u istom prostoru stanja).

Uvođenje operatora "**;**" omogućuje nam da konstruiramo proizvoljno velike (ali konačne) mehanizme u kojima će jedina okolnost koja izaziva aktiviranje primitivnog mehanizma biti okončanje leksikografski prethodnog mehanizma.

Potrebno je uvesti pravila za konstruiranje mehanizama kod kojih aktiviranje osnovnih mehanizama ovisi o tekućem stanju mehanizma. U tu se svrhu uvodi pojam "prolaza" sintaksom:

prolaz : [*predikatna_formula*] *naredba* { ; *naredba* }

gdje su **[** i **]** (uglate zagrade) dio sintakse (a ne meta-simboli koje prikazujemo s **[** i **]**).

Ideja se prolaza temelji na tome da se grupa naredbi napisanih iza predikatne formule izvršava samo ako je propoziciona formula istinita. U smislu ove ideje, predikatna formula koja prethodi naredbama naziva se zaklon ("guard"). On omogućuje da se u određenim okolnostima (koje su funkcija stanja) omogući izvršavanje naredbi koje stoje iza njega. Istinitost zaklona (predikatne formule) neophodan je preduvjet za izvršavanje naredbi iza njega, ali nije dovoljna, kao što ćemo malo kasnije vidjeti.

Naredbe za selekciju i iteraciju

Potreba za uvođenjem "prolaza" motivirana je sljedećim razmišljanjem: Pretpostavlja se da je potrebno konstruirati mehanizam koji će, ako se upusti u računanje iz početnog stanja okarakteriziranog s *Q*, okončati sa zaključkom *R*. Dalje, pretpostavka je da se ne može naći jedinstvena lista naredbi koje bi obavile takav zadatak, ali se može naći kolekcija od nekoliko lista naredbi od kojih svaka ispunjava postavljeni zadatak. Ako su zakloni dovoljno tolerantni, tj. ako istinitost od *Q* implicira istinitost barem jednog zaklona, onda raspolažemo mehanizmom koji će iz svakog početnog stanja karakteriziranog s *Q* dovesti sustav u konačno stanje koje je karakterizirano s *R*. To su one liste naredbi (barem jedna) čiji su zakloni istiniti u početnom stanju. Najprije definirajmo:

zaklonjeni_skup : *prolaz* { ! *prolaz* }

gdje ćemo simbol "!" čitati "pa", a predstavljat će separator inače neuređenih prolaza. Dodajući interpunkcijske simbole, proširujemo skup našeg jezika sa:

naredba : IF *zaklonjeni_skup* FI |DO *zaklonjeni_skup* OD

gdje je:

selekcija : IF *zaklonjeni_skup* FI
iteracija : DO *zaklonjeni_skup* OD

U nastavku dajemo semantiku ovih dviju naredbi.

SELEKCIJA

Ako selekciju općenito prikažemo kao

IF $[z_1]$ LN_1
  ![z_2] LN_2
  ...
  ![z_n] LN_n
FI

gdje su z_i zakloni, a LN_i odgovarajuće pridružene liste naredbi. Semantika mehanizma za selekciju, nazovimo ga ***if***, dana je sa:

$$wp(\mathbf{if},R)=(\exists j: 1 \le j \le n: z_j) \wedge (\forall j: 1 \le j \le n: z_j \Rightarrow wp(LN_j,R))$$

za svaki zaključak R. Ili, riječima: "Predikat $wp(\mathbf{if},R)$ istinit je u svakoj točki prostora stanja za koju egzistira bar jedan j u intervalu $[1,n]$ takav da je z_j istinito i ako je za svaki j u istom intervalu za koji je z_j istinit, istinit i predikat $wp(LN_j,R)$". Ista definicija može se napisati i u drugom obliku:

$$\begin{aligned} wp(\mathbf{if},R) = {} & (z_1 \vee \ldots \vee z_n) \\ & \wedge (z_1 \Rightarrow wp(LN_1,R)) \\ & \quad \ldots \\ & \wedge (z_n \Rightarrow wp(LN_n,R)) \end{aligned}$$

U slučaju da nijedan zaklon (logički izraz) nije istinit, selekcija je semantički ekvivalentna naredbi **ABORT**, jer je tada $wp(\mathbf{if},R)$=F. Dalje, zahtijeva se da svako stanje karakterizirano sa $wp(\mathbf{if},R)$ bude karakterizirano i sa $z_n \Rightarrow wp(LN_n,R)$ za svaki j.

Za one vrijednosti j za koje je z_j=F implikacija je istinita po definiciji (bez obzira je li $wp(LN_j,R)$ jednako T ili F). Drugim riječima, za te vrijednosti j nevažno je što bi aktiviranje LN_j moglo ustanoviti. To se, u realizaciji, odražava nebiranjem LN_j za aktiviranje.

Za one j za koje je z_j=T u početnom stanju implikacija je istinita samo ako je $wp(LN_j,R)$=T. S obzirom na to da formalna definicija zahtijeva istinitost implikacije za svako j, u općem se slučaju, kad su dva ili više zaklona istinita, može nasumice izabrati lista naredbi za koju je z_j=T. Iz toga se zaključuje da je mehanizam ***if*** nedeterministički.

ITERACIJA

Semantika naredbe za iteriranje nešto je složenija od semantike ***if*** mehanizma. U slučaju da nijedan zaklon nije istinit, naredba za iteriranje semantički je ekvivalentna naredbi

SKIP. Nasuprot tome, naredba za iteriranje ne može okončati sve dok je barem jedan zaklon istinit. Neka je ***do*** naziv naredbe za iterириranje, koju se može općenito napisati kao

DO $[z_1]$ LN_1
$![z_2]$ LN_2
...
$![z_n]$ LN_n
OD

Neka je ***if*** naziv selekcije s istim zaklonjenim skupom i neka su predikatne formule $H_k(R)$ dane rekurzivnom definicijom:

za $k=0$ $H_0(R) = R \wedge \neg (j: 1 \le j \le n: z_j)$
za $k>0$ $H_k(R) = wp(\textbf{\textit{if}}, H_{k-1}(R)) \vee H_0(R)$

tada je semantika mehanizma ***do*** jednaka:

$wp(\textbf{\textit{do}}, R) = (\exists k: k \ge 0: H_k(R))$

Predikatna formula $H_k(R)$ najširi je preduvjet da će mehanizam ***do*** okončati poslije najviše k selekcija prolaza mehanizma ***do***, ostavljajući sustav u konačnom stanju koje zadovoljava zaključak R.

Za $k=0$ zahtijeva se da mehanizam ***do*** okonča odmah, tj. bez selekcije bilo kojeg prolaza. To znači da nijedan zaklon ne smije biti istinit, što odražava drugi član konjunkcije koja definira H_0. No, tada je jasno da zaključak R mora biti istinit i u početnom stanju, što odražava prvi član konjunkcije koja definira H_0. Za $k>0$ moguća su dva slučaja:

1) Nije istinit nijedan zaklon. Tada R mora biti ispunjeno početnim stanjem, što izražava drugi član disjunkcije koja definira H_k.

2) Istinit je barem jedan zaklon. Ono što se tada događa je kao da se aktivira mehanizam ***if***, koji sigurno neće "abortirati" po pretpostavci ovog slučaja. Poslije izbora jednog prolaza i izvršenja te naredbe, moramo biti sigurni da je mehanizam stigao u stanje iz kojeg je potrebno najviše $k-1$ selekcija da se osigura okončanje sa zaključkom R. Definicija to iskazuje postavljajući $H_{k-1}(R)$ za najširi preduvjet mehanizmu ***if***.

Na kraju definicija $wp(\textbf{\textit{do}}, R)$ iskazuje da mora egzistirati takva vrijednost k da najviše k selekcija prolaza iz ***do*** osigurava okončanje sa zaključkom R.

8.4 VARIJABLE

Prva inačica jezika FORTRAN, na primjer, imala je samo dva tipa varijabli: cjelobrojne i realne. Prvom pojavom varijable u tekstu programa, po fiksnom dogovoru, bilo je uvedeno i novo ime. Tip je bio određen početnim slovom imena te varijable (početno slovo `I`, `J`, `K`, `L`, `M` ili `N` označivalo je cjelobrojni tip, ostala slova – realni). To je u praksi dovodilo do velike nesigurnosti. Ako, na primjer, program koristi varijablu `SIGN`, pojava krivog imena, kao što je na primjer `SING`, neće upozoriti korisnika, jer je time bila legalno uvedena nova varijabla (realnog tipa) s imenom `SING`.

Ideja o potrebi deklariranja varijabli prvi put je bila uvedena u jeziku ALGOL 60, 1963. godine. To je značilo da se u naredbama programa smije referirati samo na unaprijed deklarirane varijable. Primjenjujući to na naš primjer, nailaskom na SING detektirala bi se pogreška. Deklaracije u ALGOL-u bile su korisne ne samo zbog uvođenja "rječnika" s imenima varijabli koja su bila dopuštena za uporabu u naredbama programa, već i zbog toga što su svakom uvedenom imenu pridružili tip. Time je bio ukinut originalni dogovor iz FORTRAN-a da je tip varijable određen njezinim početnim slovom.

Veće odvajanje od FORTRAN-a bilo je i zbog uvođenja koncepta blokova. Blok se rabio za ograničavanje tekstualnog dosega deklaracija. Granice dosega bile su između otvorene `begin` i zatvorene `end` "zagrade". Poslije `begin` deklarira se jedno ili više imena koja su lokalna za taj blok. Blok je, međutim, jedna od mogućih naredbi i zbog toga se može pojaviti jedan unutar drugog. Pravila dosega u ALGOL-u 60 štite lokalne varijable unutarnjeg bloka od vanjskih uplitanja, ali ne i suprotno.

Poslije ALGOL-a, u posljednjih pedeset godina, dizajnirani su mnogi jezici za programiranje. Jedni su, kao na primjer Pascal i C, zahtijevali obvezno deklariranje varijabli. U Pascalu je doseg značenja imenâ (varijabli, konstanti, tipova itd.) bio definiran unutar bloka koji je predstavljao tijelo procedure. U nekim je drugim jezicima, kao na primjer u GW BASIC-u, imenu varijable dodavan sufiks $ ili %, koji je određivao tip varijable itd.

Poslije tridesetak godina strogog pristupa problemu deklariranja varijabli pojavili su se ponovo jezici koji ne samo da nisu tražili deklariranje varijabli, već su dopuštali i promjenu njihova tipa (ili strukture!) unutar istog bloka programa. Takav je, na primjer, jezik Python.

A što je s varijablama u jeziku **DDH**? S obzirom na to da je Dijkstra u vrijeme dok je definirao svoj jezik imao uzore u samo nekoliko tada poznatih jezika za programiranje, poslije pomne analize zaključio je da deklariranje varijabli u njima ne zadovoljava zahtjevima strogog pristupa korektnosti programa. Kao rezultat svega toga bilo je njegovo originalno rješenje problema varijabli programa, njihovog dosega, inicijalizacije i prava pristupa.

Uz nastojanje da se izbjegnu slabosti tekstualnog dosega ALGOL-a 60 i tada njemu sličnih jezika, Dijkstra se odlučio učiniti prvi korak u pravcu tekstualnog ugniježđenja konteksta pa je uveo blok u kojem imena imaju jedinstveno značenje. Za razliku od ALGOL-a, imena prvi put uvedena u nekom bloku nedostupna su izvan njega. U izrazima nije dopuštena uporaba neinicijaliziranih varijabli. Uvode se i dva osnovna tipa: cjelobrojni (`int`) i logički (`bool`), te struktura polja. Struktura polja parametrizirana je po svom osnovnom tipu i opisana je atributima (svojstvima) i funkcijama (metodama), pa se može slobodno zaključiti da je jezik **DDH** i mini objektni jezik.

Inicijalizacija i tekstualni doseg varijabli

Varijable imaju svoje ime pa najprije definirajmo pravilo tvorbe imenâ:

ime : *slovo* { [*slovo* | *brojka*] }

gdje se ***slovo*** i ***brojka*** u proširenoj notaciji regularnih izraza mogu prikazati kao

```
slovo  : [A-Za-z]
brojka : [0-9]
```

Ovom pravilu treba dodati dva kontekstna uvjeta:

1) Ime ne smije biti iz skupa rezerviranih riječi. Rezervirane riječi su imena naredbi, na primjer, **SKIP** i **ABORT**, ili riječi koje se pojavljuju kao dio sintakse naredbi, na primjer **IF**, **FI**, **DO**, **OD** itd. Uvijek se pišu samo velikim slovima.
2) Velika i mala slova sadržana u imenu imaju jedinstveno značenje. Na primjer, `A` i `a` su dva različita imena. Otuda slijedi da su, na primjer, `skip`, `Skip`, `do`, `Do` ili `oD` legalno izabrana imena.

Za svaki blok postavljen je, konstantnom nomenklaturom, njegov tekstualni kontekst u kojem imena imaju jedinstveno značenje. Imena koja se pojavljuju u tekstualnom kontekstu bloka ili su lokalna ("privatna"), sa značenjem koje vrijedi samo u tekstu tog bloka, ili su globalna, tj. naslijeđena sa svojim značenjem iz neposredne okoline. To znači da se u tekstu nekog bloka ne smije pojaviti ime koje nije deklarirano u istom bloku.

Pored imena, varijable imaju jedinstveno svojstvo promjene svoje vrijednosti. Ovo vodi do pitanja: Kolika je vrijednost lokalne varijable na ulazu u blok? Za razliku od mnogih jezika, Dijkstra ne dopušta nikakve unaprijed dogovorene vrijednosti, kao, na primjer, nulu za cjelobrojne ili `True` za logičke varijable. Praksa je pokazala da se takvim načinom inicijalizacije mogu načiniti fatalne pogreške u programima, koje se teško uočavaju.

Uzimajući u obzir i slijedeći Dijkstrin koncept, zahtijevat ćemo da ni na jednom mjestu programa napisanog u našem jeziku pri uporabi varijable ne smije postojati neizvjesnost je li bila ranije inicijalizirana. Jedan od načina da se to postigne jest da se načini obvezna inicijalizacija svih lokalnih varijabli na ulazu bloka, zajedno sa željom da se ne inicijaliziraju s besmislenim vrijednostima.

Tekstualni doseg lokalne varijable u bloku protezat će se od otvaranja bloka s **BEGIN** sve do zatvaranja s **END**. Podijelit ćemo ga na "pasivni doseg", u kojem nije dopušteno referiranje na tu varijablu, i "aktivni doseg", u kojem varijabla može biti rabljena. Za svaku varijablu vrijede sljedeća pravila:

1) Od ulaza u blok pa do izlaza iz njega može biti izvedena samo jedna naredba za inicijalizaciju.
2) Od početka bloka do naredbe za inicijalizaciju ne smije se izvršiti nijedna naredba iz aktivnog dosega.

Blok, prije svega, možemo promatrati kao sintaksnu strukturu u kojoj je nabrajanje njegove lokalne nomenklature slijeđeno listom naredbi odvojenih s ";". Takva lista naredbi mora imati sljedeća svojstva:

a) Mora sadržavati jedinstvenu naredbu za inicijalizaciju dane lokalne varijable.
b) Naredbe (ako ih ima) koje prethode naredbi za inicijalizaciju u pasivnom su dosegu varijable, a ako su to unutarnji blokovi, ne smiju je naslijediti.

c) Naredbe (ako ih ima) koje slijede naredbu za inicijalizaciju u aktivnom su dosegu varijable, a ako su to unutarnji blokovi, smiju je naslijediti.
d) Za samu naredbu za inicijalizaciju postoje tri mogućnosti:

   1) Naredba za inicijalizaciju je primitivna.
   2) Ona je unutarnji blok. U tom slučaju nasljeđuje dotičnu varijablu, a lista naredbi koja slijedi nabrajanje unutarnje nomenklature ponovo ima svojstva od (a) do (b).
   3) Ona je selekcija. U tom slučaju sve liste naredbi koje slijede uvjete pojedinih prolaza selekcije moraju zadovoljavati svojstva od (a) do (d).

Poslije svega ovoga možemo proširiti sintaksu našeg jezika sa:

```
naredba          : blok | inicijalizacija
blok             : BEGIN nomenklatura naredba { ; naredba } END
nomenklatura     : element_nomenklature ; { element_nomenklature ; }
element_nomenklature :
                    tip_nomenklature varijabla { , varijabla }
```

Sljedeća stvar o kojoj treba odlučiti je: Treba li pri ulazu u blok biti bilo kakav nagovještaj o prirodi posredovanja s naslijeđenim varijablama? Ako, na primjer, takva varijabla pri svakom aktiviranju unutarnjeg bloka djeluje kao globalna konstanta, željeli bismo onemogućiti bloku da mijenja njezinu vrijednost. Preciznije: on je može samo ispitivati.

Suzimo, na trenutak, našu pozornost na unutarnji blok, s kontekstom ***in*** ("unutar"), koji je kompletan u aktivnom dosegu varijable koju nasljeđuje od neposrednog okolnog bloka, s kontekstom ***out*** ("izvan"). Ako je varijabla na razini konteksta ***out*** promjenljiva, u kontekstu ***in*** može biti promjenljiva ili konstantna.

Činjenica da je neka globalna konstanta upravo konstantna od ključnog je značenja kada nas interesira što može utjecati na promjenu njezine vrijednosti u nekom bloku. Tada slobodno možemo preskočiti tekst tog bloka. Ako je na razini konteksta ***out*** varijabla već konstanta, unutarnji blok može je naslijediti samo kao takvu. Unutarnji blok koji je kompletan u aktivnom dosegu naslijeđene varijable bit će uspješno realiziran, a ako je u pasivnom dosegu, ne predstavlja nikakvu poteškoću, jer ne može naslijediti tu varijablu iz svog okoliša.

Treći slučaj, kada blok počinje u pasivnom, a završava u aktivnom dosegu varijable, zaslužuje malo više pozornosti. To nameće obvezu unutarnjem bloku da je mora inicijalizirati. Blok je naslijedio ono što ćemo nazvati "virgin" (netaknuta) varijabla. U oba konteksta, ***out*** i ***in***, možemo se pitati je li varijabla promjenljiva u svom aktivnom dosegu. U slučaju da se nasljeđuje ***virgin*** varijabla, ova dva pitanja potpuno su neovisna. Okolnost da varijabla na razini tekstualnog konteksta ***out*** neće promijeniti svoju vrijednost ne isključuje mogućnost da je samo građenje njezine inicijalne vrijednosti višestupno.

Imena varijabli u nekom bloku mogu biti lokalna, tj. ne odnose se na vanjski kontekst, ili su naslijeđena od okolnog konteksta s obvezom inicijalizacije ili bez nje. Za svaku varijablu razlikujemo tri slučaja:

PRI "privatna" (lokalna), tj. ne odnosi se na vanjski kontekst,
VIR "virgin" (netaknuta), tj. nasljeđuje se iz okolnog konteksta s obvezom inicijalizacije u bloku ***in*** i
GLO "globalna", tj. stečena je od okolnog konteksta bez obveze i bez dopuštenja da se inicijalizira.

Bez obzira na to je li varijabla navedena kao **PRI** ili **VIR**, mora biti inicijalizirana unutar bloka u kojem su deklarirana. Jedina je razlika u tome što varijabla **PRI**, navedena na početku nekoga bloka, nema veze s kontekstom ***out*** (nije naslijeđena od okolnog konteksta), a varijabla **VIR** mora se pojaviti u kontekstu ***out*** i unutarnji blok mora započeti u njezinom pasivnom, a završiti u aktivnom dosegu, tj. unutarnji blok obvezan je inicijalizirati takve varijable u svom kontekstu.

Varijabla **GLO** navedena u nomenklaturi unutarnjeg bloka mora se pojaviti u kontekstu ***out***, a unutarnji blok može biti samo u njezinom aktivnom dosegu.

Dijkstra je vanjskim aspektima bloka dodao i unutarnje: svakoj varijabli unutar bloka bit će dopušteno ili ne da mijenja svoju vrijednost. Ako se vrijednost varijable smije mijenjati, označivat će se s **VAR**, a ako se ne smije mijenjati, s **CON**. Kombinirajući vanjske i unutarnje aspekte varijabli, imamo sljedeća pravila prijenosa varijabli iz ***out*** u ***in*** kontekst:

in	***out***
PRIVAR, **PRICON**	ne smije se pojaviti,
GLOVAR	**PRIVAR**, **VIRVAR**, samo ako je unutarnji blok u aktivnom dosegu,
GLOCON	**PRIVAR**, **PRICON**, **VIRVAR**, **VIRCON**, samo ako je unutarnji blok u aktivnom dosegu, **GLOVAR**, **GLOCON** bez ograničenja,
VIRVAR, **VIRCON**	**PRIVAR**, **PRICON**, **VIRVAR**, **VIRCON**, samo ako je unutarnji blok započeo u pasivnom dosegu.

Kao posljedica prednjeg, aspekt **CON** u kontekstu ***out*** nakon inicijalizacije isključuje mogućnost da unutarnji blokovi mijenjaju vrijednost varijable. Ovaj aspekt ne isključuje inicijalizaciju u unutarnjem bloku na čijoj razini varijabla može zadovoljavati aspekt **VAR**. Na primjer, u kontekstu ***out*** može biti **PRICON** x, čija se inicijalizacija prenosi unutarnjem bloku u čijem će kontekstu **VIRVAR** x biti izgrađena. Kada je inicijalizacija jednom završena, vrijednost varijable x ostat će konstantna tijekom izvršavanja naredbi vanjskih blokova.

Posljedice danih pravila su:

1) Sve varijable u prvom (osnovnom) bloku moraju biti deklarirane kao **PRIVAR** ili **PRICON**, jer prvi blok nema vanjskog konteksta.
2) Varijabla pri svom prvom pojavljivanju mora biti deklarirana sa **PRIVAR** i **PRICON**.
3) Deklaracija varijable pridružuje joj samo svojstvo "**VAR**" ili "**CON**".

4) Varijabla mora biti deklarirana s imenom nomenklature **VIRVAR** ili **VIRCON** samo ako prije početka novog bloka nije bila inicijalizirana (nalazi se u svom pasivnom dosegu). Prefiks "**VIR**" je od engleske riječi "virgin", što znači "netaknuta". Deklariranjem varijable kao **VIRVAR** ili **VIRCON** istovremeno je prenesena obveza bloku u kojem je to napisano da do izlaska iz bloka navedene varijable moraju biti inicijalizirane.

5) Aspekt "**CON**" u kontekstu "vanjski" isključuje mogućnost da, nakon inicijalizacije, unutarnji blokovi mijenjaju vrijednost varijable. To ne isključuje inicijalizaciju u unutarnjem bloku na čijoj razini varijabla može zadovoljiti aspekt "**VAR**". Na primjer, u kontekstu "vanjski" može biti **PRICON** X, čija se inicijalizacija prenosi unutarnjem bloku u kojem će varijabla biti izgrađena, pa smije stajati **VIRVAR**. Ali, kad je inicijalizacija jednom završena (izlaskom iz unutarnjeg bloka), vrijednost od X ostat će konstantna pri izvršavanju naredbi vanjskih blokova.

Na temelju svega naprijed rečenog, možemo definirati sintaksu tipa nomenklature:

tip_nomenklature : PRIVAR | PRICON | VIRVAR | VIRCON | GLOVAR | GLOCON

Dvije globalne konstante definirane su unaprijed i dostupne su u svim blokovima. To su logičke konstante `False` i `True`.

Evo dva primjera pravilnog prenošenja varijabli iz bloka u blok. Mjesta inicijalizacije naznačili smo komentarima napisanim između navodnika.

♣ Primjer 8.6

Analizirajmo dosege varijabli u sljedećem programu:

```
BEGIN PRICON A, B, C;
  "Inicijalizacija varijable A";

  BEGIN VIRVAR B, C; GLOCON A;
    "Inicijalizacija varijable B";

    BEGIN VIRVAR C; GLOCON A, B;
       "Inicijalizacija varijable C";
       C = C+A+B
       END;

     B = A*A; C = B/2
     END;
  ...
  END
```

U prvom su bloku, na razini *r*, deklarirane tri privatne kontante s imenima A, B i C. U njemu je inicijalizirana varijabla C i prenosi se u novi blok, na razini *r*+1, kao globalna konstanta. U taj se blok prenose i varijable B i C, ali s obvezom da moraju biti inicijalizirane. S obzirom da njihov tip nomenklature ima sufiks **VAR**, dopušteno je inicijalizirati ih u više koraka. Prvo je inicijalizirana varijabla B. Slijedi novi blok, na razini *r*+2. U njega se prenosi varijabla C uz obvezu inicijalizacije. Prenose se i varijable A i B kao globalne konstante. Inicijalizira se varijabla C i mijenja joj se prvobitna vrijednost. Izlazi se iz bloka na razini *r*+2 i vraća u blok na razini *r*+1. Tu se izgrađuju konačne vrijednosti varijabli B i C. Izlazi se u osnovni blok.

♣ **Primjer 8.7**

```
BEGIN PRICON A, B;
  "Inicijalizacija varijable A";
  BEGIN VIRVAR B; GLOCON A; PRIVAR X;
    "Inicijalizacija varijable B";
    "Inicijalizacija varijable X";
    END;
  BEGIN PRICON X; GLOCON B;
    "Inicijalizacija varijable X"
     END;
  ...
  END
```

U prvom su bloku, na razini *r*, deklarirane dvije privatne konstante s imenima A i B. U njemu je inicijalizirana varijabla A i prenosi se u novi blok, na razini *r*+1, kao globalna konstanta. U taj se blok prenosi i varijabla B, ali s obvezom da mora biti inicijalizirana, te deklarira privatna varijabla X. U sljedećem bloku, također na razini *r*+1, deklarira se privatna varijabla s imenom X (što je dopušteno jer je novi blok izvan dosega značenja imena X iz prethodnog bloka) i prenosi globalna konstanta B iz vanjskog bloka.

Tipovi varijabli

Jezik **DDH** ima dva osnovna (bazična) tipa podataka:

- cjelobrojni i
- logički

Ako s ***tip*** označimo tipove podataka, može se definirati:

```
tip : INT | BOOL
```

gdje je **INT** identifikator cjelobrojnog, a **BOOL** logičkog tipa. Imena su tipova rezervirane riječi.

CJELOBROJNI TIP

Cjelobrojni tip je podskup cijelih brojeva (ovisno o ciljnom jeziku). Pravila pisanja cijelih brojeva, sintaksna kategorija ***broj***, dana je sa:

```
broj     : predznak brojka { brojka }
predznak : [ + | - ]
```

Izgrađivanje cjelobrojnih vrijednosti postiže se izračunavanjem brojčanog (cjelobrojnog) izraza. Osnovno pravilo pisanja cjelobrojnih izraza dano je sljedećom sintaksom:

```
cjelobrojni_izraz    :  operand { operacija operand }
operacija            :  + | - | * | / | MOD
operand              :  broj | cjelobrojna_varijabla |
                        cjelobrojna_konstanta |
                        - cjelobrojni_izraz |  ( cjelobrojni_izraz )
cjelobrojna_varijabla : ime
cjelobrojna_konstanta : ime
```

Značenje operacija je sljedeće:

+ zbrajanje
- oduzimanje
***** množenje
/ cjelobrojno dijeljenje
MOD ostatak cjelobrojnog dijeljenja

Izračunavanje cjelobrojnog izraza izvodi se slijeva nadesno, izvršavajući operacije prema sljedećem prioritetu:

(1) podizraz u zagradi
(2) predznak (unarna operacija "−" ili "+")
(3) množenje, dijeljenje ili ostatak cjelobrojnog dijeljenja (**MOD**)
(4) zbrajanje ili oduzimanje

LOGIČKI TIP

Logički tip definiran je nad skupom {**FALSE**, **TRUE**}, gdje konstanta **FALSE** predstavlja „laž“, a **TRUE** „istinu“. Izgrađivanje logičkih vrijednosti postiže se izračunavanjem logičkog izraza koji se piše prema sljedećim pravilima:

```
logički_izraz :  log_operand { log_operacija log_operand }
log_operacija :  AND | OR
log_operand   :  FALSE | TRUE | log_varijabla | log_konstanta |
                 relacijski_izraz | NOT logički_izraz
                 ( logički_izraz )
log_varijabla :  ime
log_konstanta :  ime
logički_izraz :  cjelobrojni_izraz relacija cjelobrojni_izraz
relacija      :  == | > | < | >= | <= | <>
```

Značenje logičkih operacija je:

NOT negacija,
AND konjunkcija i
OR disjunkcija.

Ove su operacije definirane kao u matematici, [Dov2012a], pa ih nećemo posebno objašnjavati. Značenje relacija je:

== jednako,
> veće,
< manje,
<= manje ili jednako,
>= veće ili jednako i
<> različito.

Izračunavanje logičkog izraza izvodi se slijeva nadesno, izvršavajući operacije prema sljedećem prioritetu:

(1) podizraz u zagradi
(2) negacija (**NOT**)
(3) relacije (relacijski podizraz)
(4) konjunkcija (**AND**)
(5) disjunkcija (**OR**)

Inicijalizacija

Nomenklaturom se deklariraju varijable programa pridružujući im atribut "**VAR**" ili "**CON**", bez navođenja tipa i strukture varijable. Tek će naredbom za inicijalizaciju biti definiran tip i struktura varijable. Inicijalizacija vrijednosti varijabli programa postiže se isključivo naredbom za inicijalizaciju, prema pravilima:

inicijalizacija **:** ***primitivna_inicijalizacija*** | ***inicijalizacija_polja***

Pravila pisanja naredbe za primitivnu inicijalizaciju je:

primitivna_inicijalizacija **:** ***ime*** **VIR** ***tip*** **=** ***izraz***

Primitivnom se inicijalizacijom zadanom imenu (primitivne) varijable pridružuje tip i pridružuje inicijalna vrijednost dobivena izračunavanjem izraza. Tip izraza mora odgovarati tipu varijable. Dakako, sve varijable koje se pojavljuju u izrazu moraju biti prethodno inicijalizirane.

♣ Primjer 8.8

Naredbama:

```
X0 VIR int  = 666;  X VIR int = X0; Y0 VIR int  = 404;  Y VIR int = Y0;
Ok VIR bool = False; D  VIR bool = X > 0 and Y > 0;
```

cjelobrojnoj je varijabli `X0` pridružena inicijalna vrijednost `666`, cjelobrojnoj varijabli `X` vrijednost varijable `X0`, cjelobrojnoj varijabli `Y0` pridružena je inicijalna vrijednost `404`, cjelobrojnoj varijabli `Y` vrijednost varijable `Y0`, logičkoj varijabli `Ok` vrijednost `False` i logičkoj varijabli `D` pridružena je vrijednost `True`, jer je rezultat izračunavanja danog logičkog izraza jednak `True`. Ne može se, na primjer, napisati:

```
B VIR int = B+1;
```

jer je `B` na desnoj strani (u izrazu) još uvijek u pasivnom dosegu, a prijeći će u aktivni tek po završetku ispravne inicijalizacije.

♣ Primjer 8.9

Euklidov algoritam za određivanje najveće zajedničke mjere (najvećeg zajedničkog djelitelja) dvaju pozitivnih cijelih brojeva može se napisati u jeziku DDH kao:

```
BEGIN PRICON x0, y0; PRIVAR x, y;
  x0 VIR int, y0 VIR int = 105, 15; x VIR int, y VIR int = x0, y0;
  DO [ x >= y ] x = x-y ! [ x <= y ] y = y-x OD; "print x"
  END
```

Varijable sa strukturom polja

Varijable uvedene u prethodnom dijelu, koje su cjelobrojnog ili logičkog tipa, mogu imati samo jednu vrijednost iz skupa vrijednosti deklariranoga tipa, pa smo ih nazvali primitivne varijable.

Pored primitivnih varijabli promatraju se i njihovi skupovi koji imaju isto ime, a različit „indeks". To je struktura podataka koju nazivamo „polje" (v. drugo poglavlje). Povijesno gledajući, u FORTRAN-u su, na primjer, takve varijable morale biti deklarirane s DIMENSION kojeg slijedi ime (ili imena) varijable i u zagradi cijeli broj veći ili jednak 1 koji je predstavljao „dimenziju" varijable. U Pascalu su takve varijable bile deklarirane s ARRAY iza kojeg su u uglatoj zagradi bile navedene domene indeksa općenito višedimenzionalnog polja.

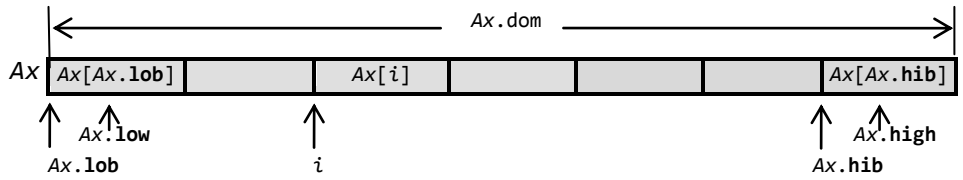
Svi nedostaci vezani za primitivne varijable u jezicima za programiranje u vrijeme Dijkstrinog definiranja jezika DDH, a i danas, vrijede i ovdje: FORTRAN niti Pascal ne zahtijevaju obaveznu inicijalizaciju varijabli sa strukturom polja prije uporabe. Tip je u FORTRAN-u, opet po fiksnom dogovoru, bio određen početnim slovom imena varijable. U Pascalu se morao definirati u samoj deklaraciji, ali u oba jezika postojala je mogućnost referiranja na nepostojeću komponentu polja, izvan domene indeksa, bez upozorenja da se to dogodilo. Pritom se moglo otići u nedopuštena područja radne memorije, što je moglo dovesti do blokiranja rada računala.

Dijkstra, ne držeći se strogo rješenja u tadašnjim jezicima za programiranje, definira polje kao strukturu koja je opisana određenim atributima (svojstvima) i funkcijama (metodama), svojom domenom i semantikom. Sada se može reći da je takva struktura podataka imala obilježja klase.

Polje može biti parametrizirano po bazičnom tipu (**int** ili **bool**). Prednost parametrizacije je u lakšem uvođenju novih tipova (pri proširenju jezika). Takvim pristupom u definiranju varijabli sa strukturom polja možemo smatrati da su strukture nad tipom **int** i **bool** jednake sve do razine parametara, tj. do elemenata strukture koji su iz domene mogućih vrijednosti bazičnog tipa.

ATRIBUTI POLJA

Neka je Ax varijabla sa strukturom polja, kao što je prikazano na sl. 8.1.



Sl. 8.1 – Varijabla sa strukturom polja i njezini atributi.

U aktivnom dosegu polja Ax možemo izlučiti kao prvi atribut domenu i pišemo:

```
Ax.dom
```

a to je cijeli broj koji govori koliko elemenata sadrži polje *Ax* na određenom mjestu u programu. Indeksi polja *Ax* bit će iz neprekinutog intervala cijelih brojeva. Atributi

Ax.**lob** i *Ax*.**hib**

označivat će donju i gornju granicu domene polja Ax. Time je polje *Ax* definirano samo za indeks *i* koji zadovoljava uvjet:

$$Ax.\mathbf{lob} \le i \le Ax.\mathbf{hib} \ \wedge \ Ax.\mathbf{dom} \ge 0$$

Ova tri dosad uvedena atributa zadovoljavaju uvjet:

$$Ax.\mathbf{dom} = Ax.\mathbf{hib} - Ax.\mathbf{lob} + 1 \ge 0$$

Vrijednost polja *Ax* na mjestu i domene označivat ćemo s *Ax*[*i*]. Za *Ax*.**dom** > 0 uvodimo dva atributa:

Ax.**low** definiran kao Ax[Ax.**lob**] i
Ax.**high** definiran kao Ax[Ax.**hib**]

Oni predstavljaju vrijednost polja Ax u najnižoj i najvišoj točki domene.

INICIJALIZACIJA POLJA

Identifikator polja je rezervirana riječ **ARRAY** i pridružuje se varijabli naredbom za inicijalizaciju polja:

```
inicijalizacija_polja : ime VIR tip ARRAY = ( broj {, konstanta } )
broj_elemenata        : brojka { brojka }
konstanta             : broj | log_vrijednost
log_vrijednost        : FALSE | TRUE
```

Inicijalna domena polja određena je brojem navedenih konstanti. Ako nije navedena nijedna konstanta, inicijalna domena polja jednaka je nuli.

Pri pisanju konstanti mora biti zadovoljen kontekstni aspekt: tip konstante mora biti jednak bazičnom tipu polja. Referiranje na pojedine komponente polja piše se prema pravilu:

```
komponenta_polja : ime_polja [ cjelobrojni_izraz ]
```

uz uvjet da je vrijednost cjelobrojnog izraza unutar domene indeksa polja.

♣ Primjer 8.10

Polje A inicijalizirano naredbom:

```
A VIR int array = (-7, 10, -12)
```

ima sljedeće vrijednosti svojih atributa:

```
A.dom  = 2  A.lob  = -7  A.hib  = -6  A.low  = 10  A.high = -12
```

Indeks komponenti polja mora biti u intervalu od -7 do -6, pa su definirane sljedeće vrijednosti:

```
A[-7] = 10  A[-6] = -12
```

♣ Primjer 8.11

Naredbom

```
S VIR int array = (-2);
```

inicijalizirano je cjelobrojno polje sa sljedećim vrijednostima atributa:

```
S.lob = -2  S.hib = -3  S.dom =  0
```

Primijetimo da se **S.hib** dobije iz **S.dom + S.lob -1 = -3**. Ostali atributi polja S, **S.low** i **S.high**, nisu definirani.

♣ Primjer 8.12

Iz inicijalizacije

```
P VIR bool array = (1, False, True, False, True);
```

imamo:

```
P.lob  = 1         P [1] = False
P.hib  = 4         P [2] = True
P.dom  = 4         P [3] = False
P.low  = False     P [4] = True
P.high = True
```

Atributi **dom**, **lob** i **hib** su cjelobrojne vrijednosti i mogu se pojaviti kao operandi u cjelobrojnim izrazima. Atributi **low** i **high** i inicijalizirane komponente polja imaju tip jednak bazičnom tipu polja pa se, ovisno o svom bazičnom tipu, mogu pisati kao operandi u cjelobrojnim ili logičkim izrazima. Dakle, možemo prvobitnoj definiciji operanda u cjelobrojnim izrazima dodati:

```
operand : ime_polja1 . atribut | ime_polja1 [ domena ]
atribut : dom | lob | hib | low | high
```

i proširiti operande u logičkim izrazama sa:

```
log_operand : ime_polja2 . atribut2 | ime_polja2 [ domena ]
atribut2    : low | high
```

OPERATORI NAD POLJEM

Nad strukturom polja, u njegovom aktivnom dosegu, postoji nekoliko operatora (ili funkcija). Ako je *Ax* varijabla sa strukturom polja i *k* cjelobrojni izraz, pravilo pisanja prvog operatora je sljedeće:

```
Ax : shift (k)
```

Operator **shift** će, ako je $k>0$, prouzročiti pomicanje granica domene navedenog polja *Ax* "udesno" za k mjesta; ako je $k<0$, za k mjesta "ulijevo", a ako je $k=0$, transformacija je ekvivalentna **SKIP**. Značenje znaka "**:**" jest promjena vrijednosti (odnosno atributa) polja *Ax*, pa je semantika operacije **shift**:

$$wp(\text{"}Ax\text{: shift } (k)\text{"}, R) = R_{Ax'\to Ax}$$

tj. kopija od R u kojem je svako pojavljivanje od *Ax* zamijenjeno s *Ax'*, gdje je:

$$Ax'.\mathbf{lob} = Ax.\mathbf{lob} + k$$
$$Ax'.\mathbf{hib} = Ax.\mathbf{hib} + k$$
$$Ax'.\mathbf{dom} = Ax.\mathbf{dom}$$
$$Ax'[i] = Ax[i-k]$$

Iz ovih definicija slijedi:

$$Ax'.\mathbf{low} = Ax.\mathbf{low} \quad \text{i} \quad Ax'.\mathbf{high} = Ax.\mathbf{high}$$

uz napomenu da nedefiniranost desnih strana u gornjim jednakostima implicira nedefiniranost lijevih strana.

Sljedeća dva operatora proširuju definiranost polja s "gornjeg" ili "donjeg" kraja. Vrijednost funkcije u novoj točki dana je kao parametar x, a to je izraz koji mora biti iz bazičnog tipa polja. Pišu se prema pravilu:

$$Ax : \mathbf{hiext}\ (x) \quad \text{i} \quad Ax : \mathbf{loext}\ (x)$$

Semantika operatora **hiext** je:

$$wp(\text{"}Ax\text{: hiext } (x)\text{"}, R) = R_{Ax'\to Ax}$$

gdje su:

$$Ax'.\mathbf{lob} = Ax.\mathbf{lob}$$
$$Ax'.\mathbf{hib} = Ax.\mathbf{hib} + 1$$
$$Ax'.\mathbf{dom} = Ax.\mathbf{dom} + 1$$
$$Ax'[i] = x \qquad \text{za } i = Ax.\mathbf{hib} + 1$$
$$= Ax[i] \quad \text{za } i \neq Ax.\mathbf{hib} + 1$$

Semantika operatora **loext** je:

$$wp(\text{"}Ax\text{: loext } (x)\text{"}, R) = R_{Ax'\to Ax}$$

gdje su:

$$Ax'.\mathbf{lob} = Ax.\mathbf{lob} - 1$$
$$Ax'.\mathbf{hib} = Ax.\mathbf{hib}$$
$$Ax'.\mathbf{dom} = Ax.\mathbf{dom} + 1$$
$$Ax'[i] = x \qquad \text{za } i = Ax.\mathbf{hib} - 1$$
$$= Ax[i] \quad \text{za } i \neq Ax.\mathbf{hib} - 1$$

♣ Primjer 8.13

Ako se nad poljem A iz primjera 8.10, inicijaliziranim naredbom:

```
A VIR int array = (-7, 10, -12)
```

izvrše sljedeće operacije

```
A : loext (-3);  A : hiext (77);  A : hiext (33)
```

imali bismo:

```
A.dom  = 5
A.lob  = -8
A.hib  = -4
A.low  = -3
A.high = 33
```

Ako sada nad istim poljem izvršimo operaciju:

```
A : shift (8)
```

Bilo bi:

```
A.lob  = 0
A.hib  = 4
```

dok bi vrijednost preostalih atributnih funkcija ostala nepromijenjena.

Sada uvodimo dva operatora, **hirem** i **lorem**, koje smanjuju domenu za jedan, odnosno ukidaju vrijednost na početku ili kraju polja. Definirane su pod uvjetom da je *Ax*.**dom**>0. Pokušaj da se izvrše nad poljem za koje je domena jednaka nuli ekvivalentno je s **ABORT**. Semantika operacije **hirem** je:

$$wp(\text{"}Ax\text{: hirem"}, R) = (Ax.\mathbf{dom} > 0 \wedge R_{Ax'\to Ax})$$

gdje su:

```
Ax'.lob = Ax.lob
Ax'.hib = Ax.hib -1
Ax'.dom = Ax.dom -1
Ax' [i] = nije definirano    za i = Ax.hib
        = Ax [i]             za i ≠ Ax.hib
```

Semantika operacije **lorem** je:

$$wp(\text{"}Ax\text{: lorem"}, R) = (Ax.\mathbf{dom} > 0 \wedge R_{Ax'\to Ax})$$

gdje su:

```
Ax'.lob = Ax.lob +1
Ax'.hib = Ax.hib
Ax'.dom = Ax.dom -1
Ax' [i] = nije definirano    za i = Ax.lob
        = Ax  [i]            za i ≠ Ax.lob
```

♣ Primjer 8.14

Poslije izvršenja operacije

```
A : lorem
```

nad poljem A iz prethodnog primjera, imali bismo:

```
A.dom  = 4
A.lob  = 1
A.hib  = 4
A.low  = 10
A.high = 33
```

Ako sada nad istim poljem izvršimo operaciju:

```
A : hirem
```

bilo bi:

```
A.dom  = 3
A.lob  = 1
A.hib  = 3
A.low  = 10
A.high = 77
```

Dalje se uvode još dva operatora:

x, Ax : **hipop** i *x, Ax* : **lopop**

Prvi je semantički ekvivalentan s:

x = *Ax*.**high**; *Ax* : **hirem**

a drugi s:

x = *Ax*.**low**; *Ax* : **lorem**

Uz to, svakako, mora biti ispunjen uvjet da je tip primitivne varijable *x* jednak bazičnom tipu polja *Ax*.

Sljedeći operator preuređuje polje izmjenjujući međusobno dvije vrijednosti na mjestima *i* i *j* unutar domene. Označuje se sa **swap**, semantika mu je:

$$wp(\text{"}Ax\text{: swap } (i, j)\text{"}, R) = (Ax.\mathbf{dom} > 0 \wedge Ax.\mathbf{lob} \le i \le Ax.\mathbf{hib} \wedge Ax.\mathbf{lob} \le j \le Ax.\mathbf{hib} \wedge R_{Ax' \to Ax})$$

gdje su:

$$\begin{aligned} Ax'.\mathbf{lob} &= Ax.\mathbf{lob} \\ Ax'.\mathbf{hib} &= Ax.\mathbf{hib} \\ Ax'.\mathbf{dom} &= Ax.\mathbf{dom} \\ Ax'[k] &= Ax[j] \quad \text{za } k = i \\ &= Ax[i] \quad \text{za } k = j \\ &= Ax[k] \quad \text{za } k \neq i \wedge k \neq j \end{aligned}$$

♣ Primjer 8.15

Polje **X** inicijalizirano je naredbom:

```
X VIR int array = (1, 10,20,30,40,50,60,70,80,90)
```

što možemo prikazati kao

X	10	55	30	66	50	77	70	88	90
	1	2	3	4	5	6	7	8	9

gdje su:

```
X.dom = 9   X.lob = 1   X.hib = 9   X.low = 10   X.high = 90
```

Uz pretpostavku da je:

```
i VIR int = X.lob
```

poslije izvršenja dijela programa:

```
DO [i < (X.lob + X.hib)/2] X: swap (i, X.hib –i +1); i = i +1 OD
```

polje će **X** sadržavati vrijednosti u reverznom nizu:

X	90	88	70	77	50	66	30	55	10
	1	2	3	4	5	6	7	8	9

Domena, donja i gornja granica domene ostali su nepromijenjeni, a nove vrijednosti atributa `low` i `high` su:

```
X.low = 90   X.high = 10
```

Posljednji operand koji ćemo uvesti jest promjena vrijednosti varijable sa strukturom polja na mjestu i unutar njezine domene. Pisat ćemo:

$$Ax \texttt{ : } \textbf{alt}\ (i,x)$$

pri čemu tip varijable (izraza) mora biti jednak bazičnom tipu polja Ax. Semantika ove operacije je:

$$wp(\texttt{"}Ax\texttt{:}\textbf{alt}(i,x)\texttt{"},\ R) = (Ax.\textbf{lob} \leq i \leq Ax.\textbf{hib} \wedge R_{Ax' \to Ax})$$

gdje su:

$$\begin{aligned} &Ax'.\textbf{lob} = Ax.\textbf{lob} \\ &Ax'.\textbf{hib} = Ax.\textbf{hib} \\ &Ax'.\textbf{dom} = Ax.\textbf{dom} \\ &Ax'\ [k] = x \qquad \text{za } k = i \\ &\qquad\ = Ax\ [k] \quad \text{za } k \neq i \end{aligned}$$

8.5 NAREDBE ZA UNOS I ISPIS

Da bismo mogli proširiti primjenljivost izvorno definiranog Dijkstrinog jezika, proširujemo ga uvodeći dvije korisne naredbe: naredbu za unos i naredbu za ispis. Prvom od tih naredbi moći će se inicijalizirati primitivne varijable u vrijeme izvršavanja programa. Druga će naredba omogućiti prikaz (ispis) rezultata programa.

Za obje ćemo naredbe trebati niz znakova da bi se mogao komentirati unos, odnosno ispis vrijednosti. Za to uvodimo niz znakova koji se piše prema pravilu:

```
niz_znakova : ' { znak } '
```

gdje je ***znak*** bilo koji ASCII znak (što uključuje i razmak ili blank). Naredba za unos bit će dio naredbe za inicijalizaciju:

```
naredba_za_unos    : input ( niz_znakova )
naredba_za_ispis   : ( Write | WriteLn ) ( ispis )
ispis              : [ izraz { , izraz } ]
izraz              : cjelobrojni_izraz | logički_izraz | niz_znakova
```

P R O G R A M I

Na kraju ovoga poglavlja u ovom ćemo dijelu dati nekoliko primjera programa napisanih u jeziku **DDH**. To su neki poznati problemi – algoritmi teorije brojeva i grafova koji u dovoljnoj mjeri prikazuju upotrebu tipova podataka, strukture polja i operatora jezika **DDH** u njihovom kreiranju.

EUCLIDOV ALGORITAM

Euclidov algoritam za traženje najveće zajedničke mjere dvaju pozitivnih cijelih brojeva ne treba posebno objašnjavati. U jeziku DDH može se realizirati na sljedeći način:

```
PROGRAM Euclid;
BEGIN
  PRIVAR x0, y0, x, y;
  x0 VIR int = input ('Prvi broj ');  y0 VIR int = input ('Drugi broj');
  x  VIR int = x0;                    y  VIR int = y0;
  IF [x0 > 0 and y0 > 0]
     DO [x > y] x = x -y
       ![y > x] y = y -x
       OD;
     WriteLN ('NZD(', x0, ', ', y0, ') = ', x)
    ![x0 <= 0 or y0 <= 0] Write ('POGREŠKA')
    FI
  END.
```

HAMMINGOV NIZ

Potrebno je generirati rastući niz:

```
1 2 3 4 5 6 8 9 10 12 15 16 18 ...
```

svih brojeva djeljivih samo s 2, 3 ili 5, što se može zadati aksiomima:

1) 1 je član niza,
2) ako je x u nizu, tada su 2*x, 3*x i 5*x također u nizu, i
3) niz sadrži samo one brojeve koji zadovoljavaju aksiome (1) i (2).

Detaljno je i postupno rješenje ovog problema dao Dijkstra, [Dij1976], u poglavlju 17. S obzirom na to da se u ovoj knjizi ne bavimo algoritmima koji su izvan teorije formalnih jezika, za naše potrebe dovoljno je imati konačno rješenje – program u jeziku **DDH**:

```
PROGRAM Hamming; { Hammingov niz } BEGIN
  PRIVAR AQ, i2, i3, i5, x2, x3, x5, N, i;
    AQ VIR int array = (1, 1);
    i2 VIR int = 1;  i3 VIR int = 1;  i5 VIR int = 1;  x2 VIR int = 2;
    x3 VIR int = 3;  x5 VIR int = 5;  N  VIR int = 30;
  DO [AQ.dom # N]
    IF [x3 >= x2 and x2 <= x5] AQ: hiext [x2]
      ![x2 >= x3 and x3 <= x5] AQ: hiext [x3]
      ![x2 >= x5 and x5 <= x3] AQ: hiext [x5]
      FI;
    DO [x2 <= AQ.high] i2 = i2 +1; x2 = 2 *AQ[i2] OD;
    DO [x3 <= AQ.high] i3 = i3 +1; x3 = 3 *AQ[i3] OD;
    DO [x5 <= AQ.high] i5 = i5 +1; x5 = 5 *AQ[i5] OD
    OD;
  i VIR int = AQ.lob;
  Writeln ('Prvih ', N, ' članova Hammingova niza:');
  DO [i <= AQ.hib] Write (AQ[i]); i := i +1 OD
  END.
```

PROBLEM n-te PERMUTACIJE

Dijkstra problem *n*-te i sljedeće permutacije opisuje u osmom i trinaestom poglavlju. Prije nego što prijeđemo na rješenje problema, dajemo nekoliko definicija. Neka je:

$p = (p_0, p_1, \ldots, p_{n-1})$

permutacija od n, $n>1$, različitih vrijednosti p_i, $0 \le i < n$, i neka je

$q = (q_0, q_1, \ldots, q_{n-1})$

permutacija različita od p ali koju čini isti skup vrijednosti kao u p. Izjavu: „permutacija p prethodi permutaciji q u alfabetskom uređenju“ permutacije p i q ispunjavaju onda i samo onda ako za minimalnu vrijednost k za koju je $p_k \neq q_k$ imamo $p_k < q_k$. Na primjer, za $n=3$ i skup vrijednosti 2, 4 i 7, imamo:

$\text{indeks}_3\ (2, 4, 7) = 0$ $\text{indeks}_3\ (2, 7, 4) = 1$ $\text{indeks}_3\ (4, 2, 7) = 2$
$\text{indeks}_3\ (4, 7, 2) = 3$ $\text{indeks}_3\ (7, 2, 4) = 4$ $\text{indeks}_3\ (7, 4, 2) = 5$

Permutacija $(4, 2, 7)$ prethodi permutaciji $(4, 7, 2)$, jer za $p_2 \neq q_2$ vrijedi $p_2 < q_2$.

indeks_n permutacije n različitih elemenata je indeks_n alfabetski prve s istom krajnjom vrijednošću uvećan za indeks_{n-1} permutacije preostalih $n-1$ krajnjih desnih vrijednosti. Na primjer:

$\text{indeks}_3\ (4, 7, 2) = \text{indeks}_3\ (4, 2, 7) + \text{indeks}_2(7, 2) = 2 + 1 = 3$

Početna permutacija ima indeks jednak 0, a posljednja $n!-1$, gdje je n broj članova. Kompletno rješenje se može naći u spomenutoj Dijkstrinoj knjizi.

```
PROGRAM Nperm; BEGIN
  PRIVAR C, N;
  C VIR int array = (1, 0,1,2,3,4,5,6,7,8,9);
  N VIR int = input ('Računam permutaciju broj? >') # 3628799 je posljednja
  WriteLn (0); WriteLn (C);
  BEGIN
    GLOVAR C, N;  PRIVAR S, Kfac, i, j, k;
    S VIR int = 0;  Kfac VIR int = 1;  k VIR int = 1; i VIR int = 0;  j VIR int = 1;
    DO [ k <> C.hib ] Kfac = Kfac*(k+1); k = k+1 OD;
    DO [ S <> N ]
      DO [ N < S+Kfac ] Kfac = Kfac/k; k = k-1 OD;
      i = C.hib-k; j = i+1;
      DO [ S+Kfac <= N ] S = S+Kfac; C : swap (i,j); j = j+1 OD
      OD;
    WriteLn (N); Writeln (C)
    END
  END.
```

ALGORITAM ZA NALAŽENJE PRIM BROJEVA

Sljedeći program predstavlja realizaciju poznatog algoritma za nalaženje prim brojeva poznat kao Eratostenovo sito.

```
PROGRAM Eratos; BEGIN
  PRIVAR P, Q, X, S, N, i;
  P VIR int =   2;  Q VIR int = 0;  N VIR int = 100;  X VIR int = 0;
  i VIR int =   0;  S VIR int array = (1);
  DO [ i <> N ] i = i+1; S : hiext (i) OD;
  BEGIN
    GLOVAR P, Q, X, S; GLOCON N; PRIVAR m, r;
    m VIR int = 0;  r VIR int = 0;
    DO [ P**2 <= N ] Q = P;
      DO [ P*Q <= N ] X = P*Q;
        DO [ X <= N ] S : (X) = 0; X = P*X OD;
        m = Q+1;
        DO [ S[m] == 0 ] m = m+1 OD;  Q = S[m]
        OD;
      r = P+1;
      DO [ S[r] == 0 ] r = r+1 OD; P = S[r]
      OD
    END;
  i = 2
  DO [ i <= N ]
    IF [ S[i] <> 0 ] Write (S[i])
      ![ S[i] == 0 ] SKIP
      FI;
    i = i+1
    OD
  END.
```

9.
PREDPROCESOR JEZIKA DDH

— Hublement il est venu

Ponizno je došao
Nitko ga nije prepoznao
Bio je loše odjenut
Nije imao ni cipele ni kaput
Zato što bos bje
Nitko ga prepoznao nije

Ponizno je došao
Kao da je s oblaka pao
Pretiho je riječi izgovorio
Nije se sve razumjelo
Ali nikoga na zemlji ne bi
Tko bi ga mogao poslušati

Ponizno je došao
Dobrodošlicu je trebao
Zamoliti vina i kruha
I jedan ležaj samo do jutra
Ništa više od toga nije zaželio
Ipak ništa nije dobio

Ponizno je došao
Ponizno je nestao
To nije samo stranac bio
Kojeg se više nije tražilo
To nije samo neznanac bio
Kojeg se nije zadržati željelo

Bilo je to prije 2000 godina ili više
I nikad se ponoviti neće
Ali ipak je ostao u sjećanju
I cijeli je svijet u iščekivanju

Djevojke koje ga žele dobiti
Moraju se svaku večer uljepšati
Djeca o njemu pričaju
Kao što se priča o svom prijatelju

Ljudi nisu ništa rekli
Ali su mu jednu čašu vina sačuvali
On će je jednoga dana popiti
U ljubavi i radosti

Ponizno je došao
Humblement il est venu

(Georges Moustaki/
Zdravko DOVEDAN HAN)

*Na kraju ćemo ove knjige primijeniti sva stečena znanja u realizaciji predprocesora jezika **DDH**. No premda, kao što je opisano u prethodnom poglavlju, ima prilično jednostavnu sintaksu, jezik **DDH** je sa strogom definicijom semantike svojih naredbi i jakim kontekstnim aspektima podesan za proučavanje u teoriji formalnih jezika, sintaksnoj i semantičkoj analizi, te prevođenju.*

Ovdje je u potpunosti opisan postupak njegove leksičke analize, sintaksne analize i prevođenja. Sintaksna analiza realizirana je kao prepoznavač jezika sa svojstvima, a to je prepoznavač upravljan tablicom prijelaza i akcija. Predprocesor je realiziran u Pythonu. I ciljni jezik je Python.

9.1 UVOD

Prije nego što prijeđemo na opis predprocesora jezika **DDH** dajemo nekoliko povijesnica vezanih za prve njegove verzije.

Ideja za realizaciju prvog predprocesora jezika **DDH** (tada nazvan „DIKTRAN“) nastala je još u vrijeme izrade magistarskog rada, [Dov1982], u razdoblju od 1980. do 1982. godine. Prva verzija predprocesora bila je gotova u ljeto 1981. godine. Predprocesor je bio napisan u jeziku **ANSI FORTRAN** na računalu CDC 3170. Za ciljni je jezik bio izabran **FORTRAN MS**, na istom računalu. Za sintaksnu analizu koristio se rekurzivni spust.

Analizom rezultata zaključilo se da je dobiven velik i neefikasan programski kôd. Razlog tomu bio je veliki broj selekcija u modulu sintaksne analize. To je uvjetovano samim algoritmom – rekurzivnim spustom, ali i jezikom implementacije – **ANSI FORTRAN**-om koji nije imao mogućnost rekurzivnih poziva potprograma.

FORTRAN MS je imao deklaraciju polja sa zadanim brojem elemenata što je dovelo do predefiniranja inicijalizacije polja jezika **DDH**:

```
inicijalizacija_polja :
  ime VIR tip ARRAY ( broj_elemenata ) = ( broj {, konstanta } )
```

Uveden je ***broj_elemenata*** kojim je limitiran broj elemenata polja. To je novi atribut **lim**. Ako je *Ax* polje koje se inicijalizira, njegova je inicijalna domena određena brojem navedenih konstanti i mora biti ispunjen uvjet $Ax.\mathbf{dom} \leq Ax.\mathbf{lim}$. Ovim se proširenjem naredbe za inicijalizaciju polja promijenila i semantika onih operacija nad poljem koje su proširivale njegovu domenu. Na primjer, semantika funkcija **hiext** i **loext**:

$wp(\text{"}Ax\text{: hiext } (x)\text{"}, R) = R_{Ax' \to Ax}$

$wp(\text{"}Ax\text{: loext } (x)\text{"}, R) = R_{Ax' \to Ax}$

bila bi:

$wp(\text{"}Ax\text{: }\mathbf{hiext}\ (x)\text{"}, R) = (Ax.\mathbf{dom} < Ax.\mathbf{lim} \wedge R_{Ax' \to Ax})$

$wp(\text{"}Ax\text{: }\mathbf{loext}\ (x)\text{"}, R) = (Ax.\mathbf{dom} < Ax.\mathbf{lim} \wedge R_{Ax' \to Ax})$

Jezik **DIKTRAN** je bio proširen i naredbom za ispis. Bila je to naredba WRITE koja se pisala prema pravilima jezika FORTRAN (imala je definiran „format“ ispisa).

U drugoj je inačici predprocesor jezika **DIKTRAN** bio implementiran u FORTRAN-u 77, ali na računalu CYBER 170/825 i postao je dostupan za javnu uporabu, [Sta1985]. Posebno je bio prikladan za kolegije koji su se bavili teorijom programiranja, [Ost1989].

Prošlo je preko trideset godina od prve implementacije jezika **DDH**. Od tada su se pojavili mnogi novi jezici za programiranje, a postojeći proširili i prilagodili razvoju kompjuterske tehnologije. Uveden je i pojam „objektnog programiranja“ pa ćemo imati širok spektar jezika u izboru „pravog“ ciljnog jezika našeg predprocesora jezika **DDH**.

9.2 OPIS JEZIKA DDH

U prethodnom smo poglavlju definirali jezik **DDH**, njegove tipove podataka te sintaksu i semantiku njegovih naredbi. Ovdje ćemo dati sumarni pregled u BNF-u. Prije toga uvodimo neznatna proširenja neophodna za realizaciju predprocesora.

Leksička struktura

Da bismo definirali osnovnu leksičku strukturu jezika **DDH**, treba definirati njegov alfabet, rječnik i leksička pravila.

ALFABET

Alfabet jezika **DDH** čine sljedeće skupine znakova:

- velika slova engleskog alfabeta: `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
- mala slova engleskog alfabeta: `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- brojke: `0 1 2 3 4 5 6 7 8 9`
- posebni znakovi: `+ - * ( ) [ ] { } = ! < > , : ; '`

RJEČNIK

Nad alfabetom jezika **DDH** definiran je rječnik sačinjen od sljedećih skupina riječi ili simbola:

- rezervirane riječi:

ABORT	**AND**	**ARRAY**	**BEGIN**	**BOOL**	**DO**	**DOM**	**END**
FALSE	**FI**	**GLOCON**	**GLOVAR**	**HIB**	**HIGH**	**HIPOP**	**HIREM**
IF	**INT**	**LOB**	**LOEXT**	**LOPOP**	**LOREM**	**LOW**	**MOD**
NOT	**OD**	**OR**	**PRICON**	**PRIVAR**	**PROGRAM**	**SKIP**	**TRUE**
VIR	**VIRCON**	**VIRVAR**	**WRITE**	**WRITELN**			

- imena:

```
ime      : slovo { slovo | brojka }
  slovo  : veliko_slovo | malo_slovo
  brojka : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Identifikator mora biti različit od rezerviranih riječi.

- cijeli brojevi:

```
cijeli_broj   : predznak broj
  predznak    : + | - | ε
  broj        : brojka { brojka }
```

- nizovi znakova:

```
niz_znakova   : ' { znak | razmak } '
```

- posebni simboli:

  - komponirani: `== <> >= <=`
  - jednostavni: `+ - * / = > < ! ( ) [ ] { } . ; : '`

LEKSIČKA PRAVILA

Sve se riječi pišu kompaktno, bez razmaka između znakova. Velika i mala slova imaju jednako značenje.

Sintaksna struktura

Na najvišoj se razini osnovna sintaksna struktura jezika **DDH** može predočiti sa:

```
program :   PROGRAM ime_programa; { komentar; } blok.
blok    :   BEGIN
              nomenklatura {; nomenklatura }{; naredba }
            END
naredba :   blok |SKIP | ABORT | komentar | inicijalizacija |dodjeljivanje |
            selekcija | iteracija | modifikacija_polja |naredba_za_unos |
             naredba_za_ispis
```

Sada preostaje da se defineniraju tipovi i strukture podataka, te pravila pisanja pojedinih naredbi i njihova semantika.

```
nomenklatura       : tip_nomenklature ime {, ime }
  tip_nomenklature : PRIVAR | PRICON | VIRVAR | VIRCON | GLOVAR | GLOCON
komentar           : { { znak } }
inicijalizacija    : ime VIR tip = ulaz | inic_polja
  ulaz             : izraz | unos
    unos           : INPUT ( niz_znakova )
```

```
  inic_polja       : ime VIR tip ARRAY
                     = (cijeli_broj { , konstanta })
    konstanta      : cijeli_broj | log_vr
    log_vr           : FALSE | TRUE

dodjeljivanje      : varijabla = izraz | varijabla, dodjeljivanje, izraz
  varijabla          : ime_prim_var
    ime_prim_var       : ime
  izraz              : cjel_izraz | log_izraz
    cjel_izraz         : operand { operacija operand }
      operacija          : + | - | * | / | MOD
      operand            : broj | cjel_var | cjel_kon | - cjel_izraz |
                           cjel_polje . atribut | cjel_polje [ domena ] |
                           ( cjel_izraz )
        atribut            : DOM | LOB | HIB | LOW | HIGH
        cjel_var           : ime
        cjel_kon           : ime
        cjel_polje         : ime
        domena             : cjel_izraz
    log_izraz          : log_operand { log_operacija log_operand }
      log_operacija      : AND | OR
      log_operand        : FALSE | TRUE | log_var | log_kon |  rel_izraz | NOT log_izraz |
                           log_polje . atribut2 | log_polje [ domena ] |( log_izraz )
        atribut2           : LOW | HIGH
        log_var            : ime
        log_var            : ime
        log_polje          : ime
        rel_izraz          : cjel_izraz relacija cjel_izraz
        relacija           : == | > | < | >= | <= | <>

selekcija          : IF zaklonjeni_skup FI
  zaklonjeni_skup    : prolaz { ! prolaz }
    prolaz             : [ log_izraz ] naredba { ; naredba }

iteracija          : DO zaklonjeni_skup OD

modifikacija_polja : polje , varijabla : pop_operator |
                     polje : rem_operator |
                     polje : ext_operator ( izraz ) |
                     polje : SHIFT ( cjel_izraz ) |
                     polje : SWAP ( domena , domena ) |
                     polje : ( domena ) = ulaz
  polje              : ime_polja
    ime_polja          : ime
  pop_operator       : LOPOP | HIPOP
  rem_operator       : LOREM | HIREM
  ext_operator       : LOEXT | HIEXT

naredba_za_unos    : varijabla = unos

naredba_za_ispis     : ( WRITE | WRITELN ) ( ispis { , ispis } )
  ispis                : cjel_izraz | log_izraz | niz_znakova
```

Dana sintaksna struktura (beskontekstna gramatika) ne opisuje jezik DDH u potpunosti, ali će nam korisno poslužiti u definiranju njegova prepoznavača.

9.3 IZBOR CILJNOG JEZIKA

Temeljno pravilo za izbor ciljnog jezika jest: sintaksa i semantika ciljnog jezika trebaju biti što bliži sintaksi i semantici izvornog jezika. Da bismo bili sigurni da smo izabrali „pravi" ciljni jezik trebamo analizirati postojeće jezike, biti potpuno upoznati s njihovom sintaksom i semantikom, te se potom odlučiti za „optimalno" rješenje. Na primjer, ako imamo zadatak realizirati predprocesor jezika PL/0, za koji se može reći da je „mali Pascal", očigledno bismo za ciljni jezik izabrali Pascal.

Kao ciljni jezik u realizaciji prva dva predprocesora jezika DDH, kao što smo spomenuli u uvodu, bili su izabrani ANSI FORTRAN i FORTRAN 77. Pojavom osobnih računala, posebno Pascala i njegovih poznatih inačica Turbo Pascal 3.0, 4.0 do 7.0 tvrtke Borland (a danas aktualne inačice pod nazivom Free Pascal), Pascal bi bio zadovoljavajuće rješenje za implementaciju predprocesora, a i kao ciljni jezik: ima strogo definirane tipove i strukture podataka, selekciju i WHILE petlju dovoljnu za implementaciju selekcije i DO naredbe jezika **DDH**, te strukturu sloga, odnosno objekte, za implementaciju svojstava polja i funkcija nad njime. Blokovskoj strukturi izvornoga jezika odgovarale bi procedure Pascala u koje bi se globalne varijable prenosile preko ulazno-izlazne parametarske liste procedure, a privatne bi varijable bile lokalne varijable procedure. Na primjer, `PROGRAM Hamming` za generiranje Hammingova niza, dan u dijelu *PROGRAMI* prethodnoga poglavlja, mogao bi se prevesti u Pascal:

```
PROGRAM Hamming; { Hammingov niz }

USES Crt, DDH_GCT;

CONST
  Lim_ = 1000;

TYPE
  arr_int = RECORD
    a   : ARRAY [1..Lim_] OF longint;
    lob, hib, lim : integer
    END;
  arr_bool = RECORD
    a   : ARRAY [1..Lim_] OF boolean;
    lob, hib, lim : integer
    END;

CONST
  i2  : longint = 1;  i3 : longint = 1;
  i5  : longint = 1;  x2 : longint = 2;
  x3  : longint = 3;  x5 : longint = 5;

VAR
  AQ   : arr_int;
  N, i : longint;
```

```
PROCEDURE Init_; BEGIN
  AQ.lim := 30;
  AQ.lob := 1; AQ.hib := AQ.lob;
  AQ.a[AQ.lob] := 1;
  N := AQ.lim;
  END;

BEGIN
  ClrScr; Init_;
  WHILE AQ.hib -AQ.lob +1 <> N DO BEGIN
    IF (x3 >= x2) AND (x2 <= x5) THEN BEGIN
      AQ.hib := AQ.hib +1; AQ.a[AQ.hib] := x2
      END
    ELSE IF (x2 >= x3) AND (x3 <= x5) THEN BEGIN
      AQ.hib := AQ.hib +1; AQ.a[AQ.hib] := x3
      END
    ELSE IF (x2 >= x5) AND (x5 <= x3) THEN BEGIN
      AQ.hib := AQ.hib +1; AQ.a[AQ.hib] := x5
      END;
    WHILE x2 <= AQ.a[AQ.hib] DO BEGIN i2 := i2 +1; x2 := 2 *AQ.a[i2] END;
    WHILE x3 <= AQ.a[AQ.hib] DO BEGIN i3 := i3 +1; x3 := 3 *AQ.a[i3] END;
    WHILE x5 <= AQ.a[AQ.hib] DO BEGIN i5 := i5 +1; x5 := 5 *AQ.a[i5] END;
    END;
  i := AQ.lob;
  WriteLN ('Prvih ', N, ' članova Hammingova niza:');
  WHILE i <= AQ.hib DO BEGIN Write (AQ.a[i] :8); i := i +1 END;
  ReadKey
  END.
```

gdje modul `DDH_GCT` sadrži globalne konstante, strukture i tipove podataka, te procedure za rad s poljima.

Danas postoji nekoliko jezika koji su besplatni ili su s pristupačnom cijenom, pa se pri projektiranju predprocesora pružaju ogromne mogućnosti za izbor. „Pravi“ izbor ciljnog jezika ovisit će o izvornom jeziku. Na primjer, ako bismo željeli dizajnirati predprocesor jezika **PL/0** čiji smo interpretator opisali u sedmom poglavlju, sigurno bismo se odlučili za neku od verzija Pascala.

Koji jezik izabrati za ciljni jezik u implementaciji predprocesora jezika DDH? Nije teško odgovoriti. Procjenjujemo da je to Python! Štoviše, Python je podesan i za pisanje programa samog predprocesora.

Python

Jezik Python postao nam je „glavni“ jezik za prikaz većine algoritama u prethodne dvije knjige, `[Dov2012a]` i `[Dov2012b]`. Koristili smo ga u implementaciji većine algoritama iz prethodne dvije knjige i u svim algoritmima ove knjige. Nismo ga nigdje posebno opisivali jer je pretpostavka da su ga korisnici prethodne dvije i ove knjige naučili ili u predmetima koji se bave problemima algoritama, struktura podataka i programiranja ili su to učinili samostalno. Ovdje ćemo dati samo one naredbe i strukture podataka Pythona koje su neophodne za razumijevanje prevođenja.

U sljedećoj je tablici dan prijevod naredbi jezika **DDH** u jezik Python. Izostavljeni su izravni prijevodi, kao što su, na primjer, relacije, jer smo ih u jeziku **DDH** definirali da budu jednake relacijama u Pythonu.

DDH	Python
BEGIN ...	ε (za razinu 0) **def** BEGIN_*broj* () : (za razinu > 0) Poslije završetka bloka generira se BEGIN_*broj* () (poziv procedure procedure)
(PRI|VIR)(CON|VAR) *ime* {, *ime*}	ε
GLO(CON|VAR) *ime* {, *ime*}	**global** *ime* {, *ime*}
ime **VIR** *tip* = *izraz*	*ime* = *izraz*
ime **VIR** *tip* **ARRAY** = *n-torka*	*ime* = **array** (*'tip'*, *n-torka*)
IF [u_1] n_1 ![u_2] n_2 ... ![u_k] n_k **FI**	**if** u_1 : n_1 **elif** u_2 : n_2 ... **elif** u_k : n_k
DO [u_1] n_1 ![u_2] n_2 ... ![u_k] n_k **OD**	**while** True : **if** u_1 : n_1 **elif** u_2 : n_2 ... **elif** u_k : n_k **else** : **break**
ABORT	**exit** (99)
v_1, v_2, …, v_k = i_1, i_2, …, i_k	v_1, v_2, …, v_k = i_1, i_2, …, i_k
HIREM, LOREM	**hirem** (), **lorem** ()
HIPOP, LOPOP	**.hipop** (), **.lopop** ()
Ax : **SWAP** (*i*, *j*)	*Ax*.**swap** (*i*, *j*)
x, *Ax* : **HIPOP**	*x* = *Ax*.**high**; *Ax* : **hirem**
x, *Ax* : **LOPOP**	*x* = *Ax*.**low**; *Ax* : **lorem**
WRITELN (*argumenti*)	**print** *argumenti*
WRITE (*argumenti*)	**print** *argumenti*,

9.4 STRUKTURA PREDPROCESORA

Da bismo mogli pristupiti problemu dizajniranja predprocesora bilo kojeg jezika, neophodno je znati sintaksu i semantiku ulaznog i ciljnog jezika. U prethodnom smo poglavlju opisali sintaksu i semantiku jezika DDH, a u ovom smo je prikazali sažeto. Sada preostaje da konstruiramo predprocesor. Prvo ćemo definirati leksičku analizu koja će biti realizirana pretvaračem koji na ulazu prihvaća niz znakova i prevodi ih u niz riječi (simbola) rječnika jezika DDH.

Rječnik jezika DDH

Rječnik će sadržavati nepromjenljivi dio, kojeg čine rezervirane riječi i ostali simboli, i promjenljivi dio kojeg čine identifikatori (imena) s određenim svojstvima. Sve riječi imaju kôd (prijelaz), a to je cijeli broj od 1 do 33 za rezervirane riječi, 34 do 48 za identifikatore i 49 do 69 za ostale riječi (i znakove).

REZERVIRANE RIJEČI I KODOVI

PROGRAM	**1**	GLOVAR	**8**	WRITE	**15**	SHIFT	**22**	NOT	**29**
BEGIN	**2**	VIR	**9**	IF	**16**	SWAP	**23**	FALSE	**30**
PRICON	**3**	INT	**10**	DO	**17**	LOPOP, HIPOP	**24**	TRUE	**31**
PRIVAR	**4**	BOOL	**11**	END	**18**	LOREM, HIREM	**25**	AND	**32**
VIRCON	**5**	ARRAY	**12**	FI	**19**	LOB, HIB	**26**	OR	**33**
VIRVAR	**6**	ABORT	**13**	OD	**20**	DOM, LIM	**27**		
GLOCON	**7**	SKIP	**14**	LOEXT,HIEXT	**21**	LOW, HIGH	**28**		

IDENTIFIKATORI SA SVOJSTVIMA

Ako program u jeziku **DDH** općenito napišemo kao:

```
    PROGRAM Ime;
┌──{0}BEGIN
│       a;
│   ┌─{1}BEGIN
│   │       b;
│   │   ┌─{2}BEGIN
│   │   │       c
│   │   └─────  END;
│   │       d
│   └─────────END;
│       e;
│   ┌─ {1}BEGIN
│   │       f
│   └─────────END
└──────────END.
```

gdje su *a*, *b*, *c*, *d*, *e* i *f* skupovi naredbi u pojedinima blokovima, na određenoj razini. Prvi **BEGIN** otvara blok s razinom 0. Svaki novi blok imat će razinu za 1 veću od bloka u kojem je ugniježđen itd. Zatvaranjem bloka (s END) vraća se na razinu manju za 1.

Može se reći da je sintaksom jezika **DDH** uvedena "blokovska struktura" programa. U prethodnom smo poglavlju uveli pravilo da se svaka varijabla smije upotrebljavati samo u okvirima svog tekstualnog konteksta. Na primjer, ako je varijabla X bila deklarirana u nomenklaturi osnovnog bloka (razina 0) bit će pristupačna u naredbama *a* i *e*. Ako je želimo koristiti u nekom od unutarnjih blokova, moramo je redefinirati u njihovim nomenklaturama. Time se uvode izvjesna kontekstna svojstva koje identifikatori moraju zadovoljavati u svim sintaksnim strukturama u kojima se pojavljuju, pa zaključujemo da je **DDH** jezik sa svojstvima.

Osnovnu sintaksnu strukturu možemo definirati beskontekstnom gramatikom, ali to neće biti dovoljno za izvođenje sintaksne analize. S obzirom da ćemo u sintaksnoj analizi koristiti prepoznavač, preciznije prepoznavač jezika sa svojstvima, potrebno je na neki način osigurati provjeru svojstava identifikatora na određenim mjestima u programu i mogućnost njihove promjene. Najprije uvodimo razinu bloka sa sljedećim značenjem:

1) Prvi blok ima razinu jednaku 0.
2) Ostali blokovi imaju razinu za jedan veću od bloka u čijem su kontekstu.

Ako razinu promatramo kao globalnu varijablu RZ koja na početku programa ima vrijednost 0, onda joj se nailaskom na **BEGIN** vrijednost mijenja u:

```
RZ = RZ +1
```

a nailaskom na END u:

```
RZ = RZ -1
```

Dalje, svakom identifikatoru koji predstavlja ime varijable, u pasivnom ili aktivnom dosegu, pridružit ćemo vektor svojstava:

$$s = (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9)$$

gdje su:

$s_0, \ldots, s_7$ deklariranost u bloku razine i, $i = 0, \ldots, 7$:
- s_i = * nije deklarirana u bloku razine i
- s_i = c deklarirana kao CON
- s_i = v deklarirana kao VAR

s_8 osnovni (bazični) tip:
- * nije definiran osnovni tip
- I osnovni tip je int
- B osnovni tip je bool

s_9 struktura:
- p primitivna varijabla
- a polje

s_{10} broj referiranja:
- 0 prvi put
- j j-ti put ($j > 1$)

U bloku koji ima razinu jednaku i sve ćemo identifikatore podijeliti u klase karakterizirane sljedećim predikatima:

i_0 : "nije deklariran"
i_d^I : $s_i \neq * \wedge s_8 = * \wedge s_9 = *$
i_d^0 : $s_{i-1} \neq * \wedge s_i = * \wedge s_8 = * \wedge s_9 = *$
i : $s_{i-2} = * \wedge s_{i-1} = *$
i_r : $s_0 = * \wedge s_1 = * \wedge \ldots \wedge s_{i-1} = *$
i_c^0 : $s_{i-1} = c \wedge s_8 \neq * \wedge s_9 \neq *$
i_v^0 : $s_{i-1} = v \wedge s_8 \neq * \wedge s_9 \neq *$
B_c^p : $s_i = c \wedge s_8 = B \wedge s_9 = p$
B_c^a : $s_i = c \wedge s_8 = B \wedge s_9 = a$
B_v^p : $s_i = v \wedge s_8 = B \wedge s_9 = p$
B_v^a : $s_i = v \wedge s_8 = B \wedge s_9 = a$
I_c^p : $s_i = c \wedge s_8 = I \wedge s_9 = p$
I_c^a : $s_i = c \wedge s_8 = I \wedge s_9 = a$
I_v^p : $s_i = v \wedge s_8 = I \wedge s_9 = p$
I_v^a : $s_i = v \wedge s_8 = I \wedge s_9 = a$

♣ Primjer 9.1

Razmotrimo program:

```
PROGRAM Ident;

BEGIN PRIVAR A, B; PRICON C;
  A VIR int = 66;
  BEGIN GLOCON A; VIRVAR B, C;
    B VIR int = A;
    C VIR bool array = (-1, False);
    { naredbe u bloku 1 }
    END
  { naredbe u bloku 0 }
  END.
```

Na početku prvog bloka svi su identifikatori sa svojstvom i_0. Nomenklaturom identifikatori A, B i C prelaze u klasu sa svojstvima označenim s i_d^I. Daljnja promjena svojstava je:

A : $i_d^I \rightarrow I_v^p \rightarrow i_v^0 \rightarrow I_c^p$

B : $i_d^I \rightarrow i_d^0 \rightarrow i_d^I \rightarrow I_v^p$

C : $i_d^I \rightarrow i_d^0 \rightarrow i_d^I \rightarrow B_v^a$

U sljedećoj su tablici dane skupine identifikatora sa zajedničkim svojstvima i pridružene im oznake i kodovi (prijelazi):

Oznaka	*Opis – svojstva* 012i4567 89	*Kod*	*Oznaka*	*Opis – svojstva* 012i4567 89	*Kod*
i_0	"nije deklariran"	**34**	I_v^a	***v**** Ia	**42**
B_c^p	***c**** Bp	**35**	i_d^0	**+***** **	**43**
B_c^a	***c**** Ba	**36**	i_c^0	**c***** ++	**44**
B_v^p	***v**** Bp	**37**	i_v^0	**v***** ++	**45**
B_v^a	***v**** Ba	**38**	i	******* **	**46**
I_c^p	***c**** Ip	**39**	i_d^I	---+**** **	**47**
I_c^a	***c**** Ia	**40**	i_r	******* **	**48**
I_v^p	***v**** Ip	**41**			

gdje je + oznaka definiranog svojstva, a – definiranog ili nedefiniranog svojstva.

OSTALE RIJEČI

I (broj)	**49**	+	**54**	)	**59**	!	**64**	input	**69**
.	**50**	-	**55**	[	**60**	N (niz)	**65**		
:	**51**	*	**56**	]	**61**	<	**66**		
,	**52**	/	**57**	{	**62**	>	**67**		
;	**53**	(	**58**	}	**63**	=	**68**		

RJEČNIK

Implementacija rječnika u Pythonu, modul `DDH_V.py`, dana je u nastavku. Rječnik je realiziran kao mapa V koja sadrži simbole, njihove kodove (prijelaze) i prijevode u Python. Tu su i rezervirane riječi Pythona `KeyW` koje se ne smiju koristiti kao identifikatori u jeziku DDH.

DDH_V.py

```
# -*- coding: cp1250 -*-
# Rječnik jezika DDH
import os
import string

Slova  = string.ascii_uppercase +string.ascii_lowercase; Brojke = string.digits

#      simbol      kod  DDH       -> Python
V = { 'ABORT'   : (13, 'ABORT',   'exit (99)'        ),
      'AND'     : (32, 'and',     ' and '            ),
      'ARRAY'   : (12, 'array',   'array ('          ),
      'BEGIN'   : ( 2, 'BEGIN',   'def BEGIN_* () :' ),
      'BOOL'    : (11, 'bool',    ''                 ),
      'DO'      : (17, 'DO',      'while True : '    ),
      'DOM'     : (27, 'dom',     'dom'              ),
      'END'     : (18, 'END',     'BEGIN_* ()'       ),
      'FALSE'   : (30, 'False',   'False'            ),
```

```
      'FI'      : (19, 'FI',      ''                ),
      'GLOCON'  : ( 7, 'GLOCON',  'global '         ),
      'GLOVAR'  : ( 8, 'GLOVAR',  'global '         ),
      'HIB'     : (26, 'hib',     'hib'             ),
      'HIEXT'   : (21, 'hiext',   'hiext '          ),
      'HIGH'    : (28, 'high',    'high'            ),
      'HIREM'   : (25, 'hirem',   'hirem ()'        ),
      'HIPOP'   : (26, 'hipop',   '.hipop ()'       ),
      'IF'      : (16, 'IF',      'if '             ),
      'INPUT'   : (69, 'input',   'input'           ),
      'INT'     : (10, 'int',     ''                ),
      'LOB'     : (26, 'lob',     'lob'             ),
      'LOEXT'   : (21, 'loext',   'loext'           ),
      'LOPOP'   : (26, 'lopop',   '.lopop ()'       ),
      'LOREM'   : (25, 'lorem',   'lorem ()'        ),
      'LOW'     : (28, 'low',     'low'             ),
      'NOT'     : (29, 'not',     'not '            ),
      'OD'      : (20, 'OD',      'else : break'    ),
      'OR'      : (33, 'or',      ' or '            ),
      'PRICON'  : ( 3, 'PRICON',  ''                ),
      'PRIVAR'  : ( 4, 'PRIVAR',  ''                ),
      'PROGRAM' : ( 1, 'PROGRAM', '# PROGRAM '      ),
      'SHIFT'   : (24, 'shift',   'shift'           ),
      'SKIP'    : (14, 'SKIP',    'pass'            ),
      'SWAP'    : (23, 'swap',    'swap'            ),
      'TRUE'    : (31, 'True',    'True'            ),
      'VIR'     : ( 9, 'VIR',     ''                ),
      'VIRCON'  : ( 5, 'VIRCON',  'global '         ),
      'VIRVAR'  : ( 6, 'VIRVAR',  'global '         ),
      'WRITE'   : (15, 'Write',   'print '          ),
      'WRITELN' : (15, 'WriteLn', 'print '          ) }

KeyW = ['as',     'assert', 'break',  'class',  'continue', 'def',
        'del',    'elif',   'else',   'except', 'exec',     'finally',
        'for',    'from',   'global', 'import', 'in',       'is',
        'lambda', 'pass',   'print',  'raise',  'return',   'try',
        'while',  'with',   'yield']
SvId = { 'C Bp' : 35,  'C Ba' : 36,  'V Bp' : 37,  'V Ba' : 38,
         'C Ip' : 39,  'C Ia' : 40,  'V Ip' : 41,  'V Ia' : 42,
         'C **' : 47,  'V **' : 47  }
Znak  = { "." : (50, "."),  ":" : (51, "."  ),  "," : (52,    ""),
          ";" : (53,  ""),  "+" : (54, "+"  ),  "-" : (54,   "-"), "–" : (54,  "-"),
                                                "–" : (54,   "-"),
          "*" : (56, "*"),  "/" : (56, "/"  ),  "(" : (58,    ""),
          ")" : (59,  ""),  "[" : (60,  ""  ),  "]" : (61,  " :"),
          "{" : (62,  ""),  "}" : (63,  ""  ),  "!" : (64,    ""),
          "'" : (65,  ""),  "<" : (66, " < "),  ">" : (66, " > "),
          "=" : (68,  "") }
```

Leksička analiza

Program leksičke analize `Get_Sym()` dan u nastavku prepoznavat će simbole u ulaznom nizu znakova, `Get_Ch()`, i vraćat će njihov kod. Pritom će koristiti podatke iz modula `DDH_V.py`. i tablicu identifikatora, `Ident`. Ako se učitani niz znakova ne može prihvatiti kao riječ jezika, dojavljuje se pogreška i prekida daljnje prevođenje.

Leksicka_analiza

```
def Nadji_id () :
  global Ss, Gr, Kraj, Cc, Line, Ch, LN, RZ, \
         BZ, PZ, Ident, C, VC, Ime, Tr, Tr0

  if Ident == {} :
    s = '*' *10
    at = (s, Sym)
    Ident.update({ Sym : at })
    C = 34
    return C

  if Sym in Ident :
    s, Ime = Ident[Sym]
    TS = s[8:]
    S  = s[RZ] +' ' +TS
    C  = 34
    if S in SvId :
      C = SvId [S]
    else :
      if RZ > 0 :
        if s[RZ-1: RZ+1] == '**' and TS == '**'  : C = 43
        if s[RZ-1: RZ+1] == 'C*' and TS <> '**'  : C = 44
        if s[RZ-1: RZ+1] == 'V*' and TS <> '**'  : C = 45
        if s[RZ-1: RZ+1] <> '**' and TS == '**'  : C = 47
      if RZ > 1 :
        if s[RZ-2: RZ+1] == '***' and TS    == '**' : C = 46
        if s[RZ] <> '*'  and TS == '**'             : C = 47
      if RZ > 2 :
        if s[RZ-3: RZ+1] == '****' and TS == '**'   : C = 48
    return C

  else :
    '* Novi identifikator *'
    if RZ == 0 : s = VC +'*' *9
    else       : s = '*' *RZ +VC +'*' *(9 -RZ)
    at = (s, Sym)
    Ident.update ({ Sym : at })
    Ime = Sym
    C = 34
    return  C

def Get_Ch () :
  global Line, Cc, P, Ii, Ch, Kraj
  if Cc == len (Line) -1 and Ii < len (P)-1 :
    Ii   += 1;
    Line = Ucitaj_L (Ii); Cc = -1
  Cc   += 1;  Ch = Line[Cc]
  Kraj = Ii == len(P) -1 and Cc == len (Line) -1
  return

def Get_Sym () :
  global Ss, Gr, Kraj, Cc, Line, Ch, LN, \
         RZ, BZ, PZ, Povrat, Ident, C, Sym, py, PRIKAZ, Tr

  Sym = '';  Get_Ch ()
  while ord (Ch) <= 32 and not Kraj : Get_Ch()
```

```
if Ch in Slova :
  ' Identifikator ili rezervirana riječ '
  while Ch in Slova +Brojke: Sym = Sym +Ch;  Get_Ch ()
  Sym_ = Sym.upper()
  if Sym_ in V : C = V[Sym_][0]   # rezervirana riječ
  elif Sym not in KeyW : C = Nadji_id ()  # identifikator
  else :
    print (Cc -len(Sym) +3)*' '+chr(24), '*' , Sym, 'je rez. riječ Pythona!'
    exec (77)
  Cc -= 1
else :
  if Ch in Brojke : # broj
    while Ch in Brojke: Sym = Sym +Ch;  Get_Ch ()
    C   = 49;  Cc -=  1
  else :
    if Ch == '{' : # komentar
      while Ch <> '}' : Get_Ch ()
      Get_Sym ();  print
    elif Ch == "'" : # niz znakova
      Sym = "'"
      while True :
        Get_Ch ();  Sym += Ch
        if Ch == "'" : break
      C = 65
    else :
      Sym = Ch
      if Ch in ['<', '>', '='] :
        Ch0 = Ch;  Get_Ch ();  Sym = Ch0 +Ch
        if Sym in ['<>', '>=',  '<=', '=='] :
          C = 66;  py += ' '+Sym+' ';  return
        Sym = Ch0; Cc -= 1
      if Sym in Znak : C, tr = Znak[Sym]; py += tr
      else :
        print '*** LEKSIČKA POGREŠKA U REDU', Ii+1
        print 'Sym =', '*'+Sym+'*';  print 'Ch  =', '*'+Ch+'*';  Gr = True
        # exit ()
if PRIKAZ : print Sym, ' '*(20-len(Sym)),
```

Sintaksna analiza i prevođenje

Sintaksna analiza će se izvoditi uz pomoć prepoznavača jezika sa svojstvima koji je upravljan tablicom prijelaza i akcija. Prvo treba definirati tablicu prijelaza iz osnovne sintaksne strukture jezika **DDH** uz istodobno definiranje akcija na određenim mjestima. Akcijama (a bit će to primitivne naredbe ili procedure) će se definirati ili predefinirati svojstva identifikatora, provjeravati kontekstna svojstva pojedinih naredbi i/ili generirati prijevod u ciljnom jeziku.

Znamo da tablica prijelaza ima stanja i prijelaze. U našem je slučaju to 69 prijelaza i *n* stanja. Problem je zajedno prikazati sve prijelaze. Osim toga, radi se o rijetko zaposjednutoj matrici. To su bili razlozi da tablicu prijelaza prikažemo po dijelovima, prateći osnovnu sintaksnu strukturu. Uz kôd narednog stanja dodali smo kôd akcije (0 za praznu akciju). Treba napomenuti da općenito postupak definiranja tablice prijelaza i akcija jezika za koji želite realizirati predprocesor nije "bezgrešan". Sigurno

će već na samom početku biti pogrešaka. Da bi se to umanjilo, treba paralelno provjeravati valjanost definicije izvođenjem programa za sintaksnu analizu (prepoznavanje) jezika sa svojstvima kojeg smo dali u devetom poglavlju druge knjige.

zaglavlje_programa:

	PROGRAM	BEGIN	i_0	;
	1	**2**	**34**	**53**
1	2, 0			
2			3, 1	
3				4, 0
4		5, 2		

blok:

	PRICON	PRIVAR	VIRCON	VIRVAR	GLOCON	GLOVAR	END	i_0	i_d^0	i_c^0	i_v^0	i_r	,	;
	3	**4**	**5**	**6**	**7**	**8**	**18**	**34**	**43**	**44**	**45**	**48**	**52**	**53**
5	6, 3	6, 3	8, 3	8, 3	10, 8	12, 8								
6								7, 0				7, 0		
7							16, 7						6, 0	14, 0
8									9, 0					
9							16, 7						8, 0	14, 0
10										11, 13	11, 13			
11							16, 7						10, 9	14, 12
12											13, 13			
13							16, 7						12, 9	14, 12

	BEGIN	PRICON	PRIVAR	VIRCON	VIRVAR	GLOCON	GLOVAR	ABORT	SKIP	WRITE	IF	DO	B_V^p	B_V^a	I_V^p	I_V^a	i_d^I
	2	**3**	**4**	**5**	**6**	**7**	**8**	**13**	**14**	**15**	**16**	**17**	**37**	**38**	**41**	**42**	**47**
14	5, 0	6, 3	6, 3	8, 3	8, 3	10, 3	12, 3	16, 0	16, 0	63, 8	18, 0	18, 0	55, 0	59, 0	55, 0	59, 0	19, 9

naredba:

	BEGIN	ABORT	SKIP	WRITE	IF	DO	END	FI	OD	B_V^p	B_V^a	I_V^p	I_V^a	i_d^I	.	;	[	!
	2	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**37**	**38**	**41**	**42**	**47**	**50**	**53**	**60**	**64**
15	5,2	16,8	16,8	63,17	18,6	18,6				55,24	59,9	55,24	59,9	19,9				
16							16, 7	16,7	16,7						1,10	15,12		18,51
17																		
18																	50,0	

inicijalizacija:

	VIR	INT	BOOL	ARRAY	END	FALSE	TRUE	I	,	;	+ -	(	)	N	=
	9	**10**	**11**	**12**	**18**	**30**	**31**	**49**	**52**	**53**	**54**	**58**	**59**	**65**	**68**
19	20,0														
20		21,4	33,4												
21				22,5											46,9
22															26,21
23												24,9			
24														25,9	
25													16,9		
26												27,9			
27								29,9			28,9				
28								29,9							
29									30,9				45,22		
30								32,9			31,9				
31								32,9							
32									30,9				45,22		
33				34,5											50,9
34												35,0			
35								36,0							
36													37,0		
37															38,9
38												39,0			
39						40,0	40,0	41,0							
40								41,0							
41									42,9				45,0		
42						43,0	43,0	44,0							
43								44,0							
44									42,9				45,0		
45					16,7					15,12					

cjelobrojni izraz:

	B_C^a	B_V^a	I_C^p	I_C^a	I_V^p	I_V^a	I	+ -	(	input
	36	**38**	**39**	**40**	**41**	**42**	**49**	**54**	**58**	**69**
46	48,9	48,9	47,9	48,9	47,9	48,9	47,9	46,0	46,14	23,9

	END	FI	OD	LOB, HIB	DOM, LIM	LOW, HIGH	AND	OR	.	,	;	+ -	* /	(	)	]	!	relacija
	18	**19**	**20**	**26**	**27**	**28**	**32**	**33**	**50**	**52**	**53**	**54**	**56**	**58**	**59**	**61**	**64**	**66**
47	16,7	16,7	16,7				50,8	50,8		46,9	15,12	46,0	46,0		47,15	15,11	18,51	46,0
48									49,0					46,19				
49				47,9	47,9	47,9												

logički_izraz:

	NOT	FALSE, TRUE	B_C^p	B_C^a	B_V^p	B_V^a	I_C^p	I_C^a	I_V^p	I_V^a	I	+ -	(	input
	29	**30**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**49**	**54**	**58**	**69**
50	50,8	52,8	52,9	53,9	52,9	53,9	47,9	48,9	47,8	48,9	47,9	46,0	50,9	23,0
	END	FI	OD	LOB	DOM	LOW	AND, OR	.	,	;	(	)	]	!
	18	**19**	**20**	**26**	**27**	**28**	**32**	**50**	**52**	**53**	**58**	**59**	**61**	**64**
51											50,9			
52	16,7	16,7	16,7				50,8		50,9	15,0		52,9	15,0	18,51
53								54,0			46,0			
54				47,0	47,0	51,0								

modifikacija polja:

	LOEXT	SHIFT	SWAP	LOPOP	LOREM	B_V^p	B_V^a	I_V^p	I_V^a	:	,	(	=
	21	**22**	**23**	**24**	**25**	**37**	**38**	**41**	**42**	**51**	**52**	**58**	**68**
55											56,25		46,9
56						55,24	57,27	55,24	57,27				
57										58,0			
58				16,8									
59										60,0			
60	62,8	62,8	62,8		16,8							46,28	
61													46,29
62												46,9	

naredba za ispis:

	LOB	DOM	LOW	B_C^p	B_C^a	B_V^p	B_V^a	I_C^p	I_C^a	I_V^p	I_V^a	I	.	:	,	(	)	,
	26	**27**	**28**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**49**	**50**	**51**	**52**	**58**	**59**	**65**
63																64,0		
64				65,9	68,9	65,9	68,9	65,9	68,9	65,9	68,9							65,9
65															64,9		16,30	
66	67,0											67,0						
67															64,9		16,0	
68													69,0			46,19	16,23	
69	65,0	65,0	65,0															

Za implementaciju tablice prijelaza i akcija u Pythonu najbolje je koristiti mapu čiji će elementi imati ključ, par `(q,C)`, `q`-tekuće stanje; `C`-prijelaz, a vrijednost će biti također par `(Q,A)`, `Q`-naredno stanje; `A`-kod akcije. Modul `DDH_TPA.py` sadrži tablicu prijelaza i akcija jezika **DDH**. Program `SAiP` sadrži proceduru za prepoznavanje, `Sint_An()`, izvršavanje akcija, `Akcija(A)` i generiranje linija ciljnog programa.

DDH_TPA.py

```
TPA = {
  ( 1,  1) : ( 2,  0),
  ( 2, 34) : ( 3,  1),
  ( 3, 53) : ( 4,  0),
  ( 4,  2) : ( 5,  2),
  ( 5,  3) : ( 6,  3), ( 5,  4) : ( 6,  3), ( 5,  5) : ( 8,  3),
  ( 5,  6) : ( 8,  3), ( 5,  7) : (10,  8), ( 5,  8) : (12,  8),
  ( 6, 34) : ( 7,  0), ( 6, 48) : ( 7,  0),
  ( 7, 18) : (16,  7), ( 7, 52) : ( 6,  0), ( 7, 53) : (14,  0),
  ( 8, 43) : ( 9,  0),
  ( 9, 18) : (16,  7), ( 9, 52) : ( 8,  0), ( 9, 53) : (14,  0),
  (10, 44) : (11, 13), (10, 45) : (11, 13),
  (11, 18) : (16,  7), (11, 52) : (10,  9), (11, 53) : (14, 12),
  (12, 45) : (13, 13),
  (13, 18) : (16,  7), (13, 52) : (12,  9), (13, 53) : (14, 12),
  (14,  2) : ( 5,  0), (14,  3) : ( 6,  3), (14,  4) : ( 6,  3),
  (14,  5) : ( 8,  3), (14,  6) : ( 8,  3), (14,  7) : (10,  3),
  (14,  8) : (12,  3), (14, 13) : (16,  0), (14, 14) : (16,  0),
  (14, 15) : (63,  8), (14, 16) : (18,  0), (14, 17) : (18,  0),
  (14, 37) : (55,  0), (14, 38) : (59,  0), (14, 41) : (55,  0),
  (14, 42) : (59,  0), (14, 47) : (19,  9),
  (15,  2) : ( 5,  2), (15, 13) : (16,  8), (15, 14) : (16,  8),
  (15, 15) : (63, 17), (15, 16) : (18,  6), (15, 17) : (18,  6),
  (15, 37) : (55, 24), (15, 38) : (59,  9), (15, 41) : (55, 24),
  (15, 42) : (59,  9), (15, 47) : (19,  9),
  (16, 18) : (16,  7), (16, 19) : (16,  7), (16, 20) : (16,  7),
  (16, 50) : ( 1, 10), (16, 53) : (15, 12), (16, 64) : (18, 51),
  # 17 prazno
  (18, 60) : (50,  0),
  (19,  9) : (20,  0),
  (20, 10) : (21,  4), (20, 11) : (33,  4),
  (21, 12) : (22,  5), (21, 68) : (46,  9),
  (22, 68) : (26, 21),
  (23, 58) : (24,  9),
  (24, 65) : (25,  9),
  (25, 59) : (16,  9),
  (26, 58) : (27,  9),
  (27, 49) : (29,  9), (27, 54) : (28,  9), (27, 54) : (28,  9),
  (28, 49) : (29,  9),
  (29, 52) : (30,  9), (29, 59) : (45, 22),
  (30, 49) : (32,  9), (30, 54) : (31,  9), (30, 54) : (31,  9),
  (31, 49) : (32,  9),
  (32, 52) : (30,  9), (32, 59) : (45, 22),
  (33, 12) : (34,  5), (33, 68) : (50,  9),
  (34, 58) : (35,  0),
  (35, 49) : (36,  0),
  (36, 59) : (37,  0),
  (37, 68) : (38,  9),
  (38, 58) : (39,  0),
  (39, 30) : (40,  0), (39, 31) : (40,  0), (39, 49) : (41,  0),
  (40, 49) : (41,  0),
  (41, 52) : (42,  9), (41, 59) : (45,  0),
  (42, 30) : (43,  0), (42, 31) : (43,  0), (42, 49) : (44,  0),
  (43, 49) : (44,  0),
```

```
  (44, 52) : (42,  9), (44, 59) : (45,  0),
  (45, 18) : (16,  7), (45, 53) : (15,  0),
  (46, 36) : (48,  9), (46, 38) : (48,  9), (46, 39) : (47,  9),
  (46, 40) : (48,  9), (46, 41) : (47,  9), (46, 42) : (48,  9),
  (46, 49) : (47,  9), (46, 54) : (46,  0), (46, 58) : (46, 14),
  (46, 69) : (23,  9),
  (47, 18) : (16,  7), (47, 19) : (16,  7), (47, 20) : (16,  7),
  (47, 32) : (50,  8), (47, 33) : (50,  8), (47, 52) : (46,  9),
  (47, 53) : (15, 12), (47, 54) : (46,  0), (47, 56) : (46,  0),
  (47, 59) : (47, 15), (47, 61) : (15, 11), (47, 64) : (18, 51),
  (47, 66) : (46,  0), (47, 68) : (46,  0),
  (48, 50) : (49,  0), (48, 58) : (46, 19),
  (49, 26) : (47,  9), (49, 27) : (47,  9), (49, 28) : (47,  9),
  (50, 29) : (50,  8), (50, 30) : (52,  8), (50, 35) : (52,  9),
  (50, 36) : (53,  9), (50, 37) : (52,  9), (50, 38) : (53,  9),
  (50, 39) : (47,  9), (50, 40) : (48,  9), (50, 41) : (47,  9),
  (50, 42) : (48,  9), (50, 49) : (47,  9), (50, 54) : (46,  0),
  (50, 58) : (50,  9), (50, 69) : (16,  0),
  (51, 58) : (50,  9),
  (52, 18) : (16,  7), (52, 19) : (16,  7), (52, 20) : (16,  7),
  (52, 32) : (50,  8), (52, 52) : (50,  9), (52, 53) : (15,  0),
  (52, 59) : (52,  9), (52, 61) : (15,  0), (52, 64) : (18, 51),
  (53, 50) : (54,  0), (53, 58) : (46,  0),
  (54, 26) : (47,  0), (54, 27) : (47,  0), (54, 28) : (51,  0),
  (55, 52) : (56, 25), (55, 68) : (46,  9),
  (56, 37) : (55, 24), (56, 38) : (57, 27), (56, 41) : (55, 24),
  (56, 42) : (57, 27),
  (57, 51) : (58,  0),
  (58, 24) : (16,  8),
  (59, 51) : (60,  0),
  (60, 21) : (62,  8), (60, 22) : (62,  8), (60, 23) : (62,  8),
  (60, 25) : (16,  8), (60, 58) : (46, 28),
  (61, 68) : (46, 29),
  (62, 58) : (46,  9),
  (63, 58) : (64,  0),
  (64, 35) : (65,  9), (64, 36) : (68,  9), (64, 37) : (65,  9),
  (64, 38) : (68,  9), (64, 39) : (65,  9), (64, 40) : (68,  9),
  (64, 41) : (65,  9), (64, 42) : (68,  9), (64, 65) : (65,  9),
  (65, 51) : (66,  0), (65, 52) : (64,  9), (65, 59) : (16, 30),
  (66, 26) : (67,  0),
  (67, 52) : (64,  9), (67, 59) : (16,  0),
  (68, 50) : (69,  0), (68, 58) : (46, 19), (68, 59) : (16, 23)   }
```

SAiP

```
def Trans (red): # Generiranje reda prijevoda
  global PY, py, SP
  if red <> '' and red <> len(red) *' ' : PY.append ([red])
  py = ' '*SP

def Sint_An () : # Sintaksna analiza (Prepoznavanje)
  global Ss, Gr, Kraj, Cc, Line, Ch, LN, RZ, BB, BZ, PZ, \
         Povrat, Ident, C, PY, VC, Ime, Sym, PRIKAZ, Post, Tr, Tr0

  def Error (CE) :
    i = Cc -len(Sym) +1
```

```
    if PRIKAZ :
      print 'Ss =', Ss, 'S =', S, 'C =', C
      print '*** SINTAKSNA POGREŠKA br. '+str(CE) +' u redu '+str(Ii),
      print 'na poziciji '+str(i) +' simbol '+Sym
    else :
      print ' '*(i-1), '*** SINTAKSNA POGREŠKA '
    Gr = True

  def Akcija (A) : # Izvršavanje akcija
    global Ss, RZ, BB, BZ, EOS, VC, Ime, PY, py, SP, \
           Ime, Sym, Kraj, KD, PZ, Povrat, Post, Tr, Tr0

    def Gen_pov (Q, Ch): global Povrat, PZ; Povrat = [Q] +Povrat; PZ = [Ch] +PZ
    def Pop_pov (): global Povrat, PZ; Povrat = Povrat[1:]; PZ = PZ[1:]

    if   A == -1 : return True
    elif A ==  0 : pass
    elif A ==  1 : # -> zaglavlje programa
      Trans ('# -*- coding: cp1250 -*-');  Trans ('# PROGRAM '+Sym);  Trans ('')
      Trans ('from arr import *'       );  Trans (''              );  Trans ('')
    elif A ==  2 : # novi blok
      if RZ < 7 :
        RZ += 1;  BB += 1
        if BB > 0 :
          Post = V[Sym.upper()][2]; Post = Post.replace ('*', str(BB))
          py += Post;  SP += 2;  Trans (py)
      else      : Error (99);  return False
      EOS = 'E' +EOS
    elif A == 3 : # PRIVAR, PRICON, VIRVAR, VIRCON, GLOVAR ili GLOCON
      VC = Sym[3].upper(); Akcija (8)
    elif A == 4 : # osnovni tip
      s, Ime = Ident[Ime];  s = s[ :8] +Sym[0].upper() +'p'
      Ident.update ( {Ime : (s, Ime)} )
    elif A == 5 : # polje
      s, Ime = Ident[Ime];  s = s[ :9] +'a';
      Ident.update ( {Ime : (s, Ime)} )
    elif A ==  6 : # IF ili DO
      py += Tr0; Tr0 = '';  EOS = Sym[1] +EOS;  Trans (py);  py += V[Sym][2]
      if Sym.upper() == 'DO' :
        SP += 2; Trans (py);  py += V['IF'][2]
    elif A ==  7 : # END, FI ili OD
      py += Tr0;  Tr0 = '';  Sym = Sym.upper()
      if Sym[0] == EOS[0] : EOS = EOS [1:]
      else                : Error (10);  return False
      if Sym == 'END' :
        if RZ >= 0 :
          RZ -= 1
          if BB > 0 :
            SP -= 2; Trans(py)
            Post = V[Sym.upper()][2]; Post = Post.replace ('*', str(BB))
            py += Post;  Trans (py);  BB -= 1
        else      : Error (99);  return False
      else :
        SP -= 2; Trans(py)
        if Sym == 'OD' : SP -= 2
        py += V[Sym][2]; Trans(py)
```

```
  elif A ==  8 : # generiranje prijevoda
    py += V [Sym.upper()][2]
    if Sym.upper() in ['LOEXT','HIEXT','SHIFT','SWAP'] : Gen_pov(16,')'); Tr = ')'
  elif A ==  9 :  py += Sym  # niz znakova
  elif A == 10 : # KRAJ PROGRAMA
    Kraj = EOS == ''
    if Kraj : Trans (py[:-1])
    else    : Error (9);  return False
  elif A == 11 : py += Tr; Tr = ''; SP += 2;  Trans (py)  # Alternativa
  elif A == 12 : # kraj naredbe -> emitiraj prijevod
    py += Tr0; Tr0 = ''; Trans (py)
  elif A == 13 : # dodaj definiranost varijable u novom bloku
    s, Ime = Ident[Ime]; s = s[:RZ] +VC +s[RZ+1:]; Ident.update ({Ime : (s, Ime)})
    py += Sym
  elif A == 14 : py += Sym;  BZ += 1  # otvorena zagrada
  elif A == 15 : # zatvorena zagrada
    if BZ > 0 : BZ -= 1; py += Sym
    else      :
      if PZ[0] == ')' :
        if Tr <> '' : py += Tr;  Tr = ''
        else        : py += Tr0; Tr0 = ''
        Ss = Povrat[0]; Pop_pov()
        if Ss == 61 : py = py[:-1] +', '
      else          : Error (2);  return False
  elif A == 16 : Gen_pov (47, ')')  # indeks polja
  elif A == 17 : # generiranje prijevoda
    py += V [Sym.upper()][2]
    Gen_pov (16, ')');  Tr0 = ''
    if Sym.upper() == 'WRITE' : Tr0 = ','
  elif A == 19 : # referiranje na komponentu polja
    py += ".ind (";  Gen_pov (47, ')'); Tr = ')'
  elif A == 20 : Ime = Sym  # ime identifikatora
  elif A == 21 : py += Sym +"array ('int', "  # inicijalizacija polja
  elif A == 22 : py += ", ))"; Trans (py)
  elif A == 23 : py += ".val"; Trans (py)  # ispis polja
  elif A == 24 : # dodjeljivanje
    py += Sym;  s, Ime = Ident[Sym];  KD += s[8]
  elif A == 25 : py += Sym; # konkurentno pridruživanje
  elif A == 26 : pass # =
  elif A == 27 : py = py[:-1] +'=' +Sym # konkurentno pridruživanje
  elif A == 28 : py += 'alt ('; Gen_pov (61, ')'); Tr0 = ')' # (...)
  elif A == 29 : Tr0 = ')'  # (...)
  elif A == 30 : pass # izbaci )
  elif A == 41 : Gen_pov (16, ')'); BZ = 0  #
  elif A == 51 : py += Tr;  Tr = '';  SP -= 2;  Trans (py); py += 'elif '
  else         : print '* NIJE DEFINIRANA AKCIJA BR.', A; return False
  return True

S  = Ss;  Qt = (S, C)
if PRIKAZ : print Qt,
if Qt in TPA :
  Ss, A = TPA [Qt]
  if PRIKAZ  : print '->', Ss, A
  if Ss == 0 : Error (1);  return
  Ok = Akcija (A)
else : print 'GREŠKA '; exit (99)    #, S, C, Sym, Ss
return
```

Generiranje prijevoda (ciljnog programa u Pythonu) počinje poslije učitavanja imena programa. Generira se zaglavlje:

```
# -*- coding: cp1250 -*-
# PROGRAM <ime_programa>;

from arr import *
```

Modul `arr.py` sadrži klasu `array` u kojoj su definirani atributi, procedure i funkcije varijabli sa strukturom polja u jeziku **DDH**:

arr.py

```
class array :

  def __init__ (self, t, x):
    self.tip = t
    self.val = []
    self.lob = x[0]
    x        = x[1:]
    if len (x) > 0 :
      for i in range (0, len(x)) : self.val.append(x[i])
    self.dom = len (self.val)
    self.hib = self.lob +self.dom -1

  def in_dom (self, i) :
    d  = -self.lob;  i += d;  return d <= i <= self.hib +d

  def ind (self, i) :
    y  = 'NIJE DEFINIRANO'
    if self.in_dom (i) : i -= self.lob;  y = self.val[i]
    return y

  def alt (self, i, x) :
    if self.in_dom (i) : i -= self.lob;  self.val[i] = x
    else : exit

  def loext (self, x) : # Ax : loext (x)
    self.val = [x] +self.val;  self.dom += 1;  self.lob -= 1;  self.low = x

  def hiext (self, x) : # Ax : hiext (x)
    self.val.append (x);  self.dom += 1;  self.hib += 1;  self.high = x

  def low (self) :   # Ax.low
    if self.dom > 0 : self.low  = self.val[self.lob]
    else            : self.low  = 'nije definirano'

  def high (self) :  # Ax.high
    if self.dom > 0 : self.high = self.val[-1]
    else            : self.high = 'nije definirano';  exit (99)

  def lorem (self) : # Ax : lorem
    if self.dom > 0 : self.val  = self.val[1:];  self.dom -= 1;  self.lob += 1
    else            : print 'nije definirana operacija LOREM'; exit (99)
```

```
def hirem (self) : # Ax : hirem
  if self.dom > 0 : self.val  = self.val[:-1]; self.dom -= 1;  self.hib -= 1
  else            : print 'nije definirana operacija HIREM'; exit (99)

def lopop (self) : # x, Ax : lopop -> x = Ax.low;  Ax : lorem
  if self.dom > 0 : x = self.low;  self.lorem();  return x
  else            : print 'nije definirana operacija LOPOP';  exit (99)

def hipop (self) : # x, Ax : hipop -> x = Ax.high;  Ax : hirem
  if self.dom > 0 : x = self.high;  self.hirem();  return x
  else            : print 'nije definirana operacija HIPOP';  exit (99)

def swap (self, i, j) : # Ax : swap (i, j)
  if self.in_dom (i) and self.in_dom (j) :
    x            = self.ind(i);         y            = self.ind(j);
    i0           = i -self.lob;         j0           = j -self.lob;
    self.val[i0] = y;                   self.val[j0] = x
    self.low     = self.val[self.lob];  self.high    = self.val[-1]
  else           : print 'indeks polja van domene';  exit (99)
```

9.5 PREDPROCESOR

Slijedi kompletan kôd predprocesora. Da bismo izbjegli ponavljanja, komentarom smo označili mjesta gdje treba kopirati dijelove programa leksičke i sintaksne analize s prevođenjem.

DDH.py

```
# PROGRAM DDH; { PREDPROCESOR jezika DDH }

from gramatika import *
from DDH_V     import *
from DDH_TPA   import *

def Init () : # ' Inicijalizacija varijabli '
  global Ss, Gr, Kraj, Cc, Line, Ch, LN, RZ, BB, BZ, PZ, Povrat, \
         Ident, PY, EOS, VC, py, SP, KD, Post, Tr, Tr0

  Ss = 1;    Gr     = False;  Kraj = False;  Cc = -1;  Line = ''
  Ch = ' ';  LN     = 0;      RZ   = -1;     BZ =  0;  BB   = -1
  PZ = ['']; Povrat = [];     Ident = {};    PY = [];  EOS  = ''
  VC = '*';  py     = '';     SP    = 0;     KD = '';  Post = ''
  Tr = '';   Tr0    = ''

def Ucitaj_PROG (Poruka, Tip): # Učitavanje programa
  G = fileopenbox(Poruka, None, Tip, '*')
  if G == None: G = ''
  if os.path.exists(G):
    P = []
    for line in open (G, 'r') : P.append (line[:-1])
    Ok = True
    if G.rfind('\\') > 0: G = G[G.rfind('\\')+1:]
    PRIKAZ = raw_input ('Prikaz prijelaza tijekom prepoznavanja (D/N)? ')
    PRIKAZ = PRIKAZ.upper()[0] == 'D'
  else :  Ok = False
  return G, Ok, P, PRIKAZ
```

```
def Ucitaj_L (i) :
  red = P[i] + ' '
  print "%02d" %(i+1), P[i]
  return red

### LEKSIČKA ANALIZA

### SINTAKSNA ANALIZA

def Upamti_PROG (P, Ime): # Pamćenje programa
  for L in P : print L[0]
  Ime = Ime.upper().replace ('.TXT', '')
  Ime = 'DDH_' +Ime.replace ('.DDH', '') +'.py'

  R = ''
  for i in range (len(P)):
    R += P[i][0] +'\n'
  open (Ime, 'w').write (R)

  print
  print 'Program je upamćen pod imenom', Ime, '\n'
  try :
    execfile (Ime)
  except :
    print "LOGIČKA POGREŠKA ILI POGREŠKA U PRIJEVODU"

while True:
  Ime_PROG, Ok, P, PRIKAZ = Ucitaj_PROG ('***', '*.DDH;*.TXT')
  if Ok :
    print Ime_PROG, '\n'
    # for i in range (len(P)) : print "%02d" %(i+1), P[i]
  else : print '*** ne postoji program', Ime_PROG;  exit (99)
  print;  Init();  Ii = 0;  Cc = -1;  Line = Ucitaj_L (Ii)
  while not Gr and not Kraj:
    Get_Sym()
    if Gr : print '*** LEKSIČKA POGREŠKA U REDU', Ii+1
    else  :
      if Sym <> '' : Sint_An ()
  if not Gr :
    print '\n', 'Nema pogrešaka', '\n\n'
    Upamti_PROG (PY, Ime_PROG)
  else : print "*** SINTAKSNA POGREŠKA U PROGRAMU"
  print '\n'
  Ponovi = raw_input ('UČITAVAM DRUGI PROGRAM (D/N)? ')
  if Ponovi.upper()[0] <> 'D' : break
```

9.6 PRIMJERI

Ulazni program u jeziku DDH mora biti napisan i zapamćen u tekstualnoj datoteci, s ekstenzijom DDH ili TXT. Poslije njegova odabira iz menua možemo uključiti ili ne prikaz sintaksne analize. Tada će biti prikazan svaki korak leksičke i sintaksne analize. To će nam biti važno ako želimo razumjeti što se događa u fazi projektiranja predprocesora, definiranja tablice prijelaza i akcija ili pri proširenju postojećeg predprocesora.

Procesor je jednoprolazni. Pojavom bilo koje pogreške (leksičke ili sintaksne) prekida se daljnje prevođenje. Ako nije bilo pogrešaka, bit će generiran program u izlaznom jeziku (Pythonu) s imenom

```
DDH_<ulazno_ime_bez_ekstenzije>.py
```

Potom će generirani program biti učitan i izvršen. Na primjer, program za izračunavanje *n*-te permutacije, `NPERM.DDH` bit će preveden u `DDH_NPERM.py`.

NPERM.DDH

```
PROGRAM Nperm;

BEGIN
  PRIVAR C, N;
  C VIR int array
    = (1, 0,1,2,3,4,5,6,7,8,9);
  N VIR int =
    input ('Permutacija ( >');
  WriteLn (0); WriteLn (C, '\n');

  BEGIN
    GLOVAR C, N;  PRIVAR S, Kfac, i, j, k;
    S VIR int = 0;  Kfac VIR int = 1;
    k VIR int = 1;  i    VIR int = 0;
    j VIR int = 1;
    DO [ k <> C.hib ]
      Kfac = Kfac*(k+1); k = k+1
      OD;
    DO [ S <> N ]
      DO [ N < S+Kfac ]
        Kfac = Kfac/k; k = k-1
        OD;
      i = C.hib-k; j = i+1;
      DO [ S+Kfac <= N ]
        S = S+Kfac; C : swap (i,j);
        j = j+1
        OD
      OD;
    WriteLn (N); Writeln (C)
    END
  END.
```

DDH_NPERM.py

```
# -*- coding: cp1250 -*-
# PROGRAM Nperm
from arr import *
C=array ('int', (1,0,1,2,3,4,5,6,7,8,9, ))
N=input('Permutacija (1 do 3628799)? ')
print C.val
def BEGIN_1 () :
  global C,N
  S=0
  Kfac=1
  k=1
  i=0
  j=1
  while True :
    if k <> C.hib :
      Kfac=Kfac*(k+1)
      k=k+1
    else : break
  while True :
    if S <> N :
      while True :
        if N < S+Kfac :
          Kfac=Kfac/k
          k=k-1
        else : break
      i=C.hib-k
      j=i+1
      while True :
        if S+Kfac <= N :
          S=S+Kfac
          C.swap(i,j)
          j=j+1
        else : break
    else : break
  print N
  print C.val
BEGIN 1 ()
```

```
Program je upamćen pod imenom DDH_NPERM.py

Permutacija (1 do 3628799)? 3628799
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3628799
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Najkraći putovi u grafu

Zadan je graf s *n* čvorova i udaljenostima između njih. U programu je to matrica susjedstva, M. Potrebno je odrediti najkraće putove od svakog do nekog zadanog čvora.

```
PROGRAM Putovi;

BEGIN
  PRICON q, N, M; PRIVAR j, Ud, Slj, Pre;
  N    VIR int = 5; { broj čvorova }
  q    VIR int = 5; { početni čvor }
  M    VIR int array = (1,   0,   1, 100,   4,   5,
                            1,   0,   2, 100,   8,
                          100,   2,   0,   3, 100,
                            4, 100,   3,   0,   6,
                            5,   8, 100,   6,   0 );
  Pre VIR int array = ( 1 );
  Ud  VIR int array = ( 1 );
  Slj VIR int array = ( 1 );
  j   VIR int       = 1;
  DO [ j <= N ] Ud : hiext (100000); Pre : hiext (0); j = j +1 OD;
  Slj : hiext (q); Ud : (q) = 0; j = 1;
  DO [ Slj.hib >= 1 ]
     BEGIN
       GLOCON N, M; GLOVAR j, Ud, Slj, Pre; PRIVAR LL, KK;
       LL VIR int = Slj.high;
       KK VIR int = (LL -1)*N +j;
       IF [ Ud(j) –Ud(LL) >  M(KK) ]
          Ud:  (j) = Ud(LL) +M(KK); Slj: hiext (j); Pre: (j) = LL; j = 1
         ![ Ud(j) –Ud(LL) <= M(KK) ]
          IF [ j <  N ] j = j +1
            ![ j == N ] Slj: hirem; j = 1
            FI
         FI
       END
     OD;
  j = 1;
  DO [j <= Pre.dom ] WriteLn ('(', j, ') ->', Pre(j)); j = j+1 OD
END.
```

```
# -*- coding: cp1250 -*-
# PROGRAM Putovi
from arr import *
N=5
q=5
M=array ('int', (1,0,1,100,4,5,1,0,2,100,8,100,2,0,3,100,4,100,3,0,6,5,8,100,6,0, ))
Pre=array ('int', (1, ))
Ud=array ('int', (1, ))
Slj=array ('int', (1, ))
j=1
while True :
  if j <= N :
    Ud.hiext (100000)
    Pre.hiext (0)
    j=j+1
  else : break
Slj.hiext (q)
```

```
Ud.alt (q, 0)
j=1
while True :
  if Slj.hib >= 1 :
    def BEGIN_1 () :
      global N,M
      global j,Ud,Slj,Pre
      LL=Slj.high
      KK=(LL-1)*N+j
      if Ud.ind (j)-Ud.ind (LL) > M.ind (KK) :
        Ud.alt (j, Ud.ind (LL)+M.ind (KK))
        Slj.hiext (j)
        Pre.alt (j, LL)
        j=1
      elif Ud.ind (j)-Ud.ind (LL) <= M.ind (KK) :
        if j < N :
          j=j+1
        elif j == N :
          Slj.hirem ()
          j=1
    BEGIN_1 ()
  else : break
j=1
while True :
  if j <= Pre.dom :
    print '(',j,') ->',Pre.ind (j)
    j=j+1
  else : break
```

```
Program je upamćen pod imenom DDH_PUTOVI.py
```

```
( 1 ) -> 5
( 2 ) -> 1
( 3 ) -> 2
( 4 ) -> 3
( 5 ) -> 0
```

Interpretacija rezultata programa. Brojevi u prvom stupcu označuju čvorove, a u drugom vezu na sljedeći čvor. Na primjer, najkraći put od čvora (3) do (5) jest:

```
( 3 ) -> ( 2 ) -> ( 1 ) -> ( 5 )
```

U nastavku pogledajmo kako će naš predprocesor prevesti nekoliko programa napisanih u jeziku **DDH**, od kojih su neki dani u prethodnom poglavlju.

```
Prikaz prijelaza tijekom prepoznavanja (D/N)? n
Euclid0.txt

01    PROGRAM Euclid;
02    BEGIN
03      PRIVAR x0, y0, x, y;
04      x0 VIR int = input ('Prvi broj  ');  y0 VIR int = input ('Drugi broj ');
05      x  VIR int = xO;                     y  VIR int = y0;
GREŠKA
```

Euclidov algoritam

```
Prikaz prijelaza tijekom prepoznavanja (D/N)? d

01    PROGRAM Euclid;
PROGRAM               (1, 1) -> 2 0
Euclid                (2, 34) -> 3 1
;                     (3, 53) -> 4 0
02    BEGIN
BEGIN                 (4, 2) -> 5 2
03      PRIVAR x0, y0, x, y;
PRIVAR                (5, 4) -> 6 3
x0                    (6, 34) -> 7 0
,                     (7, 52) -> 6 0
y0                    (6, 34) -> 7 0
,                     (7, 52) -> 6 0
x                     (6, 34) -> 7 0
,                     (7, 52) -> 6 0
y                     (6, 34) -> 7 0
;                     (7, 53) -> 14 0
04      x0 VIR int = input ('Prvi broj  ');  y0 VIR int = input ('Drugi broj ');
x0                    (14, 47) -> 19 9
VIR                   (19, 9) -> 20 0
int                   (20, 10) -> 21 4
=                     (21, 68) -> 46 9
input                 (46, 69) -> 23 9
(                     (23, 58) -> 24 9
'Prvi broj  '         (24, 65) -> 25 9
)                     (25, 59) -> 16 9
;                     (16, 53) -> 15 12
y0                    (15, 47) -> 19 9
VIR                   (19, 9) -> 20 0
int                   (20, 10) -> 21 4
=                     (21, 68) -> 46 9
input                 (46, 69) -> 23 9
(                     (23, 58) -> 24 9
'Drugi broj '         (24, 65) -> 25 9
)                     (25, 59) -> 16 9
;                     (16, 53) -> 15 12
05      x  VIR int = x0;                    y  VIR int = y0;
x                     (15, 47) -> 19 9
VIR                   (19, 9) -> 20 0
int                   (20, 10) -> 21 4
=                     (21, 68) -> 46 9
x0                    (46, 41) -> 47 9
;                     (47, 53) -> 15 12
y                     (15, 47) -> 19 9
VIR                   (19, 9) -> 20 0
int                   (20, 10) -> 21 4
=                     (21, 68) -> 46 9
y0                    (46, 41) -> 47 9
;                     (47, 53) -> 15 12
06      IF [x0 > 0 and y0 > 0]
IF                    (15, 16) -> 18 6
[                     (18, 60) -> 50 0
x0                    (50, 41) -> 47 9
>                     (47, 66) -> 46 0
```

```
0                    (46, 49) -> 47 9
and                  (47, 32) -> 50 8
y0                   (50, 41) -> 47 9
>                    (47, 66) -> 46 0
0                    (46, 49) -> 47 9
]                    (47, 61) -> 15 11
07         DO [x > y] x = x -y
DO                   (15, 17) -> 18 6
[                    (18, 60) -> 50 0
x                    (50, 41) -> 47 9
>                    (47, 66) -> 46 0
y                    (46, 41) -> 47 9
]                    (47, 61) -> 15 11
x                    (15, 41) -> 55 24
=                    (55, 68) -> 46 9
x                    (46, 41) -> 47 9
-                    (47, 54) -> 46 0
y                    (46, 41) -> 47 9
08           ![y > x] y = y -x
!                    (47, 64) -> 18 51
[                    (18, 60) -> 50 0
y                    (50, 41) -> 47 9
>                    (47, 66) -> 46 0
x                    (46, 41) -> 47 9
]                    (47, 61) -> 15 11
y                    (15, 41) -> 55 24
=                    (55, 68) -> 46 9
y                    (46, 41) -> 47 9
-                    (47, 54) -> 46 0
x                    (46, 41) -> 47 9
09           OD;
OD                   (47, 20) -> 16 7
;                    (16, 53) -> 15 12
10         WriteLN ('NZD(', x0, ', ', y0, ') = ', x)
WriteLN              (15, 15) -> 63 17
(                    (63, 58) -> 64 0
'NZD('               (64, 65) -> 65 9
,                    (65, 52) -> 64 9
x0                   (64, 41) -> 65 9
,                    (65, 52) -> 64 9
', '                 (64, 65) -> 65 9
,                    (65, 52) -> 64 9
y0                   (64, 41) -> 65 9
,                    (65, 52) -> 64 9
') = '               (64, 65) -> 65 9
,                    (65, 52) -> 64 9
x                    (64, 41) -> 65 9
)                    (65, 59) -> 16 30
11        ![x0 <= 0 or y0 <= 0] Write ('POGREŠKA')
!                    (16, 64) -> 18 51
[                    (18, 60) -> 50 0
x0                   (50, 41) -> 47 9
(47, 66) -> 46 0
0                    (46, 49) -> 47 9
or                   (47, 33) -> 50 8
y0                   (50, 41) -> 47 9
(47, 66) -> 46 0
```

```
0                    (46, 49) -> 47 9
]                    (47, 61) -> 15 11
Write                (15, 15) -> 63 17
(                    (63, 58) -> 64 0
'POGREŠKA'           (64, 65) -> 65 9
)                    (65, 59) -> 16 30
12        FI
FI                   (16, 19) -> 16 7
13      END.
END                  (16, 18) -> 16 7
.                    (16, 50) -> 1 10


# -*- coding: cp1250 -*-
# PROGRAM Euclid
from arr import *
x0=input('Prvi broj  ')
y0=input('Drugi broj ')
x=x0
y=y0
if x0 > 0 and y0 > 0 :
  while True :
    if x > y :
      x=x-y
    elif y > x :
      y=y-x
    else : break
  print 'NZD(',x0,', ',y0,') = ',x
elif x0 <= 0 or y0 <= 0 :
  print 'POGREŠKA',

Program je upamćen pod imenom DDH_EUCLID.py

Prvi broj  444
Drugi broj 111
NZD( 444 ,  111 ) =  111
```

Hammingov niz

```
01 PROGRAM Hamming; { Hammingov niz }
02

03 BEGIN
04   PRIVAR AQ, i2, i3, i5, x2, x3, x5, N, i;
05     AQ VIR int array = (1, 1);
06     i2 VIR int = 1;  i3 VIR int = 1;
07     i5 VIR int = 1;  x2 VIR int = 2;
08     x3 VIR int = 3;  x5 VIR int = 5;
09     N  VIR int = 30;
10   DO [AQ.dom <> N]
11     IF [x3 >= x2 and x2 <= x5] AQ: hiext (x2)
12       ![x2 >= x3 and x3 <= x5] AQ: hiext (x3)
13       ![x2 >= x5 and x5 <= x3] AQ: hiext (x5)
14       FI;
15     DO [x2 <= AQ.high] i2 = i2 +1; x2 = 2 *AQ(i2) OD;
```

```
16     DO [x3 <= AQ.high] i3 = i3 +1; x3 = 3 *AQ(i3) OD;
17     DO [x5 <= AQ.high] i5 = i5 +1; x5 = 5 *AQ(i5) OD
18     OD;
19   i VIR int = AQ.lob;
20   Writeln ('Prvih ', N, ' članova Hammingova niza:');
21   DO [i <= AQ.hib] Write (AQ(i)); i = i +1 OD
22   END.
```

```
# -*- coding: cp1250 -*-
# PROGRAM Hamming
from arr import *
AQ=array ('int', (1,1, ))
i2=1
i3=1
i5=1
x2=2
x3=3
x5=5
N=30
while True :
  if AQ.dom <> N :
    if x3 >= x2 and x2 <= x5 :
      AQ.hiext (x2)
    elif x2 >= x3 and x3 <= x5 :
      AQ.hiext (x3)
    elif x2 >= x5 and x5 <= x3 :
      AQ.hiext (x5)
    while True :
      if x2 <= AQ.high :
        i2=i2+1
        x2=2*AQ.ind (i2)
      else : break
    while True :
      if x3 <= AQ.high :
        i3=i3+1
        x3=3*AQ.ind (i3)
      else : break
    while True :
      if x5 <= AQ.high :
        i5=i5+1
        x5=5*AQ.ind (i5)
      else : break
  else : break
i=AQ.lob

print 'Prvih ',N,' članova Hammingova niza:'
while True :
  if i <= AQ.hib :
    print AQ.ind (i),
    i=i+1
  else : break
```

```
Program je upamćen pod imenom DDH_HAMMING.py

Prvih  30  članova Hammingova niza:
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 50 54 60 64 72 75 80
```

Eratostenovo sito

```
01 PROGRAM Eratos;
02
03 BEGIN
04   PRIVAR P, Q, X, S, N, i;
05   P VIR int =   2;  Q VIR int = 0;  N VIR int = 100;  X VIR int = 0;
06   i VIR int =   0;
07   S VIR int array = (1);
08   DO [ i <> N ] i = i+1; S : hiext (i) OD;
09   BEGIN
10     GLOVAR P, Q, X, S; GLOCON N; PRIVAR m, r;
11     m VIR int = 0;  r VIR int = 0;
12     DO [ P*P <= N ] Q = P;
13       DO [ P*Q <= N ] X = P*Q;
14         DO [ X <= N ] S : (X) = 0; X = P*X OD;
15         m = Q+1;
16         DO [ S(m) == 0 ] m = m+1 OD;
17         Q = S(m)
18         OD;
19       r = P+1;
20       DO [ S(r) == 0 ] r = r+1 OD;
21       P = S(r)
22       OD
23     END;
24   i = 2;
25   DO [ i <= N ]
26     IF [ S(i) <> 0 ] Write (S(i))
27       ![ S(i) == 0 ] SKIP
28       FI;
29     i = i+1
30     OD
31   END.
```

```
# -*- coding: cp1250 -*-
# PROGRAM Eratos
from arr import *
P=2
Q=0
N=100
X=0
i=0
S=array ('int', (1, ))
while True :
  if i <> N :
    i=i+1
    S.hiext (i)
  else : break
def BEGIN_1 () :
  global P,Q,X,S
  global N
  m=0
  r=0
  while True :
    if P*P <= N :
      Q=P
```

```
        while True :
          if P*Q <= N :
            X=P*Q
            while True :
              if X <= N :
                S.alt (X, 0)
                X=P*X
              else : break
            m=Q+1
            while True :
              if S.ind (m) == 0 :
                m=m+1
              else : break
            Q=S.ind (m)
          else : break
        r=P+1
        while True :
          if S.ind (r) == 0 :
            r=r+1
          else : break
        P=S.ind (r)
      else : break
BEGIN_1 ()
i=2
while True :
  if i <= N :
    if S.ind (i) <> 0 :
      print S.ind (i),
    elif S.ind (i) == 0 :
      pass
    i=i+1
  else : break

Program je upamćen pod imenom DDH_ERATOS.py

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Zadaci

*1) Proširite jezik **DDH** uvodeći znakovni tip (char). Vrijednosti toga tipa neka budu znakovi iz ASCII skupa znakova, kodna stranica 852.*

2) Proširite predprocesor jezika DDH uvođenjem optimalizacije koda. Na primjer, ako DO petlja sadrži samo jedan zaklonjeni skup:

```
DO [ uvjet ] naredbe OD
```

što smo prevodili u:

```
while True :
  if uvjet :
    naredbe
  else : break
```

poslije optimalizacije bilo bi:

```
while uvjet :
  naredbe
```

3) *Proširite predprocesor jezika DDH uvođenjem nedeterminističke iteracije (kako ju je Dijkstra definirao).*

4) *Proširite jezik DDH i njegov predprocesor tako da prihvaća složenu inicijalizaciju (konkurentno pridruživanje) primitivnih varijabli i polja prema sintaksi:*

```
inic     : v = ulaz | a = inic_a | v, inic, ulaz | a, inic, inic_a
  v      : ime VIR tip
  a      : ime VIR tip ARRAY
  inic_a : (cijeli_broj { , konstanta })
```

gdje su ***ulaz, ime, tip, cijeli_broj*** *i* ***konstanta*** *definirani u osnovnoj sintaksnoj strukturi jezika DDH. Na primjer, pravilno su napisane sljedeće naredbe:*

```
X VIR int, Y VIR int = input ('Zadaj X '), 10
A VIR int, B VIR bool, C VIR int ARRAY = Y*Y, X < 10, (0, 1,2,1)
```

5) *Proširite jezik PL/0 naredbama za ulaz i izlaz i projektirajte predprocesor novodobivenog jezika.*

Pogovor

Došli smo do kraja treće knjige iz niza FORMALNI JEZICI I PREVODIOCI. U prvoj smo knjizi uveli osnovne pojmove i bavili se generiranjem jezika. Opisali smo tri formalizma za to: regularne izraze, gramatike i generatore (automate). U drugoj smo se knjizi bavili problemom sintaksne analize: parsiranjem i prepoznavanjem.

U ovoj smo, trećoj knjizi, obradili teme koje se odnose na problem prevođenja. Prvo smo opisali prevođenje linearnih i beskontekstnih jezika, potom prevođenje kontekstnih jezika od kojih su posebno važni jezici za programiranje. Definirali smo jezike za programiranje i opisali faze prevođenja. Izabrali smo dvije klase prevodilaca: interpretatore i predprocesore.

Mislim da će prikazani postupak definiranja jezika za programiranje i njihovo svođenje na problem definiranja jezika sa svojstvima, uveden u devetom poglavlju prve knjige, [Dov2012a], potom problem sintaksne analize definiran u devetom poglavlju druge knjige, [Dov2012b] i detaljna primjena u definiranju, sintaksnoj analizi (prepoznavanju) i prevođenju, dana na primjeru jezika **DDH** u osmom i devetom poglavlju ove knjige, biti motiv i veliki izazov za daljnje dogradnje primjene u dizajnu i implementaciji predprocesora mnogih drugih "mini" jezika. Na primjer, vjerujem da bi dizajn predprocesora jezika **PL/0** bio prilično trivijalan.

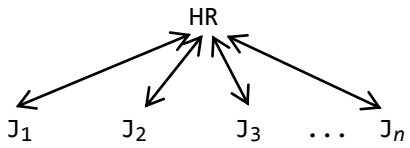
Što dalje? Mislim da se nadgradnja može nastaviti u dva pravca:

- projektiranju kompilatora i
- obradi prirodnih jezika

Oni koji bi se željeli baviti projektiranjem netrivijalnih kompilatora (kao što je, na primjer, za jezike C i Pascal) morali bi imati dodatna znanja iz

- jezika za programiranje (prijenos parametara, doseg varijabli, alociranje memorije itd.)
- algoritama i struktura podataka (raspršeno ("hash") sortiranje i pretraživanje, dinamičko programiranje itd.)
- izabranih poglavlja diskretne matematike (grafovi, stabla itd.)
- arhitekture računala (asemblerskih jezika)
- operacijskih sustava itd.

Ono što želim posebno istaknuti jest moguće primjene danog postupka definiranja, prepoznavanja i prevođenja jezika sa svojstvima u obradi prirodnih jezika: u definiranju njihove leksičke strukture (uvođenjem svojstava), dizajniranju programa leksičke analize, definiranju prepoznavača i, napose, u definiranju prevodioca iz jednog prirodnog jezika u drugi. U početku to može biti prevodilac "sličnih" jezika, kao što su na primjer hrvatski i srpski jezik, potom ga treba proširivati na druge slavenske jezika i postupno na neke neslavenske jezike (engleski, francuski, njemački itd). Uvijek bi trebalo prevoditi tako da je hrvatski jezik HR ulazni, drugi jezik J_1, J_2,..., J_n ciljni i obrnuto.



Tada bismo, prihvaćajući hrvatski jezik kao meta-jezik, imali istodobno riješen i poblem prevođenja bilo kojeg jezika J_i u J_k, $i \neq k$:

$$J_i \rightarrow HR \rightarrow J_k$$

No, odgovorno tvrdim da se problem prevođenja jednog prirodnog jezika u drugi ne može u potpunosti riješiti bez primjene teorije formalnih jezika, lingvistike, matematike, teorije baza podataka i teorije algoritama i struktura podataka. Mislim da veći dio teorije formalnih jezika dan u tri moje knjige može biti solidan početak.

Preostaje da udružite snagu s drugim znalcima za spomenute discipline i počnete rješavati problem prevođenja koji postaje sve aktualniji ulaskom Hrvatske u Europsku uniju.

Tada sigurno ne bismo često čitali prijevode uputa na mnogim proizvodima, pa i na facebooku, bez pravog padeža (sintakse) i značenja (semantike), kao što čitam na kraju uputa svoga radnog stolca: "Nepoštivanje ovih preporuka svibanj uzrokovati ozbiljne ozljede!". Sretno!

U Zagrebu, rujna 2013. godine

Autor

Literatura

[Aho1986] AHO, V.A.; SETHI, R.; ULLMAN,D.J.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[Aho1979] AHO, V. A.; ULLMAN, D. J.: *Principles of Compiler Design*, Addison-Wesley Publishing Company, 1979.

[Aho1973] AHO, V. A.; ULLMAN, D. J.: *The Theory of Parsing, Translation, and Compiling*, vol.I I: *Compiling*, Prentice-Hall, 1973.

[Aho1972] AHO, V. A.; ULLMAN, D. J.: *The Theory of Parsing, Translation, and Compiling*, vol. I: *Parsing*, Prentice-Hall, 1972.

[Bac1979] BACKHOUSE, C. R.: *Syntax of Programming Languages: Theory and Practice*, Prentice Hall, 1979.

[Ber1981] BERRY, E. R.: *Programming Language Translation*, Ellis Horwood Limited, 1981.

[Cre2009] CRESPI REGHIZZI, S.: *Formal Languages and Compilation*, Springer-Verlag, London, 2009.

[Dij1976] DIJKSTRA, E.W.: *A Discipline of Programming*, Prentice-Hall, 1976.

[Dov2012a] DOVEDAN HAN, Z.: *FORMALNI JEZICI I PREVODIOCI • regularni izrazi, gramatike, automati*, Element, Zagreb, 2012.

[Dov2012b] DOVEDAN HAN, Z.: *FORMALNI JEZICI I PREVODIOCI • sintaksna analiza i primjene*, Element, Zagreb, 2012.

[Dov2011] DOVEDAN HAN, Z.: *Pascal s tehnikama programiranja (1)*, VVG, Velika Gorica, 2011.

[Dov1995] DOVEDAN, Z.: *Pascal i programiranje (1)*, don, Zagreb, 1995.

[Dov1992] DOVEDAN, Z.: *Jedan model sintaktičke analize jezika za programiranje*, disertacija, Filozofski fakultet, Zagreb, 1992.

[Dov1983] DOVEDAN, Z.: *Sintaktička analiza jezika sa svojstvima*, Informatica 82/3, Ljubljana, 1983.

[Dov1982] DOVEDAN, Z.: *Sinteza i realizacija interaktivnog jezika s formalno definiranom semantikom*, mag. rad, Elektrotehnički fakultet, Zagreb, 1982.

[Flo1967] FLOYD, R.W.: *Assigning Meaning to Programs*, American Mathematical Society, 1967., p. 19-31

[Goo1976] GOOS, G.; HARTMANIS, J., editors: *Compiler Construction, An Advanced Course*, Springer-Verlag, 1976.

[Gru1990] GRUNE, D.: *Parsing Techniques – A Practical Guide*, Ellis-Horwood, 1990.

[Hoa1969] HOARE, C.A.R.: *An Axiomatic Basis for Computer Programming*, CACM, 12(10), 1969., p. 576-583

[Hop2001] HOPCROFT, E. J.; MOTWANI, R.; ULLMAN, D. J.: *Introduction to Automata Theory, Languages, and Computation*, second edition, Addison-Wesley, 2001.

[Man1968] MANNA, Z.: *The Correctness Problem of Computer Programs*, Computer Sc. Research Review, 1968., p. 34-36

[Ost1989] OSTOJIĆ, N.: *Matematički principi programiranja*, metodička zbirka zadataka, VVTŠ KoV, Zagreb, 1989.

[Slo1995] SLONNEGER, K.; KURTZ, B. L: *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Publishing Company, 1995.

[Sta1985] STANIĆ, Z.: *Predprocesor jezika DIKTRAN*, diplomski rad, VVTŠ KoV, Zagreb, 1985.

[Sto2012] STOJANOVIĆ, A.: *ELEMENTI RAČUNALNIH PROGRAMA s primjenom u Pythonu i Scali*, Element, Zagreb, 2012.

[Ull1976] ULLMAN, D.J.: *Fundamental Concepts of Programming Systems*, Addison-Wesley Publishing Company, 1976.

[Wai1984] WAITE, M. W.; GOOS, G.: *Compiler Construction*, Springer-Verlag, 1984.

[Wir1976] WIRTH, N.: *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Kazalo

Bilješka o autoru

chanson d'amour et d'amitié
chanson d'un vieux routier
de la vieille rengaine

chanson des rues et des pavés
perdue ou retrouvée
sur le bord de la seine

chanson qui vit
dans ma mémoire
et vient dans ma guitare
me jouer la chansonnette

chanson des nappes de papiers
chanson qui fait rêver
musique un peu simplette...

Georges Moustaki

Dr. sc. Zdravko DOVEDAN HAN (1952), redoviti je profesor na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta Sveučilišta u Zagrebu. Diplomirao je (1975) politehniku (aerodinamiku). Magistrirao je (1982) i doktorirao (1992) s temama iz računarskih znanosti, discipline formalni jezici i prevodioci.

Počeo je programirati u FORTRANu još za vrijeme studija politehnike, 1972. godine. Od 1977. godine bio je asistent na predmetima programiranja (BASIC i FORTRAN) i *Matematičkih principa programiranja*, a od 1979. godine predavač iz predmeta *Strukturno programiranje* (Pascal). Od 1984. godine predaje *Jezične procesore*, a od 1990. godine uvodi predmet *Formalni jezici i prevodioci* kojeg od 2005. godine čine tri kolegija: *Uvod u formalne jezike i automate*, *Teorija sintaksne analize i primjene* i *Teorija prevođenja i primjene*. Predaje ih na preddiplomskom, diplomskom i doktorskom studiju na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta Sveučilišta u Zagrebu. Također predaje *Algoritme i strukture podataka* i *Objektno i vizualno programiranje* na preddiplomskom studiju istog odsjeka.

Autor je niza znanstvenih i stručnih radova te članaka iz područja informacijskih i računarskih znanosti, posebno teorije formalnih jezika i programiranja. Vodio je tri znanstvena projekta MZO iz područja obrade i razumijevanja prirodnih jezika. Tvorac je nekoliko informacijskih sustava koji su u razdoblju od 1988. do 2013. godine bili instalirani u preko pedeset tvrtki diljem Hrvatske.

Kao autor ili koautor objavio je dvanaest knjiga od kojih su najpoznatije *BASIC*, Ljubljana (1986); *FORTRAN 77 s tehnikama programiranja*, Ljubljana (1987); *PASCAL i programiranje*, Ljubljana (1989); *GW-BASIC*, Zagreb (1990); *FORMALNI JEZICI – sintaksna analiza*, Zagreb (2003); *PASCAL s tehnikama programiranja*, Velika Gorica (2011); *FORMALNI JEZICI I PREVODIOCI – regularni izrazi*, *gramatike*, *automati* i *FORMALNI JEZICI I PREVODIOCI – sintaksna analiza i primjene*, Zagreb (2012).

Zagreb, prosinac 2013.