Transliatorius. Įvadas

Transliatorius – tai programa, kuri verčia *pradinę programą* į jai ekvivalenčią *objektinę programą*. Pradinė programa yra parašoma tam tikra *pradine kalba*, o objektinė programa formuojama tam tikroje *objektinėje kalboje*. Transliatoriaus programa vykdoma *transliavimo* metu.

.

Kai pradinė kalba aukšto lygio (pvz. Fortan, Pascal), o objektinė – žemo (autokodas, mašininė kalba), tai toks transliatorius vadinamas kompiliatoriumi.

Kai pradinė kalba autokodas arba asemblerio kalba, o objektinė – mašininė kalba, tai šis transliatorius vadinamas asembleriu..
Autokodas yra kalba, labai artima mašininei kalbai. Dauguma autokodo instrukcijų yra simbolinis mašininių komandų vaizdavimas.

Programa, kuri ne tik transliuoja programą, bet taip pat ir įvykdo, vadinama interpretatoriumi. Interpretatorius paima programą, parašytą pradinėje kalboje, kaip pradinę informaciją ir ją įvykdo. Interpretatorius nesukuria objektinės programos, kuri turėtų būti vėliau vykdoma, bet vykdo ją tiesiogiai pats.

Gramatikos ir kalbos

Kad būtų galima suprasti kaip veikia transliatorius, būtina žinoti kaip apibrėžiama programavimo kalba, jos konstrukcijos, simboliai.

Neformaliai kalbos gramatika vadiname sintaksinį programavimo kalbos apibrėžimą.

Produkcija arba gramatikos *taisykle* vadinama sutvarkyta pora (U,x), kuri paprastai užrašoma taip:

U ::= x

Čia U – simbolis, o x – netuščia baigtinė simbolių eilutė. U vadinama taisyklės *kairiaja puse*, o x – *dešiniaja puse*.

Gramatika G[Z] vadinama baigtinė netuščia taisyklių aibė. Z – tai simbolis, kuri turi būti bent vienos taisyklės dešiniojoje pusėje. Jis vadinamas gramatikos pradiniu simboliu. Visi simboliai, esantys kairiosiose ir dešiniosiose taisyklių pusėse sudaro *žodyną V* (kartais dar vadinamą *alfabetu*).

Simboliai, kurie sutinkami taisyklių kairiosiose pusėse vadinami *neterminaliniais*. Jų aibė žymima VN. Simboliai iš V, kurie neįeina į VN, vadinami *terminaliniais simboliais* jų aibė žymima VT. Taigi V = VN + VT. Neterminalus rašysime skliausteliuose < >.
Gramatikos pavyzdys:

```
<sakinys>::=<kintamasis>:=<išraiška>
<išraiška>::=<išraiška>+<termas> | <termas>
<termas>::=<termas>*<daugiklis> | <daugiklis>
<daugiklis>::= (<išraiška>) | <operandas>
<operandas>::=<identifikatorius> | <konstanta>
<kintamasis>::=<identifikatorius>
```

Eilutė – baigtinė alfabeto (arba žodyno) elementų seka.

Kalba L(G[Z]) – tai visų galimų terminalinių simbolių eilučių poaibis. T.y. eilutės iš simbolių, priklausančių VT. Terminalinių simbolių eilutė priklauso kalbai tik tada, kai ją galima išvesti iš pradinio gramatikos simbolio.

Chomsky N. apibrėžė 4 kalbų klases gramatikų terminais. Gramatika nurodoma kaip sutvarkytas ketvertas (V, T, P, Z). Čia V – alfabetas; $T \subseteq V$ – terminalinių simbolių alfabetas; P – baigtinis taisyklių sąrašas; Z – pradinis simbolis, $Z \in V - T$. Gramatikų rūšys:

1. Reguliarios (3 klasės).
2. Laisvo konteksto (2 klasės).
3. Kontekstinės (1 klasės).
4. Frazinės struktūros (0 klasės).

Šių gramatikų skirtumai kyla iš to kokios yra jų taisyklės. Gramatika yra 0 klasės arba gramatika su frazine struktūra, jei jos taisyklės turi tokį pavidalą:

$u ::= v$, čia $u \in V^+$, $v \in V^*$.

Kontekstinės gramatikos taisyklių pavidalas:

$xUy ::= xuy$, kur $U \in V - T$, $u \in V^+$ ir $x,y \in V^*$.

Laisvo konteksto gramatikos taisyklių pavidalas:

$U ::= u$, čia $U \in V - T$, $u \in V^*$.

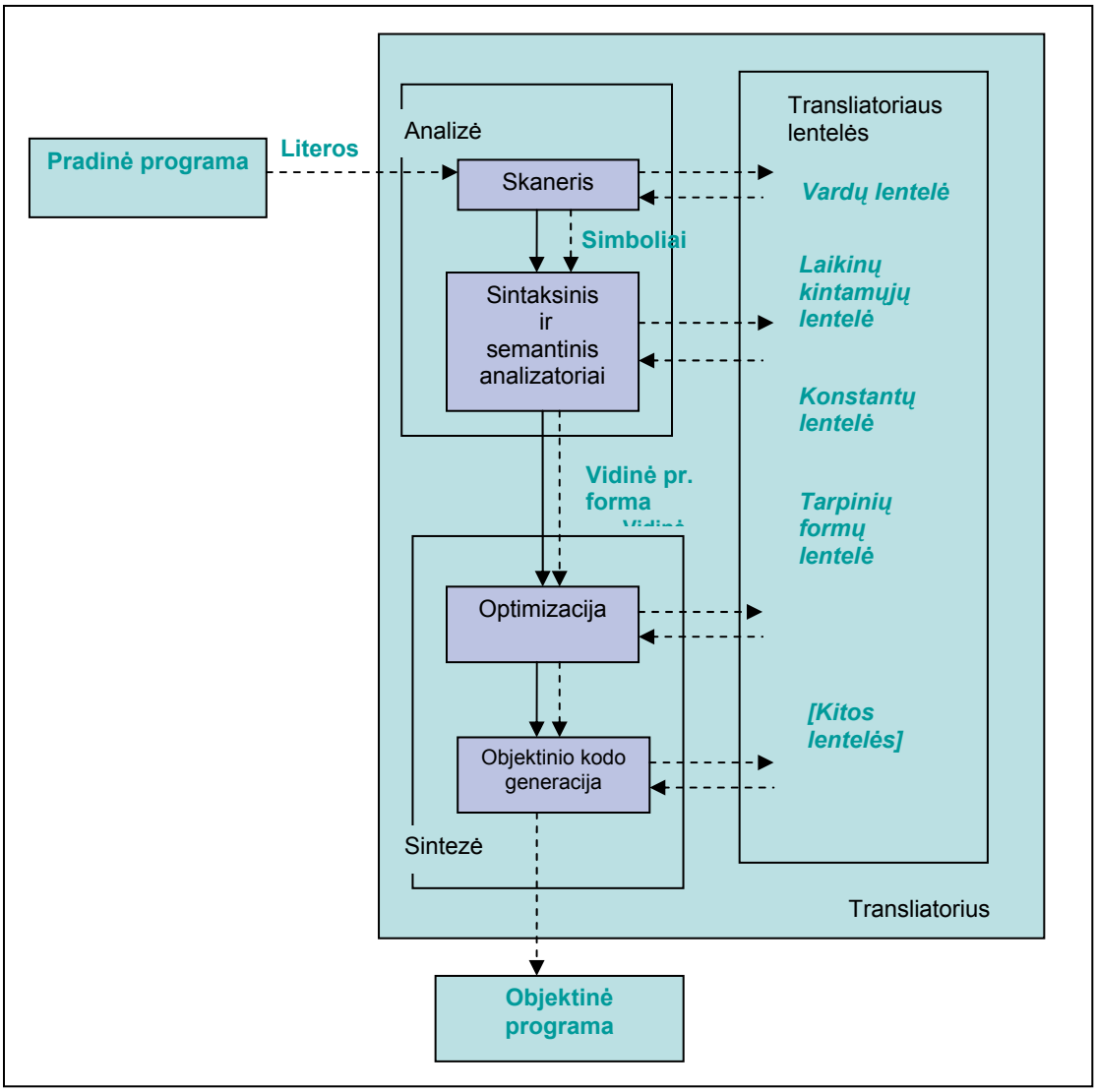
Reguliariosios gramatikos taisyklių pavidalas:

$U ::= N$ arba $U ::= WN$, čia $N \in T$, U ir $W \in V - T$.

Dažniausiai naudojama laisvo konteksto gramatika.
Paprasčiausia - reguliari gramatika. Ja aprašomi kalbos žodžiai, t.y. identifikatoriai, konstantos, skirtukai.

Transliavimo procesas

Kompiliatorius iš pradžių vykdo pradinės programos analizę, o tada objektinio kodo sintezę. Pradžioje pradinė programa skaidoma į jos sudedamąsias dalis, tada iš šių dalių kuriama atitinkama objektinė programa. Analizės metu transliatorius kuria lenteles, kurios vėliau naudojamos tolimesnėje analizėje bei sintezėje. Pav. vaizduoja transliavimo procesą. Punktyrinės rodyklės simbolizuoja informacijos srautą, o ištisinės – programos vykdymo tvarką.



1 pav.

Informacinės lentelės

Analizuojant programą , iš procedūrų, kintamųjų aprašų, ciklų antraščių ir kt. paimama informacija ir išsaugoma tolimesniam naudojimui. Šie duomenys saugomi atskirose programos vietose ir organizuojami taip, kad į juos būtų galima kreiptis iš bet kurios transliatoriaus dalies. Ką konkrečiai reikia saugoti, priklauso nuo pradinės kalbos, objektinės kalbos ir transliatoriaus sudėtingumo. Tačiau kiekvienas transliatorius turi vardų (arba identifikatorių) lentelę. Tai lentelė visų identifikatorių, sutinkamų pradinėje programoje, su jų atributais (tipas, adresas objektinėje programoje ir kita informacija, kurios galėtų prireikti generuojant objektinį kodą). Taip pat gali būti kuriamos konstantų, *for*- ciklų antraščių lentelės ir pan.

Skaneris

Skaneris – pati paprasčiausia transliatoriaus dalis, kartais dar vadinama *leksiniu analizatoriumi.* Skaneris iš kairės į dešinę peržiūri pradinės programos literas ir sudaro (atskiria iš pradinės programos teksto) programos *simbolius* – sveikus skaičius, identifikatorius, tarnybinius žodžius, dvigubus simbolius, tokius kaip **, //, /*, */ ir kt. Tada simboliai perduodami analizuojančiai programai. Skanerio lygyje gali būti išimami komentarai, simboliai gali būti siunčiami į vardų lentelę. Skaneris gali atlikti ir kitus nesudėtingus uždavinius, kuriems nereikalinga pradinės *programos* analizė.

Paprastai skaneris perduoda analizatoriui simbolius vidinėje formoje. Pavyzdžiui kiekvienas skyriklis (tarnybinis žodis, operacijos arba punktuacijos ženklas) gali būti vaizduojamas sveiku

skaičiumi. Kitus simbolius galima žymėti dviem skaičiais – pirmasis skaičius, nesutampantis su jokiu skyriklį atitinkančiu skaičiumi, žymi, kad turimas simbolis – identifikatorius ar konstanta, ar koks kitas, antras skaičius nurodo identifikatoriaus, konstantos ar kt. adresą atitinkamoje informacinėje lentelėje.

Sintaksinis ir semantinis analizatoriai

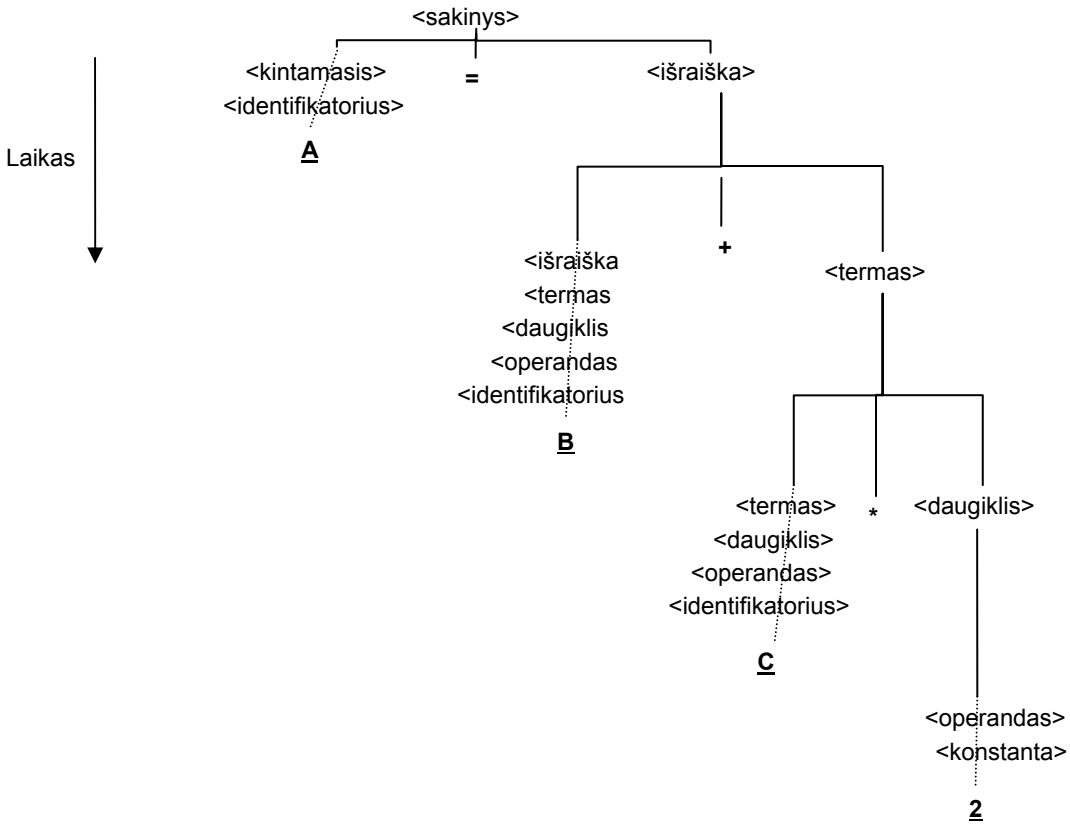
Analizatoriai skaido pradinę programą į sudedamąsias dalis, formuoja programos vidinę formą ir pildo informacines lenteles. Tai atliekama vykdant pilną pradinės programos sintaksinę ir semantinę analizę. Paprastai analizatorius yra sintaksiškai valdoma programa.
Stengiamasi kiek tik įmanoma atskirti sintaksę nuo semantikos. Kai sintaksinis analizatorius atpažįsta pradinės kalbos konstrukciją, jis iškviečia atitinkamą *semantinę procedūrą* arba *semantinę programą*. Pastaroji atlieka semantinę analizę ir išsaugo informaciją apie šią konstrukciją informacinėse lentelėse ir/arba vidinėje programos formoje. Pavyzdžiui, kai sutinkamas kintamojo aprašymas, semantinė programa patikrina ar šis identifikatorius nėra aprašomas antrąkart ir išsaugo informaciją apie jį vardų lentelėje.

Sentencialinės formos analizė – tai išvedimo kelio jai ieškojimas arba sintaksinio medžio sudarymas. Yra du sintaksinės analizės būdai – *kylantis* ir *žemėjantis*. Šie pavadinimai nusako būdą kaip sudaromi sintaksiniai medžiai. Žemėjančio analizatoriaus atveju sintaksinis medis sudaromas nuo šaknies – pradinio simbolio. Kylančio analizatoriaus esmė yra siekti nuo užduotą simbolių eilutę suvesti į pradinį gramatikos simbolį.

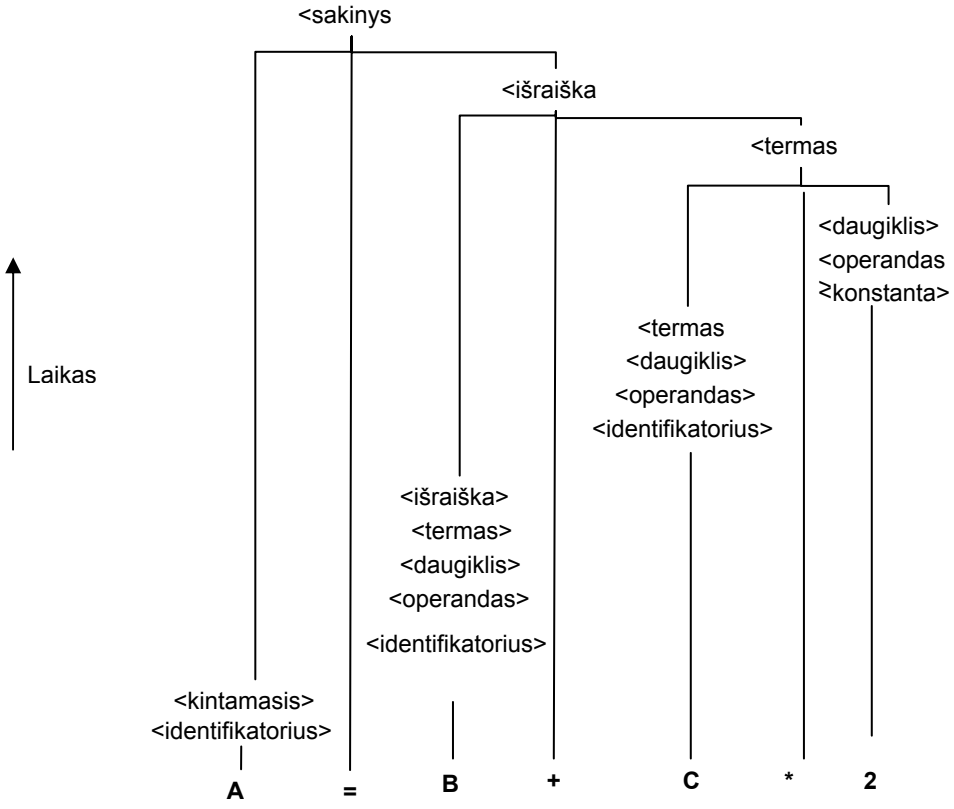
Pavyzdys.
Tarkim, kad turime tokią kalbos gramatiką:

```
<sakinys>::=<kintamasis>=<išraiška>
<išraiška>::=<išraiška>+<termas> | <termas>
<termas>::=<termas>*<daugiklis> | <daugiklis>
<daugiklis>::= (<išraiška>) | <operandas>
<operandas>::=<identifikatorius> | <konstanta>
<kintamasis>::=<identifikatorius>
```

Atpažinsime sakinį: A = B + C * 2.
Atlikdami analizę žemėjančiu analizatoriumi, gausime tokį sintaksinį medį (sintaksinis medis A) :



Atlikdami kylančią analizę, gausime tokį sintaksinį medį(sintaksinis medis B):



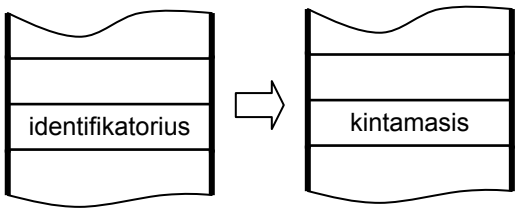
Kaip matome, baigus analizę, sintaksiniai medžiai visai nesiskiria.

Atliekant sintaksinę analizę, naudojamas sintaksinis stekas. Jo pildymo būdas priklauso nuo pasirinkto sintaksinės analizės būdo.
Tarkime duota eilutė A = B + C * 2. Atlikime kylančią sintaksinę analizę. Tada, pagal pateiktos kalbos gramatiką, atliekame veiksmus, kurių seka nurodyta pav. sintaksinis medis B (nurodyta, į kokius neterminalus ir kuriame analizės etape, suvedami simboliai).
Pradžia.
Į steką užrašome ‚identifikatorius‘ . Tai iš tikrųjų yra A vidinėje formoje. Tokius simbolius (vidinėje formoje) paprastai jau pateikia skaneris. *Ši* vidinė forma turi žymę, kad tai identifikatorius, ir nuorodą į vietą identifikatorių (vardų) lentelėje.

Tada panaudojama gramatikos taisyklė

<kintamasis>::=<identifikatorius>

ir vietoje ‚identifikatorius‘, į steką užrašoma ‚kintamasis‘. Daugiau taisyklių panaudoti negalime, nes nėra tokios, kur dešinėje pusėje būtų tik <kintamasis>.

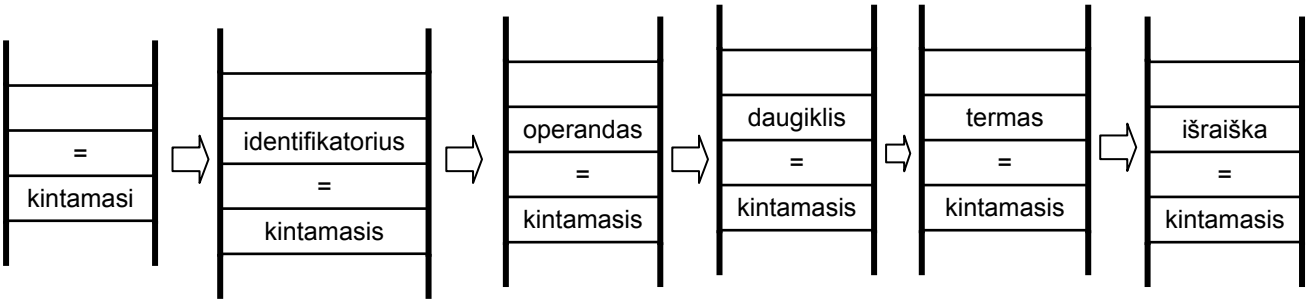


Į steką (sekančią ląstelę) užrašoma ‚=‘ .

Į steką užrašoma ‚identifikatorius‘ (B vidinėje formoje). Panaudojame gramatikos taisykles

<operandas>::=<identifikatorius>
<daugiklis>::= <operandas>
<termas>::= <daugiklis>
<išraiška>::= <termas>

ir ‚identifikatorius‘ pakeičiame į ‚operandas‘, tada į ‚daugiklis‘, paskui į ‚termas‘ ir po to į ‚išraiška‘.



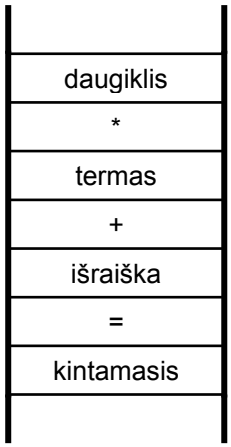
Ir taip tęsiame, visur kur galime pritaikydami gramatikos taisykles.

Kai į steką užrašysime ‚konstanta‘ (skaičius 2 vidinėje formoje), tai steke turi būti:

Pagal taisykles

```
<operandas>::= <konstanta>
<daugiklis>::= <operandas>
```

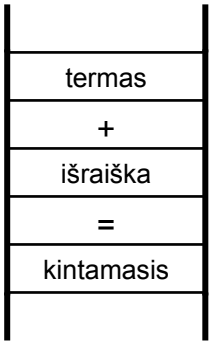
‚konstanta‘ keisime į ‚operandas‘, ‚daugiklis‘. Steke bus toks turinys:



Dabar galime panaudoti taisyklę

```
<termas>::=<termas>*<daugiklis>
```

ir „sutraukti“ steko turinį:



Pritaikome taisyklę

```
<išraiška>::=<išraiška>+<termas>
```

Pagaliau taikome taisyklę

```
<sakinys>::=<kintamasis>=<išraiška>
```

Štai baigėme sintaksinę analizę.

Atliekant sintaksinę redukciją („sutraukimą“, taisyklės pritaikymą – dešiniąją pusę keičiant kairiąja), iškviečiama atitinkama semantinė programa (semantinis analizatorius). Ši generuoja *vidinę programos formą*, pildo informacines lenteles.

Vidinė programos forma

Vidinė programos forma labai priklauso nuo jos tolimesnio naudojimo. Pradinės programos sintaksė gali būti vaizduojama: medžiu; lenkiška forma (operacija, operandas, operandas); tetradomis (operacija, operandas, operandas, rezultatas).
Pavyzdžiui, priskyrimo sakinys <A = B + C * 2> tetradomis atrodys taip:

*, C, 2, T1
+, B, T1, T2
=, T2, A,

Čia T1 ir T2 yra laikini kintamieji. Operandais šiame pavyzdyje bus ne patys simboliniai vardai, bet nuorodos į šiuos elementus (arba indeksai) vardų lentelėje, kur jie aprašyti.

Optimizacija
Prieš generuojant objektinį kodą, reikia optimizuoti vidinę programą – tam tikru būdu pakeisti. Pvz. daugybos operacijos keičiamos sudėties operacijomis. Be to reikia išskirti atminties pačios programos kintamiesiems.

Objektinio kodo generacija
Šiame etape vidinė programos forma yra „verčiama“ į autokodą arba mašininę kalbą. Tarkime, kad vidinė forma yra tetradų sąrašas, kaip ankstesniame pavyzdyje. Tada turime generuoti komandas kiekvienai tetradai iš eilės. Tarkime mūsų objektinė kalba yra asembleris IBM 360, tada anksčiau parašytom tetradom galėtume sugeneruoti šias komandas:

L	5, C	C pakrauti į registrą 5.
M	4, 2	Sandaugos rezultatą užrašyti į registrus 4 ir 5.
A	5, B	Prie sandaugos rezultato pridėti B.
ST	5, A	Įsiminti rezultatą.

Interpretatoriuje ši transliatoriaus dalis pakeičiama programa, kuri faktiškai vykdo programą vidinėje formoje. Todėl programa vidinėje formoje labai panaši į kompiliacijos metu gaunamą objektinį kodą.

Skaneris

Skaneris yra transliatoriaus dalis, kuri skaito pradinės programos literas ir konstruoja pradinės programos žodžius: identifikatorius, tarnybinius žodžius, skaičius, skyriklius. Kartais šie žodžiai yra vadinami leksemomis. Iš tiesų, tai skaneris vykdo pradinės programos teksto leksinę analizę. Todėl skaneris dar vadinamas *leksiniu analizatoriumi.*
Skanerio darbo objektą – žodžius aprašo reguliariosios gramatikos. Skaneris sintaksiniam analizatoriui vietoje žodžių grąžina fiksuoto formato (žodžių vidinė forma) informaciją. Gali kilti klausimas, kodėl nesujungus sintaksinės ir leksinės analizės. Skanerio atskyrimui yra kelios svarbios priežastys. Žodžių sintaksę galima aprašyti labai paprastomis gramatikomis, todėl galima sukurti labai efektyvų analizės algoritmą. Kadangi skaneris sintaksiniam analizatoriui pateikia žodžius vidinėje formoje, tai sintaksinis analizatorius gauna žymiai daugiau informacijos kaip toliau elgtis. Be to skaneris gali patikrinti kontekstą ir konfliktinėse situacijose nustatyti tikrąją literų reikšmę. Pvz.:
Tarkime analizuojamas Fortan kalbos programos tekstas. Skaneris paėmė literą „DO10I =“. Kad nustatyti ar čia prasideda priskyrimo kintamajam sakinys, ar ciklo antraštė, skaneris gali patikrinti kas eina pirmiau “, “ ar “(“.
Be to pačiai kalbai gali būti keletas realizacijų, kurios skiriasi tik programos teksto užrašymo detalėmis (pvz. tarpų ignoravimas arba neignoravimas, didžiųjų raidžių interpretavimas – skiriasi

nuo mažųjų arba nesiskiria ir pan.). Todėl tam pačiam transliatoriui galima parašyti kelis skanerius, nereikia perrašinėti viso transliatoriaus.

Naudojamos dvi skanerio schemos – nuosekli ir lygiagreti.
Nuoseklus skaneris. Galima skanerį programuoti taip, kad jis peržiūrėtų visą programos tekstą ištisai. Tada jis pateikia sintaksiniam analizatoriui lentelę su visa pradine programa vidinėje formoje.
Lygiagretus skaneris. Galima skanerį programuoti kaip procedūrą SCAN, į kurią kreipiasi sintaksinis analizatorius kiekvieną kartą, kai prireikia naujos leksemos. Šis būdas efektyvesnis, nes nereikalauja saugoti skanerio pateiktos ištisos programos.

Skanerio darbo objektą aprašo reguliariosios gramatikos, t.y. taisyklės yra tokio pavidalo:
U : : = N arba U : : = WN, čia N - terminalinis, U ir W neterminaliniai simboliai.

Pavyzdys.
Leksemų gramatika:

<identifikatorius> : : = <raidė> | <identifikatorius> <raidė> | <identifikatorius> <skaitmuo>
<raidė> : : = A | B | …
<skaitmuo> : : = 1 | 2 | … | 9
<sveikas> : : = <skaitmuo> | <sveikas> <skaitmuo>
<skirtukas> : : = <SLASH> / | <SLASH> * | <AST> * | <COLON>=
<SLASH> : : = /
<AST> : : = *
<COLON> : : = :

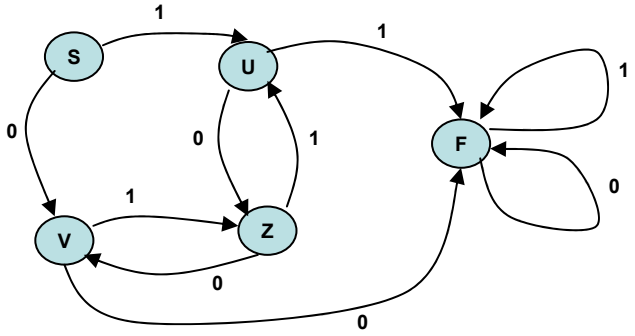
Pavyzdys.
Turime tokią reguliariąją gramatiką G[Z]:

Z : : = U0 | V1
U : : = Z1 | 1
V : : = Z0 | 0

Kalba pagal šią gramatiką yra aibė tokių terminalinių eilučių: 0110, 1010, 10100101, …
Formaliai kalbą galima apibrėžti taip:

$L(G) = \{ B^n \mid n > 0\}$, čia B = {01, 10}

Atpažinimo palengvinimui nubraižykime būsenų diagramą:



Kaip matome – visi neterminaliniai simboliai yra žymimi mazgais ir atitinka *būsenas*, o terminaliniai simboliai žymi briaunas. Pridėtos *pradinė* ir *klaidos* būsenos.

Ši būsenų diagrama atitinka baigtinį automatą DA({S,Z,U,V,F},{0,1},M,S,{Z}).

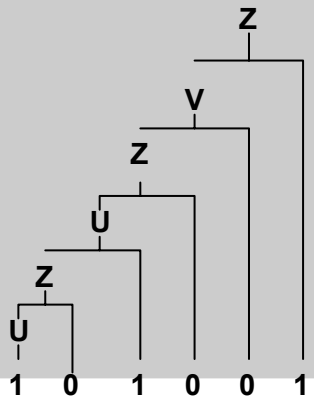
Baigtinis automatas – tai penketas (K,VT,M,S,Z).
K – būsenų alfabetas
VT – terminalinių simbolių alfabetas
M – atvaizdavimo funkcija: $M : K \times VT \rightarrow K$. Funkcija užrašoma taip:
$M(Q,T) = R$. Čia $Q,R \in K$, $T \in VT$, $R \in K$.
S – pradinė būsena
Z – galutinių būsenų aibė.

Automate DA M yra toks atvaizdis:

M (S,0) = V	M (S,1) = U
M (V,0) = F	M (V,1) = Z
M (U,0) = Z	M (U,1) = F
M (Z,0) = V	M (Z,1) = U
M (F,0) = F	M (F,1) = F

Patikrinkime, ar eilutė 101001 priklauso kalbai L(G). Tikrinsime imdami iš eilės po vieną simbolį. Taip pat sudarykime sintaksinį medį.

Žingsnis	Einamoji būsena	Eilutės likutis
1	S	101001
2	U	01001
3	Z	1001
4	U	001
5	Z	01
6	V	1
7	Z	



Eilutė priklauso kalbai, jei atliekant sintaksinę analizę pasiekiama galutinė būsena (Z) Taigi nagrinėta eilutė priklauso kalbai.

Betkuriai reguliariajai gramatikai galima sukonstruoti baigtinį automatą. Betkuriam baigtiniam automatui egzistuoja reguliari gramatika.

Baigtinį automatą galima vaizduoti matrica.
Tarkime, kad turime sunumeruotą būsenų aibę $S_1, S_2, \dots, S_n$, sunumeruotą terminalinių simbolių aibę $T_1, T_2, \dots, T_m$ ir atvaizdavimo funkciją M. Tada baigtinį automatą galime vaizduoti matrica B(n×m).
$B(i, j) = k$, jei $M(S_i, T_j) = S_k$.

Skanerio programavimas

Programuosime skanerį nesudėtingai programavimo kalbai.

Jos simboliai (leksemos):

Skirtukai arba operacijų ženklai: /, +, -, *, (,) , ir //.

Tarnybiniai žodžiai: BEGIN, ABS, END.

Identifikatoriai.

Sveiki skaičiai.

Komentarai yra išskiriami dvigubais simboliais /* ir */.

Tegu vidinėje formoje leksemos bus vaizduojamos kaip parodyta lentelėje dešinėje.

Vidinis vaizdavimas (leksemos kodas)	Leksema	Mnemoninis vardas
0	Neapibrėžtas simb.	$UND
1	Identifikatorius	$ID
2	Sveikas	$INT
3	BEGIN	$BEGIN
4	ABS	$ABS
5	END	$END
6	/	$SLASH
7	+	$PLUS
8	-	$MINUS
9	*	$STAR
10	(	$LPAR
11	)	$RPAR
12	//	$SLSL

Mnemoninis vardas reikalingas programuojant. Tai gali būti kintamojo, saugančio leksemos kodą vardas.

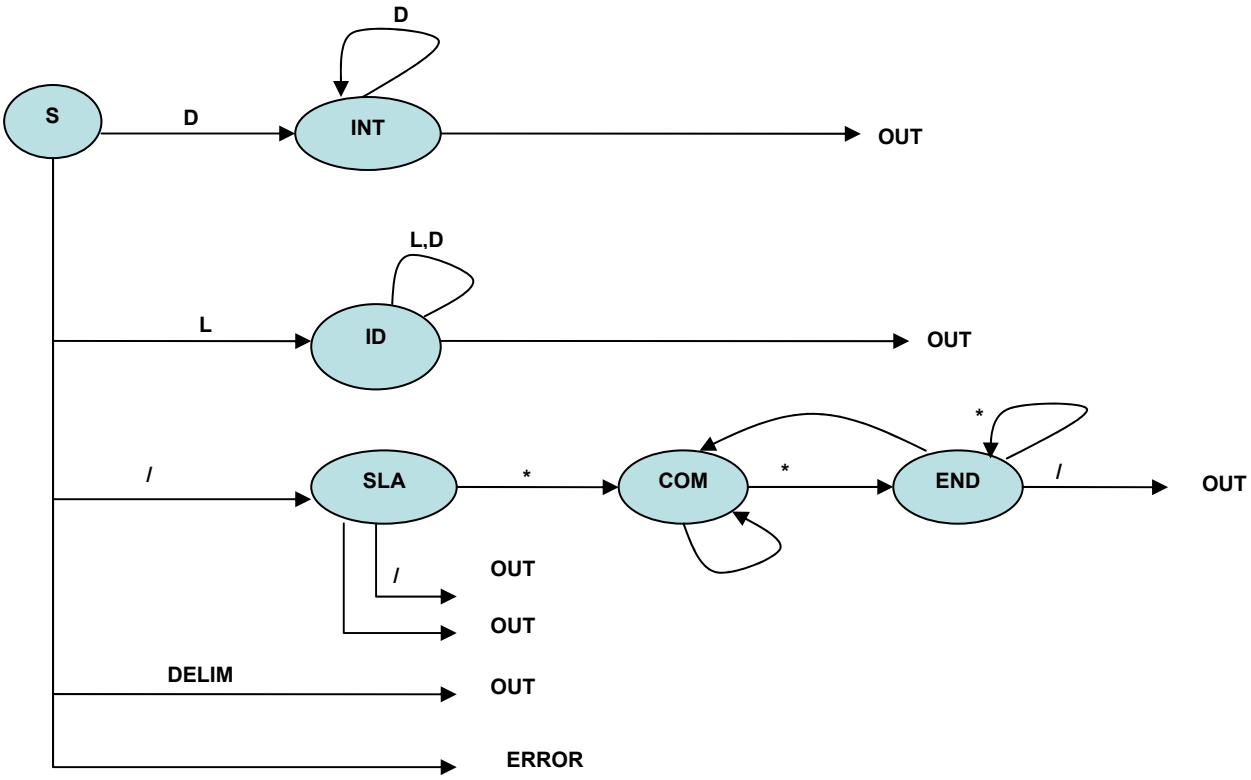
Tarkime pradinė programa yra tokia:

BEGIN A+/BC// /*COMMENT ++*/END 11

Tada skaneris turi grąžinti tokią lentelę:

Žingsnis	Rezultatas
1	3, `BEGIN`
2	1, `A`
3	7, `+`
4	6, `/`
5	1, `BC`
6	12, `//`
7	5, `END`
8	2, `11`

Pradžioje nubraižykime baigtinį automatą , vaizduojantį, kaip vyksta skanerio atliekama simbolio leksinė analizė:



INT – sveikų skaičių būsena.
ID – identifikatoriaus būsena.
SLA – `/` būsena.
COM – komentaro būsena.
END - komentaro pabaigos būsena.

Skanerio darbui reikalingi šie globalūs kintamieji ir paprogramės:

`CHARACTER CHAR`
Saugo einamuoju momentu skanuojamą pradinės programos literą.

`INTEGER CLASS`
Saugo sveiką skaičių, nurodantį literos, saugomos `CHAR`, klasę (D (skaitmuo) – 1; L (raidė) – 2, `/` - 3, DELIM - 4).

`STRING A`
Saugo eilutę literų, sudarančių simbolį.

`GC ~GETCHAR`
Procedūra. Paima naują literą ir priskiria `CHAR`, literos klasę priskiria kintamajam CLASS. Gali atlikti ir kitus darbus – spausdinti sekančią pradinės programos eilutę ir pan.

`GETNONBLANK`
Procedūra tikrina ar CHAR turinys nėra tarpas. Jei taip kviečia `GETCHAR` tol, kol bus ne tarpas.

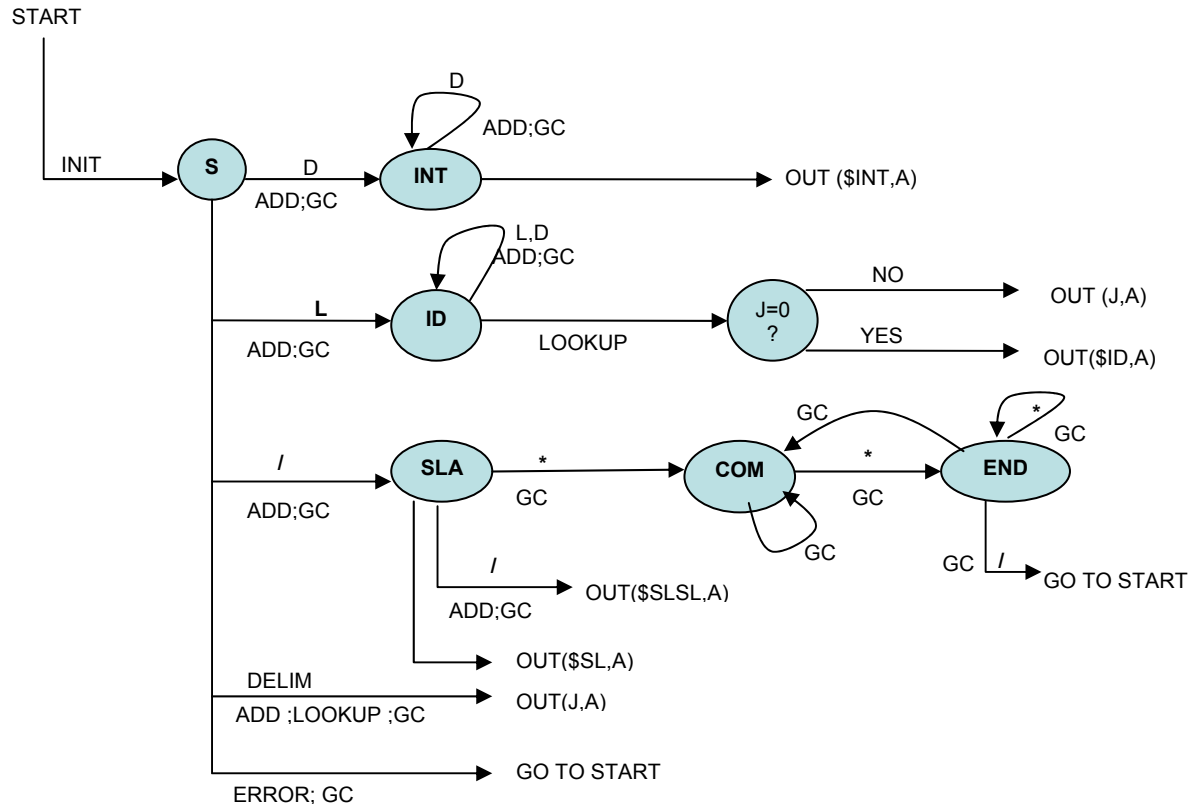
`ADD`
Literą iš `CHAR` prijungia prie A, t.y. atlieka : `A := A CAT CHAR.`

`OUT (C,D)`
Grįžtama į procedūrą, iškvietusią skanerio programą. C – leksemos kodas, D – leksema.

`LOOKUP`
Ieško kintamajame A surinktos leksemos tarnybinių žodžių lentelėje. Jei rado tokį tarnybinį žodį, tai globaliam kintamajam J priskiria jo kodą, jei ne – J priskiria 0 (šiuo atveju leksema yra identifikatorius).

`INIT`
Iškviečia `GETNONBLANK`. Išvalo kintamąjį A.

Dabar pridėkime aprašytas komandas į skanerio darbo diagramą



Skanerio programa

Skanerio procedūra turi du parametrus. Pirmasis – vidinis leksemos kodas, antrasis – pati leksema – simbolių eilutė, sudaranti leksemą.

```
PROCEDURE SCAN (INTEGER SYN; STRING SEM)

START: GETNONBLANK; A:= ` `;
CASE CLASS OF
   BEGIN
       BEGIN  WHILE CLASS = 1 DO
          BEGIN     A := A CAT CHAR;
                    GETCHAR;
          END;
```

```
         SYN:= $INT;
      END;
      BEGIN  WHILE CLASS ≤ 2 DO
         BEGIN     A := A CAT CHAR;
                   GETCHAR;
         END;
         SYN:= $ID;
         LOOKUP (A);
         IF J≠0 THEN SYN:= J;
      END;
      BEGIN  A:= CHAR; GETCHAR;
         IF CHAR = `*` THEN
                   BEGIN
                       B: GETCHAR;
                       C: IF CHAR ≠ `*` THEN GOTO B;
                       GETCHAR;
                       IF CHAR ≠ `/` THEN GOTO C;
                       GETCHAR;
                       GOTO START;
                   END;
         IF CHAR = `/` THEN
                   BEGIN
                       A:= A CAT CHAR;
                       GETCHAR;
                       SYN:= $SLSL
                   END;
                      ELSE  SYN:= $SLASH;
       END;
       BEGIN ERROR;
         GETCHAR;
         GOTO START;
       END;
    END;
SEM:= A;
```

Gramatikos ir kalbos (Papildymas)

Tegu G – gramatika. Sakoma, kad eilutė v *betarpiškai* generuoja eilutę w, ir žymima:

$$v \Rightarrow w$$

jei tam tikroms eilutėms x ir y galima parašyti:

$$v = xUy,\ w = xuy$$

ir yra gramatikos G taisyklė U : : = u.
Taip pat sakoma, kad w betarpiškai išvedama iš v arba w betarpiškai redukuojasi į v.

Sakoma, kad eilutė v generuoja w arba w redukuojasi į v, ir rašoma $v \Rightarrow^+ w$, jei egzistuoja betarpiškai generuojamų eilučių seka tokia, kad

$$v= u0 \Rightarrow u1 \Rightarrow u2 \Rightarrow \ldots \Rightarrow u[n] = w$$

Čia n>0.
Šis išvedimas vadinamas ilgio n išvedimu.

Žymima $v \Rightarrow^* w$, jei $v \Rightarrow^+ w$ arba v = w.

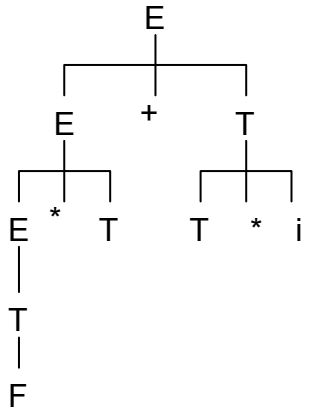
Tegu G[Z] – gramatika. Eilutė x vadinama *sentencialine forma,* jei x išvedama iš pradinio simbolio Z, t.y. jei $Z \Rightarrow^* x$. Sakinys – tai sentencialinė forma, susidedanti tik iš terminalinių simbolių. Kalba L(G[Z]) tai aibė sakinių:

$$L(G[Z]) = \{x \mid Z \Rightarrow^* x, x \in VT^+\}.$$

Tegu G[Z] – gramatika ir w = xuy – sentencialinė forma. Tada u vadinama sentencialinės formos w *fraze* neterminaliniam simboliui U, jei $Z \Rightarrow^* xUy$ ir $U \Rightarrow^+ u$, o jei $Z \Rightarrow^* xUy$ ir $U \Rightarrow u$, tai u vadinama *paprastąja fraze.*

Reikia būti atidiems. Faktas, kad teisinga $U \Rightarrow^+ u$, visai nereiškia, kad u yra sentencialinės formos xuy frazė. Būtina sąlyga, kad būtų išvedimas $Z \Rightarrow^* xUy$.

Pavyzdys.
Turime gramatiką G[E] ir tokį sintaksinį medį :



Kaip matome, eilutė F*T + T*i yra sentencialinė forma, nes iš vedama iš pradinio simbolio.
F*T yra šios eilutės frazė neterminaliniam simboliui E. i yra frazė neterminalinio simbolio F atžvilgiu, be to – tai paprastoji frazė.
Frazės – tai sintaksinio medžio, kurio šaknis yra pradinis simbolis, pomedžiai.

Sentencialinės formos pagrindu vadinama pati kairioji paprastoji frazė.

Jei gramatikoje galimas toks išvedimas: $U \Rightarrow^+ \ldots U \ldots$, sakoma, kad gramatika yra *rekursinė* U atžvilgiu. Jei $U \Rightarrow^+ U \ldots$, tai turime kairiąją rekursiją U atžvilgiu. Jei $U \Rightarrow^+ \ldots U$, tai turime dešiniąją rekursiją U atžvilgiu.

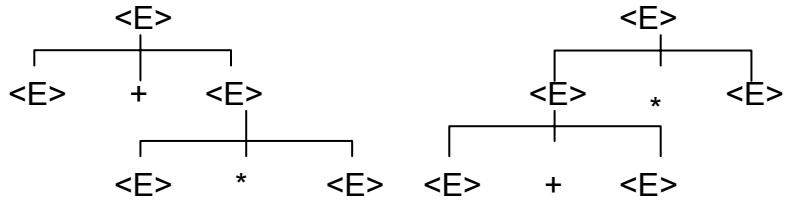
Gramatikos sakinys yra *nevienareikšmis,* jei jo išvedimui egzistuoja du skirtingi sintaksiniai medžiai. Gramatika vadinama *nevienareikšme*, jei joje galimi nevienareikšmiai sakiniai.

Kalbos, kurioms neegzistuoja vienareikšmės gramatikos, vadinamos *esminiai nevienareikšmėmis kalbomis*.

Gramatikos nevienareikšmiškumo problema algoritmiškai neišsprendžiama. Tai reiškia, kad neegzistuoja tokio algoritmo, kuris betkuriai Bekaus-Nauro formos gramatikai baigtiniu žingsnių skaičiumi nustatytų, ji vienareikšmiška ar ne.

Pavyzdys.
Turima tokia gramatika G[<E>]:
<E> ::= <E> +<E> | <E> * <E> | (<E>) | i

Parodysime, kad gramatikos sentencialinė forma <E> + <E> * <E> yra nevienareikšmė.
Nubraižykime sintaksinius medžius:



Sentencialinė forma <E> + <E> * <E> nėra vienareikšmė, nes šios sentencialinės formos išvedimui, pagal pateiktą gramatiką, galima sudaryti du skirtingus sintaksinius medžius. Taip pat ir gramatika G[<E>] yra nevienareikšmė.

Betarpiškas išvedimas xUy $\Rightarrow$ xuy vadinamas *kanoniniu*, jei eilutę y sudaro tik terminalai. Išvedimas w $\Rightarrow$ + v vadinamas kanoniniu, jei kiekvienas betarpiškas išvedimas šioje sekoje yra kanoninis.

Ne kiekviena sentencialinė forma, bet kiekvienas sakinys turi kanoninį išvedimą.

Sentencialinė forma, kuri turi kanoninį išvedimą, vadinama *kanonine* sentencialine forma
Kanoninė sentencialinė forma pasižymi tuo, kad dešiniau pagrindo yra tik terminaliniai simboliai.

Vykdant kylančią sintaksinę analizę, sentencialinėje formoje nustatomas pagrindas (paprastoji frazė, esant arčiausiai eilutės pradžios – „kairiausia“) ir jis keičiamas taisyklės kairiąja puse. Toks išvedimas ir vadinamas *kanoniniu.*

Programos konstrukcijų atpažinimo programa vadinama sintaksiniu analizatoriumi.
Pagrindinės sintaksinės analizės problemos:

1. Žemėjančio analizatoriaus problema:
   Kuo pakeisti patį kairįjį neterminalą V, jei yra kelios dešiniosios pusės?
2. Kylančio analizatoriaus problema:
   Kiekviename žingsnyje yra redukuojamas pagrindas. Kaip rasti tą pagrindą ir kaip nustatyti į kokį neterminalą jį redukuoti?

Žemėjantys analizatoriai

Žemėjantis analizatorius sudaro sintaksinį medį, pradėdamas nuo šaknies – pradinio gramatikos simbolio, palaipsniui besileisdamas iki sakinio lygio. Apie tai jau kalbėta aukščiau.

Žemėjantis analizatorius su grįžimais

Kad šio analizatoriaus algoritmas būtų lengviau suprantamas, aprašysime šį algoritmą naudodamiesi suprantama *žmogaus* sąvoka. Tarsime, kad kiekviename analizės etape -kiekviename konstruojamo sintaksinio medžio mazge, yra po vieną žmogų, kuris atlieka tam tikrą darbą – ieško išvedimo terminalinei eilutei.
Sakykim, kokiam tai žmogui reikia išanalizuoti sakinį x. Pradžioje jis turi surasti išvedimą $Z \Rightarrow^+ x$, kur Z – pradinis gramatikos simbolis. Tegu neterminalui Z yra tokios taisyklės:

$$Z ::= X_1 X_2 \ldots X_n \mid Y_1 Y_2 \ldots Y_m \mid Z_1 Z_2 \ldots Z_l$$

Pradžioje žmogus bando pritaikyti taisyklę $Z ::= X_1 X_2 \ldots X_n$. Jei neišeina sudaryti išvedimo medžio naudojantis šia taisykle, tai žmogus bando taisyklę $Z ::= Y_1 Y_2 \ldots Y_m$. Ir taip toliau.
Galime sakinį x užrašyti taip: $x_1 x_2 \ldots x_n$. Žmogus įsisūnija žmones M_i, kad jie rastų išvedimą atitinkamai $X_i \Rightarrow^* x_i$, i= 1 … n. Jei išvedimą rado M_1 (sakoma, kad jis uždengė eilutę x_1), tai isisūnijamas M_2, ir taip toliau. Jei M_c išvedimo nerado, tai atsisakoma sūnaus M_c ir vyresniojo sūnaus M_{c-1} paprašoma rasti kitą išvedimą (vyksta grįžimas). Jei ir šis nerado kito išvedimo, tai vėl atsisakoma sūnaus M_{c-1} ir sūnaus M_{c-2} paprašoma rasti kitą išvedimą ir t.t.. Sūnūs savo ruožtu taip pat vykdo įsisūnijimus, ieškodami išvedimo savo tikslui. Taip tęsiama kol bus rastas išvedimas visai eilutei x.

Įsisūnijimo imitacijai naudojamas stekas

Algoritmo vykdymo metu aktyvus yra tik vienas žmogus.

Kalbos gramatiką užrašysime į masyvą `GRAMMAR` tokiu būdu:
Jei turima tokia taisyklių aibė: `U :: = x | y | … | z`, tai ji masyve atrodys taip : Ux|y|…|z|$

Imkime tokią gramatiką:

```
Z :: = E #
E :: = T+E | T
T :: = F*T | F
F :: = (E) | i
```

Masyve `GRAMMAR` ji atrodys taip: `ZE#|$ET+E|T|$TF*T|F|$F(E)|i|$`

Kiekvienas steko elementas atitinka vieną žmogų ir susideda iš penketo

```
(GOAL, i , FAT, SON, BRO)
```

`GOAL` – simbolis, kurio ieško žmogus. Jis turi rasti neuždengtoje eilutėje pagrindą, kuris redukuojasi į GOAL, ir uždengti tą eilutės dalį.
i – indeksas masyve `GRAMMAR`, rodantis į taisyklės, kurios kairiojoje pusėje yra `GOAL`, tą simbolį dešiniojoje pusėje, su kuriuo žmogus dirba einamu momentu.
`FAT` – tėvo vardas (numeris steko elemento, atitinkančio žmogaus tėvą).
`SON` – vardas pačio jauniausio iš sūnų.
`BRO` – vyresniojo brolio vardas.

Dar algoritme naudosime šiuos kintamuosius:

j – paties kairiausio neuždengto (neatpažinto) simbolio eilutėje x numeris. Eilutė x – tai tokia seka INPUT(1), INPUT(2), … , INPUT(n).

c – einamu momentu dirbančio žmogaus vardas (steko elemento numeris).

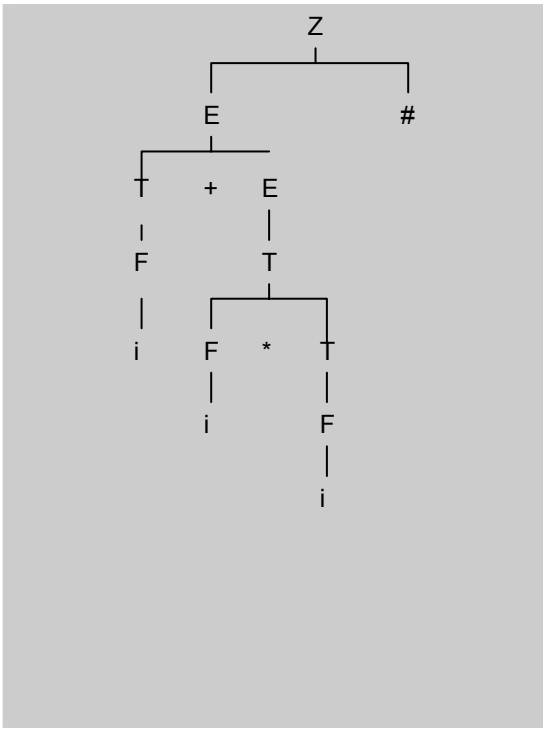
v – visų einamu momentu esančių žmonių skaičius (steko elementų skaičius).

Algoritmas:

```
INICIALIZACIJA
           S(1):= (Z,0,0,0,0);
           c:=1;
           v:=1;
           j:=1;
           GO TO NAUJAS ŽMOGUS
NAUJAS ŽMOGUS
           IF GOAL terminalas THEN
                      IF INPUT(j) = GOAL
                                 THEN
                                  BEGIN
                                     j:= j+1;
                                     GO TO SĖKMĖ
                                  END
                                  ELSE   GO TO NESĖKMĖ
           i:= taisyklės simboliui GOAL dešiniosios pusės pradžios indeksas
masyve GRAMMAR;
           GO TO CIKLAS
CIKLAS
           IF GRAMMAR(i) = "|" THEN
                      IF FAT ≠ 0 THEN GO TO SĖKMĖ
                                 ELSE  STOP – kalbos sakinys;
           IF GRAMMAR(i) = "$" THEN
                      IF FAT ≠ 0 THEN GO TO NESĖKMĖ
                                 ELSE STOP – ne kalbos sakinys;
           v:= v+1;
           S(v):= (GRAMMAR(i),0,c,0,SON);
           SON:= v;
           c:= v;
           GO TO NAUJAS ŽMOGUS
SĖKMĖ
           c:= FAT;
           i:= i + 1;
           GO TO CIKLAS;
NESĖKMĖ
           c:= FAT;
           v:= v-1;
           SON:= S(SON).BRO;
           GO TO DAR KARTĄ
DAR KARTĄ
           IF SON = 0 THEN
                      BEGIN
                        WHILE GRAMMAR(i) ≠ "|" DO i:= i+1;
                        i:= i+1;
                        GO TO CIKLAS;
                      END
           i:= i-1;
           c:= SON;
```

```
IF GOAL neterminalas THEN GO TO DAR KARTĄ
j:= j – 1;
GO TO NESĖKMĖ
```

Naudodamiesi šiuo algoritmu, atlikime sakinio i + i * i analizę.
Turime gauti tokį steko turinį, kuris atspindi sintaksinio išvedimo medžio šiam sakiniui struktūrą:



Steko elemento numeris (v)	GOAL	i	FAT	SON	BRO
1	Z	4	0	15	0
2	E	10	1	7	0
3	T	20	2	4	0
4	F	28	3	5	0
5	I	0	4	0	0
6	+	0	2	0	3
7	E	12	2	8	6
8	T	18	7	13	0
9	F	28	8	10	0
10	I	0	9	0	0
11	*	0	8	0	9
12	T	20	8	13	11
13	F	28	12	14	0
14	I	0	13	0	0
15	#	0	1	0	2

Išanalizavę algoritmą, galime pamatyti, kad šis algoritmas turi didelį trūkumą – pilnus grįžimus. Problemos dėl jų iškils, kai kartu su sintaksine analize taip pat bus atliekamos ir semantinės programos. Apie tai bus kalbama vėliau.

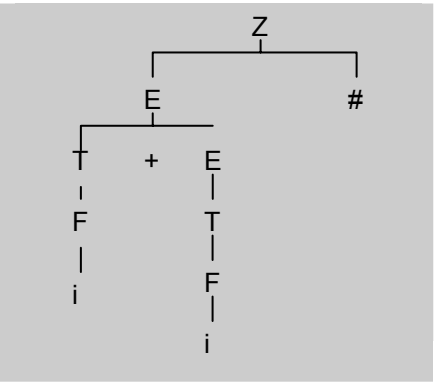
Modifikuokime šį algoritmą taip, kad jo darbo metu būtų kuo mažiau grįžimų. Tam būtina taip sutvarkyti taisykles masyve GRAMMAR, kad galimos dešiniosios pusės tam pačiam neterminalui būtų išdėstytos mažėjančia tvarka. (Masyvas GRAMMAR jau yra taip sutvarkytas, nors ankstesniame algoritme tai nebuvo svarbu.) Dabar iš kart bus stengiamasi pritaikyti teisingą taisyklę. Dėl šio pakeitimo galime atsisakyti steko elemento komponento BRO, bet bus saugomas einamu momentu analizuojamo simbolio sakinyje indeksas j.
Steko elementas atrodys taip: (GOAL, i, FAT, SON, j)

Algoritmas.

```
INICIALIZACIJA
          S(1):= (Z,0,0,0,1);
          c:= 1;
          v:= 1;
          GO TO NAUJAS ŽMOGUS
NAUJAS ŽMOGUS
          i:= indeksas, nurodantis taisyklės simboliui GOAL pirmosios
          dešiniosios pusės pradžią masyve GRAMMAR;
          GO TO CIKLAS
CIKLAS
          IF GRAMMAR(i) = "|"
                    THEN IF FAT ≠ 0 THEN GO TO SĖKMĖ;
                                    ELSE STOP - kalbos sakinys;
          IF GRAMMAR(i) = "$"
                    THEN IF FAT ≠ 0 THEN GO TO NESĖKMĖ
                                    ELSE STOP - ne kalbos sakinys;
          IF GRAMMAR(i) terminalas THEN
                    IF INPUT (j) = GRAMMAR (i)
                              THEN
                                 BEGIN
                                    j:= j+1;
                                    i:= i+1;
                                    GO TO CIKLAS;
                                 END
                              ELSE GO TO ALTERNATYVA
          v:= v+1;
          S(v):= (GRAMMAR(i),0,c,0,j);
          SON:= v;
          c:= v;
          GO TO NAUJAS ŽMOGUS
SĖKMĖ
          c:= FAT;
          j:= S(SON).j;
          i:= i+1;
          GO TO CIKLAS;
NESĖKMĖ
          c:= FAT;
          SON:= 0;
          GO TO ALTERNATYVA
ALTERNATYVA
          v:= c;
          IF FAT <> 0 THEN j:= S(FAT).j
                      ELSE j:= 1;
          WHILE GRAMMAR(i) <>"|" DO i:= i+1;
          i:= i+1;
```

```
GO TO CIKLAS;
```

Šiuo algoritmu išanalizavę eilutę `i + i #`, turėtume gauti tokį steko turinį (palyginkite jį su sintaksiniu medžiu):



Steko elemento numeris (v)	GOAL	i	FAT	SON	j
1	Z	4	0	2	5
2	E	10	1	5	4
3	T	20	2	4	2
4	F	28	3	0	2
5	E	12	2	6	4
6	T	20	5	7	4
7	F	28	6	0	4

Žemėjančios analizės problemos ir jų sprendimas

Aukščiau aprašyti analizatoriai netinkami naudoti, kai gramatikos taisyklėse yra betarpiška kairioji rekursija (taisyklės X : : = X …) .Tada gaunamas begalinis ciklas. Kad išvengti betarpiškos kairiosios rekursijos, galima taisykles užrašyti naudojantis iteracijos ženklais:

Vietoje `E:: = E + T` rašyti `E :: = T {+T}`
Vietoje `T:: = T * F | T / F | F` rašyti `T :: = F {* F | / F}`

Bendru atveju, tiesioginė kairioji rekursija gramatikoje naikinama dviem etapais:

Faktorizacija. Jei egzistuoja tokio pavidalo taisyklės:

```
U :: =  xy | xw | … | xz
```

tai jas keičiame į

```
U :: = x ( y | w | … | z )
```

Po faktorizacijos gramatikoje kiekvienam neterminalui lieka ne daugiau kaip viena dešinioji pusė su kairiąja tiesiogine rekursija.

Iteracijos panaudojimas. Tarkime turime tokią taisyklę, su kairiąja rekursija:

```
U :: = x | y | … | z | Uv
```

Ją keičiame į:

```
U :: = ( x | y | … | z ) {v}
```

Gramatikos vaizdavimas atmintyje

Vienas iš būdų – jau naudotas masyvas `GRAMMAR`. Bet jis nepatogus, nes reikalauja ieškoti dešiniosios pusės pradžios indekso kiekvienam neterminalui.

Gramatikos vaizdavimui galime naudoti sąrašinę struktūrą, vadinamą sintaksiniu grafu. Kiekvienas mazgas atitinka simbolį S iš dešiniosios taisyklės pusės ir susideda iš komponenčių:

`VARDAS` – pats simbolis S vidinėje formoje.

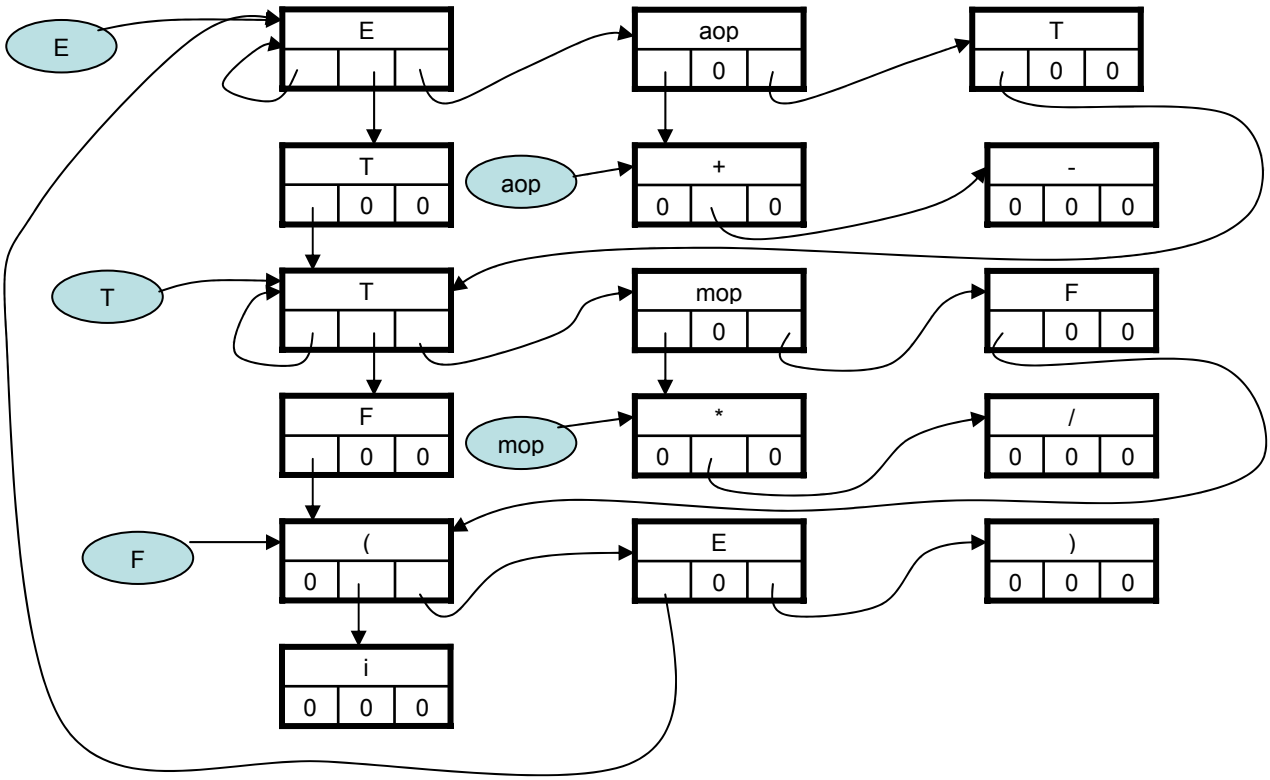
APIBRĖŽIMAS – nuoroda į mazgą, atitinkantį pirmą simbolį pirmoje iš dešiniųjų pusių simboliui S, arba, jei S – terminalas, tai reikšmė 0.
ALTERNATYVA – nuoroda į mazgą, atitinkantį pirmą simbolį dešiniosios pusės alternatyvios tai, kurios šis mazgas yra. Tai taikoma tik dešiniųjų pusių pirmiesiems simboliams. Jei S – terminalas, tai reikšmė 0.
SEKANTIS – nuoroda į mazgą, saugantį sekantį simbolį taisyklės dešiniojoje pusėje.

VARDAS		
APIBRĖŽIMAS	ALTERNATYVA	SEKANTIS

Imkime tokią gramatiką:

```
E ::= E <aop> T | T
T ::= T <mop> F | F
F ::= ( E ) | i
<aop> ::= + | -
<mop> ::= * | /
```

Sintaksinis grafas šiai gramatikai:



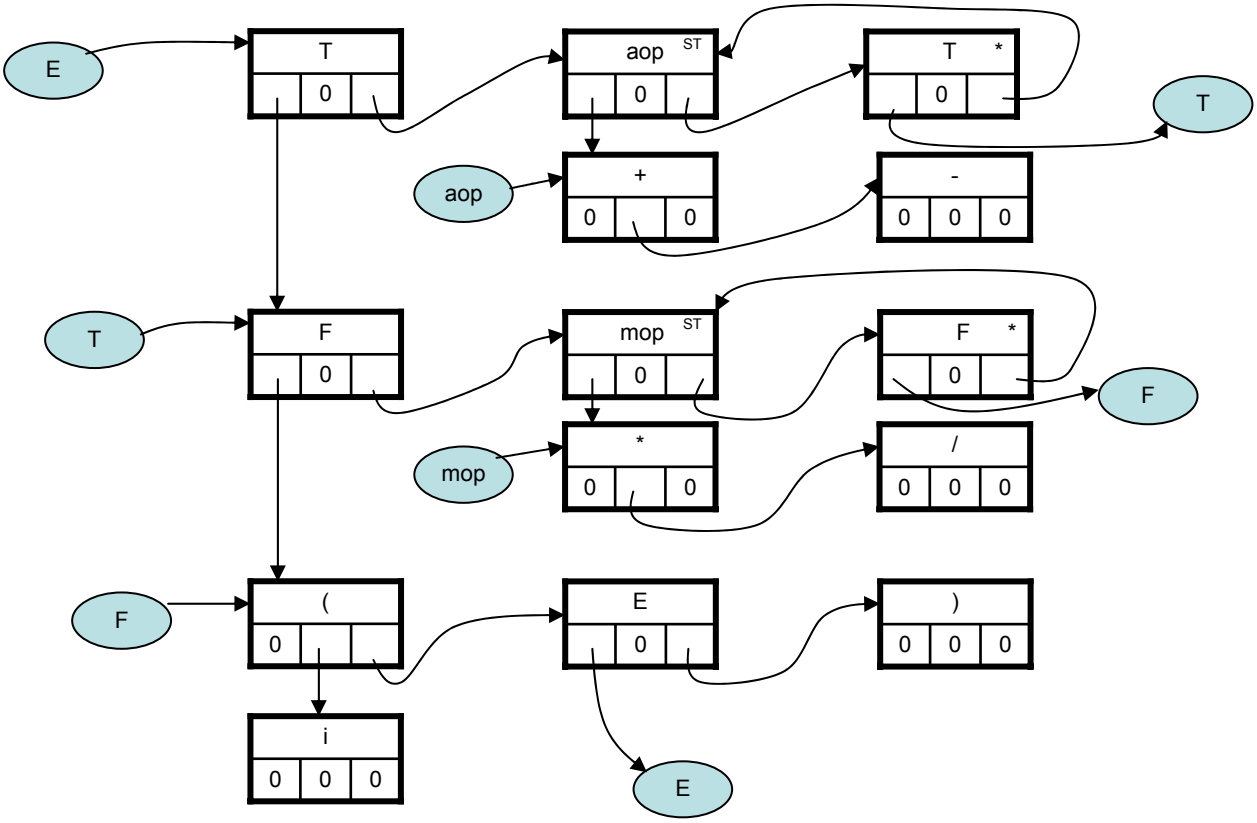
Dabar šiek tiek pakeiskime gramatiką – panaikinkime kairiąją rekursiją:

```
E ::= T { <aop> T}
T ::= F {<mop> F }
F ::= ( E ) | i
```

```
<aop> ::= + | -
<mop> ::= * | /
```

Šiek tiek pakeisime ir sintaksinį grafą. Iteracijai žymėti mazguose įvesime naujus žymenis.
ST – reiškia, kad mazgo simbolis pradeda iteracijos grandį.
* - reiškia, kad simbolis baigia iteracijos grandį.

Gausime tokį sintaksinį grafą:



Žemėjantis analizatorius be grįžimų

Analizatoriaus programa būtinai turi dirbti be grįžimų. Tai būtina, nes reikės susieti sintaksę su semantika. Analizuojant programą bus vykdomos ir semantinės užduotys. Pavyzdžiui identifikatoriai įsimenami vardų lentelėje, atliekant aritmetinius veiksmus tikrinama, ar atitinka operandų tipai, ir pan. Grįžimo atveju, reikėtų naikinti visą šią informaciją.
Kad išvengti grįžimų, reikia taip pat tikrinti ir kontekstą, kad iš karto parinkti reikalingą taisyklę. Paprastai kaip kontekstas tikrinamas sekantis neuždengtas simbolis pradinėje programoje. Toliau aprašomas analizatoriaus be grįžimų variantas.

Rekursyvus žemėjantis analizatorius

Šis analizatorius kiekvienam neterminalui turi po rekursyvią procedūrą. Kiekviena tokia procedūra atlieka analizę frazių, išvedamų iš to neterminalo. Procedūrai yra pateikiama nuo kurios pradinės programos vietos reikia pradėti ieškoti frazės, išvedamos iš neterminalo. Kad analizės metu nebūtų grįžimų, kaipo kontekstas tikrinamas simbolis, einantis po atpažintos frazės dalies.

Tarkime, kad turime tokią gramatiką:

```
<sakinys>::=<kintamasis>:=<išraiška>
           | IF <išraiška> THEN <sakinys>
           | IF <išraiška> THEN <sakinys> ELSE <sakinys>
<kintamasis>::= i | i(<išraiška>)
<išraiška>::= <termas> | <išraiška>+<termas>
<termas>::= <daugiklis> | <termas>*<daugiklis>
<daugiklis>::=  <operandas> | (<išraiška>)
<operandas>::=<identifikatorius> | <konstanta>
```

Užrašysime procedūras neterminalams iš šios gramatikos VAR, EXPR, TERM, FACTOR, STATE. Naudosime šiuos globalius kintamuosius ir papildomas procedūras:

NXTSYMB – globalus kintamasis, saugantis naują simbolį iš pradinės programos.
SCAN – procedūra, naują pradinės programos simbolį priskirianti kintamajam NXTSYMB.
ERROR – procedūra, apdorojanti klaidos situaciją.

Tam kad kalbos sakiniui pradėti sintaksinę analizę, reikia užrašyti tokią programą:
`SCAN; STATE;`

Procedūros:

```
PROCEDURE STATE
IF NXTSYMB = "IF"
    THEN BEGIN SCAN; EXPR;
               IF NXTSYMB ≠ "THEN"
                   THEN ERROR
                   ELSE BEGIN SCAN; STATE;
                              IF NXTSYMB ≠ "ELSE"
                                 THEN BEGIN SCAN;STATE; END
                         END
         END

    ELSE BEGIN  VAR;
                IF NXTSYMB ≠ ":="
                    THEN ERROR
                    ELSE  BEGIN SCAN;EXPR;  END;
         END;
```

```
PROCEDURE VAR;
IF NXTSYMB ≠ "i"
      THEN ERROR
      ELSE BEGIN
               SCAN;
               IF NXTSYMB = "(" THEN
                   BEGIN
                      SCAN;EXPR;
                      IF NXTSYMB ≠ ")"
                           THEN ERROR
                           ELSE SCAN
                   END
            END;
```

```
PROCEDURE EXPR;
BEGIN
    TERM;
    WHILE NXTSYMB = "+" DO
            BEGIN
                SCAN;
                TERM;
            END
END
```

```
PROCEDURE TERM;
BEGIN
```

```
PROCEDURE FACTOR;
IF NXTSYMB = "("
```

```
   FACTOR;
   WHILE NXTSYMB = "*" DO
           BEGIN
               SCAN;
               FACTOR;
           END
END
```

```
       THEN BEGIN
              SCAN;
              EXPR;
              IF NXTSYMB ≠ ")"
                   THEN ERROR
                   ELSE SCAN;
            END
      ELSE VAR;
```

Kylantys analizatoriai

Naudojant šį metodą sentencialinėje frazėje ieškoma pati kairioji paprastoji frazė u, kuri naudojant taisyklę U ::= u redukuojama į U. Atliekant kylančią analizę, pagrindinės problemos yra dvi – kaip rasti pagrindą ir į kokį neterminalą jį redukuoti.
Toliau mes bandysime išspręst šias problemas gramatikoms, vadinamoms paprasto dominavimo gramatikomis.

Dabar sakydami „sentencialinė forma“ turėsime omeny „kanoninė sentencialinė forma“.

Galėtume rasti pagrindą, nagrinėdami sentencialinę formą iš kairės į dešinę, vis imdami po du gretimus simbolius, ir tikrindami, ar pirmasis simbolis yra pagrindo pabaiga(angl. - *tail*).O pagrindo pradžią (angl. - *head*) galėtume rasti grįždami atgal.

Tarkime, kad turime sentencialinę formą …RS… . Arba R, arba S , arba abu simboliai priklauso pagrindui. T.y. galimos trys galimybės:

1. R- priklauso pagrindui, o S nepriklauso. Tai žymėsime taip: R o> S ir sakysime, kad R dominuoja prieš S. R yra pagrindo pabaigos simbolis tam tikroje taisyklės dešiniojoje pusėje: U ::= … R. S turi būti terminalas (nes išvedimas kanoninis).
2. R ir S priklauso pagrindui. Žymėsime R ^{o}= S ir sakysime, kad R ir S lygūs dominavimo prasme. Gramatikoje turi būti taisyklė U ::= … RS … .
3. S priklauso pagrindui, o R nepriklauso. Žymėsime R <o S ir sakysime, kad S dominuoja prieš R. S turi būti pradžios simbolis tam tikroje taisyklės dešiniojoje pusėje:
   U ::= S… .

Pastebėsime, kad nė vienas santykis nėra simetrinis

Jei neegzistuoja tokios sentencialinės formos, kur simboliai R ir S eitų greta, tai sakoma, kad dominavimo santykis tarp šių simbolių neapibrėžtas.

Paprasto dominavimo gramatikos
Gramatika vadinama paprasto dominavimo, jei

1. Betkuriai simbolių porai apibrėžtas tik vienas dominavimo santykis, t.y. dominavimo santykiai yra vienareikšmiai.
2. Taisyklių dešiniosios pusės yra unikalios.

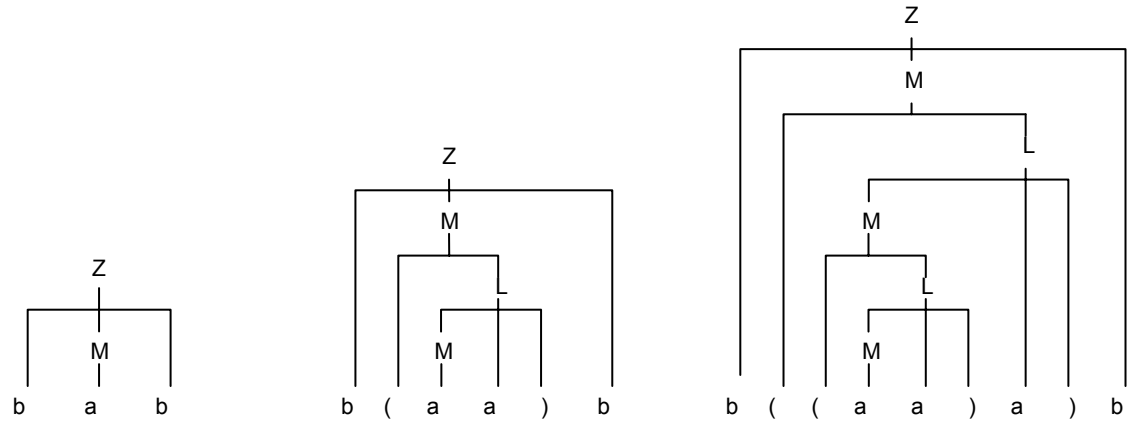
Pavyzdys.

Turime gramatiką `G[Z]:`

```
Z ::= bMb
```

```
M ::= (L | a
L ::= Ma)
```

Tai yra paprasto dominavimo gramatika. Kalbos L (G) sakiniai: bab, b(aa)b, b((aa)a)b, … , b((…(aa)a)…a)b, …. Nubrėžkime keleto sakinių sintaksinius medžius:



Nubrėšime šios gramatikos paprasto dominavimo santykių matricą:

	Z	b	M	L	a	(	)
Z							
b			$\overset{o}{=}$		<o	<o	
M		$\overset{o}{=}$			$\overset{o}{=}$		
L		o>			o>		
a		o>			o>		$\overset{o}{=}$
(			<o	$\overset{o}{=}$	<o	<o	
)		o>			o>		

Sentencialinės formos $S_1\ S_2\ \dots\ S_n$ pagrindas yra eilutė $S_i\ S_{i+1}\ \dots\ S_j$ jeigu:

1. $S_{i-1} <o S_i$
2. $S_i \overset{o}{=} S_{i+1} \overset{o}{=} S_{i+2} \overset{o}{=} \dots \overset{o}{=} S_j$
3. $S_j o> S_{j+1}$

Išrinkime eilutę b(aa)b, naudodamiesi santykių matrica.

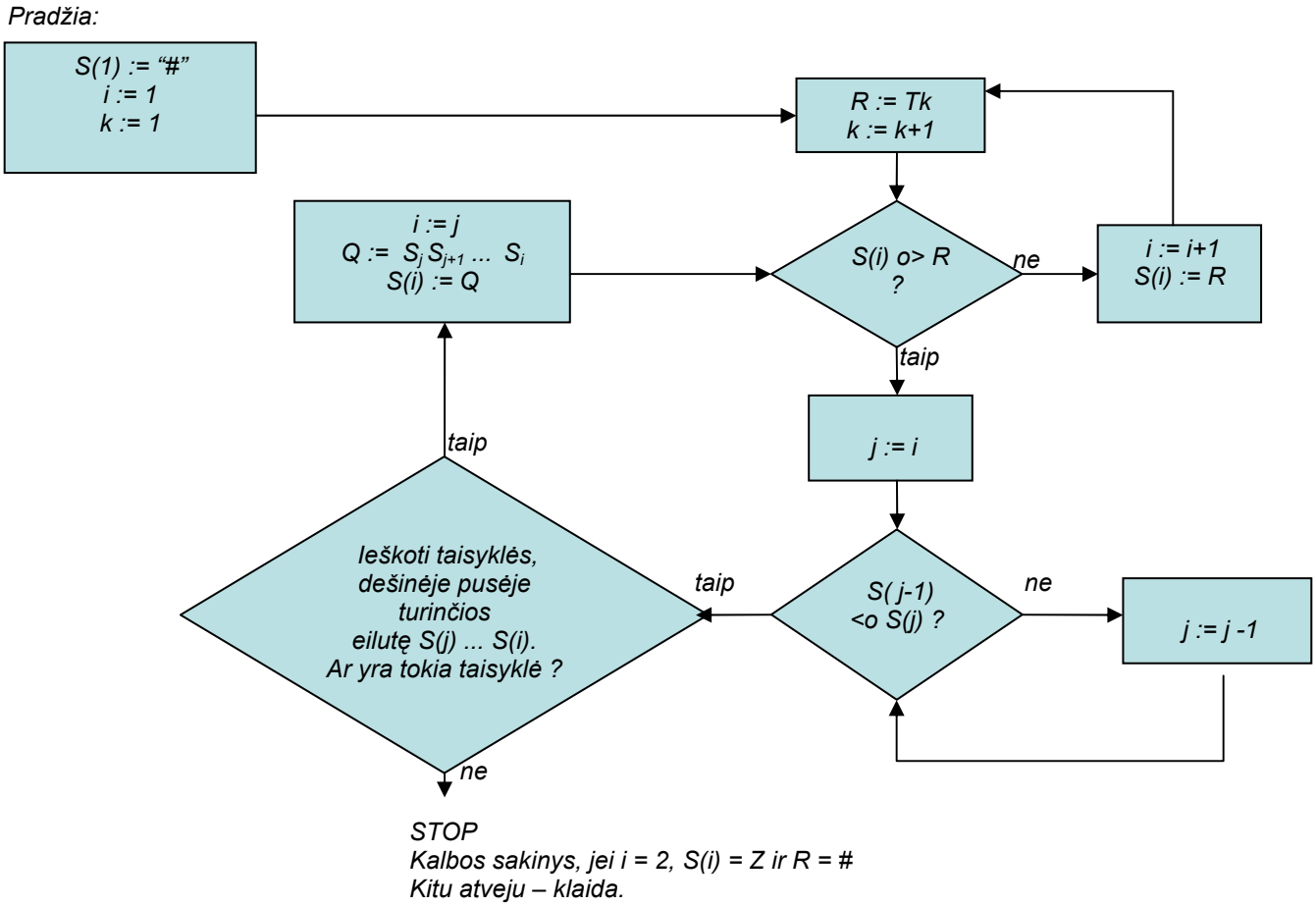
Žingsnis	Sentencialinė forma	Pagrindas	Į kokį simbolį redukuojama	Išvedimas
1	b (a a) b <o <o o> $\overset{o}{=}$ o>	a	M	b (M a) b $\Rightarrow$ b(a a) b
2	b (M a) b <o <o $\overset{o}{=}$ $\overset{o}{=}$ o>	M a)	L	b (L b $\Rightarrow$ b (M a) b
3	b (L b <o <o o>	(L	M	b M b $\Rightarrow$ b (L b
4	b M b $\overset{o}{=}$ $\overset{o}{=}$	b M b	Z	Z $\Rightarrow$ b M b

Analizės algoritmas

Pradinės eilutės simboliai iš kairės į dešinę rašomi tol, kol viršutinis steko simbolis dominuos prieš ateinantį naują simbolį. Tada pagrindas jau yra steke. Steke ieškoma pagrindo pradžios – simbolio, kuris dominuos gretimą, anksčiau į steką įrašytą simbolį. Nustačius pagrindą, surandama taisyklė ir pagrindas pakeičiamas atitinkamu neterminalu. Tai kartojama, kol steke belieka pradinis gramatikos simbolis, o ateinantis naujas simbolis yra „#“ (juo žymėsime sakinio pabaigą).

Analizatoriaus schema

Naudojame šiuos žymenis:
S – stekas. Jo skaitliukas – i.
j – steko indeksas. Naudojamas, kai reikia kreiptis į keletą viršutinių steko elementų.
T_1 T_2 … T_n – analizuojamas sakinys. Pradedant analizę, steke užrašomas simbolis # - žymėti sakinio pradžiai. Toks pat simbolis užrašomas ir į steką užrašius visą sakinį. Visi gramatikos simboliai dominuoja prieš #.
Q ir R – kintamieji saugoti simboliams darbo metu.



Analizuokime sakinį b(aa)b, naudodamiesi šia schema.

Žingsnis	S_j S_{j+1}…	Dominavimo santykis	R	T_j T_{j+1} …
0	#	<o	b	(a a) b #

1	# b	<o	(	a a) b #
2	# b (	<o	a	a) b #
3	# b (a <o	o>	a	) b #
4	# b (M	°=	a	) b #
5	# b (M a	°=	)	b #
6	# b (M a) <o	o>	b	#
7	# b (L <o	o>	b	#
8	# b M	°=	b	#
9	# b M b <o	o>	#	
10	# Z	o>	#	

Santykiai

Binarinis santykis aibėje.

Jei tarp tam tikrų elementų c ir d kažkokioje aibėje yra santykis R rašysime cRd (infiksinė užrašymo forma). Santykį galima įsivaizduoti kaip aibę sutvarkytų porų, kuriems tas santykis teisingas, t.y. $(c,d) \in R$ tada ir tik tada, kai cRd.

Pvz. santykis MAŽIAU: i < j, natūralių skaičių aibėje.

Sakoma, kad santykis R *apima* santykį P, jei iš to, kad $(a,b) \in R$ seka, kad $(a,b) \in P$.

Santykis, *atvirkštinis* R ir užrašomas R^{-1} , yra apibrėžiamas taip: a R^{-1} b tada ir tik tada, kai bRa. Pvz. Santykis MAŽIAU $^{-1}$ = DAUGIAU arba $<^{-1} = >$.

Santykis R *vadinamas* refleksyviu, jei teisinga cRc visiems elementams aibėje. Pvz. santykis MAŽIAU ARBA LYGU (≤) teisingas visiems skaičiams natūralių skaičių aibėje.

Santykis R vadinamas *tranzityviu*, jei iš to, kad aRb ir bRc seka, kad aRc.

Kiekvienam santykiui R galima apibrėžti santykį R^+, kuris vadinamas santykio R *tranzityviu uždariniu.*

Dviem santykiams R ir P, apibrėžtiems toje pačioje aibėje galime apibrėžti naują santykį RP, vadinamą santykių R ir P *sandauga*: cRPd tada ir tik tada, kai egzistuoja toks elementas e, kad teisinga cRe ir ePd. Binarinių santykių sandauga yra asociatyvi, t.y. R(PQ) = (RP)Q betkokiems santykiams R, P ir Q.

Pvz.
aRb tada ir tik tada, kai b = a + 1.
aPb tada ir tik tada, kai b = a + 2.

Aišku, kad aRPb tada ir tik tada, kai egzistuoja toks c, kad aRc ir cPb.
aRc tada ir tik tada, kai c = a + 1; cPb tada ir tik tada, kai b = c + 2
Tada aRPb tada ir tik tada, kai b = (a+1) +2 = a + 3.

Galima apibrėžti santykio R laipsnius:
$R^1 = R$, $R^2 = RR$, $R^n = RR^{n-1} = R^{n-1}R$, $n > 0$.

Santykį R^0 apibrėžiame kaip vienetinį santykį: a R^0 b tada ir tik tada, kai a = b.

a R^+ b tada ir tik tada, kai egzistuoja toks $n > 0$, kad a R^n b.
Turėdami omeny, kad santykis – aibė sutvarkytų porų, tai tranzityvų uždarinį galima užrašyti taip:
$R^+ = R^1 \cup R^2 \cup R^3 \ldots$.

Apibrėžkime *refleksyvų tranzityvų uždarinį* R*:
$R^* = R^0 \cup R^1 \cup R^2 \ldots$.

Santykių naudojimas su gramatikos elementais
Išsiaiškinę santykio sąvoką, galime suprasti, kad U $\Rightarrow$ u reiškia santykį tarp simbolio U ir eilutės u. Galime jį pavadinti išvedimo santykiu. O simbolis $\Rightarrow+$ reiškia santykio $\Rightarrow$ tranzityvų uždarinį. Atitinkamai $\Rightarrow*$ reiškia santykio $\Rightarrow$ refleksyvų tranzityvų uždarinį.

Tarkime, kad turime gramatiką ir neterminalinį simbolį U.
Galime apibrėžti aibę simbolių, kurie yra pirmieji eilutėse, išvedamose iš U.

Pav. Jei U $\Rightarrow+$ Sx, tai eilutė Sx yra išvedama iš U ir S priklausytų tai aibei.

Pažymėkime ją H(U) (angl. *head*) . Aibę H(U) galime apibrėžti tokiu būdu:

```
H(U) ::= { S | U ⇒+ S x}
```

Čia x ─ bet kokia (gali būti ir tuščia) eilutė.

Reikia pastebėti, kad apibrėžime santykis $\Rightarrow+$ yra užduodamas begalinei aibei eilučių, priklausančių žodynui V, ir nors pati aibė H(U) yra baigtinė, gali kilti problemų ją sudarant.

Apibrėžkime santykį `PIRMAS` baigtiniame žodyne `V`.
`U PIRMAS S` tada ir tik tada, kai egzistuoja taisyklė `U::= S … .`

Tada U PIRMAS$^+$ S tada ir tik tada, kai egzistuoja netuščia taisyklių seka:
`U ::= S1 … , S1::= S2 … , S2 ::= S3 …, … , Sn ::= S … .`
Taigi, matome, kad `U PIRMAS`$^+$ `S` tada ir tik tada, kai `U` $\Rightarrow+$ `S … .`

Dabar galime naujai apibrėžti aibę `H(S)`:

```
H(U) ::= { S |  (U,S) ∈ PIRMAS+ }
```

Pavyzdys.
Pailiustruosime, kaip kiekvienai gramatikos taisyklei nustatyti santykį `PIRMAS`:

```
A ::= Af          A FIRST A
```

```
A ::= B            A FIRST B
B ::= DdC          B FIRST D
B ::= De           B FIRST D
C ::= e            C FIRST e
D ::= Bf           D FIRST B
```

Aibėje $FIRST^+$ turėsime : (A,A), (A,B), (A,D), (B,D), (B,B), (C,e), (D,D), (D,B).
Taip pat gauname tokias aibes:

```
H(A) = {A,B,D}
H(B) = {B,D}
H(D) = {B,D}
H(C) = {e}
```

Analogiškai galime apibrėžti santykį PASK (paskutinis) ir aibę T(U) (angl. - *tail*):

U PASK S tada ir tik tada, kai egzistuoja taisyklė U ::= … S.
T(U) ::= { S | (U,S) ∈ $PASK^+$}

Bulio matricos
Santykiams vaizduoti naudosime Bulio matricas. Bulio matricos elementų reikšmės yra 0 arba 1. Bulio n – tos eilės matricų sudėtis atliekama naudojant operaciją ARBA atitinkamiems matricos elementams.
Matricos D = B + C elementas D(i,j) randamas taip:

```
D(i,j) := if  B(i,j) = 1 then 1 else C(i,j)
```

```
{ D(i,j) := B(i,j) or C(i,j) }
```

Bulio matricų sandauga atliekama taip pat kaip ir skaitinių matricų sandauga, tik vietoje sandaugos naudojama operacija IR, o vietoje sumos – operacija ARBA.
Jei B ir C yra n-tos eilės matricos, o D= BC, tai D(i,j) apskaičiuojamas taip:

```
D(i,j) := B(i,1) * C(1,j) + B(i,2) * C(2,j) + … + B(i,n) * C(n,j).
```

Čia * reikšmė:
a * b ::= **if** a = 0 **then** 0 **else** b

```
{ D(i,j) := B(i,1) and C(1,j) or B(i,2) and C(2,j) or … or B(i,n) and C(n,j)}
```

Santykių matricos
Tarkime, kad santykis R apibrėžtas aibėje V, kurioje yra n simbolių:
$V = \{S_1, S_2, \dots, S_n\}$.
Galime santykį R pavaizduoti n – osios eilės Bulio matrica B.
B(i,j) = 1 tada ir tik tada, kai $S_i R S_j$.

Pavyzdys.
Sudarykime santykio PIRMAS matricą gramatikai G[Z]

Z ::= bMb
M ::= (L | a
L ::= Ma)

PIRMAS	Z	b	M	L	a	(	)
Z	0	1	0	0	0	0	0
b	0	0	0	0	0	0	0
M	0	0	0	0	1	1	0
L	0	0	1	0	0	0	0
a	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0

Santykio R^{-1} matrica gaunama transponuojant santykio R matricą.

Jeigu B – santykio R n – tos eilės Bulio matrica, tai matrica B^+ apskaičiuojama taip:

$B^+ := B + BB + BBB + \ldots + B^n$

vaizduoja santykio R tranzityvų uždarinį R^+.

Warshall S. algoritmas. Tranzityvaus uždarinio matricą taip pat galima gauti naudojantis šiuo efektyviu algoritmu:

```
for i=1, n do
   begin for j=1,n do
            begin  if  B(j,i) = 1 then
                       begin  for k=1, n  do
                                  B(j,k) := B(j,k) + B(i,k)
                       end
            end
   end
```

Raskime santykio PIRMAS tranzityvaus uždarinio $PIRMAS^+$ matricą, gramatikai G[Z]:

$PIRMAS^+$	Z	b	M	L	a	(	)
Z	0	1	0	0	0	0	0
b	0	0	0	0	0	0	0
M	0	0	0	0	1	1	0
L	0	0	1	0	1	1	0
a	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0

Dominavimo santykių apibrėžimas
(1,1) dominavimas.
Tarkime, kad duota gramatika G. Dominavimo santykiai tarp gramatikos žodyno V simbolių R ir S apibrėžiami taip:

1. $R \,{}^{o}{=}\, S$ tada ir tik tada, kai gramatikoje G yra taisyklė U ::= … RS …
2. $R <o S$ tada ir tik tada, kai egzistuoja taisyklė U ::= … RV …, tokia, kad teisinga V $PIRMAS^+$ S

3. R o> S tada ir tik tada, kai S – terminalas ir egzistuoja taisyklė U ::= … VW … , tokia, kad teisinga V $PASK^+$ R ir W $PIRMAS^*$ S.

Dominavimo santykis <o lygus santykių $^o=$ ir $PIRMAS^+$ sandaugai:

$$R<oS \sim R(^o=)(PIRMAS^+)S$$

Tegu I – vienetinis santykis. R o> S tada ir tik tada, kai S – terminalas ir

$$R((PASK^+)^{-1})(^o=)(I + PIRMAS^+)S$$

Operatorių dominavimas

Kalbėdami apie paprasto dominavimo gramatikas, apibrėždavome santykius tarp visų gramatikos simbolių – tiek operandų, tiek operatorių. Operatorių dominavimo metode apibrėžiami dominavimo santykiai tik tarp operatorių. Šis metodas paprastai naudojamas aritmetinių išraiškų analizei.
Pvz. 2 + 3 * 5.
Reikia nurodyti, kad pirma būtų atliekama daugyba, o ne sudėtis.

Analizatoriaus programa naudoja ne vieną, o du stekus – operatorių stekas OPTOR ir operandų stekas OPAND.
OPTOR saugo binarinius ir unarinius operatorius, skliaustus, žymeklį - #, kuriuo prasideda ir baigiasi kiekviena išraiška.
OPAND – saugo identifikatorius ir kitus operandus.

Taip pat naudojamos dvi sveikaskaitinės funkcijos f ir g, kuriomis apibrėžiami dominavimo santykiai.

Algoritmas

S_1 – viršutinis steko OPTOR simbolis.
S_2 – naujas simbolis iš pradinės programos.

Algoritmo žingsniai:
1. Jei S_2 – identifikatorius, tai užrašyti jį į operandų steką OPAND ir praleisti 2 ir 3 žingsnius.
2. Jei $f(S_1) <= g(S_2)$, tai S_2 užrašyti į operatorių steką OPTOR ir paimti naują pradinės programos simbolį.
3. Jei $f(S_1) >= g(S_2)$, iškviesti semantinę programą, kuri nustatoma pagal S_1. Ši programa atlieka semantinį apdorojimą (lentelių pildymas ir kt.), išstums iš steko OPTOR operatorius S_1 ir , galbūt, kitus simbolius, išstums iš OPAND operatorius, susijusius su tais operandais ir išsaugos rezultatą gautą dėka operatoriaus S_1. Tai atitinka sentencialinės formos pagrindo redukciją.

Pavyzdys.
Turime aritmetinių išraiškų gramatiką:

```
Z : : = E #
E : : = T+E | T
T : : = F*T | F
```

```
F : : = (E) | i
```

Atliksime sakinio #A+B+C# analizę. Analizės metu priskirsime funkcijoms f ir g reikšmes.

Žingsnis	Stekas OPAND	Stekas OPTOR	S_1	Operatorių dominavimas	S_2	Eilutės likutis	Funkcijų f ir g reikšmės	
1		#	#		A	+B+C#		
2	A	#	#	f(#) < g(+)	+	B+C#	f(#)=1	g(+)=4
3	A	# +	+		B	+C#		
4	AB	# +	+	f(+) > g(+)	+	C#	f(+)=5	
5	T_1	#	#	f(#) < g(+)	+	C#		
6	T_1	# +	+		C	#		
7	T_1C	# +	+	f(+) > g(#)	#			g(#)=1
8	T_2	#	#	f(#) = g(#)	#			
	T_2	# #	#					

Analizuojant prireikė dviejų laikinų kintamųjų T_1 ir T_2 .

Pastebėsime, kad būtinai reikalingos *dvi* sveikaskaitinės funkcijos. Tai, kad negalima išsiversti su viena funkcija, rodo ši situacija:

(A) * B h()) > h(*)
(A * B) h(*) > h())

Operatorių dominavimo gramatikos
Gramatika vadinama *operatorine,* jei joje neegzistuoja tokios taisyklės U ::= … VW … , kur V ir W yra neterminalai.

Operatorinės gramatikos sentencialinė forma turi pavidalą:
N_1 T_1 N_2 T_2 … N_n T_n N_{n+1}
Čia N_i ─ neterminaliniai simboliai, T_i ─ terminaliniai. Kai kurie N_i gali būti tušti.

Tegu G – operatorinė gramatika, R ir S – betkokie du operatoriai (terminalai), o V ir W – neterminalai. Tada:

1. R ^{o}= S tada ir tik tada, kai egzistuoja tokio pavidalo taisyklė: U::= … RVS … .
2. R <o S tada ir tik tada, kai egzistuoja tokio pavidalo taisyklė: U::= … RW …, kur W ⇒+ S … arba W ⇒+ VS … .
3. R o> S tada ir tik tada, kai egzistuoja tokio pavidalo taisyklė: U ::= … WS …, kur W ⇒+ … R arba W ⇒+ … VS .

Operatorinė gramatika vadinama *operatorių dominavimo* gramatika, jei visų taisyklių dešiniosios pusės unikalios, taisyklės yra vienareikšmės.
Gramatika vadinama operatorių dominavimo gramatika*,* jei tarp betkokių dviejų terminalų yra apibrėžtas ne daugiau kaip vienas dominavimo santykis.

Pavyzdys.
Imkime gramatiką G[E]:

```
E :: = E #
```

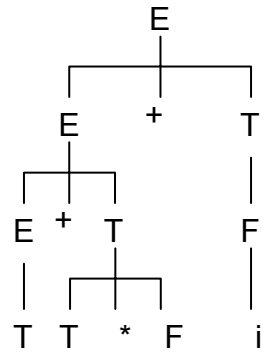
```
E ::= T+E | T
T ::= F*T | F
F ::= (E) | i
```

Sudarykime jai operatorių dominavimo matricą:

	+	*	(	)	i
+	o>	<o	<o	o>	<o
*	o>	o>	<o	o>	<o
(	<o	<o	<o	$^{o}=$	<o
)	o>	o>		o>	
i	o>	o>		o>	

Sentencialinėje formoje *pirmine fraze* vadinama tokia frazė, į kurią įeina bent vienas terminalas, o pati ši frazė neturi kitų pirminių frazių.

Pavyzdys.
Imkime sentencialinę formą `#T+T*F+i#` gramatikoje G[Z].



Joje yra šios frazės: T+T*F+i, T+T*F, i, T, T*F.
Pirminės frazės yra: T*F ir i.

Operatorių dominavimo gramatikoje pati kairiausia pirminė frazė yra tokia kairiausia eilutė $N_j\, T_j\, N_{j+1}\, T_{j+1}\, \dots\, N_i\, T_i\, N_{i+1}$, kad

$T_{j-1} <^{o} T_j$, $T_j \,^{o}= T_{j+1}$, $T_{j+1} \,^{o}= T_{j+2}$, … , $T_{i-1} \,^{o}= T_i$, $T_i \,o> T_{i+1}$.

Pavyzdys.
Naudodamiesi dominavimo matrica, analizuosime sentencialinę formą T + T * F + i.

Žingsnis	Sentencialinė forma	Santykiai	Pirminė frazė	Simbolis, į kurį redukuojama
1	# T+T*F+i #	# <o + <o * o> +	T*F	T
2	# T+T+i #	# <o + o> +	T+T	E
3	# E+i #	# <o + <o i o> #	i	F
4	# E+F #	# <o + o> #	E+F	E

Lentelėje parodyta, į kokį neterminalą redukuojama kiekviena iš pirminių frazių. Visada redukuojama pati kairiausia pirminė frazė. Atkreipkite dėmesį, kad jos ieškant, visai nekreipiama dėmesio kokie *neterminalai* yra steke.

Aukštesnės eilės dominavimas

Taip pat vadinamas (1,2) (2,1) dominavimu.

Aukštesnės eilės dominavimas leidžia kurti praktiškesnius analizatorius. Kai ieškoma pagrindo pabaiga, naudojami santykiai $\circ>$ ir $\leq\circ$. Nustatyti pagrindo pradžiai naudojami santykiai $\circ\geq$ ir $<\circ$. O pagrindui nustatyti naudojamas simbolių trejetas.

Tegu G – gramatika. R, S ir T – simboliai. Tada:

1) RS $\circ>$ T, jei egzistuoja kanoninė sentencialinė forma … RST …, kur S - pagrindo pabaiga.
2) RS $\leq\circ$ T, jei egzistuoja kanoninė sentencialinė forma … RST …, kur T įeina į pagrindą.
3) R $<\circ$ ST, jei egzistuoja kanoninė sentencialinė forma … RST …, kur S – pagrindo pradžia.
4) R $\circ\geq$ ST, jei egzistuoja kanoninė sentencialinė forma … RST …, kur S įeina į pagrindą.

Dominavimas (1,1) negali išspręsti kai kurių konfliktinių situacijų.

Pavyzdys.

Imkime gramatiką G[E]:

```
E ::= E
E ::= E+T | T
T ::= T*F | F
F ::= (E) | i
```

Dominavimas (1,1) neleidžia nustatyti vienareikšmiško santykio tarp simbolių + ir T:

1) + $\circ=$ T taisyklėje E : : = E+T. Pvz. sent. f. E+T+ …
2) + $\leq\circ$ T taisyklėje T : : = T*F. Pvz. sent. f. E+T* …

Nustatysime kai kuriuos (1,2) (2,1) dominavimo santykius gramatikai G[E]:

```
+T o>  )        (( <o= T        ( <o E+        ( o≥ E)
(F  o>)         (( <o= F        ( <o F+        T o≥ +T
+F o>  )        (( <o= i        ( <o i+        F o≥ +F
*F o>  )       +T <o= *        + <o T*        i o≥ +F
(i o>  )        (E <o= )        + <o (E        + o≥ T+
```

Gramatika, kurios dešiniosios pusės unikalios ir kurios dominavimo (1,2)(2,1) santykiai yra vienareikšmiai, vadinama *(1,2)(2,1) dominavimo gramatika.*

Kanoninės sententencialinės formos $S_1\ S_2 \ldots S_n$ pagrindas yra tokia pati kairiausia eilutė $S_j\ S_{j+1} \ldots S_{i,}$ kuri tenkina sąlygas:

$$S_{j-1} <\circ S_j\ S_{j+1}$$
$$S_j \circ\geq S_{j+1}\ S_{j+2}, \ldots, S_{i-1} \circ\geq S_i\ S_{i+1}$$
$$S_{i-1}\ S_i \circ> S_{i+1}$$

Tada ji yra kokios nors taisyklės dešinioji pusė.

Analizatorius (1,2)(2,1) dominavimo gramatikoms yra analogiškas kaip ir (1,1) dominavimo gramatikoms.

Kad įgyvendinti analizatorių galima sudaryti dvi trimates matricas:
PT, kuri skirta rasti pagrindo pabaigai ir vaizduojanti santykius $\circ>$ ir $\leq\circ$.
PH, kuri skirta rasti pagrindo pradžiai ir vaizduojanti santykius $\circ\geq$ ir $<\circ$.

Tačiau naudoti trimates matricas neekonomiška, ir net neįmanoma, turint didesnes gramatikas.

Kad būtų galima (1,2)(2,1) dominavimą naudoti ir didelėms gramatikoms, reikia atlikti šiuos žingsnius:

1. Atsisakyti matricos PH.
   Daugumai sentencialinių formų pagrindų ši matrica nereikalinga, nes galima tiesiog viršutinius steko simbolius lyginti su taisyklių dešiniosiomis pusėmis. Tai netinka tik tada, kai yra tokios dvi taisyklės U ::= ux ir V ::= x.

   Tegu G – dominavimo (1,2)(2,1) gramatika, S_i – pagrindo, sudaryto iš ne mažiau kaip 2 simbolių pabaiga. Tada sentencialinės formos pagrindas bus tokia *ilgiausia* eilutė $S_j\ S_{j+1}\ldots S_i$, kuri yra kokios nors taisyklės dešinioji pusė.

   Kai yra tokios taisyklės U ::= uX ir V ::= X, tai kiekvienai tokių taisyklių porai sudaromas sąrašas trejetų (R,X,T), kuriems $R <\circ XT$. Analizės metu, sutikę santykį $RX \circ> T$ tikriname, ar yra sąraše trejetas (R,X,T). Jei yra, tai taikoma taisyklė U ::= X, kitu atveju ieškome pačios ilgiausios eilutės, atitinkančios taisyklės dešiniąją pusę. Šį trejetų sąrašą pavadinsime TH.

2. Atsisakyti matricos PT dalies.
   Kadangi analizę atliekama iš kairės į dešinę, tai betkurio pagrindo dešinėje visada yra terminalinis simbolis. Todėl galima iš matricos panaikinti visus stulpelius su neterminaliniais simboliais. (Matrica sumažėja maždaug per pusę.)

3. Kur įmanoma naudoti paprasto dominavimo santykius.
   (Paprasto dominavimo santykius dabar žymėsime su užjuodintais skrituliukais, kad atskirtume nuo (1,2)(2,1) dominavimo santykių)
   Naudojame dominavimo (1,2)(2,1) santykius $\circ>$ ir $\leq\circ$ vietoje paprasto dominavimo santykių $\bullet>$ ir $\leq\bullet$, nes pastarieji negali išspręsti tam tikrų situacijų. Todėl galime naudoti paprasto dominavimo santykius ir *tik* konfliktinėse situacijose – dominavimo (1,2)(2,1) santykius.
   Sudarysime dvimatę matricą P, kurios elementų P(i,j) reikšmės bus:

   0, kai santykis neapibrėžtas
   1, kai santykis $\leq\bullet$
   2, kai santykis $\bullet>$
   3, kai santykis nevienareikšmis - $\leq\bullet$ arba $\bullet>$

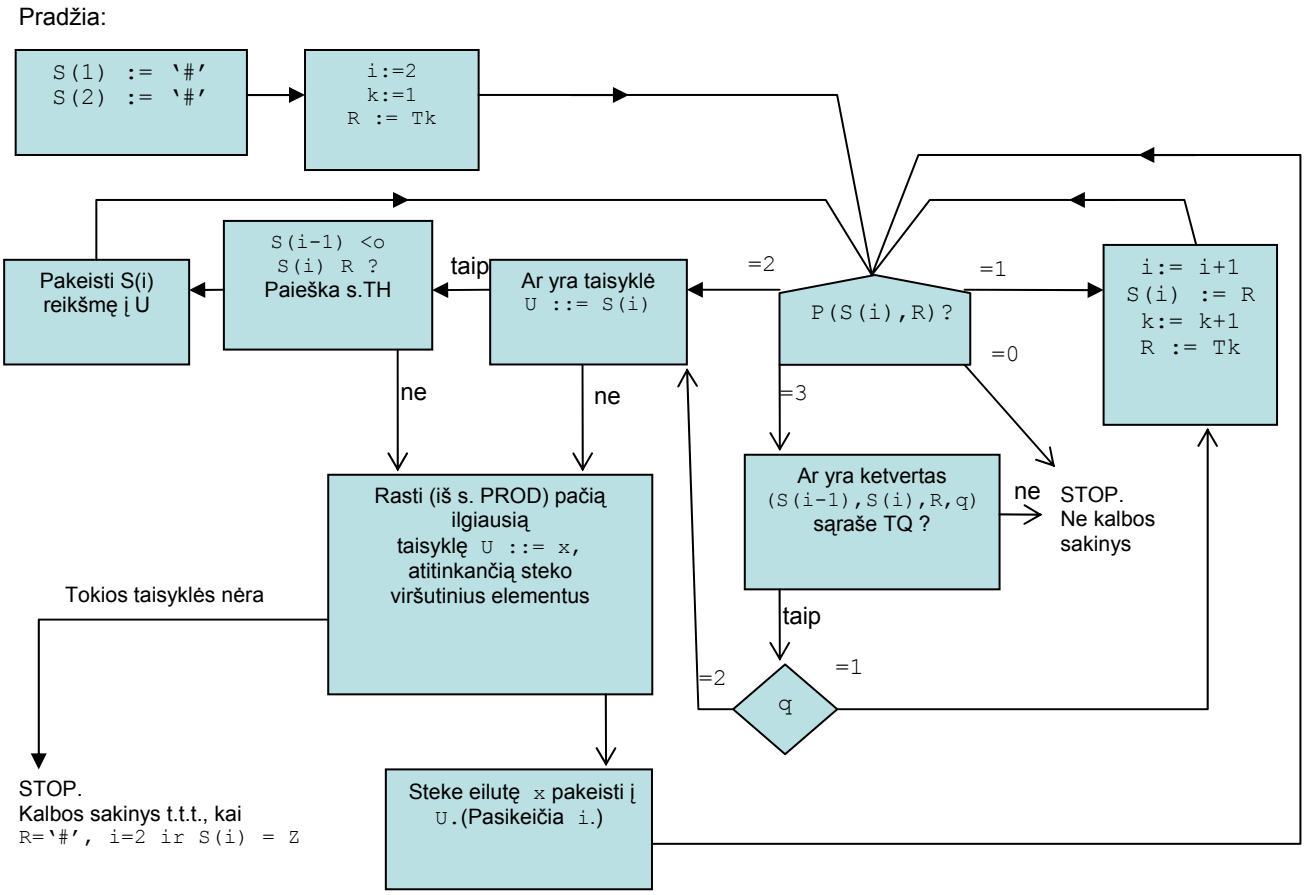
   Tuo atveju, kai P(i,j) = 3, analizatorius kreipsis į sąrašą TQ, saugantį ketvertus (S_k, S_i ,S_j, q), kur:

$q = 1$, jei $S_k \circ\!> S_i\, S_j$
$q = 2$, jei $S_k S_i \leq\!\circ\ S_j$

Algoritmas

Naudojami kintamieji ir lentelės (masyvai).

1. S – stekas. i – jo skaitliukas.
2. $T_1\, T_2 \ldots T_n$ – analizuojama eilutė.
3. P – m × k matrica . Čia m – visų simbolių skaičius, k – terminalinių simbolių skaičius. (Elementų reikšmės 1,2,3 arba 0)
4. TQ – sąrašas ketvertų (S_k, S_i ,S_j, q). (S_k, S_i - simboliai, S_j – terminalinis simbolis, q =1 arba 2)
5. PROD – taisyklių sąrašas, surūšiuotas pagal paskutinius simbolius.
6. TH – sąrašas trejetų, skirtas atvejams, kai galimas pagrindas iš vieno simbolio. (S_k, S_i ,S_j). (S_k, S_i - simboliai, S_j – terminalinis simbolis).



Pavyzdys.

Imkime gramatiką G[E]:

```
E : : = E
```

```
E : : = E+T | T
T : : = T*F | F
F : : = (E) | i
```

Šiai gramatikai reikalingos lentelės (masyvai, sąrašai):

TH:	#T#	#T+	(T+	(T)	#F#	+F#	#F+	(F+	+F+	(F)	+F)	#F*	(F*	+F*
TQ:	[*tuščias*]													
PROD:	E::= T		E::= E + T		T::= F			T::= T*F		F::= (E)		F::= i		

P	#	+	*	(	)	i
E	●>	≤●			≤●	
T	●>	●>	≤●		●>	
F	●>	●>	●>		●>	
#				≤●		≤●
+				≤●		≤●
*				≤●		≤●
(				≤●		≤●
)	●>	●>	●>		●>	
i	●>	●>	●>		●>	

Ribotas kontekstas

Ribotas kontekstas (1,1)

Tarkim, kad turime Z ⇒+ x TUR y, kur U ::= u – yra gramatikos taisyklė. Tada, jei analizės metu steke turime xTu, o naujai ateinantis simbolis yra R, tai reikėtų taikyti U ::= u.
Ar galima būtų, remiantis galimo pagrindo kontekstu T … R, nustatyti, ar tai iš tiesų yra pagrindas, ir, jei taip, tai į kokį simbolį reikėtų jį redukuoti? Kas gali būti pagrindas sentencialinėje formoje Z⇒ …TuR …. ?

Taisyklė pavidalo U ::= u vadinama *(1,1) konteksto taisykle*, jei kiekvienai simbolių porai T ir R, tokiem, kad Z ⇒+ … TuR … , sentencialinėje formoje … TuR … vienintelis galimas pagrindas yra u, ir jis gali būti redukuotas tik į U.

Gramatika vadinama *(1,1) konteksto gramatika,* jei visos jos taisyklės yra *(1,1) riboto konteksto taisyklės.*

Pavyzdys.
Gramatika

```
Z::= abU | cbV      V ::= c   U ::= c
```

nėra (1,1 konteksto gramatika, nes eilutėje abc pagal kontekstą bc negalima nustatyti kuri iš taisyklių V::= c ar U ::= c naudotina.

(1,1) konteksto gramatika:

```
Z::= aUa | bVb      V ::= c   U ::= c
```

Analizatorius

Naudojama viena lentelė, turinti tris stulpelius:
I : steko turinys, užrašomas pavidalu Tu, kur T – terminalinis simbolis arba žymeklis #, dedamas abiejuose sakinio galuose.
II: R – dešinysis kontekstas – vienas terminalinis simbolis arba #.
III: taikomos taisyklės numeris.

Algoritmas

$T_1\ T_2 \ldots\ T_n$ – analizuojamas sakinys.
S – stekas. i – jo skaitliukas.

1. i:= 1, k:=1, S(i):= #.
2. Jei viršutiniai steko simboliai atitinka nurodytą steko turinį lentelėje (I stulpelis) ir jei T_k atitinka dešinįjį kontekstą (II atitinkama stulpelio reikšmė), tai įsiminti šios eilutės numerį, kintamuoju j ir pereiti į 3 punktą. Kitu atveju – pereiti į 4 puktą.
3. Steke yra eilutė Tu, kuri atitinka eilutės j stulpelio I turinį. u redukuoti į U pagal j eilutės III stulpelyje nurodytą taisyklę.
4. Jei T_k = # pereiti į 6 punktą.
5. i := i+1; S(i):= T_k ; k:= k+1; Pereiti į 2 punktą.
6. STOP. Sakinys teisingas tada ir tik tada, kai i = 2 ir S(i) = Z.

Ribotas kontekstas (m,n)
Turime U::= u. Jei bet kurioms eilutėms v ir w, kur |v| = m ir |w| = n ir $Z \Rightarrow^*$ xvUwy (x ir y – betkokios eilutės), tarsime, kad u yra pagrindas sentencialinės formos …vuw… ir u reikia redukuoti į U. Tada ši taisyklė U ::= u yra vadinama *konteksto (m,n) taisykle.*

Gramatika vadinama *konteksto (m,n) gramatika*, jei visos jos taisyklės yra konteksto (m,n) taisyklės.

Analizatorius taip pat naudoja vieną lentelę su trimis stulpeliais. Tačiau sudaryti tą lentelę reikalingas sudėtingas konstruktorius. Didžiausias uždavinys yra panaikinti nevienareikšmiškumą, t.y. kad lentelė visais atvejais apibrėžtų vieną pagrindą, į kurį redukuojama. Tam reikia plėsti kontekstą iš kairės arba iš dešinės.
Yra dvi problemos:

- kaip nuspręsti iš kurios pusės plėsti kontekstą
- kol dar nesudaryta vienareikšmė lentelė, negalima tvirtinti, kad gramatika yra riboto konteksto.

Perėjimų matricų metodas
Šis metodas taikomas operatorinėm gramatikoms.
Turima gramatika G[<programa>]:

```
<programa> ::= <sakinys>
<sakinys> ::= IF <išraiška> THEN <sakinys>
<sakinys>::= <kintamasis>:=<išraiška>
<išraiška>::= <išraiška>+<kintamasis> | <kintamasis>
<kintamasis>::= i
```

	#	IF	THEN	:=	+	i
#						
IF						
IF <išraiška> THEN						
<kintamasis>:=						
<išraiška>+						
i						

Sudarykime perėjimų matricą, kurios eilutės atitinka taisyklių dešiniųjų pusių pradžias, besibaigiančias terminaliniais simboliais ir galinčias atsirasti steko viršūnėje. Stulpeliai atitinka terminalinius simbolius (įskaitant ir markerį #). Matricos elementai bus paprogramių numeriai. Ši paprogramė atliks reikalingą redukciją, įtrauks simbolį R į steką, perskaitys į R naują simbolį.

Pagal gramatiką G[<programa>] sudarykime kitą – ekvivalenčią gramatiką, kurios taisyklių dešiniosiose pusėse ne daugiau kaip 3 simboliai. Tokia išplėsta gramatika vadinama *augmentine gramatika.* Ji padės mums nustatyti matricos eilučių atitikmenis. Tam reikės įsivesti naujų neterminalinių simbolių. Kad atskirtume – jie bus pabraukiami. Kiekvienas naujai įvestas terminalinis simbolis bus įvedamas taip, kad būtų tam tikros *pradinės gramatikos* taisyklės dešiniosios pusės pradžia. Tokiu būdu kiekvienas šis naujai įvestas terminalinis simbolis atitinka tam tikrą perėjimų matricos eilutę.

Taisyklių keitimai (Čia T- terminalinis, U – neterminalinis, U,V – nauji neterminaliniai simboliai, y - eilutė):

(1) Jei yra taisyklė pavidalo `U::= Ty` (eilutė y gali būti tuščia), tai sukuriame `U::= T` ir visas taisykles, prasidedančias T, pavidalo `U::= Ty` keičiame taisyklėmis `U::= Uy`.
(2) Jei yra `U::= VTy ,` tai įsivedame `U::= VT` ir keičiame `U::= Uy`
(3) Jei yra `U::= UTy ,` tai įsivedame `V::= UT` ir keičiame `U::= Vy`
(4) Jei yra `U::= VVTy` , tai įsivedame `U::= VV` ir keičiame `U::= Uy`

Augmentinės gramatikos iš paprastos gavimas:
(1) keitimas kartojamas tiek kartų kiek tik įmanoma. Tada kur galima taikome (2) keitimą. Paskui kiek išeina paeiliui kartojami (3) ir (4) tipo keitimai.

Gramatikai G[<sakinys>] taikę šį algoritmą, gauname tokią augmentinę gramatiką:

```
PN1::= IF;
<sakinys>::= PN1<išraiška> THEN <sakinys>
PN2::= i;
<kintamasis>::= PN2
PN3::= <kintamasis>:= ;
<sakinys>::= <PN3> <išraiška>
PN4::= <išraiška>+;
<išraiška>::= PN4 <kintamasis>
PN5::= PN1<išraiška> THEN;
<sakinys>::= PN5 <sakinys>
```

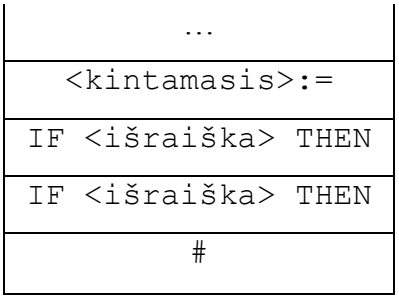
Analizatoriaus algoritmas naudos steką S. Jis saugos ne atskirus simbolius kaip anksčiau, bet simbolių eilutes – kurios yra taisyklių dešiniųjų pusių pradžios (augmentinės programos dešiniosios

pusės). Naują simbolį saugos kintamasis R, o U saugos neterminalą, į kurį buvo suvesta paskutinė paprastoji frazė.

Pavyzdys.
Tarkime, kad analizės metu steke yra toks turinys:

```
# IF <išraiška> THEN IF <išraiška> THEN <kintamasis>:= …
```

Tada stekas atrodys taip:



Kiekviename analizės etape steko viršūnėje yra eilutė, atitinkanti matricos eilutę. Naujas simbolis, esantis kintamajame R, atitinka stulpelio reikšmę taip nustatoma, kuri paprogramė bus atliekama.
Paprogramės atlieka reikalingą redukciją, įtraukia simbolį R į steką, perskaito į R naują simbolį. Naudojama procedūra SCAN, kuri užrašo naują simbolį į R. Programos pradžioje steke yra #, o U ─ tarpas. Pirmoji vykdoma paprogramė – 1.

```
1. IF U ≠ ' ' THEN ERROR;
   i:= i+1;
   S(i) := R;
   SCAN.
2. IF  U ≠ ' ' THEN ERROR;
   i:= i-1;
   U:= '<kintamasis>'.
3. IF  U ≠ '<išraiška>' OR U ≠ '<kintamasis>' THEN ERROR;
   i := i+1;
   S(i):= '<kintamasis>+';
   U:= ' '; SCAN;
4. IF U ≠ '<kintamasis>' THEN ERROR;
   i:= i-1;
   U:= '<išraiška>';
5. IF U ≠ '<programa>' OR U ≠ <sakinys> THEN ERROR;
   STOP.
6. IF U ≠ '<kintamasis>' THEN ERROR;
   U:= ' ';
   i:= i+1;
   S(i) := '<kintamasis>:=';
   SCAN.
7. IF U ≠ '<sakinys>' THEN ERROR;
   i:= i -1;
   U:= '<sakinys>'
```

```
8. IF U ≠ '<kintamasis>' OR U ≠ '<išraiška>' THEN ERROR;
    i:= i-1;
   U:= '<sakinys>';
9. IF U ≠ '<kintamasis>' THEN ERROR;
   S(i) := 'IF <išraiška> THEN';
   U:= ' ';
   SCAN.
0. ERROR; STOP.
```

Galiausiai galime užpildyti perėjimų matricą:

	#	IF	THEN	:=	+	i
#	5	1	0	6	0	1
IF	0	0	9	0	3	1
IF <išraiška> THEN	7	1	0	6	0	1
<kintamasis>:=	8	0	0	0	3	1
<išraiška>+	4	0	4	0	4	1
i	2	0	2	2	2	0

Pradinės programos vidinės formos

Vidinė programos forma – tai programos forma, kuri sukuriama, kad palengvinti objektinio kodo analizę ir generaciją, ypač tais atvejais, kai pradinė kalba sudėtinga arba atminties taupymo sumetimais.
Komandos vidinėje formoje išdėstomos ta tvarka, kuria turi būti vykdomos. Programa vidinėje formoje gali būti ir neverčiama į objektinę kalbą. Interpretatorius iškarto vykdo programą, esančią vidinėje formoje – nesukuria objektinio kodo.
Labiausiai paplitusios vidinės formos:

(1) inversinė lenkiška forma
(2) tetrados
(3) triados
(4) sintaksiniai medžiai

Paprastai visos vidinės formos susideda iš operandų ir operatorių. Skiriasi tik kaip jie apjungiami. Operandai identifikuojami nurodant tipą ir vietą atitinkamoje lentelėje. Operatoriai identifikuojami kodu.

Lenkiška forma
Naudojama aritmetinėms ir loginėms operacijoms koduoti.

Pavyzdžiai:
A*B → AB*
A*B+C → AB*C+
A*(B+C/D) → ABCD/+*
A*B+C*D → AB*CD*+

Lenkiška forma taip pat vadinama sufiksine arba postfiksine.

Lenkiškos formos gavimas:

1. Identifikatoriai rašomi ta pačia tvarka kaip ir infiksinėje formoje.
2. Operatoriai rašomi ta tvarka, kuria turi būti vykdomi.
3. Operatoriai rašomi iš karto po savo operandų.

Galima sudaryti tokias sintaksines taisykles:

```
(1)    <operandas>::= <identifikatorius> |
   <operandas><operandas><operatorius>
(2)    <operatorius>::= + | - | / | * | …
```

Unarinį minusą galima užrašyti vienu iš būdų:

- kaip binarinį operatorių. T.y. vietoje –B rašyti 0-B
- įvesti naują simbolį, pvz. @, ir pridėti dar vieną sintaksinę taisyklę:

```
(3) <operandas>::= <operandas>@
```

Tada `A+(-B+C*D)` galima užrašyti taip `AB@CD*++`

Naudojantis steku, aritmetines išraiškas lenkiškoje formoje galima apskaičiuoti vieną kartą peržiūrint išraišką iš kairės į dešinę. Skaičiuojant imamas pats kairysis simbolis, jis apdorojamas, tada imamas simbolis jam iš dešinės, ir taip toliau. Simbolio apdorojimo žingsniai:

1. Jei nagrinėjamas simbolis – identifikatorius, tai užrašome jį į steką. Paimamas sekantis simbolis.
2. Jei nagrinėjamas simbolis – binarinis operatorius, tai jis taikomas viršutiniams dviems steko elementams. Steke tie du elementai pakeičiami rezultatu.
3. Jei nagrinėjamas simbolis – unarinis operatorius, jis yra taikomas viršutiniam steko elementui. Jis yra pakeičiamas rezultatu.

Pavyzdys.
Apskaičiuosime lenkiškos formos išraišką `AB@CD*++`

Žingsnis	Išraiškos likutis	Nagrinėjamas simbolis	Sena steko būsena	Taikoma taisyklė	Nauja steko būsena
1	B@D*++	A		(1)	A
2	@CD*++	B	A	(1)	A|B
3	CD*++	@	A|B	(3)	A|-B
4	D*++	C	A|-B	(1)	A|-B|C
5	*++	D	A|-B|C	(1)	A|-B|C|D
6	++	*	A|-B|C|D	(2)	A|-B|C*D
7	+	+	A|-B|C*D	(2)	A|-B+C*D
8		+	A|-B+C*D	(+)	A+(-B+C*D)

Lenkiška forma galima užrašyti ir išraiškas su kitais operatoriais:

Priskyrimas:
`<kintamasis>:=<išraiška>`
lenkiškoje formoje:
`<kintamasis><išraiška>:=`

Pvz. `A:= B*C+D` → `ABC*D+:=`

Besąlyginis valdymo perdavimas:
`GOTO A`
lenkiškoje formoje:
`A BRL`

Čia A yra žymė. BRL – unarinis operatorius. (Branch to label)

Sąlyginis valdymo perdavimas lenkiškoje formoje bus tokio pavidalo:
`<operandas1><operandas2> BP`

(Branch on positive)
<operandas1> - aritmetinė išraiška
<operandas2> - numeris (adresas) simbolio lenkiškoje formoje.

Taip pat operatoriai:
BZ – Branch on zero
BM – Branch on minus

Sąlyginis sakinys:
`IF <išraiška> THEN <sakinys1> ELSE <sakinys2>`
lenkiškoje formoje:
`<išraiška><c1>BZ<sakinys1><c2>BR<sakinys>`

Masyvo aprašas ALGOL kalboje:
`ARRAY A [L1:U1, … ,Ln:Un]`
lenkiškoje formoje:
`L1 U1 … Ln Un A ADEC`

Čia `ADEC` – vienintelis operatorius.

Indeksuotas kintamasis ALGOL:
`A[<išraiška>, … , <išraiška>]`
lenkiškoje formoje:
`<išraiška> … <išraiška> A SUBS`

Operatoriai be operandų:
`BLOCK` – bloko pradžia
`BLCKEND` – bloko pabaiga

Pavyzdys.

Programa ALGOL kalba:

```
BEGIN     INTEGER K;
          ARRAY A[1: I-J];
          K:=0;
L:        IF I>J
          THEN K:= K + A[I-J]*6
          ELSE  BEGIN I:= I+1; I:= I+1; GOTO L END
END
```

Programa vidine (lenkiška) forma:

```
(1)       BLOCK 1 I J - A ADEC K 0 :=
(11)      I J - 29 BMZ
(16)      K K I J - A SUBS 6 * + := 41 BR
(29)      I I 1 + := I I 1 + := L BRL
(41)      BLCKEND
```

Lenkiškos formos vidinis vaizdavimas:

Operatoriai užima vieną ląstelę ir koduojami skaičiais. Identifikatoriai ir konstantos identifikuojami tipu ir adresu arba reikšme.

Operatorius	Kodas
SUBS	6
:=	7
BMZ	8
BR	9
BRL	10
BLOCK	11
BLCKEND	12
ADECK	13
+	14
*	15
-	16

Operando tipas	Operando kodas
konstanta	1
identifikatorius	2

Vidinis programos vaizdavimas:

	Vardų lentelė
1	I
2	J
3	A
4	K
5	L 25

Žodžio numeris	Žodžių turinys		Simbolis
1	11		BLOCK
2	1	1	1
4	2	1	I
6	2	2	J
7	16		–
9	2	3	A
10	13		ADEC
12	2	4	K
14	1	0	0
15	7		:=
17	2	1	I
19	2	2	J
20	16		–
22	1	45	45

Žodžio numeris	Žodžių turinys		Simbolis
23	8		BMZ
25	2	4	K
27	2	4	K
29	2	1	I
31	2	2	J
33	16		–
34	2	3	A
36	6		SUBS
37	1	6	6
39	15		*
40	14		+
41	7		:=
42	1	64	64
44	9		BR

Žodžio numeris	Žodžių turinys		Simbolis
45	2	1	I
47	2	1	I
59	1	1	1
51	14		+
52	7		:=
53	2	1	I
55	2	1	I
57	1	1	1
59	14		+
60	7		:=
61	1	5	5
63	10		BRL
64	12		BLCKEND

Tetradų forma

Patogi binarinėms operacijoms koduoti.

Bendras pavidalas:

(<operatorius>,<operandas$_1$>,<operandas$_2$>,<rezultatas>)

Pvz.
A*B ~ *, A, B, T
Čia T - kintamasis, kuriam priskiriamas operacijos rezultatas.

A*B + C*D ~ *, A, B, T_1
*, C, D, T_2
*, T_1, T_2, T_3

Unariniams operatoriams <operandas$_2$> lieka tuščiu :
–, A, , T

Programą (ALGOL):

```
BEGIN     INTEGER K;
          ARRAY A[1: I-J];
          K:=0;
L:        IF I>J
          THEN K:= K + A[I-J]*6
          ELSE  BEGIN I:= I+1; I:= I+1; GOTO L END
END
```

užrašysime tetradomis:

(1)	BLOCK
(2)	–, I, J, T_1
(3)	BOUNDS, 1, T_1
(4)	ADEC, A
(5)	:=, 0, , K
(6)	–, I, J, T_2
(7)	BMZ, 13, T_2
(8)	–, I, J, T_3
(9)	*, A[T_3], 6, T_4
(10)	+, K, T_4, T_5
(11)	:=, T_5, , K
(12)	BR, 18
(13)	+, I, 1, T_6
(14)	:=, T_6, , I
(15)	+, I, 1, T_7
(16)	:=, T_7, , I
(17)	BRL, L
(18)	BLCKEND

Triadų forma
Bendras pavidalas:
(<operatorius>, <operandas$_1$>, <operandas$_2$>)

Triada neturi rezultato lauko. Tačiau galima nurodyti triadą, kurios operacijos rezultatas dalyvauja kaip operandas:
Išraiška A + B*C

(1) *, B, C
(2) +, A, (1)

Programą (ALGOL):

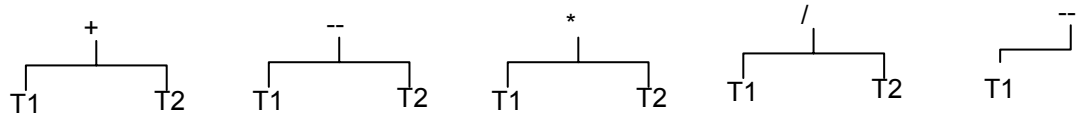
```
BEGIN    INTEGER K;
         ARRAY A[1: I-J];
         K:=0;
L:       IF I>J
         THEN K:= K + A[I-J]*6
         ELSE  BEGIN I:= I+1; I:= I+1; GOTO L END
END
```

užrašysime triadomis:

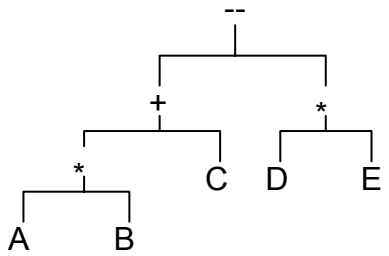
```
(1)  BLOCK
(2)  -,I,J
(3)  BOUNDS,1,(2)
(4)  ADEC,A
(5)  :=,0,K
(6)  -,I,J
(7)  BMZ,(13),(6)
(8)  -,I,J
(9)  *,A[(8)],6
(10) +,K,(9)
(11) :=,(10),K
(12) BR,(18)
(13) +,I,1
(14) :=,(13),I
(15) +,I,1
(16) :=,(15),I
(17) BRL,L
(18) BLCKEND
```

Sintaksinių medžių vidinė forma

Aritmetinėms išraiškoms medžiai apibrėžiami tokiu būdu. Paprastam kintamajam arba konstantai medis yra tiesiog kintamasis arba konstanta (medžio lapai). Jei išraiškas e_1 ir e_2 atitinka medžiai T_1 ir T_2, tai išraiškas $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, e_1 / e_2 ir $- e_1$ atitiks tokie medžiai:



Pavyzdys.
Išraiškos `A*B+C - D*E` medis:



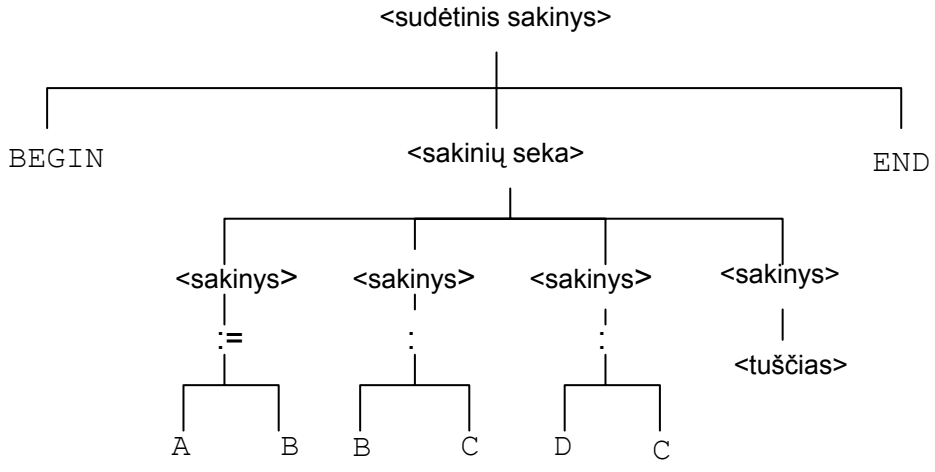
Į betkokios aritmetinės išraiškos triadas galima žiūrėti kaip į medžio vaizdavimą. Užrašykime nagrinėtos išraiškos triadas:

```
(1)       *, A, B
(2)       +, (1), C
(3)       *, D, E
(4)       –, (2), (3)
```

Paskutinė triada atitinka medžio šaknį. Kiekviena triada atitinka pomedį. Operatorius yra to pomedžio šaknis, o kiekvienas operandas – medžio lapas (konstanta arba identifikatorius) arba pomedis (numeris kitos triados).
Tačiau vaizduojant blokus, sakinius ir pan. negalime užrašyti medžio triadomis.

Pavyzdys.
Užrašykime triadas ir nubraižykime medį sudėtiniam sakiniui:

```
BEGIN A:= B; B:= C; D:= C; END
```



(1) :=,A,B
(2) :=,B,C
(3) :=,D,C

Tiesinės sritys
Optimizuoti programą dažnai atliekamas jos skaidymas į *tiesines sritis*. Tada atskirai saugomas programos grafas, kuris aprašo, kaip tos sritys siejasi tarpusavyje. Tiesinė sritis – eilė operacijų su vienu įėjimu ir vienu išėjimu (juos atitinka pirma ir paskutinė operacijos). Tiesinėje srityje visos operacijos atliekamos iš eilės – nėra valdymo perdavimo.

Padalinsime programą

```
BEGIN     INTEGER K;
          ARRAY A[1: I-J];
          K:=0;
L:        IF I>J
          THEN K:= K + A[I-J]*6
          ELSE  BEGIN I:= I+1; I:= I+1; GOTO L END
END
```

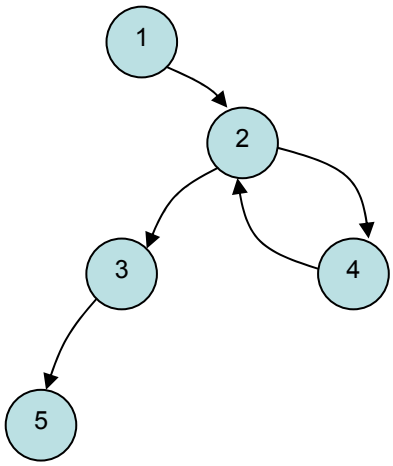
į tiesines sritis. Užrašysime tetradomis. Taip pat nubraižysime programos medį.

1. BLOCK
   −, I, J, T_1
   BOUNDS, 1, T_1
   ADEC, A
   :=, 0, , K

2. −, I, J, T_2
   BMZ, 4, 3, T_2
3. −, I, J, T_3
   *, A[T_3], 6, T_4
   +, K, T_4, T_5
   :=, T_5, , K

4. +, I, 1, T_6
   :=, T_6, , I
   +, I, 1, T_7
   :=, T_7, , I

5. BLCKEND



Pastebėsime, kad valdymo perdavimo operacijose kaip operandai dalyvauja ne tetradų numeriai, bet sričių numeriai (arba adresai). Taip pat, jei po tam tikros tiesinės srities operacijų būtinai atliekamos kažkurios kitos srities operacijos (kurios numeris nepriklauso nuo skaičiavimo rezultato), tai nereikia nurodyti, kokiai tiesinei sričiai perduodamas valdymas. Šitai nurodo programos grafas.

Semantinis analizatorius. Įvadas

Kai sintaksinis analizatorius atlieka redukciją, yra iškviečiamos semantinės programos. Semantinės programos yra sukuriamos kiekvienai gramatikos taisyklės dešiniajai pusei.

Semantinis analizatorius žemėjančiai analizei
Naudosime rekursines procedūras. Analizuosime šios gramatikos išraiškas:

```
Z::= E
E::= [−]T{(+|−)T}
T::= F{(*|/)F}
```

```
F::= I | (E)
```

Visi neterminaliniai simboliai turės atitinkamą semantinę procedūrą. Kiekvieno iš simbolių Z, E, T, F semantika yra kintamojo iš pradinės programos vardas arba laikinojo kintamojo vardas. Šį kintamąjį būtina susieti su neterminalu, kai tik į jį redukuojama. Tam kiekviena procedūra turės kintamąjį X, tipo STRING, kurį grąžins kaip neterminalo semantinę informaciją, ją iškvietusiai procedūrai.
Kiti kintamieji ir procedūros:

NXTSYMB – globalus kintamasis, saugantis naują simbolį iš pradinės programos.

SCAN – procedūra, naują pradinės programos simbolį priskirianti kintamajam NXTSYMB. Jei sutinka identifikatorių, tai į NXTSYMB užrašo I, o identifikatoriaus vardą – į NXTSEM.

ERROR– procedūra, apdorojanti klaidos situaciją.

```
PROCEDURE Z(X); STRING X;
BEGIN
  SCAN; E(X)
END
```

```
PROCEDURE E(X); STRING X;
BEGIN
  STRING Y,Z,OP;
  IF NXTSYMB='-'
          THEN BEGIN
                    SCAN; T(X);
                    j:=j+1;
                    ENTER(@,X, ,Tj);
                    Y:= Tj;
                   OP:=NXTSYMB;
               END
          ELSE BEGIN
                    T(Y);
                    OP:=NXTSYMB;
               END
          WHILE OP='+' OR OP='−' DO
               BEGIN
                    SCAN; T(Z);
                    j:=j+1;
                    ENTER(OP,Y,Z,Tj);
                    Y:= Tj;
                    OP:=NXTSYMB;
               END
          X:=Y;
END
```

```
PROCEDURE T(X); STRING X;
BEGIN
  STRING Y,Z,OP;
  F(Y); OP:= NXTSYMB;
  WHILE OP='*' OR OP='/' DO
          BEGIN
            SCAN; F(Z);j:=j+1;
            ENTER(OP,Y,Z,Tj);
            Y:=Tj ;
            OP:= NXTSYMB;
          END
  X:=Y;
END
```

```
PROCEDURE F(X); STRING X;
BEGIN
  IF NXTSYMB = 'I'
       THEN  BEGIN
               X:= NXTSEM; SCAN;
             END
       ELSE IF NXTSYMB ≠ '('
              THEN ERROR()
              ELSE BEGIN
                     SCAN; E(X);
                     IF NXTSYMB ≠ ')'
                        THEN ERROR();
                        ELSE SCAN
                   END
END
```

Infiksinės formos pervedimas į lenkišką formą

Tariama, kad kiekvieną kartą, kai sentencialinėje formoje randamas pagrindas x ir jį galima redukuoti į U pagal taisyklę U::= x, sintaksinis analizatorius iškviečia semantinę programą, susietą su šia taisykle. Semantinė programa grąžina tą lenkiškos formos dalį, kuri betarpiškai susijusi su eilute x.

1. Pagrindas visada redukuojamas, kai atliekama redukcija.
2. Jei pagrinde yra neterminalinis simbolis V, tai eilutės, kuri redukuojasi į V, lenkiška forma jau sugeneruota.

Analizuosime sakinį, priklausantį gramatikai G[Z]:

```
Z ::= E
E ::= T | E+T | E-T | -T
T ::= F | T*F | T/F
F ::= I |(E)
```

Generuojama lenkiškos formos eilutė bus kaupiama masyve P. Masyvo indeksas – p.
S(1),S(2), … , S(i) – pagrindo simboliai (esantys sintaksiniame steke S).

Sudarykime semantinę programą, susijusią su taisykle $E_1 ::= E_2 + T$.
Jei ji kviečiama, tai masyve P yra <E_1 kodas><T kodas>. Vadinasi, kad gauti lenkišką formą, reikia į masyvą rašyti '+'. Semantinės programos tekstas:

```
P(p):='+'; p:= p+1
```

Sudarykime semantinę programą, susijusią su taisykle F::= I. Čia I – „identifikatorius“. Kadangi identifikatoriai lenkiškoje formoje eina ta pačia tvarka kaip infiksinėje, ir identifikatorius dominuoja prieš savo operatorius, pagal lenkiškos formos užrašymo taisykles:

```
(1)  <operandas>::= <identifikatorius> |
     <operandas><operandas><operatorius>
(2)  <operatorius>::= + | - | / | * | …
(3)  <operandas>::= <operandas>@
```

tai mums reikia tiesiog įtraukti I į masyvą P. Programos tekstas:

```
P(p):= S(i); p:= p+1
```

Čia S(i) – viršutinis steko simbolis.

Semantinė programa, susijusi su taisykle F::=(E) nieko nedaro, nes lenkiškoje formoje skliaustų nėra, o neterminalui E lenkiška forma jau sugeneruota.
Visų taisyklių semantinės programos:

Taisyklės nr.	Taisyklė	Semantinė programa
1	`Z::= E`	[nėra]
2	`E::= T`	[nėra]
3	`E::= E+T`	`P(p):='+'; p:= p+1`
4	`E::= E–T`	`P(p):='–'; p:= p+1`
5	`E::= –T`	`P(p):='@'; p:= p+1`
6	`T::= F`	[nėra]
7	`T::= T*F`	`P(p):='*'; p:= p+1`
8	`T::= T/F`	`P(p):='/'; p:= p+1`
9	`F::= I`	`P(p):= S(i); p:= p+1`
10	`F::= (E)`	[nėra]

Pavyzdys.
Atliksime sakinio `A*(B+C)` analizę.
`S` – stekas
`R` – naujai ateinantis simbolis
T_k … - analizuojamos eilutės likutis.

`Tais ir Sem` – taikomos taisyklės numeris, semantinės programos numeris.

Žingsnis	Stekas	R	T_k …	Tais ir Sem	P
1.	`#`	`A`	`*(B+C) #`		
2.	`#A`	`*`	`(B+C)#`	9	A
3.	`#F`	`*`	`(B+C)#`	6	A
4.	`#T`	`*`	`(B+C)#`		A
5.	`#T*`	`(`	`B+C)#`		A
6.	`#T*(`	`B`	`+C)#`		A
7.	`#T*(B`	`+`	`C)#`	9	AB
8.	`#T*(F`	`+`	`C)#`	6	AB
9.	`#T*(T`	`+`	`C)#`	2	AB
10.	`#T*(E`	`+`	`C)#`		AB
11.	`#T*(E+`	`C`	`)#`		AB
12.	`#T*(E+C`	`)`	`#`	9	ABC
13.	`#T*(E+F`	`)`	`#`	6	ABC
14.	`#T*(E+T`	`)`	`#`	3	ABC+
15.	`#T*(E`	`)`	`#`		ABC+
16.	`#T*(E)`	`#`		10	ABC+
17.	`#T*F`	`#`		7	ABC+*
18.	`#T`	`#`		2	ABC+*
19.	`#E`	`#`		1	ABC+*
20.	`#Z`	`#`	STOP		ABC+*

Infiksinės formos pervedimas į tetradų formą

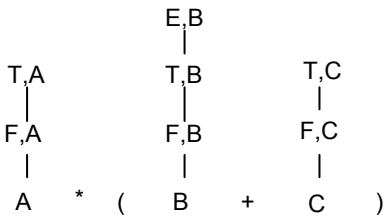
Išanalizuokime sakinį A*(B+C) taip, kad sugeneruoti tetradas:

```
+,B,C,T1
*,A,T1,T2
```

Pažiūrėkime, kokios semantinės programos tam reikalingos. Pirmoją tetradą galėtų sugeneruoti taisyklei E::=E+T skirta semantinė programa. Vykdykime kylančią analizę iki to momento, kai prireiks taikyti šią taisyklę. Čia matote sentencialines formas, kurios susidaro atliekant šią analizę (pagrindas pabrauktas):

```
A*(B+C)   T*(T+C)
F*(B+C)   T*(E+C)
T*(B+C)   T*(E+F)
T*(F+C)   T*(E+T)
```

Sekančiame žingsnyje E+T redukuojamas į E. Kartu bus vykdoma ir semantinė programa, generuojanti tetradą. Tačiau mes negalime to padaryti, nes praradome informaciją apie operandus, pritaikydami taisykles F::= B ir F::= C. Kad taip neatsitiktų, reikia su neterminalu susieti informaciją. Informacija bus saugoma sintaksinio medžio mazguose:



Semantinę informaciją, susietą su neterminalu F saugos F.SEM

Dabar dar kartą pradėkime sakinio A*(B+C) analizę. Pirmasis pagrindas yra I su semantine informacija A. Tada semantinė programa taisyklei F::=I atrodys taip:

```
F.SEM:= I.SEM
```

Semantinė programa taisyklei E_1::= E_2 + T atrodys taip:

```
i:= i+1; E1.SEM := Ti;
ENTER('+',E2.SEM, T.SEM, E1.SEM)
```

Analogiškai sudaromos semantinės programos ir kitoms taisyklėms

Semantinių programų ir stekų realizacija

Su kiekvienu neterminalu gali būti susieta keletas semantinių atributų. Beje atlikus redukciją, dalyvavusių neterminalų semantinė informacija tampa nebereikalinga. Paprastai reikalinga semantinė informacija tik tų neterminalų, kurie įeina į einamu momentu turimą (nagrinėjamą) sentencialinę formą. Pastaroji yra saugoma sintaksiniame steke S, galima taip pat sukurti kelis semantinius stekus S_1 , S_2, …. Visi stekai dirba paraleliai. Semantinė programa gali kreiptis į visus stekus. Jei tarkim S(i) saugo simbolį E, tai S_1(i), S_2(i), … gali saugoti išraiškos tipą, vykdymo metu esančio išraiškos rezultato adresą ir kt.

Kiekvieną semantinę programą galima realizuoti kaip atskirą procedūrą. Bet tada sintaksinis analizatorius turi žinoti kiekvienos jų pavadinimą. Tai gali kelti nemažai sunkumų, ypač kai gramatika didelė. Paprasčiausia sunumeruoti visas gramatikos taisykles ir sukurti vienintelę procedūrą SEMANTICS, kurią kviečiant kaip pradinė informacija pateikiamas taisyklės numeris. Bendras procedūros pavidalas gali būti toks:

```
PROCEDURE SEMANTICS(r); VALUE r; INTEGER r;
CASE r OF
BEGIN
```
[semantinė programa susieta su pirmąją taisykle]
[semantinė programa susieta su antrąją taisykle]

…

…

…
```
ENDCASE
```

Semantinis analizatorius programavimo kalbai

Darbui su vardų (arba simbolių) lentele, kur saugomi visi identifikatoriai, naudojamos šios procedūros:
`LOOKUP(NAME,P).` Simbolių lentelėje ieškomas elementas vardu `NAME`. Jo adresas grąžinamas P.
`LOOKUPDEC(NAME,P)` Dirba kaip ir `LOOKUP`, bet naudojama identifikatorių apibrėžimui. Kalboje su blokine struktūra ši procedūra peržiūri tik einamo bloko identifikatorius.
`INSERT(NAME,P)` Naujas elementas vardu `NAME` įtraukiamas į lentelę.

Simbolių lentelės elementų atributai:

1. TYPE. Reikšmės:
   0= UNDEFIND
   1= REAL
   2= INTEGER
   3= BOOLEAN
   4= LABEL
2. CLASS . Reikšmės:
   1= paprastas kintamasis
   2= masyvo vardas
   3= indeksuotas kintamasis
3. TVAR.
   1= laikinas kintamasis
   0= ne laikinas kintamasis
4. ADDR. Tetrados numeris arba kito simbolių lentelės elemento adresas.
5. DIM. Masyvo dimensija.
6. DEC
   a. 1= apibrėžtas
   b. 0= neapibrėžtas

Semantinės programos sąlyginiams sakiniams
<sakinys$_1$>::= <sąlyga>< sakinys$_2$> ELSE < sakinys$_3$> |<sąlyga>< sakinys$_2$>
<sąlyga>::= IF <išraiška> THEN

Čia <išraiška> turi būti BOOLEAN tipo. Tardami, kad nulis atitinka FALSE, o ne nulis – TRUE ir kad skaičiuojamos išraiškos rezultatas priskiriamas laikinam kintamajam T, turime sugeneruoti vieną iš tokių sekų:

(1) Tetrados skirtos T:= <išraiška>

(p) BZ, q+1, T, 0
<sakinys$_2$> tetrados

(q) BR , r, ,
(q+1) <sakinys$_3$> tetrados .

(r)

(1) Tetrados skirtos T:= <išraiška>

(p) BZ, q, T, 0
< sakinys$_2$> tetrados

(q)

Tetrada su operatoriumi BZ yra generuojama programos, susietos su gramatikos taisykle
<sąlyga>::= IF <išraiška>THEN
Generuojant tetradą nėra žinoma kur perduodamas valdymas, todėl išsaugome terados numerį kaip semantinę informaciją, susietą su neterminalu <sąlyga>. Šito dėka, vėliau galėsime prieiti prie šios tetrados. <išraiška> jau turime sugeneruotą. Nuoroda ENTRY saugo adresą elemento simbolių lentelėje, turinčio informaciją apie šią išraišką. Semantinės programos tekstas:

```
P:=<išraiška>.ENTRY;
CHECKTYPE(P,BOOLEAN);
<sąlyga>.JUMP:= NEXTQUAD;
ENTER(BZ,0,P,0)
```

Naujos procedūros ir kintamieji:
`NEXTQUAD` – sekančios tetrados numeris.
`CHECKTYPE(P,BOOLEAN)` – patikrina, ar išraiška BOOLEAN tipo.
`ENTER(BZ,0,P,0)` – tetrados redukcija.

Semantinė programa taisyklei :
<sakinys$_1$>::= <sąlyga>< sakinys$_2$>

```
I:= <sąlyga>.JUMP;
QUAD(I,2):= NEXTQUAD;
```

Nauja procedūra:
`QUAD(I,2)` – kreipiamasi į tetrados, kuris numeris I, antrąją komponentę.
Šiuo konkrečiu atveju, mes į tetradą su operandu BZ, kurią sugeneravome, užrašome į kokį adresą pereiti, jei sąlygos išraiška lygi 0 (t.y. sąlyginio sakinio sąlyga nėra tenkinama).

Kad sukurti kodą sakiniui pagal taisyklę:
<sakinys$_1$>::= <sąlyga>< sakinys$_2$> ELSE < sakinys$_3$>
reikia sukurti perėjimo komandą tarp < sakinys$_2$> ir < sakinys$_3$>. Bet, kadangi to negalima padaryti, kol < sakinys$_2$> ir < sakinys$_3$> nebus išanalizuoti, reikia pakeisti taisyklę į dvi taisykles:

```
< sakinys2>::= <pirma dalis>< sakinys3>
<pirma dalis>::= <sąlyga>< sakinys2>ELSE
```

Semantinės programos :

<pirma dalis>::= <sąlyga>< sakinys2>ELSE

```
<pirma dalis>.JUMP:= NEXTQUAD;
ENTER(BR,0,0,0);
I:= <sąlyga>.JUMP;
QUAD(I,2):= NEXTQUAD;
```

<sakinys2>::= <pirma dalis>< sakinys3>

```
I:= <pirma dalis>.JUMP;
QUAD(I,2):= NEXTQUAD;
```

Semantinės programos žymėms

```
< sakinys1> ::= I: < sakinys2>
```

Žymei reikia priskirti < sakinys2> pradžios adresą, bet tai neįmanoma pagal šią gramatiką, nes <sakinys2> buvo sugeneruotas anksčiau, nei pradėjome identifikatoriaus analizę. Todėl keičiame gramatiką:

```
<sakinys1> ::= <žymės ap.> : < sakinys2>
<žymės ap.>::= I
```

Semantinė programa

<žymės ap.>::= I

```
LOOKUPDEC(I.NAME,P);
IF P=0 THEN BEGIN
                INSERT(I.NAME,P);
                P.TYPE:= LABEL
            END
       ELSE BEGIN
                CHECKTYPE(P,LABEL);
                IF P.DEC=1 THEN ERROR()
            END
P.DEC=1;
P.ADDR:= NEXTQUAD;
```

<sakinys>::= GOTO I

```
LOOKUPDEC(I.NAME,P);
IF P=0 THEN BEGIN
                INSERT(I.NAME,P);
                P.TYPE:= LABEL;
                P.DEC:= 0;
            END
       ELSE CHECKTYPE(P,LABEL);
ENTER(BRL,P,0,0);
```

Semantinės programos ciklui

```
<sakinys>::= FOR <kintamasis>:=<for sąrašas> DO <sakinys>
<for sąrašas>::=<išraiška1>STEP<išraiška2>UNTIL <išraiška3>
```

Laikoma, kad <išraiška$_2$> visada teigiama. Tokiu atveju for- ciklą galima užrašyti tokia analogiška forma:

```
<kintamasis>:= <kintamasis1>
GOTO OVER
AGAIN: <kintamasis>:=<kintamasis>+<išraiška1>
OVER:    IF <kintamasis> ≤ <išraiška3> THEN BEGIN  <sakinys>;
                                                   GOTO AGAIN
                                            END
```

Pakeičiame taisykles:

```
<for1>::= FOR <kintamasis>:=<išraiška1>
<for2>::=<for1>STEP<išraiška2>
<for3>::=<for2>UNTIL<išraiška3>
<sakinys1>::= <for3> DO <sakinys2>
```

Semantinės programos:

<for1>::= FOR <kintamasis>:=<išraiška$_1$>

```
LOOKUP(<kintamasis>.NAME,P);
IF P=0 THEN ERROR;
P1:=<išraiška1>.ENTRY;
ENTER(:=,P1, ,P);
<for1>.ENTRY:= P;
<for1>.JUMP:=NEXTQUOAD;
ENTER(BR,0,0,0);
<for1>.JUMP1:=NEXTQUAD;
```

<for2>::=<for1>STEP<išraiška$_2$>

```
<for2>.JUMP1:=<for1>.JUMP1;
<for2>.ENTRY:=<for1>.ENTRY;
P:=<išraiška2>.ENTRY;
ENTER(+,<for2>.ENTRY,P,<for2>.ENTRY);
I:=<for1>.JUMP;
QUAD(I,2):=NEXTQUAD;
```

<for3>::=<for2>UNTIL<išraiška$_3$>

```
<for3>.JUMP1:=<for2>.JUMP1;
<for3>.JUMP:=NEXTQUAD;
ENTER(BG,0,<for2>.ENTRY,<išraiška3>.ENTRY);
```

<sakinys$_1$>::= <for3> DO <sakinys$_2$>

```
ENTER(BR,<for3>.JUMP1,0,0);
I:=<for3>.JUMP;
QUAD(I,2):=NEXTQUAD;
```

Semantinės programos kintamiesiems

<kintamasis>::=I | I(<išraiškų sąrašas>)

<išraiškų sąrašas>::= <išraiška> | <išraiškų sąrašas>,<išraiška>

<kintamasis>::=I

```
LOOKUP(I.NAME,P);
IF P=0 THEN ERROR;
<kintamasis>.ENTRY:=P;
```

Ši taisyklė nereikalauja tetradų generacijos.

Semantinėms programoms, skirtoms indeksuotam kintamajam, reikia sugeneruoti tetradas visų indeksų išraiškų ir masyvo elemento adreso apskaičiavimui. Kad galėtume tai padaryti, reikia susipažinti su atminties organizavimu masyvams.

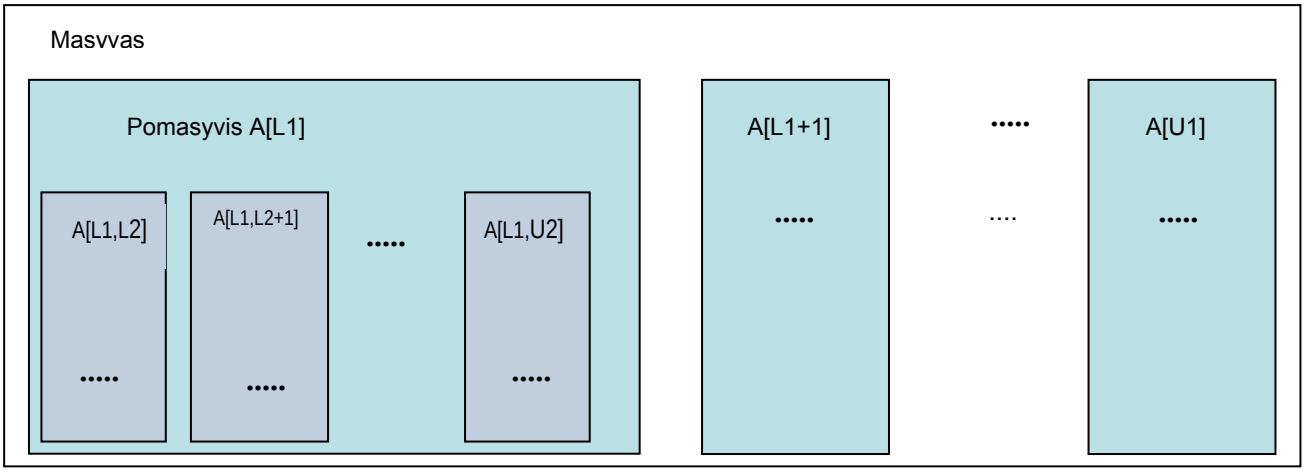
Atminties organizavimas masyvams
Masyvas ALGOL‘ e aprašomas taip:

ARRAY A[L1:U1, L2:U2, … , Ln:Un]

Masyvą A sudaro pomasyviai:
`A[L1,*, *, …, *], A[L1+1,*, *, …, *], … , A[U1,*, *, …, *]`

Pomasyvį A[i, *, *, … ,*] sudaro pomasyviai:
`A[ i,L2,*, *, …, *], A[ i,L2+1,*, *, …, *], … , A[ i,U2,*, *, …, *]`

Tai kartojasi kiekvienam masyvo išmatavimui (t.y. tiek kartų, kiek masyvas turi indeksų).



Pagrindinis uždavinys yra, kaip kreiptis į masyvo elementą A[i,j,k, … ,l,m].

Pažymėkime d_i i-tojo indekso diapazono dydį:

$d_1 = U1 - L1 + 1$
$d_2 = U2 - L2 + 1$

…
$d_n = Un - Ln + 1$

Tada pomasyvio A[i,*,*, … ,*] pirmojo elemento adresas yra:
$BASELOCK + (i - L_1) * d_2 * d_3 * \ldots * d_n$

Čia BASELOCK ─ pirmojo masyvo elemento A[L1, L2, … ,Ln] adresas.

Atitinkamai pomasyvio A[i,j,*,*, … ,*] adresas yra gaunamas prie pomasyvio A[i,*,*, … ,*] pradžios adreso pridėjus
$(j - L_2)* d_3*d_4 * \dots * d_n$.
T.y.:
$BASELOCK + (i - L_1)* d_2 d_3 * \dots * d_n + (j - L_2)* d_3*d_4 * \dots * d_n$

Tada elemento A[i,j,k, … ,l,m] adresas yra:

$BASELOCK + (i-L_1)* d_2 d_3 * \dots * d_n + (j-L_2)* d_3*d_4 * \dots * d_n + (k-L_3)* d_4*d_5 * \dots * d_n + \dots + (l-L_{n-1})*d_n + m- L_n$

Tai galime užrašyti taip CONSPART + VARPART
Čia CONSPART ir VARPART yra
$CONSPART = BASELOCK - ((\dots((L_1* d_2 + L_2)* d_3 + L_3) *d_4 + \dots + L_{n-1})*d_n + L_n)$
$VARPART = (\dots((i* d_2 + j)* d_3 + k) *d_4 + \dots + l)*d_n + m$

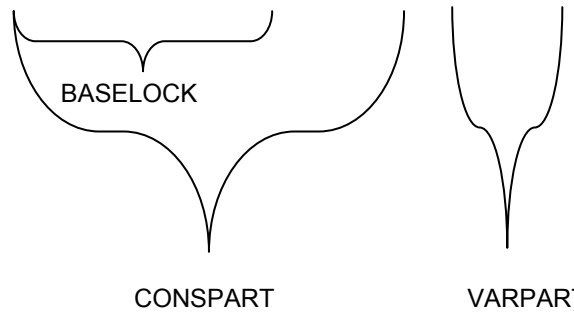
CONSPART reikia paskaičiuoti tik vieną kartą.
VARPART elementui A[i,j,k, … ,m] galima rasti atliekant tokių veiksmų seką:

$VARPART := i$
$VARPART := VARPART* d_1+j$
$VARPART := VARPART* d_2+k$

…
$VARPART := VARPART* d_n+m$

Pavyzdys.
Imkime tokį masyvą
ARRAY A[1:M,1:N]

Jo elemento A[i,j] adresas yra:
$Adresas(A[i,j])= Adresas(A[1,1]) + (i-1)*N + j-1 =$
$= Adresas (A[1,1]) - N -1 + i*N + j$



Tais atvejais, kai programavimo kalba leidžia masyvo ribas apibrėžti ne tik kompiliavimo, bet ir skaičiavimo metu, reikalingas *masyvo informacinis vektorius (dope vector)*.

Informacinis vektorius

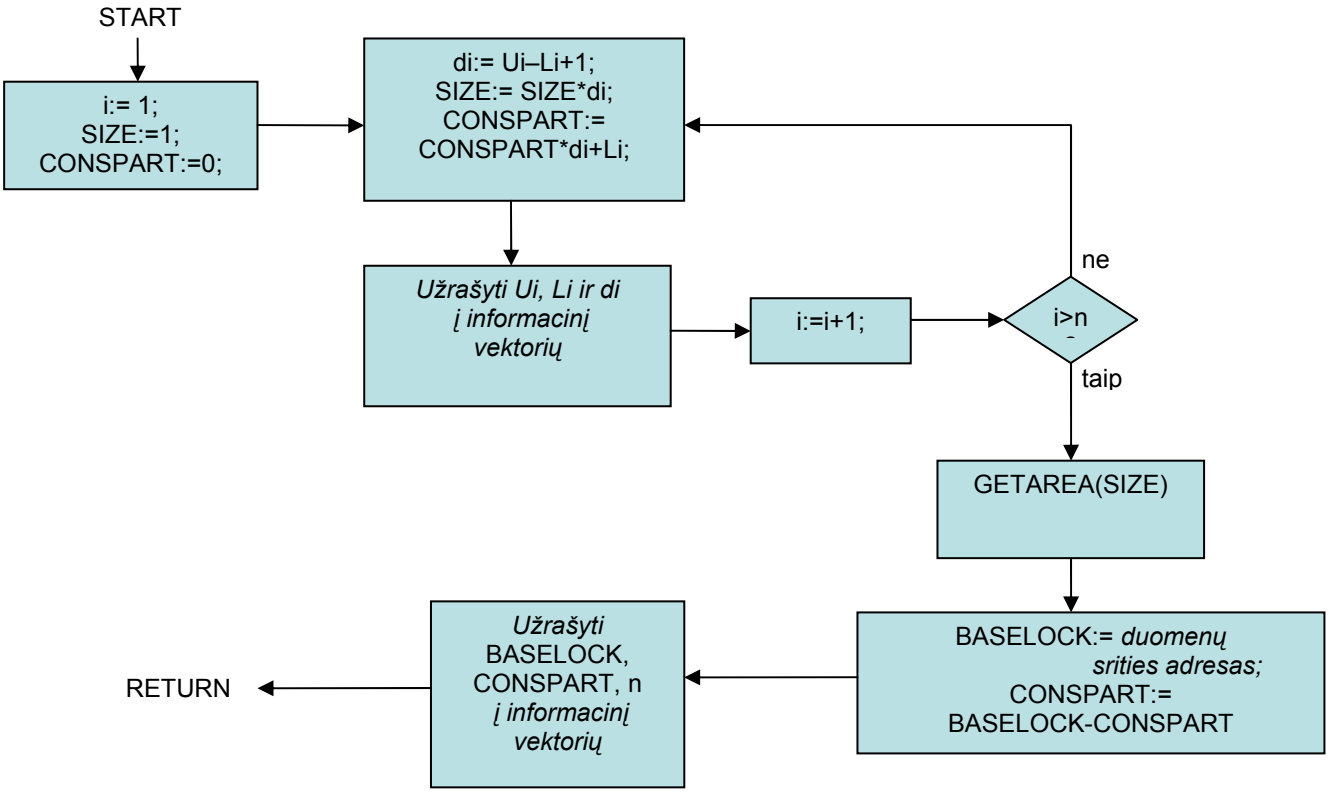
L1	U1	d_1

L2 … … Ln	U2 … … Un	d_2 … … d_n
n	CONSTPART	
BASELOCK		

Masyvo aprašas
A[L1:U1, … , Ln:Un]

Informacinis vektorius turi fiksuotą dydį, žinomą kompiliavimo metu. Atmintis masyvui gali būti išskirta tik įėjus į bloką, kur aprašytas masyvas. Tada apskaičiuojamos masyvo ribos ir vyksta kreipimasis į procedūrą, kuri skirsto masyvams atmintį. Ji paskaičiuoja reikalingų ląstelių skaičių, iškviečia GETAREA ir išsaugo reikalingą informaciją masyvo informaciniame vektoriuje.

Programos, skirstančios atmintį masyvams, schema



Semantinės programos indeksuotiems kintamiesiems

```
<kintamasis>::= I(<išraiškų sąrašas>)
<išraiškų sąrašas>::= <išraiška> | <išraiškų sąrašas>,<išraiška>
```

Šias taisykles pakeisime taip, kad galima būtų apskaičiuoti VARPART masyvo elementui A(i,j,k, … ,m) pagal:

VARPART:= i
VARPART:= VARPART* d_1+j
VARPART:= VARPART* d_2+k

…
VARPART:= VARPART* d_n+m

(žr. sk. „Atminties organizavimas masyvams“)

```
<kintamasis>::=<ind>)
<ind>::= I(<išraiška> | <ind>,<išraiška>
```

Semantinė programa taisyklei <ind>::= I(<išraiška>. Bus ieškomas masyvo identifikatorius, generuojama tetrada VARPART:= <išraiška> ir su neterminalu <ind> susiejamas adresas, kur saugomas pirmo indekso diapazonas (d_1 ~<ind>.ENTRY). Į atributą ENTRY2 bus kaupiamas VARPART.

<ind>::= I(<išraiška>

```
LOOKUP(I.NAME,P);
IF P = 0 OR P.CLASS ≠ ARR THEN ERROR();
<ind>.COUNT:= P.DIM—1;
<ind>.ARR:= P;
<ind>.ENTRY:= P+1;
GENERATETEMP(P);
P.TYPE:= INTEGER;
CONVERTRI(<išraiška>.ENTRY);
P:= <išraiška>.ENTRY;
ENTER(:=,P, ,<ind>.ENTRY2);
```

Naujos procedūros:
`CONVERTRI(<išraiška>.ENTRY)` konvertuoja `<išraiška>.ENTRY` į INTEGER tipą
`GENERATETEMP(P)` generuojamas laikinas kintamasis P

Semantinė programa taisyklei <ind>::= <ind>,<išraiška>. Dabar reikia generuoti kodą atitinkantį VARPART:= VARPART* d_i+j, jei i ─ indekso išraiška.

<ind_1>::= <ind_2>,<išraiška>

```
<ind1>.COUNT:= <ind2>.COUNT—1;
<ind1>.ARR:= <ind2>.ARR;
<ind1>.ENTRY:= <ind2>.ENTRY+1;
P:= <ind2>.ENTRY2;
ENTER(*,P, <ind1>.ENTRY, P);
P1:= <išraiška>.ENTRY;
ENTER(+,P,P1, P);
```

Semantinė programa taisyklei <kintamasis>::=<ind>) .Bus tikrinamas indeksų skaičius ir sukuriamas naujas elementas simbolių lentelėje – indeksuotas kintamasis.

<kintamasis>::=<ind>)

```
IF <ind>.COUNT ≠ 0 THEN ERROR();
GENERATETEMP(P);
P.CLASS:= INDEX;
```

```
P.TYPE:= <ind>.ARR.TYPE;
P.ADDRESS:= <ind>.ENTRY2;
P.ARR:= <ind>.ARR;
<kintamasis>.ENTRY:= P;
```

Parametrų sąryšio būdai

Kaip žinome, pradinė informaciją procedūroms yra perduodama per jų parametrus, kurių reikšmės nurodomos procedūros iškvietimo metu. Kai yra kreipiamasi į procedūrą, jai yra perduodamas sąrašas argumentų adresų. Procedūra išsaugo tuos adresus savo duomenų srityje ir naudoja nustatyti formalių ir faktinių parametrų ryšius. Yra penki parametrų tipai:

1) pagal rezultatą;
2) pagal reikšmę ir rezultatą;
3) pagal vardą;
4) pagal reikšmę;
5) pagal nuorodą (arba adresą);

Nagrinėkime programą:

```
BEGIN
    INTEGER I;
    INTEGER ARRAY B[1,2];

          PROCEDURE Q(x); INTEGER x;
          BEGIN
              I:= 1; x:= x+2; B[I]:=10;
              I:= 2; x:= x+2;
          END

    B[1]:= 1; B[2]:= 1; I:= 1;
    Q(B[I]);
END
```

Priklausomai nuo to, kokio tipo yra procedūros parametrai, įvykdžius procedūrą gauname tokias kintamųjų reikšmes:

Sąryšis pagal reikšmę:

Kintamasis	B[1]	B[2]	I	x
Procedūros pradžia	1	1	1	**1**
Kūnas	10		2	**3**
				5
Įvykdžius procedūrą	10	1	2	

Sąryšis pagal nuorodą:

Kintamasis	B[1]	B[2]	I	x
Procedūros pradžia	1	1	1	**1**
Kūnas	3		2	**3**
	10			**10**
	12			**12**

Įvykdžius procedūrą	12	1	2	

Sąryšis pagal rezultatą:

Kintamasis	B[1]	B[2]	I	x
Procedūros pradžia	1	1	1	**[neapibrėžtas]**
Kūnas	10		2	
Įvykdžius procedūrą	**[neapibrėžtas]**	1	2	

Sąryšis pagal reikšmę ir rezultatą

Kintamasis	B[1]	B[2]	I	x
Procedūros pradžia	1	1	1	**1**
Kūnas	10		2	**3**
				5
Įvykdžius procedūrą	5	1	2	

Sąryšis pagal vardą

Kintamasis	B[1]	B[2]	I	x
Procedūros pradžia	1	1	1	**1**
Kūnas	3	3	2	**3**
	10			**1**
				3
Įvykdžius procedūrą	10	3	2	