DEFINIREA FORMALĂ A SINTAXEI LIMBAJELOR DE PROGRAMARE

Limbajul de programare este o notație formală. Ea se fundamentează prin seturi de reguli care definesc forma şi înțelesul construcțiilor limbajului. Pe baza acestor reguli se stabileşte (de către programator sau de către compilator) dacă un anumit program este corect şi care este sensul lui, cu alte cuvinte ce trebuie să se întâmple atunci când el va fi executat. Regulile prin care se defineşte limbajul se referă la două aspecte majore ale acestuia: *sintaxa* şi *semantica*. Sintaxa este aceea care determină forma construcțiilor limbajului în timp ce semantica stabileşte sensul asociat construcțiilor corecte din punct de vedere sintactic.

În concluzie, un limbaj de programare poate fi specificat prin tripletul:

$L=<Sm,St,f:St\rightarrow Sm>$, unde:

- Sm reprezintă semantica limbajului;
- St este sintaxa sa;
- f este funcția de asociere a semanticii, unei sintaxe date.

Este evident că o descriere formală completă a limbajului trebuie să trateze toate aceste aspecte.

În principiu, orice metodă de definire formală a limbajelor de programare oferă soluții pentru rezolvarea următoarelor probleme:

- Definirea unui alfabet de simboluri de bază A şi, implicit, definirea mulțimii A^* reprezentând toate sirurile de simboluri posibile care pot fi construite cu elementele din A.
- Furnizarea unui set de reguli pentru selectarea mulțimii $P\subseteq A^*$ a programelor corecte din punct de vedere sintactic.
- Specificarea semanticii fiecărui element $p\in P$.

Regulile de sintaxă determină o mulțime infinită de **propoziții**, corecte din punct de vedere al formei lor. Desigur, doar o parte dintre acestea sunt corecte şi din punct de vedere al regulilor semantice. Elementele constituente ale propozițiilor se numesc **simboluri**. Modul în care caracterele care formează **alfabetul** limbajului, se grupează formând simboluri este descris prin regulile lexicale. Acestea pot fi considerate ca făcând parte din sintaxa limbajului. Totalitatea simbolurilor formează **vocabularul** limbajului. Din vocabular fac parte identificatori, cuvinte cheie (ca `begin`, `var`, `mod` în Pascal), operatori (`+`, `++`, `<=`, `==`, | în C), literale (întregi, flotante, şiruri de caractere).

Problemele ridicate de definirea formală a semanticii şi de asociere a ei cu sintaxa s-au dovedit a fi mai dificile. Dificultăţile sunt cauzate atât de complexitatea deosebită a acestor probleme cât şi de faptul că, o lungă perioadă de timp, limbajele de programare au fost definite în mod artizanal. Cu toate că, în prezent, conceptul de limbaj de programare, structura şi componentele sale, sunt corect definite şi bine fundamentate teoretic definirea noilor limbaje a trebuit să ţină cont, din motive comerciale şi de utilizare a experienţei anterioare, de compatibilitatea cu sistemele de programare existente.

Introducere în teoria limbajelor formale

Limbajele formale, ca şi capitol de matematică, au apărut cu aproximativ 10 ani mai târziu decât ştiinţa calculatoarelor. Bazele teoriei limbajelor formale s-au pus în 1956 prin crearea modelului matematic pentru reprezentarea în calculator a unui limbaj natural (limba engleză). Noam Chomsky a adaptat noţiunea de gramatică la modelarea matematică pentru calculator a unei limbi naturale; în consecinţă, termenii de specialitate utilizaţi (sintaxa, semantica, alfabet, propoziţie…) provin din lingvistică.

În 1955 – Fortran – s-a definit primul limbaj de programare.

În 1957 – s-a finalizat primul compilator Fortran (în limbaj de asamblare) de către un colectiv de la IBM condus de John Backus.

Cele doua direcţii s-au unit în 1960 prin definirea de către John Backus şi Peter Naun al sintaxei limbajului Algol 60 folosind conceptul de gramatică Chomsky.

Metalimbajul folosit de Backus şi Naun pentru definirea limbajului Algol 60 este utilizat şi în prezent, cu mici completări, ca un metalimbaj general pentru descrierea sintaxei limbajelor de programare. El se numeşte BNF (Backus Naun Form).

Definirea principalelor noţiuni:

Alfabet (A) - mulţimea finită şi nevidă a simbolurilor utilizate într-un limbaj (natural sau de programare).

Şirurile de simboluri care se pot forma cu elementele din A se numesc **propoziţii**.

Mulţimea tuturor propoziţiilor care se pot forma cu elementele din A pe baza aranjamentelor cu repetiţie, incluzând şi propoziţia vidă sau şirul vid (ε, λ) se noteaza cu A^*. Excluzând propoziţia vidă din A^* se obţine multimea A^+.

Exemplu:

$A=\{0,1\}$

$A^*=\{\varepsilon,0,1,00,01,10,11,000,001, \ldots\}$ (fără ε va deveni A^+)

În legatură cu şirurile din A^* si A^+ se poate defini operaţia de concatenare (alăturare, juxtapunere). Din punct de vedere algebric operaţia de concatenare este asociativă atât în A^+ cât şi în A^*.

$\forall x,y,z \quad \exists A^*, \ (xy)z = x(yz)$

În A^*, ε este element neutru faţă de concatenare.

$\varepsilon x = x\varepsilon = x$

Din aceste proprietăți rezultă următoarele:
(A+, concatenare) reprezintă un semigrup, numit semigrupul liber generat de A
(A*, concatenare, ε) reprezinta un monoid, numit monoidul liber generat de A

Limbaj formal: un limbaj formal peste alfabetul A este o submulțime a lui A*. Practic orice submulțime a lui A* reprezintă un limbaj formal, iar A* în ansamblu se numeşte *limbaj universal.*

Concluzie: noțiunea intuitivă de limbaj, inclusiv limbaj de programare, se poate modela ca un limbaj formal.

Problema esențială pe care o are de rezolvat un compilator, în legatură cu un limbaj de programare, este aceea de a decide dacă o anumita propoziție din A* aparține sau nu de un limbaj de programare.

Datorită faptului că propozițiile unui limbaj sunt infinite ca număr, aceasta reprezentare a limbajului - verificare devine relativ dificilă. Din acest motiv a fost necesară găsirea unei reprezentări a limbajului de programare în calculator, printr-un număr finit de reguli, care să descrie proprietățile propozițiilor din limbaj si numai ale acestora. Totalitatea acestor reguli reprezintă în esență gramatica limbajului.

Specificarea sintaxei prin gramatici

Totalitatea regulilor sintactice ale unui limbaj reprezintă *gramatica* limbajului. Se pune problema cum se descrie gramatica unui limbaj? O metodă foarte răspândită este notația folosită pentru prima dată la descrierea limbajului Algol 60 şi cunoscută sub numele **BNF** (Backus Naur Form). Vom prezenta o formă uşor extinsă a acestei notații, aşa numitul BNF extins.

BNF este un metalimbaj (un limbaj utilizat pentru descrierea altui limbaj) care foloseşte următoarele metasimboluri:

	::=		însemnând *definit ca*
	|		însemnând *sau*
<	şi	>	sunt folosite pentru a include neterminale
[	şi	]	sunt folosite pentru a include secvențe opționale
{	şi	}	sunt folosite pentru a include secvențe care se pot repeta de oricâte ori (inclusiv de zero ori).

Sintaxa este definită ca o succesiune de relații (reguli) BNF. Fiecare relație defineşte un neterminal specificat în stânga semnului **::=.** În dreapta semnului găsim o succesiune de neterminale şi terminale. Terminalele sunt simboluri ale limbajului. Fiecare neterminal găsit în partea dreaptă a unei relații trebuie să fie definit într-o altă relație. Gramatica completă a limbajului reprezintă un set de relații prin care sunt definite toate neterminalele. Un anumit neterminal particular este aşa numitul *simbol de start*; acesta se numeşte în mod tipic 〈program〉.

Un program reprezintă o succesiune de simboluri deci, în termenii de mai sus, de terminale. Programul este corect din punct de vedere sintactic, dacă succesiunea de simboluri poate fi derivată (se poate genera) pe baza regulilor gramaticale, pornind de la simbolul de start.

Exemplu de gramatică exprimată în BNF:

⟨ expresie⟩ : : = ⟨ termen⟩ {+ | – ⟨ termen ⟩ }
⟨ termen⟩ : : = ⟨ factor ⟩ { * | : ⟨ factor⟩ }
⟨ factor⟩ : : = număr | identificator | (⟨ expresie⟩)
⟨ atribuire⟩ : : = identificator : = ⟨ expresie⟩
⟨ instrucțiuni⟩ : : = ⟨ atribuire⟩ { ; atribuire}
⟨ program⟩ : : = `prog` identificator; ⟨ instrucțiuni⟩ `end.`

Programul de mai jos :

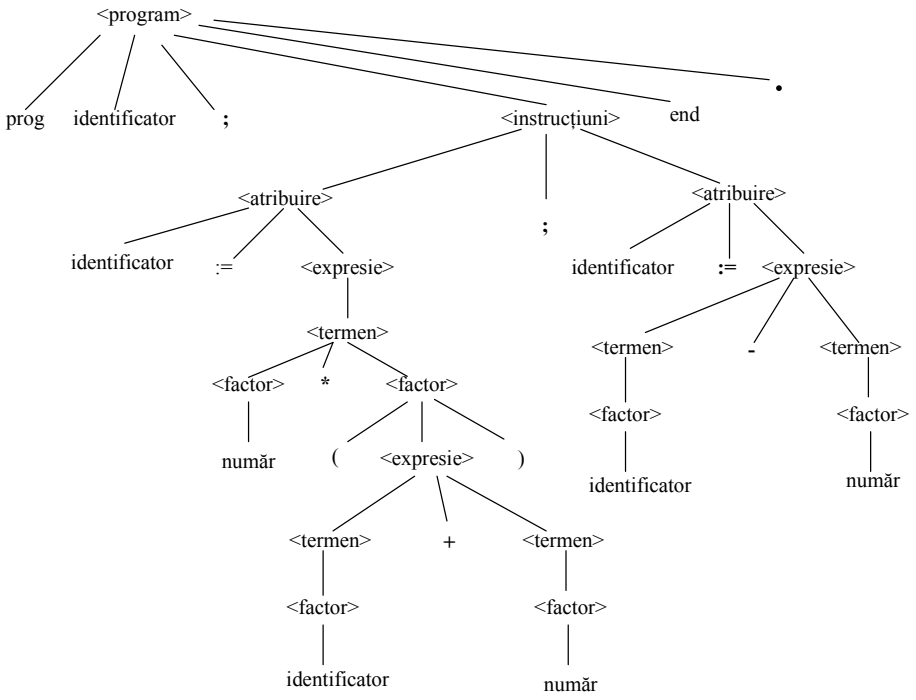
```
prog exemplu ;
   a:= 2*(x+3);
   b:= a-1
end .
```

este corect sintactic, dacă poate fi derivat pe baza regulilor de mai sus, pornind de la neterminalul ⟨ `program`⟩ care reprezintă simbolul de start. Acest proces de derivare se poate ilustra cel mai bine printr-un arbore a cărui rădăcină este simbolul de start, frunzele sunt terminale şi celelalte noduri sunt neterminale ale gramaticii (fig. de mai jos).

Un astfel de arbore se numeşte **arbore sintactic**. Verificarea corectitudinii din punct de vedere sintactic a unui program, prin derivarea unui astfel de arbore, se numeşte *analiză sintactică*. O astfel de analiză se poate desfăşura, în principiu, în două feluri:

- *De jos în sus*: se porneşte de la succesiunea de simboluri, reprezentând programul, şi se înlocuiesc succesiv secvențe reprezentând partea dreaptă a unor reguli cu partea stângă corespunzătoare, până se ajunge la simbolul de start (dacă programul este corect).

- *De sus în jos*: pornind de la simbolul de start, şi înlocuind succesiv neterminalele conform cu regulile gramaticii: se încearcă derivarea succesiunii de simboluri corespunzătoare programului.



Exemplu de arbore sintactic

Notație alternativă, simplificată, privind gramaticile:

$\mathcal{P}$ – productiile gramaticii (relații, reguli)

Forma unei producții este:

`parte stanga → parte dreapta`

sau

`parte stanga → alternativa1 | alternativa 2 | …`

Simboluri neterminale şi terminale:

- Pentru neterminale
  - Literele mari ale alfabetului latin: `A, B, C, … Z`
  - Simbolul de start sau rădăcina gramaticii: `S`
- Pentru terminale
  - Literele latine mici de la începutul alfabetului: `a, b, c, d, e, f, ...`
  - Semnele grafice care reprezintă acel terminal `+ * ( ) ...`
  - Cuvinte: plus, const, <u>ident</u> (pot să fie şi subliniate)
- Alte notații:
  - Literele latine mici de la sfârşitul alfabetului (u, v, x, y, z, w) – se folosesc pentru a reprezenta şiruri formate numai din terminale
  - Literele greceşti mici (α, β, γ, δ) – şiruri amestecate formate din terminale şi neterminale
  - Multimea tuturor neterminalelor unei gramatici – N
  - Multimea tuturor terminalelor unei gramatici – T

$N \cap T = \phi$
$V_G = N \cup T$

Considerând $\alpha \in V_{G+}$, $\beta \in V_{G*}$ putem considera că, la modul general, o relație din gramatică este de forma $\alpha \rightarrow \beta$

Gramatica $G = < N, T, \mathcal{P}, S >$

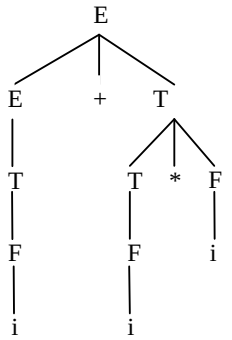
Exemplu:

```
<expresie>::=<termen>|<expresie>+<termen>|<expresie>-<termen>
<termen>::=<factor>|<termen>*<factor>|<termen>/<factor>
<factor>::= (<expresie>)|ident
T = {+,-,*,/,(,),ident}
```

devine cu noua notație:

```
E → T | E+T | E-T
T → F | T*F | T/F
F → (E) | i
```

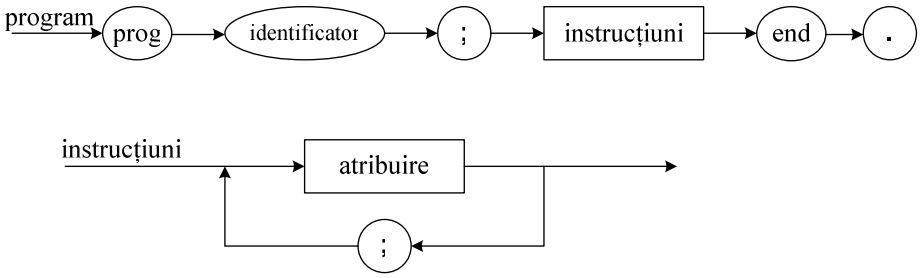
Arborele sintactic corespunzător propoziției `i+i*i` este:



Arbore sintactic pentru propoziția `i+i*i`

Diagrame de sintaxă

O altă formă de reprezentare a sintaxei, folosită pentru prima dată de N. Wirth la descrierea limbajului Pascal, sunt *diagramele de sintaxă*. Este o formă grafică de descriere echivalentă cu notația BNF. Unei reguli îi corespunde o diagramă în care neterminalele sunt reprezentate prin dreptunghiuri şi terminalele prin cercuri şi elipse. Aceste diagrame pot fi parcurse pornind de la intrare şi încheind cu o ieşire, pe una sau mai multe căi, înaintând conform sensului indicat de săgeți (vezi fig. de mai jos).



Exemplu de diagramă de sintaxă

O succesiune de simboluri este corectă din punct de vedere sintactic dacă poate fi generată prin traversarea diagramei sintactice corespunzătoare (în speță a celei corespunzătoare programului) de la intrare până la ieşire. Dacă pe parcurs se întâlneşte un dreptunghi, atunci neterminalul respectiv trebuie acceptat conform diagramei sintactice corespunzătoare neterminalului. Dacă se întâlneşte un cerc (elipsă) terminalul respectiv trebuie regăsit ca fiind cel curent în şirul de simboluri parcurse.

2.4 Transformări bazate pe gramatici

Considerând $\alpha, \beta, \gamma \in V_{G+}$ şi o relație în cadrul gramaticii $(\beta \rightarrow \delta) \in \mathcal{P}$, se poate scrie următoarea transformare $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$.

Această transformare, prin care se identifică într-un şir partea stângă a unei producții, după care această parte stângă se înlocuieşte cu partea dreaptă corespunzătoare, se numeşte **derivare**. Operația opusă derivării, de identificare într-un şir a părții drepte a unei produții şi de înlocuire a ei cu partea stângă corespunzătoare, se numeşte **reducere**.

O serie de k derivări successive aplicate asupra unui sir α, de următoarea formă:

$$\alpha \Rightarrow \beta_1 \Rightarrow \beta_2 \ldots \Rightarrow \beta_k$$

se poate scrie

$$\alpha \overset{k}{\Rightarrow} \beta_k$$

În general, dacă de la sirul α trecem la şirul β printr-un număr oarecare de derivări (necunoscut) se poate scrie:

$$\alpha \overset{*}{\Rightarrow} \beta$$

Dacă în şirul de transformări se exclude sirul vid (ε) vom nota:

$$\alpha \overset{+}{\Rightarrow} \beta$$

Şirurile care se obțin prin derivări din simbolul de start al gramaticii se numesc **forme propozitionale:**

$$S \overset{*}{\Rightarrow} \alpha$$

O formă propozițională alcătuită doar din terminale se numeşte **propozitie**.

$i+i*i$

Totalitatea propozițiilor formează **limbajul** generat de gramatica respectivă. Se poate nota astfel:

$$L(G) = \{x \mid S \overset{*}{\Rightarrow} x\}$$

Exemplu:

Mulțimea neterminalelor N = { S }

T = {0,1}

$\mathcal{P}$: S → 0S1 | 01

Aplicând prima producție de $n-1$ ori şi a doua producție o dată, în finalul transformărilor, obținem următorul şir de transformări:

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0^3S1^3 \Rightarrow \ldots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n1^n$

Această propoziție se numeşte derivata de ordinul n al lui S conform gramaticii G, iar limbajul general de gramatica $L(G)$ poate fi notat şi astfel:

$L(G)= \{ 0^n1^n/n\geq 1 \}$

Un limbaj poate fi generat de mai multe gramatici

$G_1 = \langle N_1,T,P_1,S_1\rangle$

$G_2 = \langle N_2,T,P_2,S_2\rangle$

Dacă $L(G_1) = L(G_2)$, gramaticile se numesc **echivalente**, ceea ce se notează

$G_1 \sim G_2$

Tipuri de gramatici şi limbaje

După forma producțiilor, Chomsky a împărțit gramaticile în 4 categorii (clasificarea este valabilă şi pentru limbajele generate):

- **Gramatici de tipul 0**: gramaticile în forma cea mai generala $\alpha\rightarrow\beta$, sunt gramatici fără restricții. S-au formulat, totuşi, 3 cerințe pentru acest tip de gramatici:
  a) Să se poată defini o procedură recursivă care să permită să se asocieze fiecărei propoziții câte un număr. Este şi motivul pentru care şi limbajele generate se numesc **limbaje enumerabile recursiv.**
  b) Să nu existe ambiguități
  c) Fiecare parte stângă să conțină cel puțin un neterminal.

- **Gramatici de tipul 1**: gramatici dependente de context sau sensibile la context. Producțiile sunt de forma

  $\alpha A\beta \rightarrow \alpha\gamma\beta$

  ceea ce se traduce astfel "A trece în γ numai dacă se găseşte în contextul $\alpha\beta$"

- **Gramatici de tipul 2**: gramatici independente de context (GIC). Producțiile sunt de forma

  $A \rightarrow \alpha$ (A se înlocuieste cu α indiferent de contextul în care apare)

- **Gramatici de tipul 3**: gramatici regulate (GR). Producțiile sunt de forma

  `A → aB|a` sau `A → Ba|a`

Notând cu $\mathcal{G}_i$ mulțimea gramaticilor de tipul `i` si cu $\mathcal{L}_i$, clasa limbajelor generate de G_i, există următoarele incluziuni:

$\mathcal{G}_3 \subseteq \mathcal{G}_2 \subseteq \mathcal{G}_1 \subseteq \mathcal{G}_0$

$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$

Concluzia este următoarea: atât gramaticile cât şi limbajele se pot ierarhiza conform clasificării de mai sus. Ierarhia se numeşte **ierarhie Chomsky**.

Modelul unui limbaj de programare este un limbaj independent de context. În teoria limbajelor formale se poate demonstra că pentru orice limbaj dependent de context (implicit şi pentru cele independente de context) se poate decide cu certitudine dacă o propoziție aparține sau nu limbajului respectiv. Aceasta este operația esențială pe care o are de rezolvat compilatorul la faza de analiză sintactică.

`<neterminal> ::= definitie` (relație BNF generală)

Rezultă că, BNF corespunde, ca notatie, tocmai gramaticilor independente de context.

În principiu se acceptă că un limbaj de programare este un limbaj independent de context. Acest lucru este valabil doar pentru sintaxa limbajului, care într-adevar se poate exprima printr-o gramatică independentă de context. Semantica limbajului, în schimb, conține restricții dependente de context:

- Aplicarea operatorilor în cadrul unei expresii în conformitate cu tipul operanzilor
- Definirea sau declararea unui identificator înainte de utilizare
- Referirea unui identificator doar în domeniul său de vizibilitate
- Corespondența ca număr şi tip a argumentelor cu parametrii formali corespunzatori.

În aceste condiții în compilatoare se procedează astfel: pe baza gramaticii independente de context al limbajului de programare se implementează analiza sintactică, iar restricțiile dependente de context ramân să fie verificate în faza următoare, la analiza semantică.

Unei forme propoziționale derivate în cadrul unei gramatici independente de context pot să-i corespundă mai multe derivări considerate a-i fi echivalente.

Exemplu:

```
S -> aAS | a
A -> Sba | ba
aabbaa

S => aAS => aSbAS => aabAS => aabbaS => aabbaa
S => aAS => aAa => aSbAa => aSbbaa => aabbaa
```

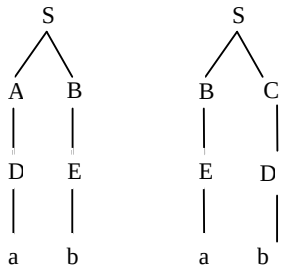
În primul caz s-a derivat în fiecare formă propozițională cel mai din stânga neterminal, derivarea se numeşte derivare stânga ($\vec{s}$). In cazul doi avem derivare dreapta ($\vec{d}$).

Cazuri particulare de gramatici independente de context

a) Gramatici ambigue

Pentru a recunoaşte dacă o propoziție face sau nu parte dintr-un limbaj, este important ca arborele ce se asociază propoziției să fie unic. Se consideră următoarea gramatică:

```
S -> AB | BC          C -> D
A -> D                D -> a | b
B -> E                E -> a | b
```



Arbori sintactici pentru propoziția *ab*

Pentru propoziția dată se pot forma 2 arbori. Se spune că gramatica este ambiguă.

O gramatică independentă de context se numeşte ambiguă dacă în limbajul generat există cel puțin o propoziție care admite mai mult decât un arbore de derivare obținut prin derivări stângi distincte. În caz contrar gramatica se consideră neambiguă. Ambiguitatea este un obstacol în anumite tipuri de analiză sintactică. În mod frecvent însă, gramaticile ambigue, sunt descrieri mai compacte şi mai clare ale limbajelor generate.

G_1 : $S \rightarrow i^5 | SS$
G_2 : $S \rightarrow i^5 | i^5 S$

Se observă că cele două gramatici sunt echivalente

$L(G_1) = L(G_2) = \{ i^{5k} \ / \ k \geq 1 \}$

G_1 este ambiguă.

i^{15}
$S \Rightarrow SS \Rightarrow SSS \Rightarrow i^5SS \Rightarrow i^{10}S \Rightarrow i^{15}$
$S \Rightarrow SS \Rightarrow i^5S \Rightarrow i^5SS \Rightarrow i^{10}S \Rightarrow i^{15}$
$\Rightarrow$ G_1 ambiguă

G_2 este neambiguă

$S \Rightarrow i^5S \Rightarrow i^{10}S \Rightarrow i^{15}$

Acelaşi limbaj poate fi generat atât de gramatici ambigue cât şi de gramatici neambigue. O mică modificare în gramatică a transformat-o dintr-o gramatica ambiguă într-una neambiguă.

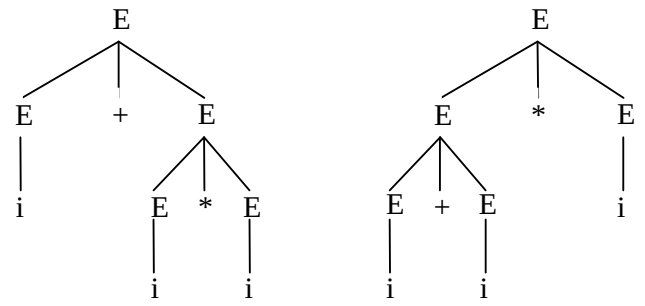
În mod frecvent, operând schimbări în setul de relații, al unei gramatici ambigue, se poate elimina ambiguitatea fără a modifica limbajul generat de gramatică.

Din păcate s-a demonstrat faptul că nu există un algoritm general care să arate, cu certitudine şi într-un timp finit, dacă o gramatică este sau nu ambiguă.

Exemplu de transformare a unei gramatici ambigue într-o gramatică neambiguă echivalentă:

G_1:E → E+ E | E * E | (E) | i

i + i * i



Arbori sintactici diferiți pentru acelaşi şir de intrare

Concluzia: gramatica G_1 este ambiguă. Ambiguitatea constă în faptul că nu ştim ce relație urmează să fie utilizată pentru prima derivare

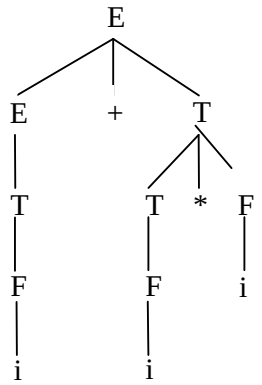
E → E + E | E * E

deoarece ambele sunt valabile. Ambiguitatea constă în neincluderea în gramatică a priorității operatorilor. Gramatica transformată este:

G_2: E → T | E + T

T → F | T * F

F → (E) | i



Arborele sintactic unic

G_2 este neambiguă, arborele prezentat este unic. Se poate demonstra şi matematic faptul că:

$L(G_1) = L(G_2),\ G_1 \sim G_2$

În plus pentru a stabili valoarea expresiei E în ansamblu, trebuie cunoscută valoarea termenului T (fiul drept al rădăcinii), adică înmulțirea trebuie efectuată înaintea adunării. Producțiile gramaticii G_2 au fost aranjate în sens invers priorității.

b) Gramatici recursive

Fie gramatica G şi un neterminal oarecare, A. Se consideră următorul şir de transformări în cadrul gramaticii:

$$A \underset{G}{\Rightarrow} \alpha A \beta,\ \alpha,\beta \in V_G^*,\quad V_G = N \cup T$$

Dacă $\alpha = \varepsilon$, A se numeşte stâng recursiv, dacă $\beta = \varepsilon$, A se numeşte drept recursiv.

O gramatică în care există cel puțin un simbol neterminal stâng recursiv, se numeşte gramatică recursivă la stânga. Similar la dreapta.

Recursivitatea la stânga este şi ea o piedică în calea analizei sintactice descendente şi va trebui eliminată. Spre deosebire de ambiguitate, pentru eliminarea recursivității de stânga există algoritmi riguroşi, precisi (vezi cap 4).

În teoria limbajelor formale se demonstrează că pentru orice gramatică independentă de context, există o gramatică echivalentă nerecursivă la stânga.

Operații asupra limbajelor

Fie A un alfabet dat şi $\mathcal{L}(A^*)$ mulțimea tuturor limbajelor care se pot forma cu elementele lui A (mulțimea submulțimilor lui A^*).

Considerând $L_1,\ L_2 \in \mathcal{L}(A^*)$ se pot defini următoarele operații de bază asupra limbajelor provenite din operații între mulțimi:

a) **Reuniunea**: $L_1 \cup L_2 = \{s \mid s \in L_1 \text{ sau } s \in L_2\}$
b) **Produsul** sau **Concatenarea**: $L_1L_2 = \{s_1s_2 \mid s_1 \in L_1 \text{ şi } s_2 \in L_2\}$
c) **Puterea**, se defineşte pe baza produsului, astfel:

$$L^n = \begin{cases} \{\varepsilon\} & n=0,\ \varepsilon\text{-şirul vid} \\ L^{n-1}L & \forall n \in N,\ n \geq 1 \end{cases}$$

d) **Închiderea Kleene** sau **Stea**:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

e) **Inchiderea pozitivă** sau **Plus**:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Observație: $L^*=L^+\cup\{\varepsilon\}$

Exemple: Se consideră următoarele mulțimi care pot fi privite atât ca alfabete cât şi ca limbaje (şiruri de lungime unu):

$L=\{A,B,\ldots,Z,a,b,\ldots,z\}$
$C=\{0,1,\ldots,9\}$

Aplicând unele operații dintre cele introduse anterior se pot defini următoarele mulțimi noi:

$L\cup C$	- mulțimea tuturor literelor şi cifrelor
LC	- toate şirurile de lungime doi în care primul caracter este literă iar al doilea, cifră
L^4	- mulțimea tuturor şirurilor formate din patru litere
L^*	- mulțimea tuturor şirurilor posibile de litere, de orice lungime, formate pe baza schemei aranjamentelor cu repetiție, inclusiv şirul vid
C^+	- mulțimea tuturor şirurilor de cifre care conțin cel puțin o cifră
$L(L\cup C)^*$	- mulțimea identificatorilor: toate şirurile de litere şi cifre care încep cu o literă.

Mulțimi regulate

Fiind dat un alfabet A, o mulțime regulată este un limbaj aparținând lui $\mathcal{L}(A^*)$ cu anumite proprietăți particulare.

Totalitatea multimilor regulate se numeste clasa multimilor regulate peste alfabetul A si se noteaza cu $\mu(A)$.

Definiție:

$\mu(A)$ este cea mai mică submulțime a lui $\mathcal{L}(A^*)$ care satisface următoarele condiții:

1. $\Phi\in\mu(A)$
2. $\{\varepsilon\}\in\mu(A)$
3. $\{a\}\in\mu(A),\ \forall a\in A$
4. $\mu(A)$ este inchisa fata de reuniune, produs si operatia *, adică:

   $\forall\ L,\ L_1,\ L_2\in\mu(A)$
   a) $L_1\cup L_2\in\mu(A)$
   b) $L_1L_2\in\mu(A)$
   c) $L^*\in\mu(A)$

Se poate demonstra că $\mu(A)=\mathcal{L}_3(A)$, adică un limbaj este regulat dacă şi numai dacă este o mulțime regulată.

Limbajele regulate sunt foarte potrivite şi în acelaşi timp suficiente pentru a exprima sintaxa simbolurilor lexicale.

Expresii regulate

Sunt modalități formale pentru specificarea mulțimilor regulate. La modul general vom nota o expresie regulată cu **r**; limbajul regulat corespunzator **L(r).**

O expresie regulată se poate construi pe baza următoarelor 3 reguli:

1. $\underline{\varepsilon}$ este o expresie regulata care desemneaza limbajul $\{\varepsilon\}$
2. $\underline{a}$ este o expresie regulată care desemnează limbajul $\{a\}$, $\forall a \in A$
3. presupunând r, r_1, r_2 ca fiind expresii regulate desemnând limbajele $L(r)$, $L(r_1)$, $L(r_2)$ atunci:
   a) $(r_1)|(r_2)$ va fi o expresie regulată care desemneaza limbajul $L(r_1) \cup L(r_2)$
   b) $(r_1)(r_2)$ va fi o expresie regulată care desemnează limbajul $L(r_1)L(r_2)$
   c) $(r)^*$ va fi expresia regulată ce desemnează $(L(r))^*$
   d) (r) desemnează limbajul L(r); parantezele nu schimbă semantica.

Dacă nu există pericolul unor ambiguități, într-o expresie regulată se poate renunța la parantezele redundante. Pentru aceasta trebuie ținut cont de următoarele proprietăți algebrice ale operatorilor:

- Toți cei trei operatori sunt asociați la stânga
- Operatorul unar stea (*) are prioritatea cea mai înaltă. Urmează, în ordine, produsul şi reuniunea (|).

Exemple: `litera(litera|cifra)*` este o expresie regulată care desemnează mulțimea identificatorilor: mulțimea $L(L \cup C)^*$ din exemplele anterioare.

`(a)|((b)*(c))` este echivalentă cu:
`a|b*c`
$(a^*b^*)^* \rightarrow A^*$

Operațiile aplicate expresiilor regulate au următoarele proprietăți algebrice:

1. Reuniunea este atât comutativă cât şi asociativă:
   $r_1|r_2 = r_2|r_1$
   $r_1|(r_2|r_3) = (r_1|r_2)|r_3$

2. Produsul este asociativ, nu şi comutativ:
   $r_1\ (r_2\ r_3) = (r_1\ r_2)\ r_3$

3. Produsul este distributiv față de reuniune atât la stânga cât şi la dreapta:
   $r_1(r_2|r_3) = r_1r_2|r_1r_3$
   $(r_1|r_2)r_3 = r_1r_3|r_2r_3$

4. ε este element unitate pentru produs:
   $\varepsilon r = r\varepsilon = r$

5. Între operația * si ε este următoarea relație:

   $r^* = (r|\varepsilon)^*$

6. Operația * este idempotentă:

   $r^{**} = r^*$

Definiții regulate

Pentru simplificarea notației este convenabil să se introducă nume desemnând expresii regulate şi apoi să se definească alte expresii regulate mai complicate, utilizând aceste nume, ca şi când ar fi simboluri. Se obține astfel o definiție regulată. Ca notație, considerând A alfabetul de simboluri de bază, se numeşte *definiție regulată* o secvență de definiții de forma:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
…
$d_n \rightarrow r_n$

unde, fiecare d_i este un nume distinct şi fiecare r_i este o expresie regulată formată numai cu simboluri din mulțimea

$A \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Restricția impusă asupra numelor (doar cele definite anterior), evită apariția definițiilor recursive asigurând finalizarea procesului de substituție.

Pentru a fi distinse de simbolurile veritabile, se adoptă convenția că numele simbolice ce desemnează expresii regulate să fie îngroşate sau subliniate.

Având în vedere faptul că anumite construcții apar frecvent în descrierea expresiilor regulate, cele trei operații de bază pot fi extinse cu următoarele notații suplimentare:

a) Unul sau mai multe exemplare

Pornind de la semnificația stabilită pentru operatorul unar postfixat **+** prin operația de închidere pozitivă, expresia ee^* este echivalentă cu e^+.

b) Zero sau un exemplar

Se introduce operatorul unar postfixat ? astfel încât expresia $e|\varepsilon$ devine echivalentă cu $e?$.

c) Clase de caractere

Notația $[c_1c_2c_3c_4]$ va desemna expresia regulată $c_1|c_2|c_3|c_4$. Pentru intervale mai lungi şi compacte de caractere se pot indica doar capetele intervalului. Astfel $a|b|...|z$ devin $[a-z]$.

Se poate considera că formalismul expresiilor regulate este o alternativă convenabilă la notația pe bază de gramatici pentru limbaje regulate (multimi regulate).

Mulțimea simbolurilor lexicale pentru un limbaj de tip Pascal poate fi exprimată prin următoarea *definiție regulată*.

cifra → [0 – 9]
litera → [A – Z, a – z]
coment → { caracter*}
sircar → ‘ caracter* ‘
delimit → spatiu| ;| , | (|) | [|] | : | .. | .
operator → + | - | * | / | < | ≤ | > | ≥ | = | <> | :=
cifre → cifra^{+}
exponent → ((E|e)(+|-)?cifre)?
fractie → (. cifre)?
numar → cifre fractie exponent
constanta → numar|sircar
identif → litera(litera|cifra)*
program → (identif|constanta|operator| delimit|coment)$^{+}$