Tema: Analiza şi simularea construirii programului prin recunoasterea gramaticei dupa anumita sintaxa si reguli de transformare a grafului in program.

Obiective:

1. de studiat şi însuşit materialul teoretic pentru evidenţierea esenţialului în elaborarea modelelor;
2. să se elaboreze scenarii de modelare şi analiza simulării elementelor caracteristice produselor soft şi modulelor (funcţiilor) în baza diferitor modele de reprezentare pentru propria variantă (Exemplu);
3. de analizat tehnicile şi metodele de analiză sintactică ca parte componentă a analizei lexicale de recunoastere a corectitudinei.
4. în baza celor studiate de elaborat modelul experimental în forma unui program de simulare a analizei semantice ca fază a compilării în care au fost stabilite gramaticile de bază a limbajului C şi au fost implementate în program. De asemenea se execută şi se efectuiază analiza sintactică a codului sursă, adică se face verificarea punctuaţiei şi a utilizării parantezelor în textul programului.

Consideraţii teoretice

1. Noţiuni generale şi definiţii.

Domeniul care se preocupă de acest aspect ar fi „*prelucrarea limbajului natural*”, situat undeva la graniţa între lingvistică şi ştiinţele cognitive.

Principalul scop constă în ceea ce priveşte limbajul natural este permiterea comunicării între calculator şi om fără a fi necesară memorarea unor comenzi sau proceduri complexe, comunicarea urmând a se face prin comenzi simple, formulate în limbajul „de zi cu zi”.

Un alt scop important ar fi traducerea automată**,** aceasta urmând să permită oamenilor de ştiinţă, oamenilor de afaceri sau populaţiei de rând să comunice liber intre ei indiferent de naţionalitate şi limbă vorbită.

Toate acestea reprezintă doar o parte din aria vastă a inteligentei artificiale şi a limbajului natural, ce include şi ştiinţele cognitive în încercarea de a explica modul în care oamenii înţeleg de fapt limbajul (înţelegerea limbajului uman).

Părintele prelucrării limbajului natural este considerat omul de ştiinţă Naom Chomsky, studiile sale începând cu 1950, influenţând dezvoltarea ulterioară a limbajelor de programare prin intermediul limbajelor formale. Este memorabilă teza acestuia potrivit căreia “ne cunoaştem cunoscând limbajul”. Plecând de la această teză, Alan Turing, tot în 1950, propune ca una dintre cerinţele pe care un calculator trebuie să le îndeplinească pentru a fi considerat inteligent, să fie aceea de a fi capabil să înţeleagă si să răspundă într-un limbaj natural.

Limbajul natural este o modalitate de comunicare om-maşină care prezintă câteva **avantaje** majore de partea utilizatorului:

- Nu necesită instruire pentru folosirea lui;
- Scuteşte utilizatorul de familiarizarea cu limbajele formale;
- Este un mijloc direct de accesare a informaţiei, independent de structura şi codificarea acestuia.

O aplicaţie de prelucrare a limbajului natural poate prezenta următoarele trei componente:

1. ***scanner-ul***

Care realizează descompunerea unei propoziții/fraze într-o listă a cuvintelor sale componente

2. ***parser-ul*** care realizează analiza propozițiilor/frazelor conform regulilor specificate de gramatica adoptată precum şi obținerea semanticii aferente acestora. Pentru aceasta, se

apelează la analiza sintactică şi morfologică, la analiza semantică şi la analiza pragmatică;

3. ***executivul*** care preia semantica obținută în pasul anterior, transformându-o în comenzi sau cunoştințe. Noi înțelegem texte mari combinând forța noastră de înțelegere pentru textele mai mici. Principalul scop al teoriei lingvistice este acela de a arăta cum pârtile mari de text sunt formate din mai multe pârti mici. Acestea sunt modelate de gramatică. Lingvistica computațională încearcă să implementeze acest proces de recunoaştere într-un mod cât mai eficient. Este tradițional să subdivizăm acest task în *sintactic* şi *semantic*, unde *sintactica* descrie cum diferite elemente formale a unui text, de cele mai multe ori

propoziții, pot fi combinate, iar *semantica* descrie cum este calculată interpretarea.

Analiza sintactică ne ajută să înțelegem modalitatea prin care cuvintele se grupează în vederea construirii unei propoziții sau fraze. În acest sens, o propoziție este formată din constituenți sintactici, unii dintre ei fiind, la rândul lor, compuşi din constituenți. Constituenții care la rândul lor sunt formați din constituenți se numesc *non-lexicali* în timp ce toți ceilalți se numesc *lexicali*. Constituenții lexicali (cuvintele vocabularului) împreună cu caracteristicile lor morfologice (număr, timp, caz etc.) formează *lexiconul*.

1. Schema generală a unui compilator. Compilatorul este un program complex, a cărui realizare presupune abordarea sistemică a procesului de traducere. In procesul de compilare, PS suferă un şir de transformări în cascadă, din ce în ce mai apropiate de CO. Conceptual, un compilator realizează două mari clase de operații:

- *analiza textului sursă;*
- *sinteza codului obiect.*

La rîndul lor, aceste operații se descompun în suboperații specializate, înlănțuite între ele şi caracterizate prin funcții bine precizate. Din punctul de vedere al structurii sale, un compilator poate fi reprezentat schematic ca în fîg. 3.5. Se observă înlănțuirea etapelor analizei şi sintezei, precum şi ordinea naturală a acestora. Facem aici observația că această reprezentare se doreşte a fi generală, şi că în practică anumite etape pot fi considerate împreună.

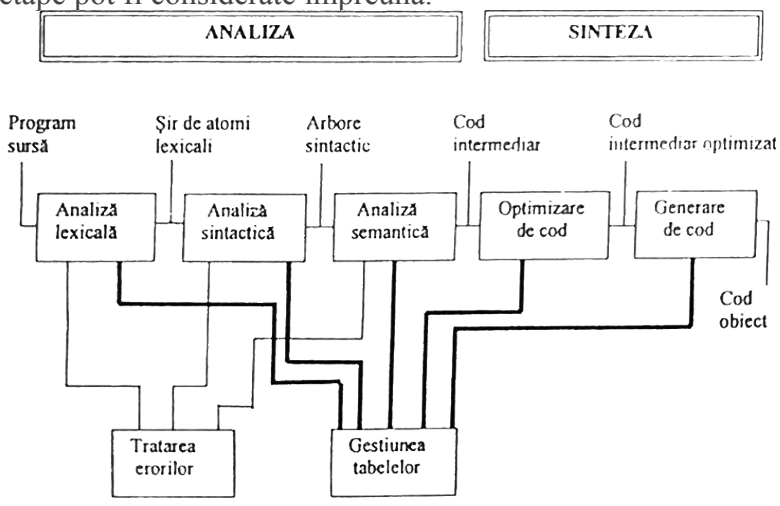

Figura 3.5. Structura unui compilator

Analiza lexicală realizează o primă parcurgere a PS (considerai ca sar de caractere), grupând aceste caractere in subsiruri. numite atomi lexicali: cuvinte cheie sau rezervate, operatori, constante, identificatori, separatori.

Analiza sintactica depistează in şirul atomilor lexicali structuri sintactice, expresii, liste, instrucţiuni, procedări s a , generând arborele sintactic (arborele de derivare), care descrie relaţiile dintre aceste structuri (de incluziune, de separare etc).

Analiza semantică foloseşte arborele sintactic pentru extragerea de informaţii privind apariţiile obiectelor purtătoare de date din PS (tipuri de date, variabile, proceduri, funcţii) şi pentru verificarea consistenţei utilizării lor Pe măsura parcurgerii arborelui sintactic, se generează codul intermediar. Acesta este un şir de instrucţiuni simple, cu format fix, în care: codurile operaţiilor sunt asemănătoare cu codurile maşină corespunzătoare, ordinea operaţiilor respectă ordinea execuţiei (dictată de apariţia acestora în PS), iar operanzii sunt reprezentaţi sub forma variabilelor din PS, şi nu sub formă de regiştri sau adrese de memorie.

Optimizarea codului intermediar are ca obiectiv eliminarea redundanţelor, a calculelor inutile, în scopul realizării unei execuţii eficiente a codului obiect Pentru atingerea acestui obiectiv se încearcă:

— realizarea tuturor calculelor posibile încă din faza de compilare (de exem-plu în expresii cu operanzi constante);

—eliminarea subexpresiilor comune (prin evaluarea lor o singură dată);

—factorizarea invarianţilor din cicluri.

Generarea programului (codului) obiect constă in alocarea de locaţii de memorie şi regiştri ai unităţii centrale pentru variabilele programului şi în înlocuirea codurilor de operaţii din codul intermediar cu cele maşină. Codul obiect rezultat poate fi:

— absolut (direct executabil);

— relocabil (care va face obiectul editării de legături, unde va fi legat de alte module obiect, rezultate din compilări anterioare sau din biblioteci);

— în limbaj de asamblare, lucru ce asigură un efort mai mic de implementare a generatorului de cod

sau

—în alt limbaj de programare în cazul preprocesoarelor.

în altă ordine de idei, generarea de cod diferă in funcţie de arhitectura maşinii (maşină cu acumulator şi cu registre de bază şi index sau maşină cu registre generale).

Cele 5 module de baza ale procesului de compilare sunt asistate de alte două componente ale compilatorului, ale căror servicii sunt utilizate pe tot parcursul compilarii: *modulul de tratare a erorilor* şi *gestionarul tabelei de simboluri:*

Modulul de tratare a erorilor este constituit dintr-o colecţie de proceduri care sunt activate ori de câte ori este detectată o eroare în timpul operaţiilor de analiza. Ele emit mesaje de diagnostic relative ia eroarea respectivă şi iau decizii privind modul de continuare a traducerii:

- traducerea se continuă cu ignorarea elementului ce conţine eroarea;
- se încearcă corectarea erorii

sau

- se abandonează procesul de traducere.

După momentul de analiză în care apar, erorile pot fi lexicale, sintactice sau semantice. Un alt criteriu de clasificare a erorilor este "gravitatea" lor; distingem avertismente (de obicei omisiuni de programare), erori care se pot "corecta" de către un compilator mai inteligent şi erori "fatale", care provoacă abandonarea

Gestionarul tabelelor este tot o colecţie de proceduri care realizenză crearea si actualizarea bazei de date a compilatorului, carc conţine două categorii de informaţii:

- proprii compilatorului (generate la implementare şi constituite din mecanismele de descriere a analizei lexicale, sintactice şi semantice)

si

- caracteristice PS (identificatori, constante, cuvinte cheie), care de obicei se memorează într-o *tabelă de simboluri.*

Există diverse modalităţi de gestionare a tabelei de simboluri, în funcţie de reprezentarea acesteia (tabel neordonat, tabel ordonat alfabetic sau dispersat -memorarea şi regăsirea făcând apel într-un astfel de caz la aşa numitele funcţii de dispersie hashing functions de tipul limbajului supus traducerii (cu sau fără structură de bloc), de convenţiile alese pentru reprezentarea atributelor în tabelă, de caracterul general (acoperitor) al tabelei (existenţa unei singure tabele de simboluri sau prezenţa unor tabele specializate pentru variabile, proceduri, constante, etichete) etc.

In faza de analiză lexicali (de obicei), la întâlnirea unui nume nou, acesta este introdus în tabela de simboluri, reţinîndu-se adresa intrării. Ori de câte ori numele este referit, informaţia prezenta în tabelă este actualizată cu informaţii sau atribute noi ale numelui respectiv, verificându-se totodată consistenţa utilizării acestuia (în analiza semantica). La generarea de cod, atributele numelui determina lungimea zonei de memorie alocata acestuia. In sfirsit, atributele
numelui pot servi si in faza de tratare a erorilor.

1.1 Analiza Faza de analiză cuprinde trei etape: *analiza lexicală, sintactica si semantica.* Aceste trei etape se pot implementa separat, aşa cum este ilustrat in figura 3.5, sau global.

1.2 Analiza lexicală. ***Analiza lexicală*** (engl. ***scanning,*** analizor lexical = scanner), transforma PS (considerat şir de caractere) într-un şir de unităţi lexicale, numite atomi ***lexicali*** sau particule lexicale (engl. ***tokens).*** Clasele de atomi lexicali, considerate ca mulţimi finite, corespund identificatorilor, constantelor întregi, constantelor reale şi operatorilor. Pe scurt, analizorul lexical realizează operaţii de detectare, clasificare şi traducere:

- detectarea în PS a subşirurilor ce respecta regulile de formare a atomilor lexicali;

- clasificarea acestor subşiruri (identificarea clasei acestora);

- traducerea subşirurilor în atomi lexicali;

- memorarea în tabela de simboluri a identificatorilor şi constantelor.

De obicei, analiza lexicală este considerată ca o etapă a analizei sintactice. Intre argumentele în favoarea separării acestor operaţii menţionăm:

- o proiectare mai simplă a analizorului sintactic şi a celui lexical, datorată principiului separării funcţiilor;
- posibilitatea scrierii analizorului lexical în limbaj de asamblare (el necesitând în general operaţii apropiate de maşină), şi în limbaje de nivel înalt a celorlalte componente ale compilatorului;
- deoarece sintaxa atomilor lexicali se poale exprima cu ajutorul unei gramatici regulate, modelul analizorului lexical poate fi un automat finit care este mai simplu de implementat decât automatele push-down utilizate în celelalte etape ale analizei;
- aportul analizorului lexical la "igienizarea" PS, deoarece:
  - numărul atomilor lexicali este mai mic decât numărul de caractere din PS;
  - se elimini blank-urile sau comentariile;
  - se poate prelua în analiza lexicală evaluarea unor construcţii dificil de implementat în analizorul sintactic, rezultatul fiind reducerea complexităţii analizei sintactice;

- creşterea portabilităţii algoritmului de compilare (de exemplu la dialecte diferite ale aceluiaşi limbaj, ce diferi prin cuvinte cheie, realizarea unui compilator înseamnă doar (scrierea analizorului lexical).

Din punctul de vedere al relaţiei cu analizorul sintactic, se deosebesc două *moduri* de abordare:

- analizor lexical independent de analizorul sintactic;

- analizor lexical comandat de analizorul sintactic, caz în care primul este o rutină a celui de al doilea.

Din punct de vedere structural, analizorul lexical poate fi *monobloc* (analiza lexicală este abordată şi rezolvată global) sau *structurat,* când gramatica atomilor lexicali este stratificată, rezultând de obicei trei subfaze: *transliterare* (clasificarea caracterelor PS), *explorare* (depistarea şi clasificarea subşirurilor de atomi lexicali) şi *selectare* (definitivarea clasificării, eliminarea caracterelor inutile şi codificarea atomilor lexicali).

Etapele ce trebuie parcurse în proiectarea unui analizor lexical sunt:

• tratarea gramaticii atomilor lexicali, prin rafinări succesive, obtinându-se o mulţime de gramatici regulate;

• stabilirea diagramei de tranziţie a automatului de recunoaştere echiva-lent gramaticilor regulate;

• completarea diagramei de tranziţie cu proceduri semantice asociate tranziţiilor (pentru semnalarea erorilor sau pentru emiterea unor ieşiri).

1.3 Analiza sintactică. *Analiza sintactică (engl. parsing)* este una dintre etapele principale ale procesului de traducere. Prin ea se realizează transformarea şirului de intrare (format din atomi lexicali) în:

- descrierea structurală a acestuia, semantic echivalentă (în cazul când şirul de intrare este corect sintactic)

respectiv

- mesaj de eroare (în caz contrar).

Descrierea structurală este echivalentă arborelui de derivare.

Construirea arborelui de derivare se face pe baza şirului atomilor lexicali şi poate fi:

- *ascendentă* (de la frunze spre rădăcină)

sau

- *descendentă* (de la rădăcină spre frunze).

Analizoarele sintactice se pot clasifica după modul de construire a arborelui de derivare (analizoare ascendente, respectiv descendente) sau după existenţa revenirilor în procesul de construire a acestui arbore (cu reveniri sau fără reveniri).

Modelul sintaxei limbajelor de programare este gramatica independentă de context, prin urmare, conform rezultatelor teoretice expuse în secţiunea precedentă, modelul analizorului sintactic va fi automatul push-down nedeter-minist. în simularea funcţionării acestuia, este importantă problema tratării revenirilor. Dacă la un *moment* dat *se* pot lua în considerare mai multe alternative de continuare a derivării, trebuie să se prevadă posibilitatea revenirii la o altă alternativă posibilă când alternativa aleasă nu conduce la un rezultat favorabil. Acest lucru măreşte timpul de realizare a analizei, motiv pentru care este de dorit ca la orice moment să fie asigurată o singură posibilitate de continuare a acesteia.

Analiza sintactică descendentă are ca model APDN şi se bazează pe derivarea stânga a şirului

acceptat Operațiile acestui automat sunt:

- *expandarea* (înlocuirea unui neterminal A din vârful stivei cu partea dreaptă a unei A-producții din gramatică) şi

- *avansul* (ştergerea unui terminal din vârful stivei atunci când el coincide cu terminalul curent din şirul de intrare, urmată de avansul intrării).

Criteriul de acceptare este criteriul stivei vide. Revenirile sunt modelate cu ajutorul unei stive auxiliare, în care se memorează configurația APDN la fiecare punct de ramificare a analizei, când sunt posibile mai multe alternative de derivare. In cazul analizoarelor sintactice descendente fără reveniri, modelul sintaxei este gramatica LL(k).

Analiza sintactică ascendentă are ca model APDN extins şi se bazează pe derivarea dreapta a şirului acceptat, parcursă în ordine inversă. Operațiile acestui automat sunt:

- *deplasarea* (căutarea în şirul de intrare, trecut succesiv în stivă, a părților drepte ale regulilor de producție ale gramaticii) şi

- *reducerea* (o parte dreaptă a unei reguli de producție este înlocuită cu neterminalul din partea stângă a regulii respective).

Criteriul de acceptare este epuizarea întregului şir de intrare, iar în stivă trebuie si rămână numai simbolul de start al gramaticii. In cazul general, când modelul sintactic al limbajului de tradus este o gramatică independentă de intext, analiza sintactică ascendentă este cu reveniri. Analiza sintactică acendentă fără reveniri utilizează ca model de sintaxă gramaticile LR(k).

1.4 Analiza semantică. Structura sintactică poate fi folosită pentru specificarea semanticii unui limbaj. în general, **semantica** (înțelesul) unei construcții poate fi exprimată prin orice cantitate sau mulțime de cantități asociate acelei construcții. O astfel de cantitate asociată se numeşte **atribut.** Ca exemple de atribute putem menționa: o mulțime de şiruri de caractere, o valoare, un tip, o configurație specifică de memorie etc. Această modalitate de abordare a semanticii se numeşte **orientată de sintaxă (syntax directed).** Regulile care definesc atributele unei construcții se numesc **reguli semantice.** O specificare de sintaxă împreună cu regulile semantice asociate realizează o **definiție orientată de sintaxă.**

De exemplu, dacă dezvoltăm un evaluator de expresii, semantica expresiei 2+3 poate fi exprimată prin valoarea ***5.*** Dacă dezvoltăm un translator din formă infixată în formă postfîxată semantica expresiei 2+3 ar putea fi exprimată sub forma tripletului (+ 2 3).

Concepută ca o finalizare a etapei de analiză a textului sursă (prin generarea codului intermediar), analiza semantică completează structurile sintactice cu valorile atributelor asociate fiecărei componente de structură. Cunoscându-se valorile atributelor, se realizează:

- validare semantică a PS;
- generarea codul intermediar echivalent

Acțiunile realizate în această ultimă fază a analizei sunt următoarele:

- atributarea arborelui de derivare (evaluarea atributelor asociate nodurilor acestuia);
- generarea de cod intermediar.

Aceste două obiective se realizează de obicei prin parcurgerea de mai multe ori a arborelui de sintaxă. In practică, analiza semantică se desfăşoară în paralel cu analiza sintactică, prin completarea acțiunilor analizorului sintactic cu acțiuni referitoare la anumite structuri de date ce reprezintă atribute ale componentelor sintactice.

Există diverse formalismc pentru analiza semantică, între care amintim:

- schemele generalizate de traducere orientate pe sintaxă (Aho);

- gramaticile de atribute extinse cu acțiuni de traducere (Marcotty),

Nu există însă un model unanim acceptat Formalismele existente depind de strategia aleasă în analiza sintactică (ascendentă sau descendenta)

1.5 **Unităţi de program.** Structura generală a unui program scris într-un limbaj de programare convențional (imperativ, dirijat de control) presupune existenta unui **program principal** şi eventual a unuia sau mai multor **subprograme** (proceduri, funcţii si subrutine) care comunică între ele şi/sau cu programul principal prin intermediul **parametrilor** şi/sau a unor **variabile globale.**

Orice program sau subprogram sursă, indiferent de limbajul în care **este** scris şi indiferent de sintaxa concretă a acestuia este divizat în două părți (nu neaparat distincte din punct de vedere fizic): partea de **declarații** şi partea imperativa Declarațiile, denumite uneori şi **instrucțiuni neexecutabile** sunt informații descriptive adresate compilatorului care descriu in principal atribute ale zonelor de date cum ar fi tipul, dimensiunea de reprezentare, eventual valori initiale etc. Partea imperativă conține instrucțiunile ce se vor executa în cursul rulării (sub)programului.

Ideea **reutilizării codului** precum şi cea a reducerii **dificultăților de proiectare şi întreținere** a programelor mari au condus în mod natural la **modularizarea** dezvoltării programelor. Corespunzător, ca rezultat al procesului de abstractizare şi factorizare, apar astfel la nivelul programelor, unitati de **program** distincte, numite **module,** cu rol bine precizat şi care aduc odată ru apariția lor numeroase avantaje. Unul dintre cele mai importante în acest sens este **compilarea separată.**

In majoritatea cazurilor activitățile pe care subprogramele le efectuează sunt independente. Astfel, unul sau mai multe subprograme pot fi grupate în module, care, fiind la rândul lor independente pot fi **compilate separat** unul de celălalt, adică la momente diferite în timp şi combinate mai târziu de către editorul de legături într-un unic program executabil. Ca urmare, dacă un modul necesită modificări şi celelalte nu, va fi recompilat doar modulul modificat, editorul de legături realizând apoi combinarea codului obiect rezultat cu versiunile deja compilate ale celorlalte module. Se economiseşte in acest fel un timp semnificativ de lucru, ideea compilării separate fiind extrem de utilă la întreținerea bibliotecilor mari de programe.

1.6 Rolul analizorului sintactic. Un analizor sintactic primeşte la intrare un şir de unităţi lexicale şi produce arborele de derivare al acestui şir în gramatica ce descrie structura sintactică (mai degrabă o anume reprezentare a acestui arbore). În practică, odată cu verificarea corectitudinii sintactice, se produc şi alte acţiuni în timpul analizei sintactice: trecerea în tabela de simboluri a unor informaţii legate de unităţile sintactice, controlul tipurilor sau producerea codului intermediar şi alte chestiuni legate de analiza semantică (unele din acestea le vom discuta în capitolul ce tratează semantica). Să ne referim în câteva cuvinte la natura erorilor sintactice şi la strategiile generale de tratare a acestora.

Dacă un compilator ar trebui să trateze doar programele corecte, atunci proiectarea şi implementarea sa s-ar simplifica considerabil. Cum însă programatorii scriu adesea programe incorecte, un compilator bun trebuie să asiste programatorul în identificarea şi localizarea erorilor. Cea mai mare parte a specificărilor limbajelor de programare nu descriu modul în care implementarea (compilatorul) trebuie să reacţioneze la erori; acest lucru este lăsat pe seama celor care concep compilatoarele. Erorile într-un program pot fi:

- lexicale: scrierea eronată a unui identificator, a unui cuvânt cheie, a unui operator etc. ;
- sintactice: omiterea unor paranteze într-o expresie aritmetică, scrierea incorectă a unei instrucţiuni, etc. ;
- semantice: aplicarea unui operator aritmetic unor operanzi logici, nepotrivirea tipului la atribuiri etc. ;
- logice: un apel recursiv infinit.

1.7 Problema recunoaşterii. Problema recunoaşterii în gramatici independente de context este următoarea: Dată o gramatică G = (V, T, S, P) şi un cuvânt w∈ T*, care este răspunsul la întrebarea w∈ L(G)? Se ştie că problema este decidabilă; mai mult, există algoritmi care în timp O(|w|) dau răspunsul la întrebare (Cooke-Younger-Kasami, Earley). Problema parsării (analizei sintactice) este problema recunoaşterii la care se adaugă: dacă răspunsul la întrebarea w e L(G) este afirmativ, se cere arborele sintactic (o reprezentare a sa) pentru w.

Fie G = (V, T, S, P) o gramatică şi w∈ L(G). Să notăm prin $\underset{st}{\overset{p}{\Rightarrow}}(\underset{dr}{\overset{q}{\Rightarrow}})$ relaţia de derivare extrem stângă (dreaptă) în care, pentru rescriere s-a aplicat producţia p e P (q∈ P).

Atunci, pentru w∈ L(G) există o derivare extrem stângă:

$$S = \alpha_0 \underset{st}{\overset{p_1}{\Rightarrow}} \alpha_1 \underset{st}{\overset{p_2}{\Rightarrow}} \ldots \alpha_n \underset{st}{\overset{p_n}{\Rightarrow}} \alpha_n = w$$

şi o derivare extrem dreaptă:

$$S = \beta_0 \underset{dr}{\overset{q_1}{\Rightarrow}} \beta_1 \underset{dr}{\overset{q_2}{\Rightarrow}} \ldots \beta m \underset{dr}{\overset{q_m}{\Rightarrow}} \beta_m = w$$

Atunci, arborele sintactic corespunzător se poate reprezenta prin secvenţa de producţii $p_1p_2\ldots p_n \in P+$ (respectiv q1q2…qm∈ P^+) care s-au aplicat, în această ordine, în derivarea extrem stângă (dreaptă) pentru obţinerea lui w.

Definiţia 1.2.1 Secvenţa de producţii π = p1p2…pn ∈ P^+ pentru care $S \underset{st}{\overset{\pi}{\Rightarrow}} = w$ se numeşte analizare stângă (parsare stângă) pentru cuvântul w∈ L(G).

Secvenţa de producţii π = q1q2…qn ∈ P^+ pentru care $S \underset{dr}{\overset{\pi}{\Rightarrow}} = w$ se numeşte analizare dreapta (parsare dreapta) pentru cuvântul w.

Exemplul 1.2.4 Fie gramatica

1. E → E+T 2. E → T 3. T→T*F
4. T→F 5. F→ (E) 6. F→id

Unde: 1, 2,…,6 identifică cele 6 producţii ale acesteia.

Pentru cuvântul w = `id+id*id`, arborele sintactic este dat în figura 1.1.

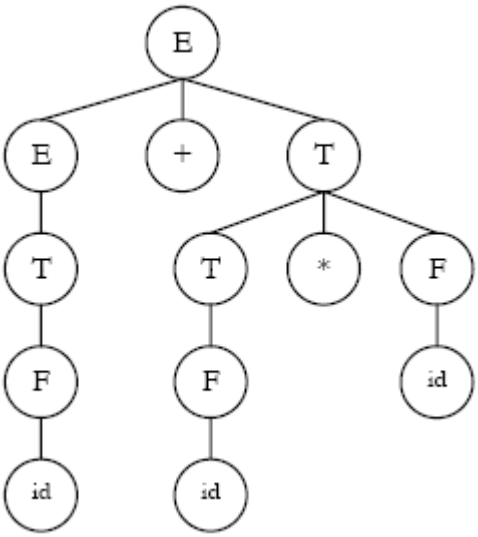

Figura 1.1

Analizarea stângă pentru acest cuvânt este: π1=12463466 iar analizarea dreaptă este π2=64264631. Aici am identificat fiecare producţie prin numărul său.

1.8 Analiza sintactică descendentă

Analiza sintactică descendentă (parsarea descendentă) poate fi considerată ca o tentativă de determinare a unei derivări extrem stângi pentru un cuvânt de intrare. In termenii arborilor sintactici, acest lucru înseamnă tentativa de construire a unui arbore sintactic pentru cuvântul de intrare, pornind de la rădăcină şi construind nodurile în manieră descendentă, în preordine (construirea rădacinii, a subarborelui stâng apoi a celui drept). Pentru realizarea acestui fapt avem nevoie de următoarea structură (figura 1.2):

- o bandă de intrare în care se introduce cuvântul de analizat, care se parcurge de la stânga la dreapta, simbol cu simbol;
- o memorie de tip stivă (pushdown) în care se obţin formele propoziţionale stângi (începând cu S). Prefixul formei propoziţionale format din terminali se compară cu simbolurile curente din banda de intrare obţinându-se astfel criteriul de înaintare în această bandă;
- o bandă de ieşire în care se înregistrează pe rând producţiile care s-au aplicat în derivarea extrem stângă care se construieşte.

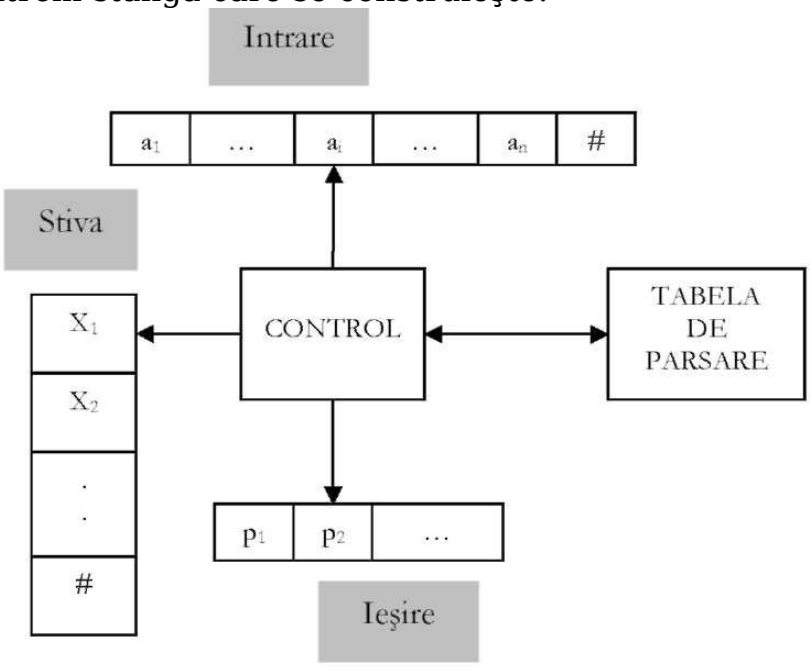

Figura 1.2

Criteriul de oprire cu succes este acela în care s-a parcurs întreaga bandă de intrare, iar memoria pushdown s-a golit. În acest caz în banda de ieşire s-a obţinut parsarea stângă a cuvântului respectiv. Iată un exemplu de cum funcţionează acest mecanism (considerăm gramatica din exemplul precedent şi cuvântul w = id+id*id). Vom considera caracterul # pentru marcarea sfârşitului benzii de intrare (marca de sfârşit a cuvântului) precum şi pentru a marca baza stivei.

Tabelul 1: Analiza sintactică descendentă pentru w = id+id*id

Pasul	Banda de intrare	Stiva de lucru	Banda de ieşire
1	id+id*id#	E#	
2	id+id*id#	E+T#	1
3	id+id*id#	T+T#	12
4	id+id*id#	F+T#	124
5	id+id*id#	id+T#	1246
6	id*id#	T#	1246
7	id*id#	T*F#	12463
8	id*id#	F*F#	124634
9	id*id#	id*F#	1246346

10	id#	F#	1246346
11	id#	id#	12463466
12	#	#	12463466

Se obţine la ieşire π= 12463466.

1.8.1 Parser descendent general. Fie G = (V, T, S, P) o gramatică care descrie sintaxa unui limbaj de programare. Să considerăm mulţimea C = T*# x Σ*# x P* numită mulţimea configuraţiilor (unui parser). Interpretarea pe care o dăm unei configuraţii este următoarea: dacă c = (u#, y#, π), atunci u# este cuvântul rămas de analizat (conţinutul benzii de intrare la un moment dat), Y# este conţinutul stivei, cu primul simbol din y în top-ul stivei (y= X1,X2,...,Xn ca în figura 1.2), iar # la baza acesteia, iar n este conţinutul benzii de ieşire: dacă π= p1,p2,...,pk, până în acest moment s-au aplicat producţiile din n în această ordine.

Definiţia 1.3.1 Parserul (analizorul sintactic) descendent ataşat gramaticii G = (V, T, S, P) este perechea (Co, ├) unde Co = {(w#, S#, s) | w∈ T* } ⊆ C} este mulţimea configuraţiilor iniţiale, iar ├⊆C x C este relaţia de tranziţie între configuraţii dată de următoarea schemă:

1. (u#, Ay#, π) ├ (u#, Py#, π r), unde r = A → P ∈ P.
2. (uv#, uy#, π) ├ (v#, Y#, π).
3. (#, #, π) este configuraţie de acceptare dacă π ≠ s.
4. O configuraţie c pentru care nu există c' astfel ca c ├ c' (în sensul 1, 2) spunem că produce eroare.

Să considerăm relaţia ├⁺ (├) închiderea tranzitivă (şi reflexivă) a relaţiei d definită mai sus. Este clar că, dacă vom considera parserul ca un sistem de rescriere, acesta este nedeterminist aşa cum este definită tranziţia de tip 1: dacă în P există producţiile A → β1 şi A → β2 cu β1≠β2, atunci au loc, în acelaşi timp: (u#, Ay#, π) d (u#, P1Y#, π), (u#, Ay#, πr1) d (u#, P2Y#, πr2) unde r1 = A →β1, r2 = A →β2. Pentru a fi corect, parserul definit mai sus va trebui să transforme (în mod nedeterminist, în general) configuraţia (w#, S#, s) în configuraţia (#, #, π) pentru cuvintele w e L(G) şi numai pentru acestea! Vom demonstra în continuare corectitudinea în acest sens.

Lema 1.3.1 Dacă în parserul descendent ataşat gramaticii G = (V, T, S, P) are loc calculul (uv#, Y#, ε) ├* (v#, ψ#, π), atunci în gramatica G are loc derivarea $\gamma = \overset{\pi}{\underset{st}{\Rightarrow}} u\psi$, oricare ar fi u, v∈ T*, y, Σ*, π∈ P*.

Demonstraţie. Să demonstrăm afirmaţia din lemă prin inducţie asupra lungimii lui π (notată |π|):

•Dacă | π | = 0 înseamnă că în calculul considerat s-au folosit doar tranziţii de tip π

2. Atunci Y = uψ şi are loc $\gamma = \overset{\pi}{\underset{st}{\Rightarrow}} u\psi$ cu π = ε ;

•Dacă | π | > 0 fie π = π_1r, unde r=A—>P este ultima producţie care se foloseşte în calculul considerat şi presupunem afirmaţia adevărată pentru calcule ce produc la ieşire n1. Aşadar, putem scrie:

(uv#,Y#,ε)=(u1u2v#,Y#,ε)├*(u2v#,AY1#,π_1)├(u2v#,βY1#,π_1r)=
(u2v#,u2ψ#,π_1r)├*(v#,ψ#, π_1r)

Atunci din (u1u2v#,Y#,ε)├*(u2v#,AY1#,π_1), conform ipotezei inductive, are loc $\gamma \overset{\pi}{\underset{st}{\Rightarrow}}$ u1Ay1. Cum în u1Ay1, u1∈ T*, iar A∈ V este cea mai din stânga variabilă, aplicând acesteia producţia A —> β, obţinem:

$\gamma \overset{\pi}{\underset{st}{\Rightarrow}} u_1 A\gamma_1 \overset{\pi}{\underset{st}{\Rightarrow}} u_1\beta\gamma_1 = u_1u_2\psi = u\psi$ Y == u1Ay1 şi demonstraţia este încheiată.

Corolarul 1.3.1 Dacă în parserul descendent are loc (w#, S#, ε)├⁺(#,#, π) atunci în gramatica G are loc $S \overset{\pi}{\underset{st}{\Rightarrow}} w$

Demonstraţie Se aplică lema precedentă pentru u = w, v = ε, Y = S, ψ= ε.

Lema 1.3.2 Dacă în gramatica G are loc derivarea $\gamma \overset{\pi}{\underset{st}{\Rightarrow}} u\psi$ şi 1: ψ∈ V, (vezi definiţia 1.3.2), atunci în parserul descendent are loc calculul:

(uv#, Y#, ε)├*(v#,ψ#,π), ∀ v ∈ T*.

Teorema 1.3.1 (Corectitudinea panerului descendent general). Se consideră gramatica redusă G = (V, T, S, P) şi w G T*. Atunci, în parserul descendent general ataşat acestei gramatici are loc (w#, S#, s) ⊢* (#, #, n) (acceptare) dacă şi numai dacă w e L(G) şi n este o analizare sintactică stângă a frazei w.

Demonstraţie Teorema sintetizează rezultatele din lemele 1.3.1 şi 1.3.2.

1.8.2 Gramatici LL(k). Parserul general introdus în paragraful precedent este un model nedeterminist iar implementarea sa pentru gramatici oarecare este ineficientă. Cea mai cunoscută clasă de gramatici pentru care acest model funcţionează determinist este aceea a gramaticilor LL(k): Parsing from Left to right using Leftmost derivation and k symbols lookahead. Intuitiv, o gramatică este LL(k) dacă tranziţia de tip 1 din Definiţia 1.3.1 se face cu o unică regulă A — P determinată prin inspectarea a k simboluri care urmează a fi analizate în banda de intrare. Dar să definim în mod riguros această clasă de gramatici.

Defininiţia 1.3.2 Fie G = (V, T, S, P) o gramatică, a ∈ 2* şi k > 1 un număr natural. Atunci definim:

k:a = if | a | < k then a else a1, unde a = a1P, | a1| = k,

a:k = if | a | < k then a else a2, unde a = ya2, | a2| = k.

Defininiţia 1.3.3 O gramatică independentă de context redusă este gramatică LL(k), k > 1, dacă pentru orice două derivări de forma:

$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\beta_1\gamma \overset{*}{\underset{st}{\Rightarrow}} ux$

$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\beta_2\gamma \overset{*}{\underset{st}{\Rightarrow}} uy$

unde u, x, y∈ T*, pentru care k: x = k: y, are loc β1 = β2:

Definiţia prezentată exprimă faptul că, dacă în procesul analizei cuvântului w=ux a fost obţinut(analizat) deja prefixul u şi trebuie să aplicăm o producţie neterminalului A (având de ales pentru acesta măcar din două variante), această producţie este determinată în mod unic de următoarele k simboluri din cuvântul care a rămas de analizat (cuvântul x).

1.8.3 O caracterizare a gramaticilor LL(1). Pentru ca un parser SLL(k) să poată fi implementat, trebuie să indicăm o procedură pentru calculul mulţimilor FIRSTk şi FOLLOWk. Pentru că în practică se foloseşte destul de rar (dacă nu chiar deloc) cazul k > 2, vom restrânge discuţia pentru cazul k = 1. Vom nota în acest caz FIRST1 şi FOLLOW1 prin FIRST respectiv FOLLOW. Aşadar, dacă a £ £+, A £ V :

$\mathrm{FIRST}(\alpha) = \{a \mid a \in T, \alpha \underset{st}{\Rightarrow} au\} \cup \text{if } (\alpha \underset{st}{\Rightarrow} \varepsilon) \text{ then } \{\varepsilon\} \text{ else } \emptyset.$

$\mathrm{FOLLOW}(A) = \{a \mid a \in T \cup \{\varepsilon\}, S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma, a \in \mathrm{FIRST}(\gamma)\}$

Mai întâi să arătăm că gramaticile SLL(1) coincid cu gramaticile LL(1).

Teorema 1.3.10 O gramatică G = (V, T, S, P) este gramatică LL(1) dacă şi numai dacă

pentru orice A £ V şi pentru orice producţii A — P1 | P2 are loc:

FIRST (P1FOLLOW (A)) 3 FIRST (P2FOLLOW (A)) = 0 Demonstraţie Să presupunem că G este LL(1). Acest lucru este echivalent cu(după Definiţia 1.3.3):

$$\left.\begin{array}{l} S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\overset{*}{\Rightarrow}} u\beta_1\gamma \underset{st}{\overset{*}{\Rightarrow}} uv_1 \\ S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\overset{*}{\Rightarrow}} u\beta_2\gamma \underset{st}{\overset{*}{\Rightarrow}} uv_2 \\ 1{:}v_1 = 1{:}v_2 \end{array}\right\} \Rightarrow \beta_1 = \beta_2$$

Să presupunem că G nu este SLL(1): există producţiile A — a1, A — a2 şi a £ FIRST (a1FOLLOW (A)) 3 FIRST (a2FOLLOW (A)) Distingem următoarele cazuri:

- a £ FIRST (a1) 3 FIRST (a2), adică $\alpha_1 \underset{st}{\overset{*}{\Rightarrow}} au_1, \alpha_2 \underset{st}{\overset{*}{\Rightarrow}} au_2$

Atunci, putem scrie derivările (G este redusă):

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_1v$$

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_2v$$

ceea ce contrazice faptul că G este LL(1).

- $a \in FIRST(\alpha_1) \cap FOLLOW(A)$ şi $\alpha_2 \underset{st}{\overset{*}{\Rightarrow}} \varepsilon$

Atunci: $\alpha_1 \underset{st}{\overset{*}{\Rightarrow}} au_1$ şi $S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma$, $a \in FIRST(\gamma)$ adică $\gamma \underset{st}{\overset{*}{\Rightarrow}} au_2$, şi putem scrie:

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_1v$$

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \underset{st}{\overset{*}{\Rightarrow}} u\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_2$$

ceea ce contrazice de asemenea ipoteza.

- $a \in FIRST(\alpha_2) \cap FOLLOW(A)$ şi $\alpha_1 \underset{st}{\overset{*}{\Rightarrow}} \varepsilon$, caz ce se tratează analog cu precedentul.
- $\alpha_1 \underset{st}{\overset{*}{\Rightarrow}} \varepsilon, \alpha_2 \underset{st}{\overset{*}{\Rightarrow}} \varepsilon$ şi atunci $a \in FOLLOW(A)$.

În acest caz avem:

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \underset{st}{\overset{*}{\Rightarrow}} u\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_2$$

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \underset{st}{\overset{*}{\Rightarrow}} u\gamma \underset{st}{\overset{*}{\Rightarrow}} uau_2$$

Deci din nou se contrazice proprietatea LL(1) pentru gramatica G.

Invers, să presupunem că pentru orice două producţii A P13 A →P2 distincte are loc FIRST (P1FOLLOW (A)) 3 FIRST (P2FOLLOW (A)) = 0 şi, prin reducere la absurd, G nu este LL(1). Asta înseamnă că există două derivări:

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \underset{st}{\overset{*}{\Rightarrow}} uav \text{ şi}$$

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \underset{st}{\overset{*}{\Rightarrow}} uav'$$

cu 1:v = 1:v' dar α1≠ α2.

Analizând cele două derivări, se constată uşor că:

$a \in FIRST(\alpha_1 FOLLOW(A)) \cap FIRST(\alpha_2 FOLLOW(A))$

ceea ce contrazice ipoteza.

1.8.4 Tabela de analiză sintactică LL(1). Pentru a implementa un analizor sintactic pentru gramatici LL(1), să considerăm o tabelă de analiză, sau tabelă de parsare LL(1):

M : V x (T U{ # }) — {(P, p) |p = A — P £ P }U {eroare} construită după algoritmul următor:

Algoritmul 1.3.6 (Tabela de analiză sintactică) Intrare:Gramatica G = (V, T, S, P); Mulţimile FIRST(P), FOLLOW(A), A — P £ P.

Ieşire:Tabela de parsare M.

Metoda: Se parcurg regulile gramaticii şi se pun în tabelă `for(A £ V )`

```
for(a£TU{#})M(A,a)=0; for(p =A—P£P){
for(a£FIRST(P)-{s})
M(A, a) = M(A, a) U{( P,p)} ;
if(s £ FIRST(B)){ for(b £ FOLLOW(A)){
if(b == s)M(A, #) = M(A, #) U{( P,p)};
10.  else  M(A, b) = M(A, b) U{( P,p)};
}//endfor
}//endif
}//endfor
11.   for(A £ V)
12.   for( a £ T U { # })
13.   if(M(A,a)=0) M(A,a)={eroare};
```

Lema 1.3.6 (caracterizarea gramaticilor LL(1)) Gramatica redusă G este LL(1) dacă şi numai dacă pentru orice A £ V şi orice a £ T U { # } M(A, a), dacă nu este eroare, conţine cel mult o pereche (P, p).

Demonstraţie Dacă G este LL(1), atunci VA £ V, oricare ar fi două producţii cu partea stângă A, A→P1, A→P2, are loc, conform teoremei 1.3.10, FIRST(P1FOLLOW(A)) 3 FIRST(P2FOLLOW(A))= O. Dacă ar exista în tabela construită A £V şi a £V U { # } astfel încât M(A, a) 3 {(P, p), (P', q)}, atunci, dacă a £ T rezultă

a £ FIRST(PFOLLOW(A)) 3 FIRST(P'FOLLOW(A))

ceea ce contrazice ipoteza. Dacă a = #, atunci s ar fi comun în FIRST(PFOLLOW(A)) şi FIRST(P'FOLLOW(A)), din nou absurd. Deci M îndeplineşte condiţia enunţată. In mod analog se dovedeşte implicaţia inversă.

1.3.5 Analizorul sintactic LL(1). Parserul (analizorul sintactic) LL(1) este un parser top-down în care tranziţiile sunt dictate de tabela de parsare. Aşadar, un astfel de parser are ca şi configuraţie iniţială tripleta (w#, S#, s) unde w este cuvântul de analizat, iar tranziţiile se descriu astfel:

1. (u#, Ay#, n) d (u#, Py#, nr) dacă M(A, 1:u#) = (P, r). (operaţia expandare)
2. (uv#, uy#, n) d (v#, y#, n). (operaţia potrivire)
3. (#, #, n) d acceptare dacă n g s.
4. (au#, by#, n) d eroare dacă a g b.
5. (u#, Ay#, n) d eroare dacă M(A, 1:u#) = eroare.

Teorema de corectitudine a parserului descendent general împreună cu teorema de caracterizare LL(1), a modului în care s-a definit tabela M, dovedesc corectitudinea parserului LL(1). Suntem acum în măsură să indicăm modul de

implementare al unui analizor sintactic LL(1). Presupunem că dispunem de o bandă de intrare (fişier de intrare) din care, cu o funcţie `getnext()`, se obţine caracterul (tokenul) următor. De asemenea dispunem de o stivă cu funcţiiile specifice `pop()` şi `push()` . Producţiile care se aplică pe parcursul analizei le scriem într-o bandă (fişier) de ieşire cu funcţia write(). Atunci algoritmul de analiză sintactică se descrie astfel: Algoritmul 1.3.6 (Parser LL(1))

Gramatica G = (V, T, S, P). Tabela de analiză LL(1) notată M. Cuvântul de intrare w#.

Analiza sintactică stângă n a lui w dacă w £ L(G), eroare în caz contrar.

Sunt implementate tranziţiile 1 — 5 folosind o stivă St.

```
St.push(#),St.push(S)//  St = S# a = getnext(),  n = s;
    do {
    X = St.pop();
    if(X == a)
    if(X   !=   '#')   getnext(); else{
    if   (n != s){write("acceptare");  exit(0);} else  {write("eroare");
exit(1);} }   //endelse else{
    if(X£T){write("eroare"); exit(1);} else{
    if(M(X,a)
    {write( else  {
    // M(X,a) = (P,r), r=X — P, P=Y1Y2...Yn //P inlocueste pe X in stiva
  for(k = n;   k>0;  --k)   push(Yk)
  write(r);   //se adauga r la n
    } //endelse
    } //endelse }   //endelse
    } while(1);
```

Exemplul 1.3.4 Să reluăm gramatica din exemplele precedente:

1. S →E
2. S → B
3. E → s
4. B → a
5. B →begin SC end
6. C →s
7. C →;SC

Mulţimile FIRST şi FOLLOW sunt date în tabelul următor:

X	FIRST(X)	FOLLOW(X)
S	a begin s	end ; s
E	s	end ; s
B	a begin	end ; s
C	; s	end

Tabela de analiză LL(1) pentru această gramatică este dată mai jos:

M	a	begin	end	;	
S					
E	eroare	eroare	(s, 3)	(s, 3)	(s, 3)
B		(begin SC end, 5)	eroare	eroare	eroare
C	eroare	eroare		(;SC, 7)	eroare

Se vede din tabelă că această gramatică este LL(1) (conform teoremei 1.3.13). Iată şi două exemple de analiză, unul pentru cuvântul `begin a;;a end` care este din limbajul generat de gramatica dată, iar altul pentru `begin aa end` care nu este corect.

PAS	INTRARE		STIVA	OPERAŢIE	IEŞIRE
1	begin a;	a end#	S#	expandare	
2	begin a;	a end#	B#	expandare	2
3	begin a;	a end#	begin SC end#	potrivire	5
4	a;	a end#	SC end#	expandare	
5	a;	a end#	BC end#	expandare	2
6	a;	a end#	aC end#	potrivire	4
7		a end#	C end#	expandare	
8		a end#	;SC end#	potrivire	7
9		a end#	SC end#	expandare	
10		a end#	EC end#	expandare	1
11		a end#	C end#	expandare	3
12		a end#	;SC end#	potrivire	7
13	a end#		SC end#	expandare	
14	a end#		BC end#	expandare	2
15	a end#		aC end#	potrivire	4
16	end#		C end#	expandare	
17	end#		end#	potrivire	6
18	#		#	acceptare	
1	begin aa end#		S#	expandare	
2	begin aa end#		B#	expandare	2
3	begin aa end#		begin SC end#	potrivire	5
4	aa end#		SC end#	expandare	
5	aa end#		BC end#	expandare	2
6	aa end#		aC end#	potrivire	4
7	a end#		C end#	eroare	

1.8.7 Eliminarea recursiei stângi. O gramatică stâng recursivă nu este LL(k) pentru nici un număr k. Este cazul, spre exemplu, al gramaticii care generează expresiile aritmetice:

E → E+T |E - T|-T |T T → T*F |T/ F| F

F → (E) | a

Prin eliminarea recursiei stângi, există posibilitatea obţinerii unei gramatici din clasa LL. Acesta este un procedeu utilizat frecvent în proiectarea analizoarelor sintactice. Să reamintim că o gramatică G este stâng recursivă dacă există un

neterminal A pentru care A = Aa. Spunem că un neterminal A este stâng recursiv imediat dacă există o producţie A — Aa în P. Să indicăm mai întâi un procedeu de eliminare a recursiei stângi imediate.

Lema 1.3.7 Fie G = (V, T, S, P) o gramatică pentru care A este stâng recursiv imediat. Să considerăm toate A — producţiile gramaticii, fie ele A — Aa1 | Aa2 | ... | Aan cele cu recursie şi A — P1 | P2 | ... | Pm cele fără recursie imediată (părţile drepte nu încep cu A). Există o gramatică G' = (V', T, S, P') echivalentă cu G, în care A nu este stâng recursiv imediat.

Demonstraţie Să considerăm un nou neterminal A' 2 V şi fie V' = V U{A} iar P' se obţine din P astfel:

- Se elimină din P toate A -producţiile;
- Se adaugă la P următoarele producţii:

A →PiA' 1 < i < m. A' — ajA' | s 1 < j < n.

•Notăm cu P' mulţimea de producţii obţinută.

Să observăm că A nu mai este stâng recursiv în G'. Noul simbol neterminal introdus, notat cu A', este de data aceasta drept recursiv. Acest lucru însă nu este un inconvenient pentru analiza sintactică descendentă.

Să observăm că, pentru orice derivare extrem stânga în gramatica G, de forma:

există în gramatica G' o derivare extrem dreapta de forma:

A = PhA' = Phak1 A' = ... = fta^. • -a^ A' = fta^.. .a^.

De asemenea, afirmaţia reciprocă este valabilă.

Aceste observaţii conduc la concluzia că cele două gramatici sunt echivalente.

Exemplul 1.3.5 În gramatica expresiilor aritmetice:

E → E+T |E-T |-T |T

T → T*F | T/F | F

F → (E) | a se fac transformările următoare:

•Producţiile E → E+T |E-T |-T |T se înlocuiesc cu:

E → TE' | -TE', E' → +T E'|-TE' | s

•Producţiile T → T*F | T/ F | F se înlocuiesc cu:

T → FT', T' → *FT' | /FT' | s

Se obţine astfel gramatica echivalentă:

E → TE' | -TE'

E' → +T E'|-TE' | s

T → FT'

T' → *FT' | /FT' | s

F → (E) | a

Aplicând algoritmii pentru determinarea mulţimilor FIRST şi FOLLOW se obţine:

X	FIRST(X)	FOLLOW(X)
E	(a -	s)
E'	+ - s	s)
T	(a	+ - s)
T'	* / s	+ - s)
F	(a	* / + - s)

Tabela de parsare pentru această gramatică este dată mai jos.

M	a	+	-	*	/	(	)	#
E	(TE', 1)	eroare	(-TE', 2)	eroare	eroare	(TE', 1)	eroare	eroare
E'	eroare	(+TE',3)	(-TE', 4)	eroare	eroare	eroare	(s, 5)	(s, 5)
T	(FT', 6)	eroare	eroare	eroare	eroare	(FT', 6)	eroare	eroare
T'	eroare	(s, 9)	(s, 9)	(*FT',7)	(/FT',8	eroare	(s, 9)	(s, 9)

					)			
F		eroare	eroare	eroare	eroare	((E) 10)	eroare	eroare

Se observă de aici că gramatica este LL(1). Să urmărim analiza sintactică pentru expresia -(a+a)*a.

PASUL	INTRARE	STIVA	OPERAŢIE	IEŞIRE
1.	-(a+a)*a#	E#	expandare	
2.	-(a+a)*a#	-TE'#	potrivire	4
3.	(a+a)*a#	TE'#	expandare	
4.	(a+a)*a#	FT'E'#	expandare	6
5.	(a+a)*a#	(E)T'E'#	potrivire	10
6.	a+a)*a#	E)T'E'#	expandare	
7.	a+a)*a#	TE')T'E'#	expandare	1
8.	a+a)*a#	FT'E')T'E'#	expandare	6
9.	a+a)*a#	aT'E')T'E'#	potrivire	11
10.	+a)*a#	T'E')T'E'#	expandare	
11.	+a)*a#	E')T'E'#	expandare	9
12.	+a)*a#	+TE')T'E'#	potrivire	3
13.	a)*a#	TE')T'E'#	expandare	
14.	a)*a#	FT'E')T'E'#	expandare	6
15.	a)*a#	aT'E')T'E'#	potrivire	11
16.	)*a#	T'E')T'E'#	expandare	
17.	)*a#	E')T'E'#	expandare	9
18.	)*a#	)T'E'#	potrivire	5
19.	*a#	T'E'#	expandare	
20.	*a#	*FT'E'#	potrivire	7
21.	a#	FT'E'#	expandare	
22.	a#	aT'E'#	potrivire	11
23.	#	T'E'#	expandare	
24.	#	E'#	expandare	9
25.	#	#	acceptare	5

Să arătăm cum eliminăm în general recursia stângă într-o gramatică. Vom face ipoteza că + gramatica G nu are s - producţii şi nu are cicluri de forma A == A. Aceasta nu este de fapt o restricţie: este ştiut faptul că, pentru orice gramatică G există G' fără s -producţii şi fără cicluri astfel încât L(G') = L(G) - { s }. (vezi [Gri86], [Juc99]). Aşadar are loc următoarea teoremă:

Teorema 1.3.13 Pentru orice gramatică G fără s - producţii şi fără cicluri, există o gramatică echivalentă G' care nu este stâng recursivă.

Demonstraţie Să considerăm următorul algoritm:

Intrare: Gramatica G = (V, T, S, P) fără s-producţii, fără cicluri.

Ieşire: Gramatica G' fără recursie stângă astfel ca L(G)=L(G').

Metoda: După ordonarea neterminalilor se fac transformări ce nu schimbă limbajul gramaticii, inclusiv prin eliminarea recursiei imediate

```
Se consideră V = [Ai, A2,  -,An|
for (i=1;  i < n;)
for (j=1;  j < i - 1;){
for (Ai — AjY e P)  {
P = P -  { Ai — AjY };
for  (Aj— p e P )  P = P U { Ai — Py};
}  //endfor
8.       Se elimina recursia imediata pentru Ai;
//endfor
}  //endfor
8.   V'= V; P'=P; // V s-a modificat la 7  iar P la 4-6
```

Să observăm că transformările din 4 - 7 nu modifică limbajul generat de gramatică încât L(G')=L(G). Să arătăm acum că G' nu are recursie stângă. Acest lucru rezultă din faptul că, după iteraţia i - 1 a buclei 2, fiecare producţie de forma Ak → Ala pentru k < i are proprietatea că l > k. Aceasta datorită transformărilor 4 - 6 şi eliminării recursiei stângi imediate în linia 7: după execuţia buclei 3 se obţin producţii A → Ama cu m > i, iar dacă m = i, producţia Ai → Aia se elimină în linia 7. Aşadar, gramatica obţinută este fără recursie stângă.

1.8.6 Funcţii de numărare şi formula multinomului. In cele ce urmează se va lucra cu mulţimi finite. Numărul de elemente din mulţimea finită X va fi notat cu |X|. Unei funcţii / de la X in Y, unde X si Y sunt mulţimi finite, astfel încât X = $[x\backslash, \ldots, x_n\}$ si Y = $[y_i, \ldots, y_m\}$, i se pune in corespondenta o aranjare a mulţimii X de obiecte in mulţimea Y de căsuţe, astfel încât in căsuţa să intre obiectele din mulţimea [x|x $E X$, /(x) = y î}. Aceasta corespondentă este o bijecţie. De asemenea, se poate stabili o bijecţie intre mulţimea funcţiilor / : X — Y si mulţimea n-uplelor formate cu elemente din Y sau a cuvintelor de lungime n formate cu litere din mulţimea Y astfel: unei funcţii / i se pune in corespondenta cuvântul /(x$_i$)/(x$_2$)... /(x$_n$) in care ordinea literelor este esenţială.

PROPOZIŢIA 1. Numărul funcţiilor / : X — Y este egal cu m^n, daca |X| = n si |Y| = m.

Prin definiţie se notează $[x]_n$ = x(x — 1)... (x — n + 1) si $[x]^n$ = x(x + 1)... (x + n — 1), ambele produse conţinând câte n factori consecutivi.

PROPOZIŢIA 2. Numărul funcţiilor injective / : X — Y, unde |X| = n si | Y| = m este egal cu [m]n.

Aranjamente de m luate cate n se numesc cuvintele de lungime n formate cu litere diferite din mulţimea Y, iar numărul lor este [m]n. Dacă m = n, aceste cuvinte de lungime n formate cu toate literele din Y se numesc *permutări* ale mulţimii Y si numărul lor este egal cu $[n]_n$, care se notează cu n! = 1 • 2 • • • n si se numeşte n *factorial.* Prin definiţie 0! = 1.

Daca mulţimea Y este o mulţime ordonată astfel încât $y_1 < y_2 < \ldots < y_n$, un cuvânt de lungime n format cu litere din Y, $y_{il} \ldots y_{in}$ se numeşte crescător dacă $y_{il} < y_{i2} < \ldots < y_{in}$ si strict crescător daca $y_{il} < y_{i2} \ldots < y_{in}$. Cuvintele strict crescătoare de lungime n formate cu m simboluri se mai numesc *combinări* de m luate câte n, iar cele crescătoare se numesc *combinări cu repetiţie* de m luate câte n.

PROPOZIȚIA 3. Numărul cuvintelor strict crescătoare de lungime n formate cu m simboluri este egal cu $[m]_n/n!$. Acest număr este egal si cu numărul submulțimilor cu n elemente ale unei mulțimi cu m elemente. Numerele $[m]_n$ se mai notează prin si se numesc *numere binomiale* cu parametrii m si n.

PROPOZIȚIA 4. Numărul cuvintelor crescătoare de lungime n formate cu m simboluri este egal cu $[m]^n/n!$.

Demonstrație. Să observam mai întâi ca $[m]^n/n!$ = (m+n-^. Putem presupune ca Y = {1,. . . ,m} si vom stabili o bijecție de la mulțimea cuvintelor crescătoare de lungime n formate cu litere din Y (cu ordinea naturala) pe mulțimea cuvintelor strict crescătoare de lungime n formate cu litere din alfabetul extins {1,. . . , m + n — 1} in felul următor: $^(y_{il}\ y_{j2}\ \ldots\ y_{in}) = y_{il}\ |y_{i2} + 1|\ldots\ |y_{in} + n — 1$. Se verifică imediat ca ^ este o bijecție, de unde rezultă propoziția, deoarece numărul cuvintelor strict crescătoare de lungime n formate cu litere dintr-un alfabet cu m + n — 1 simboluri este egal cu

Numerele binomiale apar drept coeficienți in formula binomului:

$$(a+b)^n - \sum_{k-0}^{n} \binom{n}{k} a^{n-k}b^k\ , \qquad (1)$$

care este valabilă pentru orice a , b dintr-un inel comutativ. Demonstrația formulei (1) se poate face ținând seama ca $(a + b)^n = (a + b)(a + b)\ldots (a + b)$, unde in produs sunt n factori. Pentru a obține monomul $a^{n-k}b^k$ trebuie sa îl alegem pe b din k paranteze si pe a din parantezele ramase, in număr de n — k. Dând cate un număr de ordine celor n paranteze, trebuie sa selectam k numere de ordine ale parantezelor din care îl alegem pe b din mulțimea de numere {1,. . . , n}, ceea ce se poate face in $(jkj$ moduri, deci coeficientul monomului $a^{n-k}b^k$ este egal cu ^n_k .

PROPOZIȚIA 5. Numărul de aranjări ale unei mulțimi de n obiecte X = . . . ,x_n} intr-o mulțime de *p* căsuțe Y = {y_1,... ,y_p}, astfel încât căsuța yj să conțină n obiecte pentru orice 1 < *i* < p (n_1 + . . . + n_p = n) este egal cu $\frac{n!}{n_1!n_2!\ldots n_p!}$.

Demonstrație. Obiectele din căsuța y_1 pot fi alese in(n/n1) moduri, obiectele din căsuța y_2 pot fi alese din cele $n — n_1$ obiecte rămase in ($^{n-n1/n2}$) moduri etc. Rezulta in total un număr de posibilități egal cu

$$\binom{n}{n_1}\binom{n-n_1}{n_2}\binom{n-n_1-n_2}{n_3}\ldots\binom{n_p}{n_p} - \frac{n!}{n_1!n_2!\ldots n_p!}$$

Acest raport de factoriale se mai notează $\binom{n}{n_1,n_2,\ldots,n_p}$ si se numește *număr multinomial,* care generalizează numerele binomiale (cazul p = 2) si apar in *formula multinomului* care extinde relația (1):

$$(a_1 + a_2 + \ldots + a_p)^n - \sum_{\substack{n_1,\ldots,n_p\geq 0\\ n_1+\ldots+n_p-n}} \binom{n}{n_1, n_2, \ldots, n_p} a_1^{n_1}a_2^{n_2}\cdots a_p^{n_p} \quad (2)$$

Demonstrata formulei (2) a multinomului se poate face astfel: Este clar că dezvoltarea $(a \setminus + \ldots + a_p)^n$ va conține o suma de monoame de forma a^n • • • , unde numerele naturale $n \setminus, \ldots n_p$ verifică ni+. . .+n_p = n. Trebuie găsit coeficientul acestui monom, deci de cate ori apare el in puterea a na a expresiei $a_1 + \ldots + a_p$. Pentru aceasta se procedează ca la formula binomului, dându-se numerele de ordine 1 , . . . , n parantezelor din dezvoltarea $(a_1 + \ldots + a_p)^n = (a_1 + \ldots + a_p)\ldots (a_1 + \ldots + a_p)$ si observând că pentru a obține monomul a*,[11] • • • trebuie să-l alegem pe a_1 din n_1 paranteze,... ,pe a_p din n_p paranteze. Ori aceasta este echivalent cu plasarea numerelor de ordine 1 , . . . , n ale parantezelor in p cutii, astfel încât cutia 1 sa conțină n_1 numere,...,cutia p sa conțină n_p numere, ceea ce se poate realiza in ($_{ni}\ ^n\ _n$) moduri, conform propoziției 5. □

PRINCIPIUL INCLUDERII ŞI AL EXCLUDERII ŞI APLICAȚII Principiul includerii si al excluderii constituie o generalizare a identității:

$$|A \cup B| - |A| \ | \ |B| - |A \cap B|, \quad (3)$$

unde A, B *C* X.

TEOREMA 1 (Principiul includerii si al excluderii). Dacă Aj (1 < i < q) sunt submulțimi ale unei mulțimi X, are loc relația:

$$|\bigcup_{i-1}^{q} A_i| - \sum_{i-1}^{q}|A_i| - \sum_{1\leq i<j\leq q}|A_i\cap A_j| \ | \sum_{1\leq i<j<k\leq q}|A_i\cap A_j\cap A_k| - \ldots | (-1)^{q-1}|\bigcap_{i-1}^{q} A_i|$$

(4)

Demonstrare. Vom face demonstrația prin inducție după q. Pentru q = 2 se obține relația (3). Fie q > 3, sa presupunem formula (4) adevărata pentru q — 1 mulțimi si să o demonstrăm pentru q mulțimi. Avem:

$|A_1 \cup \ldots \cup A_q| = |A_1 \cup \ldots \cup A_{q-1}| + |A_q| - |(A_1 \cup \ldots \cup A_{q-1}) \cap A_q|$. Aplicând

distributivitatea se obține $(A_1 \cup \ldots \cup A_{q-1}) \cap A_q = \bigcup_{i<q}(A_i \cap A_q)$, deci aplicând ipoteza de inducție se deduce $|\bigcup_{i=1}^{q} A_i| - |\bigcup_{i=1}^{q-1} A_i| + |A_q| - |\bigcup_{i<q}(A_i \cap A_q)| =$

$\sum_{i<q} |A_i| - \sum_{i<j<q} |A_i \cap A_j| + \sum_{i<j<k<q} |A_i \cap A_j \cap A_k \cap A_q| - \ldots + |A_q| -$

$\sum_{i<q} |A_i \cap A_q| + \sum_{i<j<q} |A_i \cap A_j \cap A_q| - \ldots$ de unde, regrupând termenii se obține (4). □ Aplicații

1) Determinarea funcției lui Euler </?(n).

Fiind dat un număr natural n, <^(n) reprezintă numărul numerelor naturale mai mici ca n si relativ prime cu n. Dacă descompunerea lui n in factori primi distincți este $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_q^{\alpha_q}$ si se notează cu A_i mulțimea numerelor naturale mai mici sau egale cu n care sunt multipli de p_i, se obține $|A_i| = n/p_i$, $|A_i \cap A_j| = n/(p_i p_j)$ etc. Deci

$\varphi(n) = n - |A_1 \cup A_2 \cup \ldots \cup A_q| =$

$n - \sum_{i=1}^{q} |A_i| + \sum_{1 \le i < j \le q} |A_i \cap A_j| - \ldots - n - \sum_{i=1}^{q} n/p_i + \sum_{1 \le i < j \le q} n/(p_i p_j) -$

$\ldots + (-1)^q n/(p_1 \cdots p_q)$, adică:

$$\varphi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_q}). \quad (5)$$

2) Determinarea numărului D(n) al permutărilor a n elemente fără puncte fixe.

Fie p o permutare a mulțimii X = {1,..., n}, adică o bijecție p : X — X. Permutarea p are un punct fix in i dacă p(i) = i. Daca notam cu A_i mulțimea permutărilor care au un punct fix in i, rezulta ca D(n) = n! — $|A_1 \cup \ldots \cup A_n|$ = n! — $\sum_{i=1}^{n} |A_i| + \sum_{1 \le i < j \le n} |A_i \cap A_j| + \ldots + (-1)^n |\bigcap_{i=1}^{n} A_i|$. Se obține $|A_{i_1} \cap A_{i_2} \cap \ldots A_{i_k}| = (n - k)!$, deoarece o permutare din mulțimea $A_{i_1} \cap \ldots A_{i_k}$ are puncte fixe in pozițiile $\ldots, i_k$, celelalte imagini putând fi alese in (n — k)! moduri. Deoarece k poziții $i_1, \ldots, i_k$ pot fi alese din mulțimea celor n poziții in moduri, rezulta:

$$D(n) \quad n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \ldots + (-1)^n \binom{n}{n} \text{ sau}$$

$$D(n) = n!(1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^k}{k!} + \ldots + \frac{(-1)^n}{n!}). \quad (6)$$

3) Determinarea numărului $x_{n>TO}$ al funcțiilor surjective.

Fie mulțimile X = ..., x_n} si Y = { $y_1, \ldots, y_m$}. Pentru fiecare i, 1 < i < m, să notăm prin A_i mulțimea funcțiilor de la X in Y pentru care y_i nu este imaginea nici unui element din X, adică A_i = {/ : X — Y| y_i </ / (X)}. Se obține $x_{n,m} = m^n - |A_1 \cup \ldots \cup A_m| = m^n - \sum |A_j| + \sum_{l<j<m} |A_j \cap A_j| - \ldots + (-1)^m |\bigcap_{i=1}^{m} A_i|$. A_i este mulțimea funcțiilor definite pe X cu valori in Y\{y_i}, deci $|A_i| = (m - 1)^n$ si in general $|A_{i_1} \cap \ldots \cap A_{i_k}| = (m - 1)^n$. Deoarece fiecare sumă $\sum_{l<i_1<\ldots<i_l<m} |A_{i_1} \cap \ldots \cap A_{i_l}|$ conține ("") termeni egali cu $(m - 1)^n$, deducem:

$$s_{n,m} = m^n - \binom{m}{1}(m-1)^n + \binom{m}{2}(m-2)^n - \ldots + (-1)^{m-1}\binom{m}{m-1} \quad (7)$$

Daca m = n se obține s_{nn} = n!, iar daca m > n nu există surjectii de la X pe Y, deci $x_{n>m} = 0$ (identitățile lui Euler).

NUMERELE LUI STIRLING, BELL, FIBONACCI ŞI CATALAN Alături de numerele binomiale si multinomiale, numerele lui Stirling, Bell si Fibonacci joacă un rol deosebit in probleme de numărare.

Pentru a defini numerele lui Stirling de prima speța, pe care le notam s(n, k), vom dezvolta polinomul $[x]_n$ in ordinea crescătoare a puterilor lui x. Coeficienții acestei dezvoltări sunt prin definiție numerele lui Stirling de prima spetă, adică

$$[x]_n = \sum_{k=0}^{n} s(n,k) x^k \quad (8)$$

Numerele s(n, k) se pot calcula prin recurenta, utilizând relațiile s(n, 0) = 0, s(n, n) = 1 si s(n + 1, k) = s(n, k — 1) — ns(n, k), ultima relație obținându-se prin egalarea coeficienților lui x^k in cei doi membri ai egalității $[x]_{n+1} = [x]_n(x - n)$. Se obține tabelul următor, unde s(n, k) = 0 pentru n < k.

s(n, k)	k = 0	1	2	3	4	5 ...
n = 1	0	1	0	0	0	0

2	0	-1	1	0	0	0
3	0	2	-3	1	0	0
4	0	-6	11	-6	1	0
5	0	24	-50	35	-10	1

Numărul lui Stirling de speța a doua, notat $S(n,$ m), este numărul partițiilor unei mulțimi cu n elemente in m clase. Să observăm că atât ordinea claselor, cat si ordinea elementelor intr-o clasă a unei partiții sunt indiferente. De exemplu, dacă avem X = $[a, b, c,$ d}, partițiile cu trei clase ale acestei mulțimi sunt: {(a), (b), (c, d)}, {(a), (c), (b, d)}, {(a), (d), (b,c)}, {(b), (c), (a, d)}, {(b), (d), (a, c)}, {(c), (d), (a, b)}, deci S(4, 3) = 6.

Numerele lui Stirling de speța a doua pot fi calculate prin recurentă astfel: Considerând mulțimea celor S(n, k — 1) partiții ale unei mulțimi cu n elemente in k — 1 clase, putem obține S(n, k — 1) partiții a n + 1 elemente in k clase, adăugând la fiecare partiție o noua clasa formata dintr-un singur element si anume al (n + 1)-lea. Să considerăm acum o partiție a n elemente in k clase. Deoarece putem adăuga al (n + 1)-lea element la clasele deja existente in k moduri diferite si toate partițiile a n + 1 elemente cu k clase se obțin fără repetiții printr-unul din cele două procedee expuse, rezultă că

$$S(n + 1, k) = S(n, k - 1) + kS(n, k)$$

pentru 1 < k < n si S(n, 1) = S(n, n) = 1. Aceste relații permit calculul prin recurenta al numerelor S(n, m), obținându-se tabelul următor:

S(n, m)	m = 1	2	3	4	5 ...
n = 1	1	0	0	0	0
2	1	1	0	0	0
3	1	3	1	0	0
4	1	7	6	1	0
5	1	15	25	10	1

Pentru calculul direct al numerelor lui Stirling de speța a doua, vom arăta ca S(n,m) = $x_{n>m}$/m!. Intr-adevăr, oricărei surjectii / a mulțimii X = . . . , x_n} pe mulțimea Y = { y_1 , . . . , y_n} ii corespunde o partiție a mulțimii X cu m clase si anume $/^{-1}(y_1)$, $/^{-1}(y_2)$, . . . , $/^{-1}(y_m)$. Deoarece intr-o partiție nu contează ordinea claselor, rezultă că n! funcții surjective de la X pe Y vor genera o aceeași partiție a mulțimii X. Ținând seama de (7) rezultă:

$$S(n,m) = \frac{1}{m!}s_{n,m} = \frac{1}{m!}\sum_{k=0}^{m-1}(-1)^k\binom{m}{k}(m-k)^n \quad (9)$$

2. Analiza sintactică ascendentă

2.1 Analiza sintactice "deplasează-reduce". Fie gramatica:

$$S \to bAcBe$$
$$A \to Aa \mid a$$
$$B \to d$$

şi cuvântul *baacde* din limbajul generat de gramatică. Vom încerca să refacem drumul parcurs de un analizor cu strategie ascendentă care pleacă de la şirul *baacde* şi găseşte în final neterminalul S. Pe parcurs vom evidenția problemele mai deosebite ale acestui tip de analiză.

Prototipul analizei sintactice ascendente este automatul "push-down" extins. După cum se ştie, el poate simula în funcționarea sa derivarea dreaptă într-o gramatică independentă de context, derivare parcursă însă în ordine inversă. Automatul "push-down" extins identifică pentru acesta în vârful stivei părți drepte ale producțiilor pe care le "reduce" la neterminalul părții stângi corespunzătoare. În vederea completării părții drepte, automatul îşi aduce pe rând în stivă simbolurile de la intrare. Şirul este acceptat dacă au rămas doar simbolul de început.

În cazul exemplului considerat, un analizor sintactic ascendent va căuta părți dreapta ale producțiilor în şirul de intrare. Le găseşte sub forma: a şi d. Desigur, analizând în mod natural, de la stânga la dreapta, după depăşirea lui b, va prefera înlocuirea primului a cu A, rezultând şirul bAacde. În termenii arborilor de derivare subşirul analizat se prezintă ca în figura .2.1.

După cum vom vedea în continuare, un analizor ascendent creează o "pădure" de arbori pe care încearcă să-i reunească în final într-un unic arbore. În cazul nostru, el a creat un arbore format dintr-un nod etichetat b şi un arbore ca în figura 2.1.

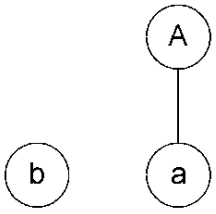

Figura 2.1

Figura 2.2.

Încercând în continuare să găsească părți dreapta (de această dată în bAacde) analizorul va găsi, în ordine : Aa, a şi d. Ordinea de analiză de la stânga la dreapta a intrării impune de data aceasta două posibilități: Aa şi a. Ambele se reduc la A, dar nu sunt totuşi la fel de bune. Într-adevăr, reducerea lui a la A conduce la şirul bAAcde care apoi prin reducerea lui d la B conduce la bAAcBe în care analizorul nu mai poate recunoaşte nici o parte dreaptă şi analiza pe această cale eşuată, fiind necesară revenirea la decizia reducerii lui a. Dacă se alege reducerea lui Aa la A se obține şirul bAcde şi corespunzător "pădurea" din figura 2.3
Concluzia este, desigur, acceptarea şirului.

S ă observăm că analiza a refăcut în ordine inversă derivarea dreaptă:

După adăugarea lui e şi d la pădure, singurul şir de redus este d. După reducerea lui obținem bAcBe şi pădurea din figura 2.4.

În fine, adăugând şi ultimul simbol la pădurea de arbori, se observă că noul şir, ca şi rădăcinile arborilor pădurii, alcătuiesc partea dreaptă a primei producții, deci putem reduce la S, adică ajungem la arborele de derivare din figura 2.5.

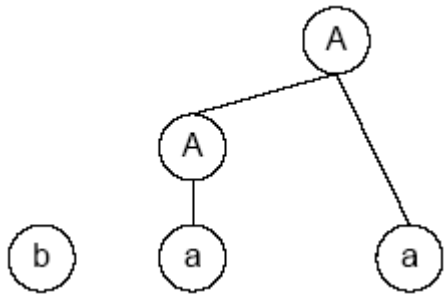

Figura 2.3

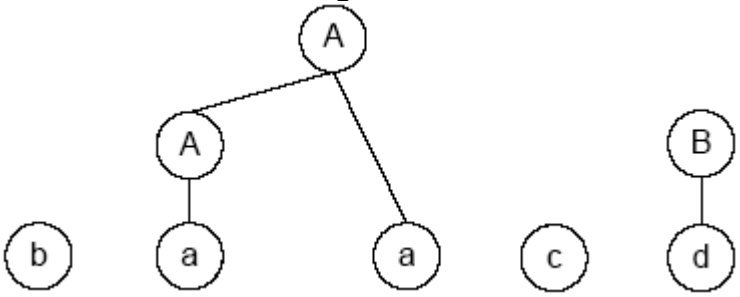

Figura 2.4

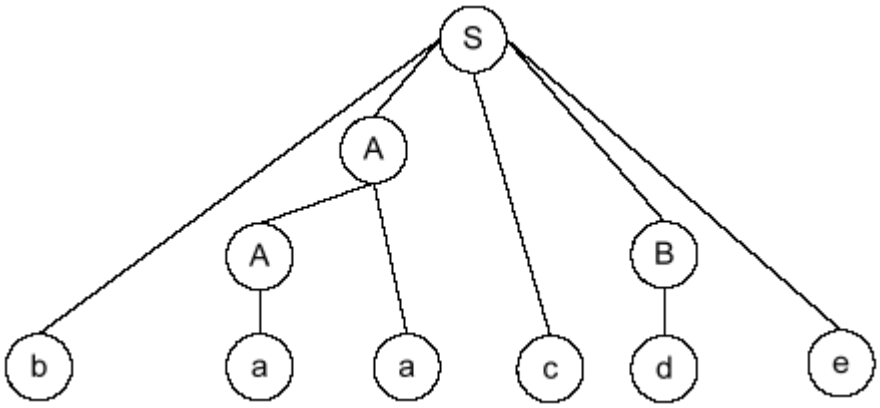

Figura 2.5

$$S \underset{d}{\Rightarrow} bAcBe \underset{d}{\Rightarrow} bAcde \underset{d}{\Rightarrow} bAacde \underset{d}{\Rightarrow} baacde$$

prin reduceri: $baacde \dashrightarrow bAacde \dashrightarrow bAcde \dashrightarrow bAcBe \dashrightarrow S$

În aceste transformări, subşirul redus joacă un rol esențial. El trebuie mai întâi identificat şi apoi înlocuit cu un neterminal. Acest subşir este întotdeauna cel mai din stânga subşir dintr-o formă propozițională dreaptă care poate fi redus astfel încât să se refacă în sens invers un pas într-o derivare dreaptă. El poartă numele de capăt (în engl.: handle).

Definiția 2.5. Capătul unei forme propoziționale dreapta $\beta\alpha\omega$ este şirul *a* dacă în gramatica respectivă există derivarea:

$$S \overset{*}{\underset{d}{\Rightarrow}} \beta A \omega \underset{d}{\Rightarrow} \beta\alpha\omega$$

În cazurile concrete capătul este dat printr-o producție A — a şi poziția lui a în */3aa>* (adică limitele lui din stânga şi din dreapta).

În arborele de derivare asociat unei forme propoziționale capătul apare ca cel mai din stânga subarbore complet (format deci dintr-un nod-rădăcină şi toți descendenții săi).

Utilizând noțiunea de capăt, analiza ascendentă apare ca un proces de formare şi identificare a capetelor formelor propoziționale dreapta obținute şi de reducere a lor la neterminale.

Exemplul 2.5. Fie gramatica:

(B_L) 1. $L \rightarrow L, E$
2. $L \rightarrow E$
3. $E \rightarrow id$
4. $E \rightarrow id(L)$

şi şirul de intrare **id**1,**id**2 **(id**3,**id**4) în care am evidențiat diferențele aparției ale terminalului **id.**

Reducerea şirului la neterminalul L are loc în următoarele etape:

Formă propozițională dreapta	Capăt	Producție folosită în reducere
id1, *id* 2 (id 3, *id* 4)	*id1*	$E \rightarrow$ id
E, *id* 2 (id 3, *id* 4)	*E*	$L \rightarrow E$ $E \rightarrow$ id
L, id2 (id3 , id4)	*id3*	*L - E*
L, *id* 2 (E, *id* 4) L, *id* 2 (L, *id* 4) L, *id* 2	*E id*4 *L, E*	$E \rightarrow$ id $L \rightarrow L, E$ $E \rightarrow$ id(L) $L \rightarrow L, E$
(L, *E*) *L, id2 (L)* id2(L) *L, E*		
L, E		
L		

Se observă că secvența reducerilor este inversă secvenței derivărilor dreapta.

Din prezentarea de mai sus remarcăm faptul că un analizor sintactic ascendent este confruntat cu două probleme : identificarea capătului în forma propozițională dreapta şi alegerea producției cu care să facă reducerea.

În ceea ce priveşte identificarea capătului, se poate arăta că într-o parcurgere de la stânga la dreapta a şirului de intrare însoţită de reduceri succesive ale capătului fiecărui forme propoziţionale dreapta, la un moment dat, analizorul sintactic nu trebuie să se întoarcă oricând în pădurea de arbori deja formată pentru a căuta capătul, ci acesta apare în mod natural în vecinătatea imediată a punctului de analiză.

Mai precis, să considerăm cele două posibilităţi de apariţie a capătului. Fie mai întâi derivarea:

$$S \underset{d}{\overset{*}{\Rightarrow}} \alpha Au' \underset{d}{\Rightarrow} \alpha\beta Byu \underset{d}{\Rightarrow} \alpha\beta\gamma yu$$

în care s-au folosit producţiile $A \to \beta By$ şi $B \to \gamma$. Se observă că şirul βBy s-a evidenţiat neterminalul cel mai din dreapta.

Pornind în sens invers, de la şirul de terminale, analizorul sintactic-ascendent a prelucrat parţial şirul de intrare aducându-l la forma : $\alpha\beta\lambda$, şi rămânându-i de parcurs şirul yu. Este momentul în care el trebuie să depisteze capătul *y* şi să-l reducă la B, şirul prelucrat devenind $\alpha.\beta B$, iar cel neanalizat rămânând *yu.* Se vede că în şirul $\alpha\beta$, deci la stânga lui B, capătul nu poate apare, B fiind cel mai din dreapta neterminal. Singura posibilitate este ca noul capăt să se completeze prin adunarea de terminale din *yu.* Această operaţie de adăugare de către analizor din şirul de intrare a noi terminale la porţiunea deja prelucrată a formei propoziţionale se numeşte deplasare. Adăugând y obţinem By în care moment se identifică un capăt By, ce se reduce la A. (z poate fi chiar şirul vid; atunci fără a se face vreo deplasare, după reducerea lui *y* la B se face reducerea lui B la A).

O altă posibilitate de derivare dreaptă o constituie secvenţa:

$$S \underset{d}{\overset{*}{\Rightarrow}} \alpha BxAu \underset{d}{\Rightarrow} \alpha Bxyu \underset{d}{\Rightarrow} \alpha\gamma xyu$$

în care s-au folosit producţiile A — y şi B — y.

Lucrând în sens invers, analizorul construieşte şirul *ay* în care momentul găseşte că *y* este capăt şi îl reduce la B. Din nou, capătul nu are ce căuta în stânga lui B, deci se vor adăuga de la intrare noi simboluri până la formarea şirului *aBxy* în care momentul se delimitează capătul *y* şi se reduce la *A*.

Din observarea celor două cazuri posibile de localizare a capătului, constatăm că orice capăt este format din cele mai recente simboluri prelucrate, reduse sau deplasate.

Deci şirul deja analizat alcătuieşte o listă LIFO a cărei implementare se poate face o memorie stivă.

În afara stivei, algoritmul de analiză va avea acces la şirul de intrare pentru a-şi deplasa terminale în stivă. De asemenea, va trebui să depună la ieşire informaţii privind analiza efectuată. Algoritmul, pe baza informaţiei de la intrare şi a conţinutului stivei, va decide asupra următoarei acţiuni ce poate fi: reducerea, deplasare, eroare, acceptare.

Un asemenea algoritm de analiză poartă numele ,,deplasează-reduce"

Maşina „deplasează-reduce" . Analizorul ,,deplaseaza-reduce" are, după cum uşor se poate observa, ca model matematic, automatul "push-down" extins. Ca şi analiza descendentă vom încerca să definim un model fizic al analizei ascendente ce lucrează prin deplasări şi reduceri : maşina ,,deplaseaza-reduce", pe scurt MDR.

Din punct de vedere al structurii, MDR este similară maşinii de analiză predictivă, MAP, prezentată în 2.1.2. Operaţiile primitive sunt aceleaşi cu cele ale MAP cu excepţia operaţiei de comparaţie a subşirurilor din stivă, necesară în identificarea capetelor din vârful stivei. În termenii acestor operaţii, efectele acţiunilor MDR sunt următoarele:

1)reducere cu *(β, A,* i):

- se şterg din vârful stivei |β| simboluri,
- se depune în vârful stivei simbolul $A \in T$,
- se depune pe banda de ieşire numărul *i.*

2)Deplasare:

- se depune simbolul curent de la intrare în vârful stivei,
- se avansează banda de intrare.

3)Eroare:

- se semnalizează eroarea prin modificarea stării procesorului.

4) Acceptare (are loc la întâlnirea marcajului $ pe banda de intrare şi a şirului *$Z0* în stivă, $Z_0 \in T$):

- se semnalizează acceptarea prin modificarea stării procesorului.

Starea maşinii este sintetizată către configuraţia sa *(aX,x, π),* unde *$aX*este conţinutul stivei cu vârful *X* reprezentat spre dreapta, *x$* este şirul de intrare cu marcajul de sfârşit al şirului *$* şi *n* este şirul numerelor de producţii cu care s-au făcut reducerile.

Configuraţia iniţială este *($,x$, Ă),*unde x este şirul de analizat, iar configuraţiile finale sunt de forma *($Z0,$, πi).*

Reducerea cu *(β,A,i)* determină o mişcare a maşinii de forma :

$$(\$\alpha\beta, x\$, \pi) \vdash (\$\alpha A, x\$, \pi i)$$

Deplasarea determina mişcarea:

$$(\$\alpha, ax\$, \pi) \vdash (\$\alpha a, x\$, \pi)$$

Eroarea întrerupe şirul mişcărilor ceea ce notăm astfel:

$(\$\alpha, x\$) \vdash ?$ eroare.

Acceptarea de asemenea întrerupe evoluţia maşinii:

$(\$S,\$, \pi) \vdash ?$ eroare.

Relaţia de mişcare $\vdash$ se poate extinde la închiderea ei tranzitivă şi reflexivă $\overset{*}{\vdash}$. MDR poate fi privită şi ea ca un translator ce defineşte o traducere astfel:

$$\tau_{MDR} = \left\{ (x, \pi) \mid (\$, x\$, \lambda) \overset{*}{\vdash} (\$S,\$, \pi) \right\}$$

Definiţia 2.6. O MDR este valabilă pentru o gramatică G dacă:

a) $L(G) = \{x \in \Sigma^* \mid (\$, x\$, \lambda) \overset{*}{\vdash} (\$S,\$, \pi)\}$

b) pentru orice $x \in L(G)$, dacă $S \Rightarrow^{\pi} x$, atunci $(x, \pi^R) \in \tau_{MRD}$.

Exemplul 2.6. Fie gramatica GL de mai sus şi şirul de intrare id, id. Considerăm $\Gamma = N \cup \Sigma$ şi $Z_0 = L$. Atunci există următoarea secvenţă de mişcări ale MDR (pentru a nu se confunda cu virgula folosită în descrierea configuraţiei, terminalul ',' a fost „prins" între apostrofuri).

$(\$, id', id\$, \lambda) \vdash (\$id, ', 'id\$, \lambda) \vdash (\$E, ', 'id\$, 3) \vdash$

$(\$L, ', 'id\$, 32) \vdash (\$L', ', id\$, 32) \vdash (\$L, ', 'id, \$, 32) \vdash$

$(\$L', 'E, \$, 323) \vdash (\$L, \$, 3231) \vdash ? acceptare.$

Principala problemă în proiectarea uni MDR valabile pentru o anumită gramatică o reprezintă algoritmul de conducere al maşinii. El trebuie să decidă când maşina face o reducere, o deplasare, când semnalizează eroare sau acceptare, care este capătul, cu ce producţie se face reducerea. Pentru toate aceste decizii este de dorit ca algoritmului să-i fie suficient un context al analizei cât mai limitat, alcătuit, de exemplu, din simbolul curent de la intrare (în cazuri mai complicate şi din alte câteva care îi urmează) precum şi simbolul din vârful stivei (eventual şi alte câteva de sub acesta). La baza unui asemenea algoritm vor exista tabele care, pentru toate combinaţiile de simboluri ce pot apărea în vârful stivei şi la intrare, să indice o unică acţiune.

Modul concret în care se construiesc asemenea tabele reprezintă subiectul următoarelor paragrafe. Astfel, vom prezenta proiectarea analizoarelor sintactice bazate pe relaţii de precedenţă între simbolurile gramaticii, precum şi cele bazate pe gramatici LR(1), cea mai largă clasă de gramatici ce permit analiza deterministă, ascendentă a limbajelor generate de ele. În toate cazurile vom urmări modul în care se stabilesc regulile referitoare la:

- delimitarea producţiei capătului prin precizarea limitei dreapta şi apoi a celei stânga ale acestuia,
- stabilirea producţiei cu care se face reducerea.

2.2. Gramatici de precedenţă. Ideea analizei bazată pe relaţii de precedenţă constă în posibilitatea depistării limitei dreapta şi a celei stânga ale capătului folosind relaţii între

simbolurile ce alcătuiesc forma propoziţională. Relaţiile, trei la număr, sunt notate de obicei cu <,=,>. În cazul precedenţei simple, într-o derivare de forma:

$$S \overset{*}{\underset{d}{\Rightarrow}} \alpha X_{K+1} A a x \overset{}{\underset{d}{\Rightarrow}} \alpha X_{K+1} X_K \ldots X_2 X_1 a x$$

capătul $XK+1XK$ ...X 2 X1 este delimitat prin cele trei relaţii astfel :

- la dreapta: X1>a,
- la stânga: XK+1<Xk,
- în interiorul capătului: Xi+1=Xi, pentru i=1,.....,k-1. De asemenea, între simbolurile consecutive din $aXK+1$ avem relaţii < sau =.

Presupunând că între două simboluri oarecare din gramatică există cel mult o relaţie, depistarea capătului de către MDR este unic determinată. Problema care rămâne este cea a alegerii producţiei cu care se face reducerea . Pentru a înlătura această nedeterminare vom presupune că gramatica folosită este unic invertibilă, adică în cadrul producţiilor ei nu există două părţi drepte identice. Cele de mai sus sunt formal introduse prin următoarele definiţii:

Definiţia 2.7. Se numesc relaţii de precedenţă Wirth - Weber relaţiile :

$$R_{<\cdot} \subset (N \cup \Sigma \cup \{\$\})^2, R_{\cdot} \subset (N \cup \Sigma \cup \{\$\})^2, R_{\cdot>} \subset (N \cup \Sigma \cup \{\$\}) \times (\Sigma \cup \{\$\})$$

definite astfel pentru $X, Y \in N \cup \Sigma$ şi $a \in \Sigma$:

a) $X < Y$ dacă există $A \to \alpha X B \lambda \in P$ a.î. $B \Rightarrow Y \gamma$,

b) $X = Y$ dacă există $A \to \alpha X Y \beta \in P$,

c) $X > Y$ dacă există $A \to \alpha B Y \beta \in P$ a.î. $B \Rightarrow \gamma X$ şi $Y \overset{*}{\Rightarrow} a\delta$. La acestea se adaugă relaţiile ce implică marcajul '$':

d) $\$ < X$ dacă există o derivare $S \overset{+}{\Rightarrow} X\alpha$

e) $X > \$$ dacă există o derivare $S \overset{+}{\Rightarrow} \alpha X$.

În ciuda asemănării lor ca notaţie cu relaţiile <,=,> din algebră, relaţiile Wirth -Weber nu au proprietăţile acestora. Reprezentarea cea mai folosită pentru relaţiile Wirth -Weber este cea tabelară

Definiţia 2.8. O gramatică independentă de context G=<N, E ,P,S> este proprie dacă nu are simboluri inutile, dacă nu există derivări de forma $A \overset{+}{\Rightarrow} A$ pentru $A \in N$ şi dacă nu are X - producţii în afară, eventual, a producţiei $S \to X$ în care caz S nu apare în nici o parte dreaptă a vreunei producţii. O gramatică independentă de context, proprie, fără X -producţii, în care între orice două simboluri există cel mult o relaţie de precedenţă Wirth-Weber, se numeşte gramatică de precedenţă simplă.

Exemplul 2.7. Fie gramatica :

$S \to bAAc$

$A \to Sa \mid a$

Mulţimea perechilor $R_{\cdot}$ este uşor de construit : se caută în părţile dreapta ale producţiilor toate perechile de simboluri consecutive:

$$R_{\cdot} = \{(b, A), (A, A), (A, c), (S, a)\}.$$

Pentru calculul mulţimii R se porneşte cu ea vidă şi cu o mulţime M iniţializată < cu R. Se caută în M perechi de forma (X,B); în exemplul nostru ele sunt: (b,A) şi (A,A).Pentru fiecare B din aceste perechi căutam în părţile dreapta ale B- producţiilor primului simbol. Fie el Y. Se adaugă (X,Y) la mulţimea R şi dacă $Y \in N$ se adaugă şi < la M. În exemplu, din (b,A) obţinem : (b,a) şi (b,S). Ambele se adaugă la R, dar (b,*S)< se adaugă şi la M. Calculul lui R se încheie când nu mai există în M nici o pereche <

(X,B) netratată. Se adaugă, în fine, perechile ($,Y), unde . În cazul nostru , ($,b). În final, relaţia R devine:

$$R_{<\cdot} = \{(b, a), (b, S), (b, b), (A, a), (A, S), (A, b), (\$, b)\}.$$

Pentru calculul perechilor din mulțimea R o vom considera inițial vidă. Fie o mulțime M inițializată cu R. În M vom căuta perechile de forma (B,Y). În exemplul nostru ele sunt: (A,A),(A,a),(S,a). Se calculează PRIM(Y). Apoi, pentru fiecare B-producție, se ia ultimul simbol al părții dreaptă, fie el X. Se adaugă la toate perechile (X,a) cu a e PRIM(Y), iar la mulțimea M perechile (X,a) cu X e N. În exemplul nostru, pentru (A,A) avem : PRIM(A)={a,b}, iar ultimul simbol al părții dreapta este a. Deci perechile adăugate la R sunt (a,a) şi (a,b). Mulțimea R rezultată este:

$$R \cdot = \{(a,a),(a,b),(a,c),(c,a),(c,\$)\}$$

Tabelul relațiilor de precedență este cel din figura III.2.16

	S	*A*	*a*	*b*	*c*	$
S			—			
A	<	=	<	<	=	
a			>	>	>	
b	<	—	<	<		
c			>			>
$				<		

Deoarece în fiecare intrare a tabelului există cel mult o relație de precedență, gramatica este de precedentă simplă.

La baza analizei limbajelor generate de gramatici de precedentă stă urătoarea teoremă:

Teorema 2.5. Fie G=<N, S ,P,S> o gramatică proprie fără λ - producții şi fie o derivare dreapta în această gramatică a şirului S:

$$\$S\$: \underset{d}{\overset{*}{\Rightarrow}} X_p X_{p-1} \ldots X_{k+1} A a_1 \ldots a_r \underset{d}{\Rightarrow} X_p X_{p-1} \ldots X_{k+1} X_k \ldots X_1 a_1 \ldots a_r$$

Atunci:

a) pentru $k < i < p$, $Xi+1 = Xi$;

b) $Xk \quad +1 < Xk$;

a) pentru $1 < i < k$ -1, $Xt+1 = Xt$;

c) X1 > a1.

Demonstrație: Folosind metoda inducției pentru numărul de paşi în derivare. Dacă derivarea este într-un pas: $\$S\$: \underset{d}{\overset{0}{\Rightarrow}} \$S\$ \underset{d}{\Rightarrow} \$X_k \ldots X_1 \$$, unde k>1, atunci din definiția 2.7. avem: $\$ < Xk$, $Xi = Xi+1$ pentru $1 < i < k$ -1 şi $Xi > \$$. Deci cerințele teoremei sunt îndeplinite.

Să considerăm afirmațiile teoremei adevărate pentru n paşi în derivare, $n>0$, şi fie derivarea dreapta în n+1 paşi:

$$\$S\$: \underset{d}{\overset{n+1}{\Rightarrow}} X_p X_{p-1} \ldots X_{k+1} A a_1 \ldots a_r \underset{d}{\Rightarrow} X_p X_{p-1} \ldots X_{k+1} X_k \ldots X_1 a_1 \ldots a_r$$

$$\underset{d}{\Rightarrow} X_p X_{p-1} \ldots X_{j+1} Y_q \ldots Y_1 X_{p-1} \ldots X_1 a_1 \ldots a_r$$

Se poate arăta uşor că: $\left[\left(X \overset{\cdot}{=} \right) \wedge (A \to Y\alpha \in P)\right] \Rightarrow (X < \cdot Y)$ Atunci avem : $Xj+1 < Yq$ *(conditia* b).

În care, în ultimul pas, Xj ($1 < j < p$) a fost înlocuit cu Xj-1X1 sunt terminale (dacă j=1, avem Xj-1,...., X1 = Ă).

Din ipoteza inductivă avem: Xj-1 < Xj sau $X j-1 = Xj$, precum şi:

Xi+1 < Xt sau $Xi+1 = Xi$ pentru $p < i < j$ (condiția a).

Deoarece : $\left[\left(A \overset{\cdot}{=} a\right) \wedge (A \to \alpha Y \in P)\right] \Rightarrow (Y \cdot > a)$ avem:

Y1 > Xj-1 sau , pentru j=1, Y1 > a (condiția c).

Corolarul 2.1. Dacă G este gramatică de precedență atunci concluziile a)-d) ale teoremei 2.5. devine:

a) pentru $k < i < p$, fie Xi+1 < Xi, fie Xi+1 = Xi;

b) $Xk+1 < Xk$;

c) pentru $1 < i < k$ -1, Xi+1 = *Xt*;

d) X1 > a1 ;

e) între orice două simboluri din şirul X_p ...X1a1 nu mai există nici o altă relație în afara celor precizate în a)..d).

Demonstrație. Evidență, datorită definiției 2.8.

2.3 Tabele de analiza LR pentru gramatici ambigue.

Un caz interesant pentru analiza LR îl prezintă gramaticile ambigue. Se ştie că într-o asemenea gramatică avea forme propoziționale pentru care există cel puțin doi arbori de derivare distincți.

De multe ori se referă pentru definirea limbajelor de programare asemenea gramatici datorită faptului că ele sunt mai compacte şi uneori chiar mai sugestive decât gramaticile neambigue echivalente.

Se poate demonstra că o gramatică ambiguă nu este LR(1), construirea tabelelor conducând la intrări cu mai multe valori în TA, deci la conflicte. Există totuşi posibilitatea prelucrării acestor tabele prin eliminarea conflictelor într-un mod convenabil proiectantului.

Unele ambiguități apar din nespecificarea prin gramatică a regulilor de precedență şi asociativitate a operatorilor din limbaj. *Exemplul* 2.22. Fie gramatica ambiguă:

$$E \rightarrow E+|E|E*E|(E)|a$$

Ea nu specifică prin producții intermediare o ordine a reducerilor care să stabilească ordinea operațiilor. Pentru următoarele două derivări distincte ale aceleiaşi propoziții:

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * a \Rightarrow E + a * a \Rightarrow a + a * a$$

$$E \Rightarrow E * E \Rightarrow E * a \Rightarrow E + E * a \Rightarrow E + a * a \Rightarrow a + a * a$$

în prima derivare secvența reducerilor reflectă prioritatea obişnuită : * este prioritar față de +, în timp cea de-a doua derivare reduce întâi E + E la E şi apoi pe E *E la E ceea ce inversează prioritatea cunoscută a operatorilor.

Ambiguitatea se reflectă în tabelele LR. Astfel, în tabela de acțiuni SLR(1) pentru gramatica ambiguă de mai sus(fig. III.2.29 a) se constată conflicte deplasare-reducere în TA(s 7,+), TA(s 7 ,*), TA(s 8,+), TA(s 8 ,*).

	a	+	*	(	)	$
	D, s3	*E*	*E*	D, *s2*	*E*	*E*
s 1	*E*	*D, S4*	*D, s5*	*E*	*E*	*A*
S 2	*D, s3*	*E*	*E*	*D, s2*	*E*	*E*
S 3	*E*	*R, 4*	*R, 4*	*E*	*R, 4*	*R, 4*
S 4	*D, s3*	*E*	*E*	*D, s2*	*E*	*E*
S 5	*D, S3*	*E*	*E*		*E*	*E*
S 6	*E*	*D, S4*	*D, s5*	*E*	*D, S9*	*E*
S 7	*E*	*D, S4 R, 1*	*D, s5 R, 1*	*E*	*R 1*	*R, 1*
S 8	*E*	*D, S4 R, 2*	*D, S5 R, 2*	*E*	*R, 2*	*R, 2*
S 9	*E*	*R, 3*	*R, 3*	*E*	*R, 3*	*R, 3*

Se consideră conflictul din TA(s7 ,*) între (D,s5) şi (R,1). Analizorul are de ales între a deplasa „* "(conform lui (D, s5)) şi a reduce cu producția E — E +E şi a deplasa ‚+' vom prefera reducerea ceea ce reflectă opțiunea pentru asociativitatea la stânga, cea mai folosită de obicei. În al doilea caz, din acelaşi motiv aplicat însă operatorului *, vom prefera tot reducerea.

Cu aceste modificări, tabela de acțiuni devine cea din fig.2.8.

	a	+	*	(	)	$
S0	D, S3	*E*	*E*	*D, s2*	*E*	*E*
s1	*E*	*D, S4*	*D, S5*	*E*	*E*	*A*
S2	*D, S3*	*E*	*E*	*D, s2*	*E*	*E*
S3	*E*	*R, 4*	*R, 4*	*E*	*R, 4*	*E, 4*
S4	*D, S3*	*E*	*E*	*D, s2*	*E*	*E*
S5	*D, S3*	*E*	*E*	*D, s2*	*E*	*E*
S6	*E*	*D, S4*	*D, S5*	*E*	*D, S9*	*E*
S7	*E*	*R, 1*	*D, S5*	*E*	*R, 1*	*R, 1*
S8	*E*	*R, 2*	*R, 2*	*E*	*R, 2*	*R, 2*
S9	*E*	*R, 3*	*R, 3*	*E*	*R, 3*	*R, 3*

Observație : Analizorul care folosește această tabelă va recunoaşte acelaşi limbaj cu limbajul recunoscut de analizorul ce folosește tabela din figura III.2.24 (Exemplul 2.10) dar va evita reducerile prin producții singulare şi va beneficia astfel de o tabelă mai mică. În mod asemănător se pot obține pentru gramatici ambigue şi tabelele LR(1) şi LALR(1).

Bibliografie

Popa Ioan "Limbaje formale", vol.1 si 2, Colectia Bursa, Bucuresti, 1997

2. Lascu Ciliana Corina " ***Locul compilatorului în programare*** ", Ed. Teora, 1996

3. Negoescu Gh., Bontas C.A. " ***Structura compilatoarelor*** ", Galati, 1999

5. Andrei Şt., Grigoraş Gh. : Tehnici de compilare. Lucrări de laborator. Editura Universității „Al.I.Cuza”, Iaşi, 1995

6. Appel, A., Modern Compiler Implementation in C, Cambridge University Press, 1997

7. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, U.S.A., 1986

8. Grigoraş, Gh.: Limbaje formale şi tehnici de compilare. Editura Universității

9. „Al.I.Cuza”, Iaşi, 1986

10. Grigoraş, Gh.: Constructia compilatoarelor - Algoritmi fundamentali, Editura Universitatii "Al. I. Cuza" Iasi, ISBN 973-703-084-2, 274 pg., 2005.

11. Jucan, T, Andrei, Şt.: Limbaje formale şi teoria automatelor. Culegere de probleme. Editura Universității „Al.I.Cuza”, Iaşi, 1997

12. Jucan, T., Limbaje formale şi automate, Editura MatrixRom, 1999.

13. Şerbănați, L.D., Limbaje de programare şi compilatoare, Editura Academiei, Bucureşti,1987.

Anexa C. Exemplu. Modelul experimental. Listingul programului

```
#include <conio.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include<string.h>
  int disparte(char S[], char *ss,char c, int a)
            {  int k=0;
  if (S[a]==c)
  {a++;
            for(;a<strlen(S);a++)
  if(S[a]!=' '&&S[a]!=')'&&S[a]!=';'&&S[a]!='\0') {ss[k]=S[a]; k++;}
  else {ss[k]='\0'; break;}
  }
            else
            {for(;a<strlen(S);a++)
  if(S[a]==c) break;
  a++;
            for(;a<strlen(S);a++)
  if(S[a]!=' '&&S[a]!=')'&&S[a]!=';'&&S[a]!=';') {ss[k]=S[a]; k++;}
  else {ss[k]='\0'; break;}
             }
            return a;
   }
void simulare_switch()
 {
  char ch,s1[256],s2[256],s3[300],s4[300],s5[300];
   int i=0,j=0;
   FILE *f;
    clrscr();
    f = fopen("switch.txt", "r");
 window(1,1,80,50);
 textbackground(1);
 textcolor(15);
 clrscr();

int  k=0,l=0;
     i=0;
  puts("\tOperatorul 'Switch' analizat este:\n");
  do
  {
   do
    {
  ch=fgetc(f);
  s1[i]=ch; i++; printf("%c",ch);
            } while(ch!='\n'&&ch!=EOF&&ch!=' ');
            s1[i-1]='\0';
//puts(s1);
            i=0;
            if(strcmp(s1,"switch")==0) {l=1;}
            if(!strcmp(s1,"case")||!strcmp(s1,"default:")) k++;
            if(!strcmp(s1,"break;")) j++;
  } while (ch!=EOF );
  puts("\n\tAnaliza sintactica a operatorului 'Switch'...\n");
  if(l)
```

```
{if (j>k) printf("\nIn exprsia data lipseshte un 'case' sau instructiunea optionala 'default'");
  else if(j<k) printf("\nExpresia switch este incorecta deoarece lipseste intreruperea 'break' a instructiunii");
 else  printf("\nDin punct de vedere sintatic operatorul SWITCH a fost oganizat corect");
 }
 else printf("\nIn instructiunea data lipseshte cuvintul chee 'switch'");
 fclose(f);
getch();
}
void main()
 { char s[100],s1[10],s2[10],s3[10],s4[10];
  char *semn[5]={"!=","<=",">=","<",">"},*oper[6]={"++","--","+=","-=","*=","/="};
  int i,j,k,l,d;
   window(1,1,80,50);
   textbackground(1);
   textcolor(15);
   clrscr();
                              simulare_switch();

 et1:

  window(1,1,80,50);
   textbackground(1);
   textcolor(15);
   clrscr();
 clrscr();
  puts("Introdu expresia for: ");
  //gets(s);
          strcpy(s,"for (i=0;i<=n;i+=2)");
          puts(s);
          for(i=0;i<strlen(s);i++)
 if(s[i]!=' '&&s[i]!='(') {s1[j]=s[i]; j++;}
 else {s1[j]='\0'; break;}
          i=disparte(s,s2,'(',i);
          i=disparte(s,s3,';',i);
          i=disparte(s,s4,';',i);
          int v=0;
          puts("Verificarea sintaxei a operatorului 'for'");
          if(!strchr(s,')) {puts("In exprsia data lipseste '('"); v++;}
           if(!strchr(s,')')) {puts("In exprsia data lipseste ')'"); v++;}
  k=0;
           for(l=0;l<strlen(s);l++)
  if(s[l]==';') k++;
           if(k!=2) {puts("In expresia data lipseste ';'"); v++;}
           if(strlen(s1)==3&&s1[1]!='=') {puts("Inexpresia data lipseste '='"); v++;}
 if (v) {puts("In expresia introdusa sunt erori doritzi sa o reintroducetzi? da-'y',nu-'n'");
           et3: char ch=getch();
           if (ch!='y'&&ch!='n') goto et3;
           if(ch=='y') goto et1;
            else exit(0);
           }
                  else puts("\n\tVerificarea sintaxe a expresiei a trecut cu succes\n");
```

```
            for(i=0;i<5;i++)
 if(strstr(s3,semn[i])) {printf("\nSemnul conditional utilizat este: %s",(semn[i])); break;}
 for(j=0;j<6;j++)
 if(strstr(s4,oper[j])) {printf("\nSemnul operational utilizat este: %s",oper[j]); break;}
 l=s2[2]-48;
 k=0; int ll=1;
          for (i=strlen(s4)-1;i>0;i--)
                  if(s4[i]>47&&s4[i]<59) {k+=(s4[i]-48)*ll; ll*=10;}
                   else break;
           int n;
 printf("Valoarea de modificare a parametrului ciclului este=%d\n\n",k);
 puts("Introdu vloarea lui n");
 scanf("%d",&n);
for(i=0;i<5;i++)
 if(strstr(s3,semn[i])) {puts(semn[i]); break;}
/////////////// Simularea lui for
          int       t;
          t=l; l=0;
                  while (1)
           { switch(i)
          { case 0: if (t==n) goto et; break;
            case 1: if (t>n) goto et; break;
            case 2: if (t<n) goto et; break;
            case 3: if (t==n) goto et; break;
            case 4: if (t==n) goto et; break;
          }
          switch(j)
          { case 0:  t++; break;
            case 1: t--;  break;
            case 2: t+=k; break;
            case 3: t-=k; break;
            case 4: t*=k; break;
            case 5: t/=k; break;
          }
          l++;
           }
          et:  printf("\nnumarul de pashi de executzie este=%d\n",l);
           /////////////// sf Simularii

          getch();

}
```