**Ioniță Vlad Cristian**

**Petrescu George-Alexandru**

**Grupa : 433A**

COMPILATOARE

Sisteme de Operare

Prof. Coordonator

Prof. Dr. Ing. Ștefan Stăncescu

Universitatea “Politehnică” București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

**Realizator : (Petrescu George-Alexandru)**

**1. Concepte generale :**

**1.1. Concepte generale legate de compilatoare…………..……...pag.3**

**1.2 Structura unui compilator. …………..…………..…………..pag4**

**Realizator : (Ioniță Vlad Cristian)**

**2. Descrierea BNF a gramaticii unui limbaj :**

**2.1. Elemente generale. …………..…………..…………..………..pag6**

**2.2. Descrierea BNF. …………..…………..…………..…………..pag7**

**2.3 Exemplu descriere BNF. …………..…………..……………...pag9**

**3. Analiza lexicală si semantică :**

**Realizator : (Ioniță Vlad Cristian)**

**3.1. Analiza lexicală. …………..…………..…………..…………..pag10**

**Realizator : (Petrescu George-Alexandru)**

**3.2. Analiza semantică. …………..…………..…………..………..pag13**

1. **Concepte generale**

**[4]**Limbajele de programare sunt notații pentru descrierea calculului pentru oameni și mașini. Lumea așa cum o știm depinde de limbaje de programare, deoarece toate software-urile care rulează pe computere au fost scrise într-un

limbaj de programare dar, înainte ca un program sa poată rula, trebuie mai intâi să fie tradus într-o formă în care poate fi executat de un calculator.

Sistemele software care fac această traducere sunt numite compilatoare .

**[5]**Pur și simplu, un compilator este un program care poate citi un program într-un singur limbaj -Limbaj sursa - și il traduce într-un program echivalent în

alt limbaj - limbaj destinație.

Un rol important al unui compilator este de a raporta orice eroare in programul sursă pe care o detectează în timpul procesului de traducere .

Un interpretor este un alt fel de procesor de limbaj. În loc de

a produce un program destinație ca o traducere, un interpretor pare să execute direct operațiile specificate în programul sursă pe intrări furnizate de

utilizator.

Programul destinație scris în limbaj masină produs de un compilator este de obicei mult mai rapid decât un interpretor la maparea intrărilor pe ieșiri. Un interpretor, cu toate acestea, poate diagnostica erori mai bine decât un compilator, pentru că poate executa programul comandă cu comanda.

Multe alte programe, în afară de compilator, pot fi necesare pentru a crea un program destinație executabil. Un program sursă poate fi împărțit în module stocate în fișiere separate . Sarcina de colectare a programului sursă revine cateodată unui program separat, numit preprocesor.

Programul sursă modificat este apoi introdus într-un compilator. Compilatorul poate produce un program în limbaj de asamblare la ieșire pentru că acesta este mai ușor de pus la ieșire și de depanat. Limbajul de asamblare este apoi prelucrat de un program numit **assembler** care produce la ieșire un cod mașină relocabil.

Programele mari sunt adesea compilate în bucăți, astfel încât codul mașină relocabil poate fi legat cu alte fișiere obiect si biblioteci într-un cod pe care rulează, de fapt, mașina. **Linker**-ul rezolvă problema adreselor de memorie externe, în cazul în care codul într-un fișier se poate referi la o locație într-un alt fișier. **Loader-**ul pune apoi împreună toate fișierele obiect executabile în memorie pentru executare .

* 1. **.Structura unui compilator**

**[6]** Până la acest moment, am tratat un compilator ca fiind o singură casetă care mapează un program sursă într-un program destinație echivalent din punct de vedere semantic. Daca aruncăm o privire în această casetă, vom vedea că există două părți componente ale acestei mapări : analiză și sinteză.

Partea de analiză sparge programul sursă în bucăți constitutive și le

impune o structură gramaticală. Aceasta utilizează apoi această structură pentru a crea o reprezentare intermediară a programului sursă. Dacă partea de analiză detectează că programul sursă este fie sintactic, fie semantic, incorect, atunci aceasta trebuie să furnizeze mesaje informative, astfel încât utilizatorul poate lua acțiuni corective. Partea de analiză colectează, de asemenea, informații despre programul sursă și le stochează într-o structură de date numită **tabel de simboluri**, care este trimis mai departe împreună cu reprezentarea intermediară către **partea de sinteză**.

Partea de sinteză construiește programul destinație din reprezentarea intermediară și informațiile din tabela de simboluri. Partea de analiză

este adesea numita capătul frontal al compilatorului; partea de sinteză este capătul din spate.

Dacă examinăm procesul de compilare în detaliu, vedem că funcționează

ca o succesiune de faze, fiecare dintre ele transformă o reprezentare a

programului sursă în alta. O descompunere tipică a unui compilator în fazele sale constituent este prezentată în figura de mai jos. În practică, mai multe faze pot fi grupate,și reprezentările intermediare între fazele grupate nu trebuie să fie construite în mod explicit.In figura de mai jos , se poate observa structura fazelor unui compilator.

Sirul de caractere

Analizor Lexical

Sirul de token-uri

Analizor Sintactic

Arbore Sintactic

Analizor Semantic

Arbore Sintactic

Tabela de Simboluri Generator de cod intermediar

Reprezentare Intermediara

Machine-Independent

Optimizator de cod

Reprezentare Intermediara

Generator de cod

Cod Masina Destinatie

Machine-Dependent

Optimizator de cod

Cod Masina Destinatie

Figura 1.2. Structura fazelor unui compilator.

Unele compilatoare au o fază de optimizare independentă de mașină, între

fazele de analiză și sinteză . Scopul acestei etape este de optimizare a

executa transformari asupra reprezentării intermediare în așa fel încât faza de sinteză să poată produce un program destinație mai bun decât în cazul unei reprezentări intermediare neoptimizate. Pentru că optimizarea este optională , una sau mai multe faze de optimizare din figura de mai sus , pot lipsi.

**2. Descrierea BNF (Backus-Naur Form) a gramaticii unui limbaj**

**2.1 Elemente generale.**

În știința calculatoarelor , BNF ( Backus Normal Form sau Backus-Naur Form) este una dintre cele două mari tehinici de notare, folosită deseori pentru descrierea sintaxei limbajelor folosite de calculator , cum ar fi : documente in format de limbaj de programare , seturi de instructiuni și protocoale de comunicație. Există anumite extensii si variante pentru forma originală a notațiilor Backus-Naur , dintre care cele mai folosite sunt Extended Backus-Naur Form (EBNF) și Augmented Backus-Naur Form (ABNF).

Definiția gramaticii :

**[9]**O gramatică liberă de context are patru componente :

* Un set de **simboluri terminale** , uneori denumite “token-uri ". Simbolurile terminale sunt simbolurile elementare ale limbajului definit de gramatica respectivă .
* Un set de **simboluri neterminale** , uneori numite " **variabile sintactice**". Fiecare simbol neterminal reprezintă un set de șiruri de simboluri terminale.
* Un set de **producții** (**reguli**) , unde fiecare unitate de producție este constituită dintr-un simbol neterminal , numit “cap”(“head”) sau “partea stângă a producției” , simbolul “::=" și o secvență de simboluri neterminale și/sau simboluri neterminale, numite “corp”(“body”) sau “partea dreapta a productiei”).
* O definire a unuia dintre simbolurile neterminale ca fiind simbolul de start.

Putem specifica gramatica prin listarea regulilor , cu regulile pentru simbolul de start ca fiind primul. Presupunem că semnele precum “<” si “<=”, cifrele și șiruri de caractere scrise îngroșat (**while**) sunt simboluri terminale. Un nume scris cu Italic este un simbol neterminal, iar orice nume sau simbol scris fără Italic este un simbol terminal. De asemenea, orice symbol scris între paranteze unghiulare (<Simbol>) este un symbol neterminal și orice simbol care nu este scris între paranteze unghiulare este un simbol terminal.

**2.2. Descrierea BNF :**

**[1]**BNF este un acronim pentru "Backus Naur Form". John Backus si Peter Naur au introdus pentru prima dată o notație formală pentru a descrie sintaxa unui limbaj dat (aceasta a fost pentru descrierea limbajului de programare ALGOL 60). Pentru a fi cat mai precis, cea mai mare parte BNF a fost introdus de Backus într-un raport prezentat la o conferință anterioară UNESCO despre ALGOL 58. Puțini au citit raportul, dar atunci când Peter Naur l-a citit , a fost surprins de unele dintre diferențele constatate între interpretarea sa si a lui Backus ,legate de ALGOL 58.. El a făcut câteva modificari care sunt folosite aproape universal și a elaborat singur propriul BNF pentru ALGOL 60 la sedința în care a fost proiectat.

**[2]**BNF este ca un fel de joc matematic: începe cu un simbol (numit simbolul de start și, prin convenție, de obicei, numit S în exemple) apoi sunt date reguli in vederea inlocurii simbolului . Limbajul definit de gramatica BNF este doar un set format din toate șirurile care pot fi produse ,folosind aceste reguli.

Regulile sunt numite reguli de producție, si arată în felul următor :

<Simbol Neterminal> ::= alternative1 | alternative2

O producție pur și simplu afirmă că simbolul din partea stângă a "::=" trebuie să fie înlocuit de unul dintre alternativele din partea dreaptă. Alternativele sunt separate de “|” . (O variantă în acest sens este de a utiliza ":: =" loc de "::=" , dar intelesul este același.) Alternativele constau, în general atât in simboluri neterminale, cât și în simboluri terminale.Simbolurile terminalele sunt pur și simplu bucăți din șirul final care nu sunt simboluri neterminale. Ele sunt numite simboluri terminale, deoarece nu există reguli de producție pentru ele : acestea termină procesul de producție.

O altă variație a gramaticii BNF este să anexăm simbolurile terminale în ghilimele pentru a le deosebi de simboluri. Unele gramatici BNF arată în mod explicit cazul în care spațiul este permis printr-un simbol atașat acestuia , în timp ce alte gramatici lasă acest lucru la discreția utilizatorului.

Dacă un șir nu poate în niciun fel să fie produs prin utilizarea regulilor, șirul nu este permis în limbajul respectiv.

**[1]**Meta-simbolurile BNF sunt:

:: = , însemnând "este definit ca"

| , însemnând "sau"

< > , paranteze unghiulare utilizate pentru a înconjura numele simbolurilor neterminale.

O regulă BNF pentru definirea unui simbol neterminal are forma :

simbol neterminal :: = sequence\_of\_alternatives , alcătuite din șiruri de simboluri terminale sau simboluri neterminale separate prin meta-simbolul ” |”.

De exemplu, producerea BNF pentru un mini-limbaj este :

<Program> :: = program

<declaration\_sequence>

 begin

                    <statements\_sequence>

                end;

Acest lucru arată că un program de mini-limbaj constă în cuvantul cheie "program", urmat de o secvența de declarații (declaration\_sequence), apoi cuvântul cheie "begin" și secvența de operații (statements\_sequence) , în cele din urmă cuvântul cheie "end" urmat de punct și virgulă.

**2.3 Exemplu descriere BNF**

**[5]** Exemplu de descriere BNF pentru o adresa poștală din Statele Unite :

**<**postal-address**>** **::=** **<**name-part**>** **<**street-address**>** **<**zip-part**>**

**<**name-part**>** **::=** **<**personal-part**>** **<**last-name**>** **<**opt-suffix-part**>** **<**EOL**>**

**|** **<**personal-part**>** **<**name-part**>**

**<**personal-part**>** **::=** **<**initial**>** "." **|** **<**first-name**>**

**<**street-address**>** **::=** **<**house-num**>** **<**street-name**>** **<**opt-apt-num**>** **<**EOL**>**

**<**zip-part**>** **::=** **<**town-name**>** "," **<**state-code**>** **<**ZIP-code**>** **<**EOL**>**

**<**opt-suffix-part**>** **::=** "Sr." **|** "Jr." **|** **<**roman-numeral**>** **|** ""

**<**opt-apt-num**>** **::=** **<**apt-num**>** **|** ""

* O adresă poștală , compusă din mai multe părți : nume , adresa străzii și codul poștal.
* **<name-part>(**simbol neterminal) poate fi definit ca :

**<Personal-part>(**simbol neterminal) ,atasat la **<last-name>(**simbol neterminal) ce specifică numele de familie, atașat **la <opt-suffix-part>**(simbol neterminal) reprezentând un sufix opțional și partea de ,atașată la **<EOL>(**simbol neterminal) .Toate aceste simbolurile neterminale reprezintă un șir de simboluri neterminale.Acest sir este separat de un al doilea șir de simboluri neterminale prin “|”. Al doilea șir este alcătuit din **<Personal-part>** alături de **<name-part>.**

* **<personal-part>** poate fi definit ca **<initial>**, urmat de “.”,care este un simbol neterminal (token) sau ca simbolul neterminal **<first-name>.**
* **<street-address>** poate fi definit ca o asociere dintre **<house-num>(**numarul casei), **<street-name>** (numele strazii) **<opt-apt-num>** (numărul apartamentului , unde este cazul) si **<EOL>** (end of line).
* **<zip-part>** poate fi definit ca <town-name> alături de “,”(simbol terminal), **<state-code>**, **<ZIP-code>** si **<EOL>.**
* **<Opt-suffix-part>** poate fi definit ca”Sr.” sau (simbolul “|”) “Jr.” sau **<roman-numeral>** sau un spațiu gol (“ “).
* **<Opt-apt-num>** poate fi definit ca **<apt-number>** sau în cazul în care nu este cazul , un spațiu gol (“ “).

**3. Analiza lexicală si semantică :**

**3.1. Analiza lexicală.**

**[7]**Prima fază a unui compilator este numită **analiza lexicală** sau **scanare**. Analizorul lexical citește sirul de caractere care alcătuiesc programul sursă și grupuri de caractere în secvențe semnificative numite **lexeme** . pentru fiecare

lexemă , analizorul lexical produce la ieșire un semn de formă :

*(token, valoare-atribut)*

pe care îl transmite către faza următoare , analiza sintactică.

Primul component **token**-ul este un simbol abstract , care este utilizat în analize sintactice. Un token este o structură ce reprezintă o lexemă care indică în mod explicit clasificarea sa în scopul de parsare. O categorie de token-uri este ceea ce în lingvistică se poate numi o parte a limbajului.

A doua componentă ,**valoarea atribut** este un pointer la o intrare din

tabela de simboluri pentru token-ul respectiv . Informații din tabela de simboluri sunt necesare pentru analiza semantică și generarea codului.

pentru analiza semantică și generare de cod .

Scopul acestei analize lexicale este de a descoperi și clasifica unitățile lexicale (token-uri), care pot fi cuvinte rezervate, operatori, identificatori, numere întregi, în virgula mobila, șiruri de simboluri, etc. Acestea provin din limbajul de programare pentru care se construiește compilatorul.

Setul existent de legi se poate completa astfel:

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>

<letter> ::= A | B | C | D | .... | Z

<digit> ::= 0 | 1 | 2 | 3 | .... | 9

Să presupunem că un program sursă conține declarația de atribuire :

**p o s i t i o n = i n i t i a l + r a t e \* 60**

**( i d , 1 ) <=) (id , 2) (+) (id , 3) (\*) (60)**

1. “**Position”** este o lexemă care ar fi mapată într-un token (id, 1), unde id

este un simbol abstract ce are rolul de identificator și “1” arata locatia in tabela de simboluri pentru “position”. Linia respectivă din tabela de simboluri pentru un identificator conține informații despre identificatorul respectiv, cum ar fi numele și tipul acestuia.

2. Simbolul de atribuire “**=** “este o lexemă care este mapată în token (=).

Deoarece acest simbol nu are nevoie de valoare-atribut, a doua componentă a fost omisă.

3. “**Inițial”** este o lexemă care este mapată în token (id, 2), unde “2” arată locația în tabela de simboluri pentru “Inițial”

4. “**+**” este un lexem care este mapat în token-ul (+).

5. “**Rate**” este o lexemă care este mapată în token-ul (id, 3), unde “3” arată locația în tabela de simboluri pentru “Rate”.

6. “**\***” este o lexemă care este mapată în token-ul (\*).

7. 60 este o lexemă care este mapată în token-ul (60).

Spațiile ce separă lexemele nu vor fi luate în cosiderare de către analizorul lexical . În această reprezentare , numele token-urilor “=” , “+” , și “\*” sunt simboluri abstracte pentru operatorii de : asignare , sumă si multiplicare.

**[13]**Un token, poate fi reprezentat printr-un cod numeric care specifică clasa din care face parte ,precum și o serie de atribute pentru fiecare clasă în parte. În acest mod, poate exista clasa operatorilor relaționali pentru care un atribut trebuie sa specifice tipul concret al operatorului. Tipul atomului lexical este necesar pentru analiza sintactică, iar valoarea atributului este semnificativă pentru analiza semantică și generarea de cod. Având un atom lexical de tip număr, atributele vor descrie tipul numărului și valoarea acestuia.

**Tratarea erorilor** :

**[14]** Faza de analiză lexicală presupune de obicei existența a două situații:

* apariția unui caracter ilegal;
* nerespectarea regulilor gramaticale.

Apariția unui caracter nedorit determină eliminarea acestuia, iar analizorul lexical va întrerupe astfel verificarea atomului lexical. Analizorul lexical va semnala eroarea și va furniza analizorului sintactic un posibil rezultat. În cazul nerespectării regulilor gramaticale se va întrerupe analiza atomului lexical curent, se va recupera ultima stare finală parcursă a contextului analizei din acel moment și construirea atomului lexical curent.

Este greu pentru un analizor lexical pentru a spune , fără ajutorul altor componente , că există o eroare in codul sursă . De exemplu , în cazul în care un șir este intâlnit pentru prima dată într-un program în C :

**f i ( a == f ( x ) )**

În contextual de mai sus , un analizor lexical nu poate spune dacă este o eroare de scriere a cuvântului cheie , sau un parametru nedeclarat al funcției.. Deoarece “f I” este un lexem valabil pentru id-ul token-ului ,analizorul lexical trebuie să returneze id-ul token-ului la analizorul sintactic și sa lase alte faze ale compilării să se ocupe de eroare .

Alte metode de recuperare in caz de erori sunt :

1. Stergerea unui caracter.
2. Adăugarea unui caracter care lipsește.
3. Înlocuirea unui caracter cu un altul.
4. Transpunerea a două caractere adiacente.

**3.2. Analiza semantică.**

**[8]**Analizorul semantic utilizează arborele sintactic și informațiile din tabela de simboluri pentru a verifica coerența semantică între programul sursă și limbajul folosit . De asemenea, adună informații de tip și le salvează fie în arborele sintactic fie în tabela de simboluri, pentru folosirea lor în timpul generării codului intermediar.

O parte importantă a analizei semantice este verificarea de tip, în care compilatorul verifică coerența dintre operatori și operanzi . De exemplu , multe definiri în limbaje de programare au nevoie de un vector index pentru a declara un număr întreg. Compilatorul trebuie să raporteze o eroare în cazul în care un număr scris în virgulă mobilă este folosit pentru a defini un vector.

Specificațiile lingvistice pot permite anumite conversiuni, numite “**coercion**”. De exemplu, un operator aritmetic binar poate fi aplicat unei perechi de numere întregi sau unei perechi de numere scrise în virgulă mobilă. Dacă operatorul este aplicat unui număr scris în virgulă mobilă și unui întreg, compilatorul va încerca să convertească numărul întreg într-unul cu virgulă mobilă.

O astfel de conversie apare în figua de mai jos. Să presupunem că “position”, “inițial” și “rate” au fost declarate ca fiind numere scrise cu virgulă mobilă și ca lexemă 60 formeaza in sine un întreg. Verificatorul de scriere în analizorul semantic din figura descoperă că operatorul “\*” este aplicat unui număr scris iî virgulă mobilă ,”rate”, și unui întreg, “60”. În acest caz, întregul poate fi convertit într-un număr scris cu virgulă mobilă. Se poate observa în figura, de asemenea, ca la ieșirea analizorului semantic se află un operator “inttofloat”, care convertește explicit argumentul întreg într-un număr scris cu virgulă mobilă.

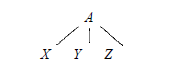
**[15]**Într-un compilator, analizorul sintactic (**parser**-ul) obține un șir de token-uri de la analizorul lexical , așa cum este prezentat în figura de mai jos și verifică dacă șirul numelor token-urilor poate fi generat de gramatica limbajului sursă. Este asteptat că analizorul sintactic să raporteze orice fel de erori de sintaxă și să reia procesul pentru partea de program care a ramas.

Conceptual , pentru programele bine formate , analizorul lexical construiește un arbore sintactic și il transmite compilatorului pentru prelucrare ulterioară . De fapt arborele sintactic nu trebuie să fie construit în mod explicit , deoarece acțiunile de verificare și traducere pot fi intercalate cu analiza sintactică.

În practică, analiza semantică ne ajută să adnotăm **arborele sintactic** cu informația de tip, să construim tabela (tabelele) de simboluri, să modificăm arborele (daca e nevoie) prin inserarea de noduri type-cast. O mare parte din analiza semantică se referă la management-ul contextelor.

**Arbori sintactici :**

**[10]**Un arbore sintactic arată sistematic cum simbolul de start al unei gramatici derivă un șir într-un limbaj . Dacă simbolul neterminal A are o regulă A - > XYZ , un arbore sintactic poate avea un nod notat A cu 3 copii notați X , Y si Z , de la stânga la dreapta, ca în figura de mai jos.



Formal , având în vedere o gramatică liberă de context (nu are un set de reguli prestabilite), un arbore sintactic, conform gramaticii este un arbore cu următoarele proprietăți :

* Rădăcina este marcată prin simbolul de start .
* Fiecare frunză este marcată de un simbol terminal sau de “e” .
* Fiecare nod interior este etichetat printr-un simbol neterminal .
* Dacă A este simbolul neterminal care etichetează un nod interior si X1 , X2, …, X sunt etichetările copiilor acelui nod ,de la stânga la dreapta, atunci trebuie să existe o regulă A - > X1X2….X.

Terminologia arborelui :

• Un arbore este format din unul sau mai multe noduri. Nodurile pot avea etichete, care vor fi simboluri gramaticale. Când trasăm un arbore , adesea reprezentăm nodurile doar prin aceste etichete.

• Exact un nod constituie rădăcina. Toate nodurile, cu excepția rădăcinii au un părinte unic; rădăcina nu are părinte. Când trasăm arborele, plasăm părintele unui nod asupra nodului respectiv și trasăm o linie între ei. Rădăcina este atunci cel mai înalt nod.

• Dacă nodul N este părintele nodului M, atunci M este un copil al lui N.

Copiii unui nod pot fi numiți “rude” . Ei au o ordine, de la

stânga, și atunci când trasăm un arbore ,coordonăm copiii unui nod dat in această manieră.

• Un nod fără copii se numește o **frunză**. Alte noduri - cei cu

unul sau mai mulți copii se numesc **noduri interioare**.

• Un descendent al unui nod N este fie N, un copil al lui N, un copil al copilului lui N, și așa mai departe, pentru orice număr de niveluri. Spunem că nodul N este un “strămoș” al nodului M , dacă M este un descendent al nodului N.

**Analiza top-down :**

**[11]**Analiza sintactică top-down poate să fie interpretată ca operația de construire a arborilor , pornind de la rădăcină și adăugând subarbori de derivare într-o ordine prestabilită. Un compilator analizează datele de intrare dintr-un limbaj de programare pentru a construi limbajul de asamblare sau o reprezentare internă prin potrivirea simbolurilor din intrare cu regulile de producţie în forma Backus-Naur.

**[16]**Un analizor sintactic top-down, aplică fiecare regulă simbolurilor intrate, lucrând de la cel mai din stânga simbol generat de o regulă şi apoi continuând cu următoarea regulă pentru fiecare simbol neterminal întâlnit. În acest fel, analiza începe din stânga rezultatului regulii (partea dreaptă) şi evaluează simbolurile neterminale începând cu stânga şi, deci, coboară în arbore pentru fiecare simbol neterminal nou înainte să continue cu următorul simbol neterminal.

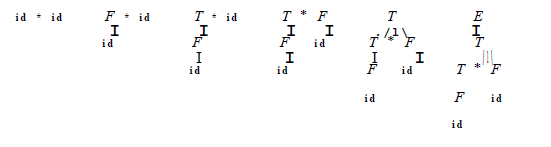
**Analiza predictivă :**

**[12]**Analiza predictivă este o metodă de analiză top-down în care un set de proceduri recursive este utilizat pentru a procesa o intrare . O procedură este asociată cu fiecare simbol neterminal al unei gramatici . Aici , avem în vedere o formă simplă de analiză top-down recursivă, numit analiză predictivă , în care simbolul considerat determină procedura de control pentru fiecare symbol neterminal. Secvența de apeluri de proceduri în timpul analizei

unui șir de intrare definește implicit un arbore sintactic pentru intrare , și poate fi folosit pentru a construi un arbore sintactic explicit, dacă se dorește, acest lucru.

**Analiza bottom-up :**

**[17]**O analiză bottom-up a construirii unui arbore sintactic pentru un șir la intrare care începe de la frunze (din partea de jos) și urcă până la rădăcină (partea de sus).Este convenabil pentru a descrie analiza ca procesul de construire a unor arbori sintactici, deși se poate efectua o traducere directă, fără construirea unui arbore explicit . Secvențele de fragmente de arbori din figura de mai jos ilustrează o analiză bottom-up pentru simbolurile terminale id\*id, respectând gramatica folosită.



Bibliografie

1. p Compilers Second Edition ***Principles, Techniques, <§ Tools*** Second edition (**Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman)**
2. <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html> [1]
3. <http://www.garshol.priv.no/download/text/bnf.html#id1.2>. [2]
4. <http://matt.might.net/articles/grammars-bnf-ebnf/> [4]
5. http://stackoverflow.com/questions/18818196/entry-rule-position-convention-in-bnf [5]

Acestea sunt capitolele pe care le-am folosit din cartea pe care ne-ati recomandat-o :

* **1.Introduction** :
* **1.1** Introduction [4]
* **1.1 Language Processors** [5]
* **1.2** The Structure of a Compiler [6]
  + - **1.2.1**Lexical Analysis [7]
    - **1.2.3** Semantic Analysis [8]
* **2.A Simple Syntax-Directed Translator**
  + - * **2.1** Introduction
      * **2.2** Syntax Definition
        + **2.2.1** Definition of Grammars [9]
        + **2.2.3** Parse Trees [10]
      * **2.4** Parsing
        + **2.4.1** Top-Down Parsing [11]
        + **2.4.2** Predictive Parsing [12]
* **3. Lexical Analysis**
  + - **3.1** The Role of the Lexical Analyzer
* **3.1.2** Tokens, Patterns, and Lexemes [13]

**3.1.4** Lexical Errors [14]

* + **4 Syntax Analysis**
    - * **4.1** Introduction

**4.1.1** The Role of the Parser [15]

* **4.4.** Top-Down Parsing [16]
* **4.5**. Bottom-Up Parsing [17]