Universitatea Politehnică Bucureşti
Facultatea de Electronica, Telecomunicatii şi Tehnologia Informatiei

Sisteme de Operare
Compilatoare

Studenti:
Robitu Paul-432A
Banu Laura-432A

2011-2012

Cuprins

1. Introducere –Banu Laura

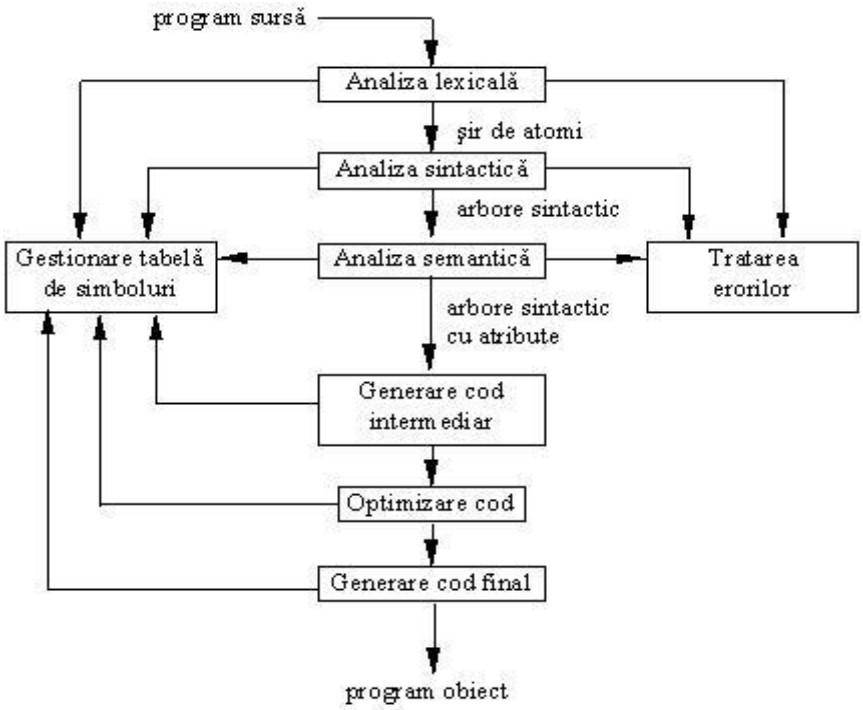
Aspecte generale

Software-ul de astazi este dependent de evolutia limbajelor de programare deoarece modul de implementare al unui software, optimizarea acestuia etc. duc in mod evident la o automatizare mai buna a sarcinilor cat si a unei necesitati din ce in ce mai reduse de resurse hardware. Limbajele de programare sunt notatii folosite la descrierea calculelor pentru oameni dar si pentru masini. Desigur, inainte ca un program sa poata rula pe o masina, trebuie translatat dintr-un limbaj accesibil omului intr-un limbaj inteles de masina pe care se executa. Aceasta translatie este realizata de catre compilator.

2. Structura unui compilator-Banu Laura

Procesul de compilare a unui program are loc in mai multe faze. O faza este o operatie unitara in cadrul careia are loc transformarea programului sursa dintr-o reprezentare in alta.

Principalele faze ale unei compilari sunt cele din figura de mai jos:



2.1 Analiza lexicala –Banu Laura

Procesul de translatare are ca prima faza analiza lexicala.Aceasta presupune cautarea cuvintelor si a altor entitati in interiorul textului si transforma programul sursa vazut ca un sir de caractere intr-un sir de unitati lexicale,numite atomi lexicali (in engleza : tokens).Asadar,la intrare,analizorul lexical primeste programul ca un sir de caractere,litere,spatii si simboluri,iar la iesire va rezulta un sir de atomi lexicali.

Atomul lexical poate fi privit ca un reprezentant al unei clase de siruri de caractere ce are reguli de formare exacte. In majoritatea limbajelor de programare, printre alte clase, vom gasi:

- clasa operatorilor
- clasa identificatorilor;
- clasa constantelor fractionare;
- clasa constantelor intregi;

Analizorul lexical (in engleza-scanner) este implementat ca un automat finit determinist,o masina cu un numar limitat de stari,care-si modifica starea la primirea unui caracter.

In concluzie, sarcina analizorului lexical este sa detecteze in sirul de caractere al programului sursa subsiruri ce respecta regulile de formare ale atomilor lexicali, sa clasifice aceste subsiruri si sa le traduca in atomi lexicali, adica in informatii codificate ce vor avea totusi in continutul lor esentialul: clasa fiecarui subsir si eventual, modul de regasire al subsirului.

Pentru acest lucru,compilatorul are o zona rezervata pentru tabela de simboluri.Practic,toti identificatorii si constantele din program se gasesc in acest « dictionar ».

In aceasta tabela se cauta identificatorul gasit in sirul de la intrare, iar in loc de indicator spre sir, in atomul lexical se plaseaza un indicator spre intarea in tabela de simboluri corespunzatoare identificatorului.

Atunci cand se proiecteaza un analizor lexical trebuie sa se tina cont de structura intregului compilator,de restrictiile lui de proiectare.Exista diverse decizii care se pot lua in privinta proiectarii analizorului lexical.

Cand se face referire la relatia cu analizorul sintactic putem deosebi :

a) Analizorul lexical comandat de analizorul sintactic.Este cea mai frecvent utilizata varianta. El apare ca o rutina a analizorului sintactic pe care acesta o apeleaza ori de cate ori are nevoie de un nou simbol.

b) Analizor lexical care nu depinde de anlizorul sintactic.Aici, analizorul lexical reprezinta un « pas » al compilatorului,zona de memorie ce contine datele prelucrate sau un fisier facand legatura cu fazele urmatoare.

Uitandu-ne la structura analizorului lexical deosebim:

a) Analizor lexical alcatuit dintr-un singur bloc. Analiza lexicala este abordata si rezolvata global.

b) Analizor lexical structurat.Spre deosebire de cel prezentat anterior,sarcinile definite la proiectare sunt oarecum independente in cadrul analizei lexicale,fiecare lucrand cu o parte distinca a analizorului. La aceasta solutie se ajunge prin stratificarea gramaticii atomilor lexicali. Fiecarui strat ii va corespunde o subfaza a analizei lexicale.

2.2. Tratarea erorilor-Banu Laura

Faza de analiza lexicala presupune de obicei existenta a doua situatii:

- nerespectarea regulilor gramaticale.
- aparitia unui caracter ilegal;

Aparitia unui caracter ilegal duce la eliminarea acestuia,analizorul lexical intrerupand verificarea atomului lexical.Tot analizorul lexical va furniza analizorului sintactic un rezultat posibil.

Importanta regulilor gramaticale este foarte mare.Nerespectarea lor duce la intreruperea analizei atomului lexical curent,recuperarea ultimei stari

parcurse integral,a continutului analizei din acel moment si,de asemenea,construirea atomului lexical curent.

Iata in cele din urma exemplul unui analizor lexical ce are ca rol recunoasterea unui caracter alfanumeric sau a unei cifre :

```
function isNumeric(c: char): boolean;
begin
isNumeric := isAlpha(c) or isDigit(c);
end;


function isAlpha(c: char): boolean;
begin
IsAlpha := upcase(c) in ['A'..'Z'];
end;


function isDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
```

2.3. Sintaxa-Banu Laura

Multimea infinita de propozitii,corecte din punct de vedere al formei lor sunt determinate de regulile de sintaxa.Daca privim din punct de vedere semantic,doar o parte din cele precizate anterior sunt corecte.

Propozitiile sunt alcatuite din elemente numite simboluri.

Modul in care caracterele care formeaza alfabetul limbajului, se grupeaza formand simboluri este descris prin regulile lexicale. Ele pot fi privite ca facand parte din sintaxa limbajului. Totalitatea simbolurilor formeaza vocabularul limbajului. Din vocabular fac parte identificatori,cuvinte cheie (ca begin, var, mod in Pascal), operatori (+,++,<=,==, | in C), literale (intregi, flotante, siruri de caractere).

Specificarea sintaxei prin gramatici

Gramatica limbajului este reprezentata de totalitatea regulilor sintactice ale unui limbaj.O metoda raspandita de descriere a unei gramatici este metoda cunoscuta sub numele de BNF(Backus Naur Form).
In ceea ce urmeaza se va prezenta forma extinsa a BNF.
BNF este un limbaj folosit pentru descrierea unui altui limbaj («metalimbaj ») care foloseste urmatoarele metasimboluri:
[si] sunt folosite pentru a include secvente optionale
< si > -sunt folosite pentru a include neterminale
::= -insemnand « definit ca »
| - insemnand « sau »
{ si } sunt folosite pentru a include secvente care se pot repeta de oricate ori (inclusiv de zero ori).

O succesiune de reguli BNF reprezinta sintaxa.

Fiecare relatie defineste un neterminal specificat in stanga semnului ::=. In dreapta semnului gasim o succesiune de neterminale si terminale. Terminalele sunt reprezentate de simboluri ale limbajului. Fiecare neterminal gasit in partea dreapta a unei relatii trebuie sa fie definit intr-o alta relatie. Gramatica completa a limbajului reprezinta un set de relatii prin care sunt definite toate neterminalele. Un anumit neterminal particular este asa numitul simbol de start; acesta se numeste in mod tipic « program » .

Conform celor spuse mai sus,o succesiune de simboluri(adica de terminale) alcatuiesc un program.Daca aceasta succesiune se poate genera pe baza regulilor gramaticale pornind de la simbolul de start atunci programul este corect din punct de vedere sintactic.

In momentul in care derivam un arbore sintactic facem,de fapt,o analiza sintactica,prin care verificam corectitudinea din punct de vedere sintactic a unui program.

2.4. Analiza sintactica-Banu Laura

Analizorul sintactic(« parser-ul »)grupeaza atomii lexicali in structuri sintactice.Rezultatul este o reprezentare arborescenta.

In multe cazuri arborele sintactic nu construieste in mod explicit arborele de derivare,ci pentru fiecare pas in constructia arborelui,prin intermediul procedurilor semantice ,declanseaza actiuni asupra bazei de date.

Putem distinge doua strategii de construire a arborelui de derivare :cea descendenta-constructia arborelui este realizata de la radacina spre frunze si cea ascendenta-invers celei anterioare.

Sirul de intrare ghideaza constructia ,atat in cazul derivarii descendente,cat si in cazul celei ascendente.

O alta clasificare a analizoarelor sintactice este data de existenta revenirilor.

O gramatica independenta de context furnizata parserului ii spune acestuoa ce structuri sintactice sa recunoasca. Parser-ul este cel mai adesea implementat ca un automat push-down,adica o masina cu numar finit de stari inzestrata si cu o stiva.Stiva ii este necesara pentru a urmari imbricarea structurilor sintactice.

2.4.1.Analiza sintactica descendenta-Banu Laura

Vom considera o gramatica G_1 :
Analiza sintactica cu reveniri :

$$S \rightarrow bAe$$
$$A \rightarrow d \mid dA$$

si avem cuvantul *bdde* din limbajul generat de gramatica.[6]

Acest exemplu va pune in evidenta functionarea analizoarelor sintactice descendente.Automatul « push-down »nedeterminist simuleaza in functionarea sa derivarea stanga a sirului acceptat .

Actiunile principale sunt :se inlocuieste un neterminal A din varful stivei cu un sir care semnifica o parte dreapta a A-productiilor gramaticii si se sterge un terminal din varful stivei doar daca acesta coincide cu terminalul curent de la intrare,dupa care se realizeaza un avans al intrarii.Stiva care nu are nici un element « beneficiaza » de un asa-numit criteriu al acceptarii.Putem spune ca un analizor sintactic incearca sa simuleze modelul automatului « push-down »,deoarece el incearca sa refaca derivarea stanga a simbolului de ineput in sirul analizat.Acest analizor pentru fiecare neterminal va avea diverse alternative de rescriere a lui.Aceasta operatie poarta numele de expandare.

Daca in urmatorul obiectiv gasim un simbol terminal,si el este acelasi cu urmatorul simbol de la intrare,analizorul efectueaza operatia de avans.

In exemplul considerat,obiectivul initial al analizorului este S si avand decat o singura structura posibila,bAc,in prima faza nu se va poate propune decat acesta,ceea ce va duce la un arbore ca cel din figura 1.

Astfel ca S,va fi inlocuit de structura formata din obiectivele : b,A,c .Acestea reprezinta etichetele descendentilor imediati ai nodului S.Un terminal trebuie sa aiba un simbol curent in sirul de intrare,ceea ce pentru exemplul considerat este adevarat.

Analizorul sintactiv va efectua o operatie de avans,dupa care se doreste gasirea in sirul de intrare (neanalizat pana la acel moment) a unei structuri a lui A.

Arborele din figura 2 se obtine prin expandarea lui « A »cu « d ».

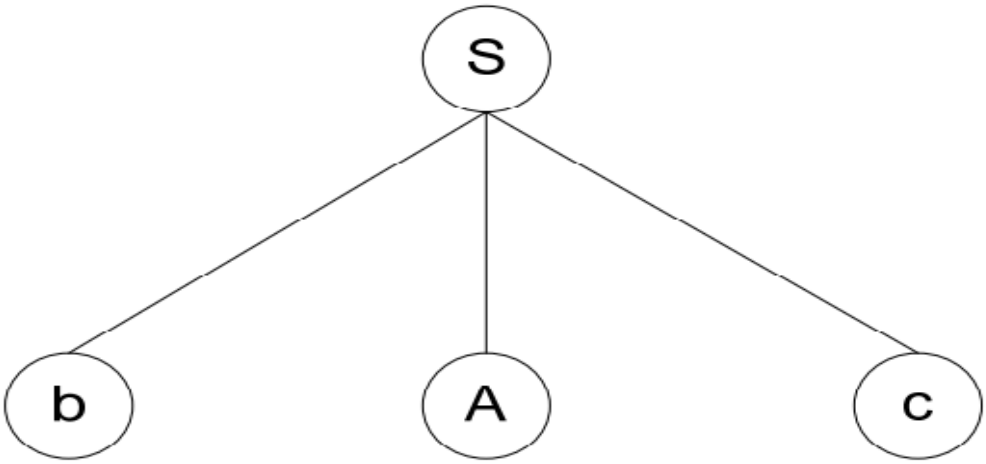


Fig.1 [6]

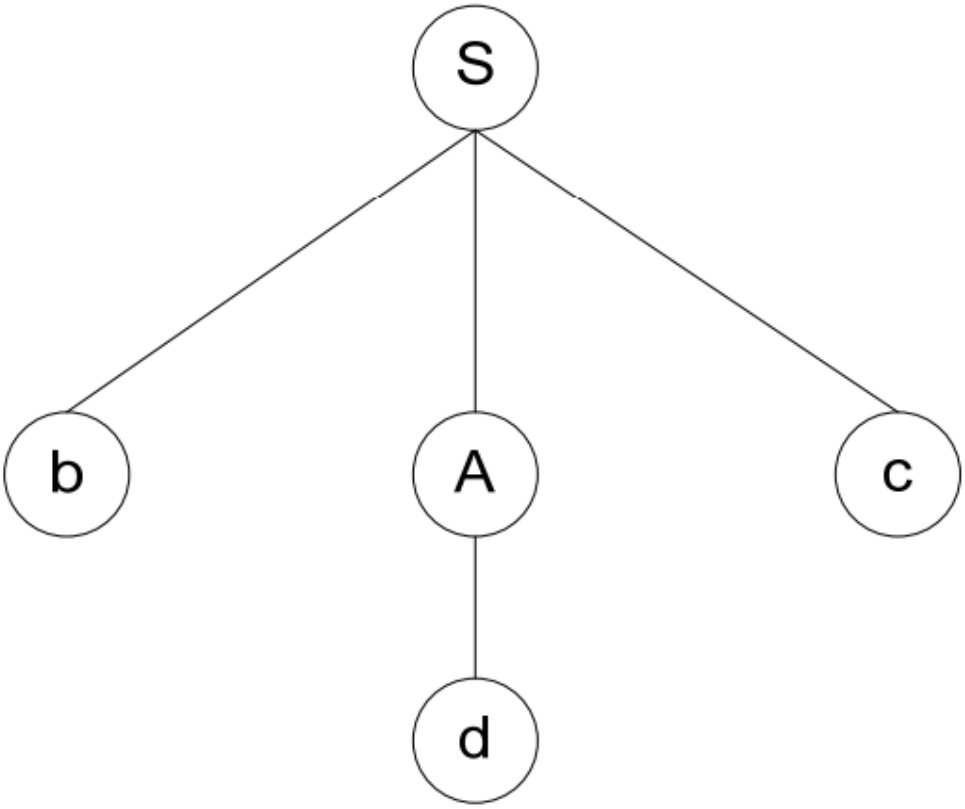


Fig.2 [6]

Avand in vedere ca obiectivul A este inlocuit cu d,rezulta ca s-a efectuat recunoasterea terminalului d in sirul de intrare.Putem observa ca urmatorul simbol de la intrare este d ceea ce reprezinta pentru analizor un avans.

Urmatoarea sarcina a analizorul sintactic este sa recunoasca in sufixul de al sirului de la intrare structura ultimului obiectiv :e.

Se va sesiza o eroare in luarea deciziilor deoarece nu se poate avansa cu e si,prin urmare,analizorul sintactic se va intoarce la cel mai recent obiectiv propus si depasit adica A,venindu-se cu o noua alternativa :dA.

Conform celor spuse mai sus,vom avea o revenire in punctul in care era analizorul cand a gasit prima alternativa de expandare a lui A adica la dde.Vom avea arborele din figura 3 in cazul expandarii cu dA.Obiectivul A va fi inlocuit cu doua obiective :d si A,care permite avansul si trecerea la obiectivul :A.Este propusa din nou expandarea cu d,urmata de avansul cu d,obtinandu-se figura 4.

Asadar,se raporteaza succes obiectivului de pe nivelul superior,care este tot A,si care,la randul lui este rezolvat pozitiv.Se trece din nou la ultimul obiectiv din structura bAe,adica e.De aceasta data,sirul de intrare neanalizat este chiar e.Datorita acestui fapt se face un avans si se raporteaza succes pt e si apoi

pentru S.Observam ca apare o coincidenta,si anume,epuizarea sirului de intrare si succesul lui S ceea ce inseamna,de fapt,acceptarea.

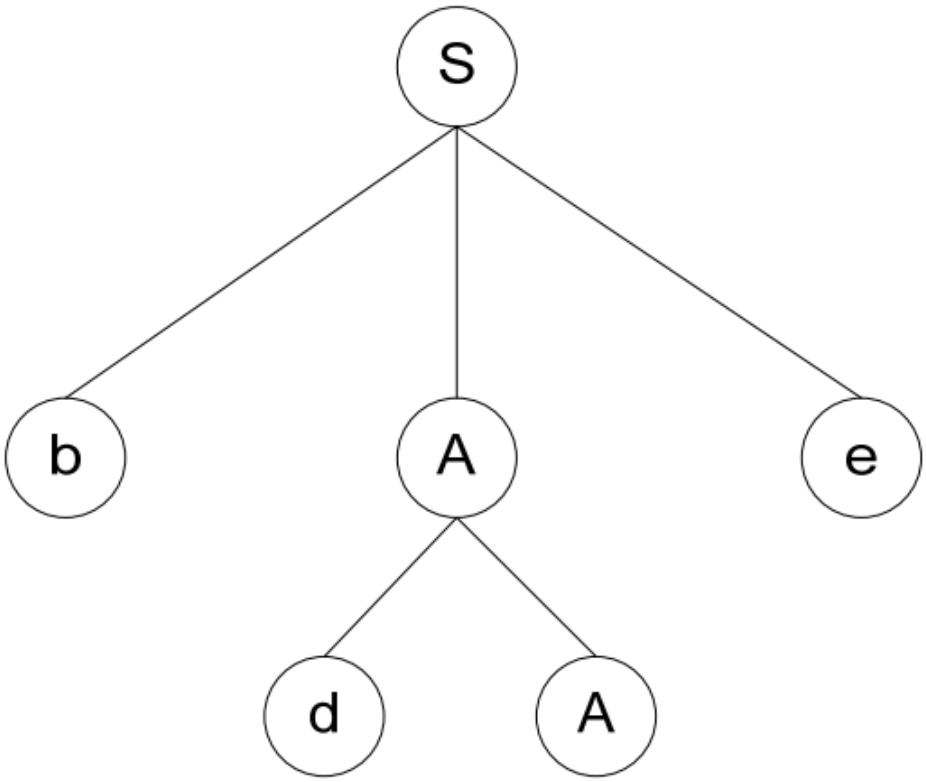


Fig.3 [6]

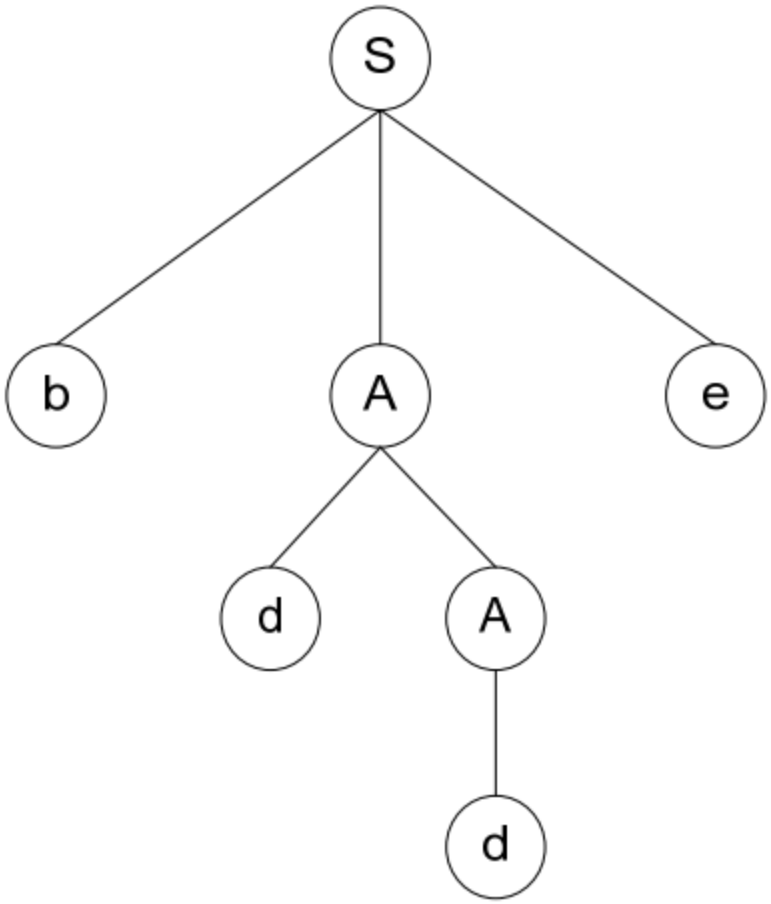


Fig.4 [6]

Consideram cazul in care sirul de la intrare ar fi fost bddb.Daca s-ar fi incercat avansul cu e in acest caz,s-ar fi inregistrat esec care ar fi condus la reveniri succesive astfel :s-ar fi cautat o alta alternativa pentru A din frontiera bAe a arborelui si,pentru ca aceasta nu mai exista,deplasarea in obiectivul b,cel mai stang descendent a lui S(care este terminal),va raporta esecul intregii alternative bAe.
Esecul lui bAe,este si esecul lui S.Pentru ca nodul S este nod radacina,aceasta situatie duce la neacceptarea sirului bddb.
Derivarea stanga a lui S in bdde este pusa in evidenta in ceea ce urmeaza :

$$S \Rightarrow bAe \Rightarrow bdAe \Rightarrow bdde \quad [6]$$

echivalenta cu construirea arborelui de derivare a sirului.

2.4.2. Analiza sintactica ascendenta –Banu Laura

Consideram gramatica:

S→ bAcBe
A→Aa |a
B→d

si cuvantul baacde din limbajul generat de gramatica.[6]

Vom urmari drumul care pleaca de la sirul baacde si gaseste in final neterminalul S,drum parcurs de un analizor cu strategie ascendenta.

Automatul «push-down » extins este modelul analizei sintactice ascendente.El poate simula in functionarea sa derivarea dreapta intr-o gramatica independenta de context ,derivare parcursa insa in ordine inversa.Pentru aceasta el gaseste in varful stivei parti drepte ale productiilor pe care le «reduce » la neterminalul partii stangi corespunzatoare.Se aduc simboluri de la intrare pentru a putea fi completata partea dreapta.Sirul este acceptat doar in cazul in care a ramas doar simbolul de la inceput.

Exemplul considerat de noi pune in evidenta cum un analizor sintactic ascendent cauta parti dreapta ale productiilor in sirul de intrare.Ceea ce va gasi va fi :a si d.Dupa depasirea lui b,daca analizam de la stanga la dreapta,se va prefera inlocuirea primului a cu A,rezultand astfel sirul bAacde.In figura 1 este reprezentat subsirul analizat.

Analizorul ascendent va creea o varietate de arbori pe care ii va reuni intr-un arbore unic.In acest caz,o varianta de arbore este cel format dintr-un nod etichetat b si o alta varianta ca cel din figura 2.

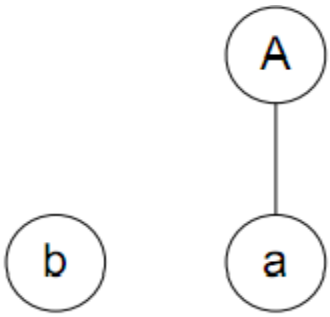


Fig.1[6]

Fig.2 [6]

Se cauta parti dreapta in bAacde si analizorul va gasi,in ordine :Aa,a si d.De aceasta data,datorita ordinii de analiza de la stanga la dreapta a intrarii,vor exista doua posibilitati :Aa si a.
Cele doua nu sunt la fel de bune,desi ambele se reduc la A.
In cazul in care se alege reducerea lui a la A se va obtine sirul bAAcde care apoi prin reducerea lui d la B conduce la sirul bAAcBe in care analizorul este incapabil sa recunoasca o parte dreapta.Acum este necesara revenirea la decizia reducerii lui a.
In figura 3 este reprezentat cazul in care se alege reducerea lui Aa la A,obtinandu-se sirul bAcde.

Adaugand e si d la « padurea »din figura 3,singurul sir redus este d.Se obtine bAcBe si « padurea »reprezentata in figura 4.

Arborele de derivare din figura 5 se obtine dupa adaugarea ultimului simbol la padurea de arbori,cu observatia ca noul sir obtinut,ca si radacinile arborilor padurii,alcatuiesc partea dreapta a primei productii,care se reduce la S.

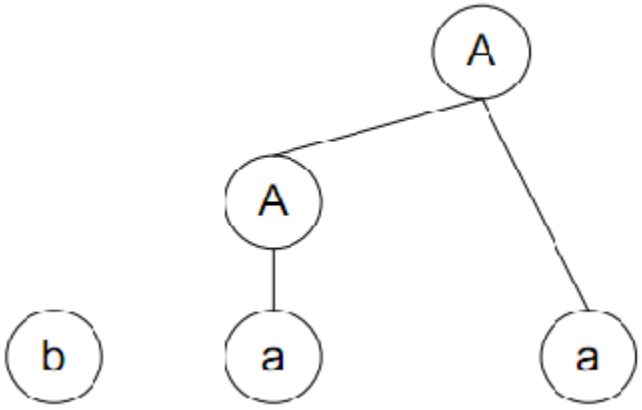


Fig.3 [6]

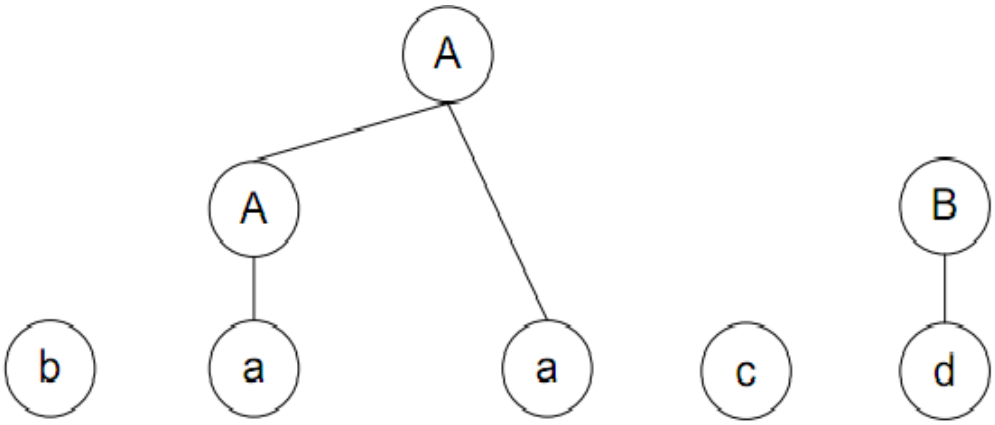


Fig.4 [6]

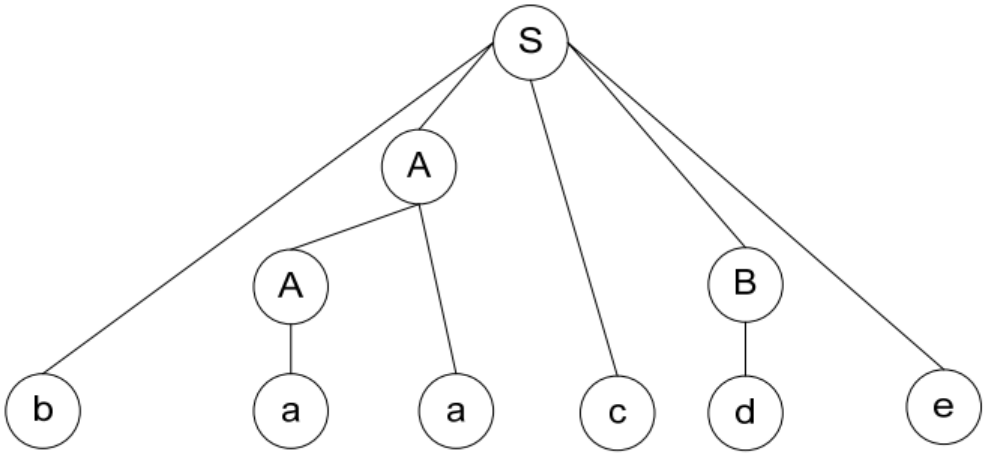


Fig.5 [6]

In final, se doreste acceptarea sirului.
Derivarea dreapta refacuta in ordine inversa :

$$S \Rightarrow bAcBe \Rightarrow bAcde \Rightarrow bAacde \Rightarrow baacde$$

prin reduceri:

$$baacde \dashrightarrow bAacde \dashrightarrow bAcde \dashrightarrow bAcBe \dashrightarrow S \quad [6]$$

Rolul principal il joaca subsirul redus.Prima etapa este identificarea subsirului , ca dupa aceea el sa fie inlocuit cu un neterminal.Acesta poarta sugestiv numele de capat (« handle »).

2.5. Analiza semantica-Banu Laura

Scopul analizei este obtinerea unui program echivalent intr-un limbaj intermediar. Toate informatiile necesare pentru urmatoarele faze se vor afla in acest program care reflecta exact structura programului.

Parte de extragere a structurii programului sursa este realizata in fazele de analiza lexicala si analiza sintactica. In cazul nostru, analiza semantica va folosi valorile atributelor cu scopul completarii structurii sintactice. Se valiedaza sau nu o data in plus din punct de vedere al corectitudinii programului sursa si se alcatuieste codul intermediar.

In prima faza, analiza sintactica ofera arborele de derivare analizei semantice, funzele fiind atomi lexicali. Mai apoi, analiza semantica va evalua atributele asociate nodurilor arborelui. In continuare, se va genera o forma intermediara pentru programul surs.

Arborele de derivatie este in general foarte voluminos, cu multe redundante, necesitand in mod evident un spatiu de memorie extrem de mare. Acest lucru atrage dupa sine si un timp prelungit de procesare datorita accesului lent la memorie. Din aceste cauze, arborele de derivatie nu este prorpiu-zis construit, ci se prefera o structura mai sintetica, aceea a arborelui sintactic. Daca se face analiza printr-o singura trecere, informatiile referitoare la structura programului se vor concentra in sirul actiunilor amalizorului.

Din motive didactice, vom prezenta pe scurt modul de constructie a arborelui in cazul analizei ascendente. Am ales acest exemplu deoarece operatia este relativ facila.

Fiecarui simbol din stiva folosita la analiza i se va asocia un indicator catre radacina unui arbore aflat practic intr-o multitudine de astfel de arbori. Cand se deplaseaza un simbol de la intrarea in stiva se va creea arborele cu un singur nod, avand eticheta chiar simbolul respectiv. Cand se reduce un capat K1, K2, ..., Km la un neterminal A, se va creea un not cu eticheta A si i se va asocia radacinile arborilor etichetati K1,K2,...,Km ca fiind descendenti directi. In plus, se mai asociaza si indicatiile de intrare ale acestor simboluri din stiva.

Figura 1 ilustreaza succesiunea etapelor constructiei arborelui de derivare in cazul sirului a1 ;a2.[6]

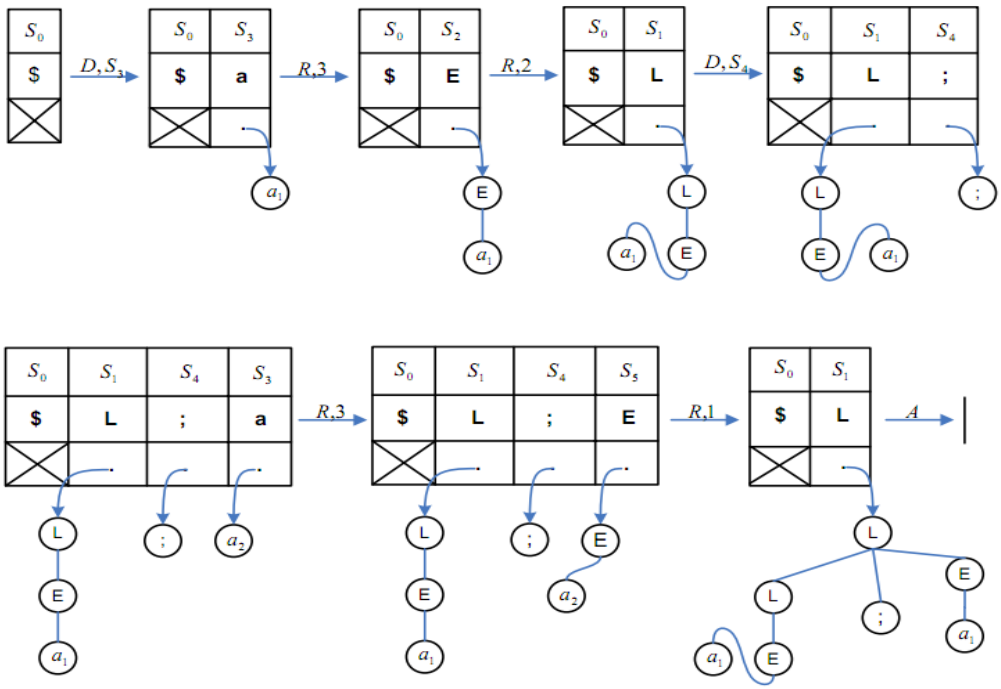


Fig.1[6]

Analiza semantica si cea sintactica au loc in acelasi timp, in compilatoarele folosite in practica.

2.6. Generarea codului intermediar-Robitu Paul

La sfarsitul fazelor de analiza (sintactica si semantica) vom regasi un fisier continand cod intermediar care se apropie mai mult de limbajul de asamblare. Avem de a face cu un program constituit dintr-o succesiune de operatii si, in mod evident, de operanzi. Ordinea de executie a operatiilor este cea intuitiva, este chiar ordinea de succesiune in programul nostru. Datorita apropierii de limbajul de asamblare, regasim operatii aritmetice, salturi, testari, atribuiri. In plus, nu vom regasi nici declaratiile, iar operanzii se gasesc in tabela de simboluri. Deosebirea majora intre limbajul de asamblare si codul intermediar este aceea ca nu vom avea operanzii sub forma de registre si nici de cuvinte stocate in memorie, acestia sunt mai degraba referinte la intrari din tabela de simboluri.

Cat priveste restul informatiilor necesare unei bune executii a programului, in operanzi vom putea regasi si modul de adresare (indirect sau direct), variabile temporare, constante, variabile indexate, variabile

simple etc. Asadar strucura poate fi de forma record unde putem defini toate cele mentionate mai sus.

Exista patru solutii pentru codul intermediar : forma poloneza, arbori sintactici, triplete si cuadruplete. Alegerea uneia dintre aceste solutii va influenta in mod direct structura secventei operatiilor cat si modul de reprezentare a unei instructiuni.

2.7. Tabela de simboluri-Robitu Paul

Tabela de simboluri este o strucura in care sunt concentrate informatii colectate de compilator in ceea ce priveste numele simbolice pe care acesta le intalneste in programul sursa. Asadar, aceasta tabela poate fi un simplu tablou de inregistrari. Totusi, aceste tabele pot avea si alte reprezentari mai complicate precum liste sau chiar arbori.

Tabela de simboluri este folosita in fazele compilarii. O intrare este constituita din numele simbolic si atributele acestui sir. Numele simbolic este pur si simplu un sir de caractere, in schimb atributele pot fi de mai multe feluri. Se pot descrie tipul simbolului, categoria din care face parte (procedura, eticheta, variabila simpla etc.) sau adresa zonei alocate in memorie.

Constatam ca putem structura tabela, sau mai corect, a constitui tabele specializate pentru variabile, proceduri, constante,etichete. O alta posibilitate ar fi diferentierea tabelelor in functie de lungimea simbolurilor.

Fazele in care este utilizata tabela de simboluri :

1) In faza de analiza, compilatorul se foloseste de tabela pentru a afla daca numele cu pricina a fost deja memorat . Daca se regaseste memorat, se va retine doar adresa intrarii. In caz contrar, acesta este introdus, adresa de intrare fiind si ea retinuta.
2) O data ce adresa a fost retinuta, se pot inspecta diversele atribute aflate in intrare sau se pot actualiza campurile atributelor. Se va determina consistenta utilizarii numelui cu declaratia lui, activitate specifica fazei de analiza semantica.
3) In faza generarii de cod, atributele servesc la determinarea lungimii zonelor alocate variabilelor.
4) Atributele memorate in tabela de simboluri sunt folosite pentru tratarea erorilor de compilare.

In ceea ce priveste proiectarea unei astfel de tabele de simboluri, trebuie avut in vedere implementarea catorva functii primitive functionale :
a) verificarea existentei unui nume simbolic in tabela,
b)adaugarea unui nou nume in tabela,
c) accesul la diferitele informatii asociate unui nume,
d)actualizarea unor informatii asociate unui nume,
e) stergerea unui nume sau mai multora din tabela.

2.8. Optimizarea codului-Robitu Paul

Codul intermediar descris mai sus nu este neaparat cel optim. Existenta unei faze de optimizare este asadat foarte necesara. Rearanjarea codului intermediar va avea ca efect eficientizarea folosirii memoriei dar in special diminuarea timpului de executie. Acest lucru se poate realiza in acelasi timp fie cu analiza semantica fie in acelasi timp cu generarea de cod. Desigur, exista si optiunea realizarii acestei operatii intr-un pas separat.

« Optimizarea se poate realiza atat asupra codului intermediar cat si asupra codului obiect.In ultimul caz, imbunatatiri importante ale eficientei executiei putem obtine daca, in paralel cu generarea de cod urmarim sa optimizam alocarea registrelor sau folosirea setului de instructiuni ale masinii. De aceea, metodele folosite pentru optimizarea codului obiect sunt puternic dependente de masina si deci mai dificil de prezentat intr-un mod unitar. » [6]

In general, imbunatatirea codului se poate realiza prin : realizarea a cat mai multor calcule inca de la compilare, eliminarea redundantelor, „factorizarea” invariantilor din cicluri.

2.9.Generarea codului obiect-Robitu Paul

Faza finala a compilarii consta in generarea codului obiect, mai precis sinteza programului obiect. Acesta este de regula executabil sau se apropie

foarte mult. Un alt procesor poate prelua acest program si tradus de acesta in cod executabil.

In general programele obiect pot avea una din formele urmatoare :

1. Programul imediat executabil disponibil in limbaj masina. Mai este numit si program absolut. Totusi, are marele dezavantaj ca nu poate fi compilat pe parti.
2. Compilatoarele comerciale prefera o forma de program remarcabil deoarece este facila editarea legaturilor a unor rutine implementate sau chiar realizate de utilizatori.
3. Forma de program in limbaj de asamblare, care are avantajul ca se poate implementa mai usor, problemele traducerii sunt preluate de faza de asamblare a programului obiect. Dezavantajul il constituie timpul crescut de compilare din cauza pasului de asamblare(suplimentar).
4. Forma simplificata intr-un alt limbaj de programare. Aceasta varianta mareste si mai mult timpul deoarece se introduce cel putin o compilare in plus. Intalnim aceasta forma in cazul preprocesoarelor de limbaje.

In Fig. 1, avem diagrama generarii codului unui obiect. Avem de a face cu un proces de parcurgere a instructiunilor in ordine, pentru fiecare apeland-se proceduri specifice de generare corespunzatoare.

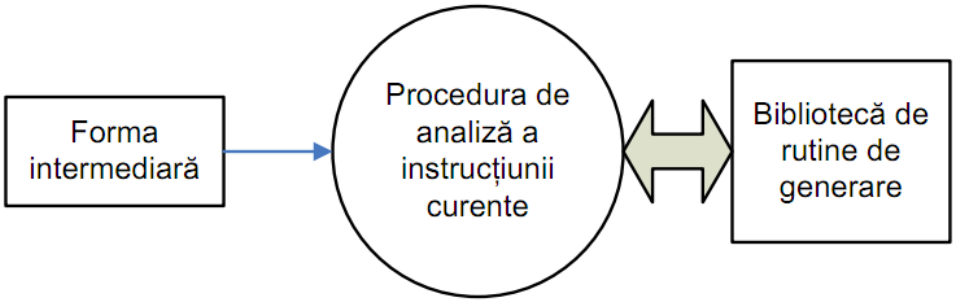


Fig.1 [6]

Sa luam exemplul unei expresii matematice (X-Y)*(M+N+P). Codul intermediar va fi :

T1 :=X-Y
T2 :=M+N+P
T3 :=T1*T2

In cazul in care folosim un calculator cu acumulator, se vor crea urmatoarele instructiuni :

1. LOAD X - incarcarea lui X in acumulator
2. SUB Y- scaderea lui Y din acumulator
3. STO T1 - memorarea acumulatorului in T1
4. LOAD M - incarcarea lui M in acumulator
5. ADD N - adunarea lui N la acumulator
6. ADD P - adunarea lui P la acumulator
7. MPY T1 - multiplicarea acumulatorului cu T1

In exemplul de mai sus se poate observa cu usurinta faptul ca avem parte de operatii din codul intermediar (ADD,SUB,MPY) dar si intructiuni care pregatesc acumulatorul, nespecifice codului intermediar (LOAD,STO). Avand partea stanga a expresie deja stocate in acumulator, ultima operatie ne-a permis sa folosim imediat operatia pe acumulatorul existent.

3. Compilatoarele GCC (linux) si PellesC (Windows x64)-Robitu Paul

Arhitectura back-end in ceea ce priveste compilatoarele este comuna atat in sistemele bazate pe UNIX cat si in sistemele Microsoft Windows. Compilatorul este cel care genereaza executabilul pentru ambele sisteme. Daca CL.EXE este cel mai popular compilator pe platforma Windows, ne-am orientat studiul catre un compilator pentru un sistem Windows pe 64 de biti datorita popularitatii in crestere.

3.1. Etapele compilarii

Desigur, etapele compilarii sub Windows sau UNIX sunt commune :

1. Preprocesare C
2. Analiza si Translatarea
3. Asamblarea
4. Link-editarea

1. Preprocesarea C

Directivele de preprocesare incep cu #. Exista mai multe tipuri de directive :

- Definirea si anularea definirii #define si #undef
- Preprocesarea conditionala cu ajutorul #if, #ifdef ”macro”, #endif, #else, #elif
- Includerea fisierelor header cu ajutorul #ifndef, #include

Atributul –e permite atat pentru gcc cat si pentru cc (PellesC) rularea etapei de preprocesare.

2. Analiza si Translatarea

Acestea sunt cele mai importate etape in compilarea codului sursa. Ele nu sunt comute pentru UNIX si Windows in ceea ce priveste sintaxei de asamblare. Codul sursa al programului este transformat in cod obiect (cod intermediar specific compilatorului de C). Se creeaza un fisier cu nume identic dar avand extensia .obj sau .o. Aici sunt semnalate eventualele erori sintactice din codul sursa.

In plus, putem remarca aici flexibilitatea compilatorului gcc in raport cu PellesC. Desi ambele permit alaturarea atributului –s, gcc permite adaugarea optiunii –fverbose –asm pentru a realiza asocierile intre variabilele si stivele folosite. De asemenea sintaxa Intel poate fi aleasa folosind –masm=Intel. Multiplele optiuni de optimizare in cazul gcc reprezinta inca un punct forte al flexibiliatii. Ele depind insa de versiunea de compilator gcc folosita. Alte setari de finete se pot realiza folosind atributul –f. Aici putem enumera :

-finline-functions = convertirea tuturor functiilor dintr-un fisier in macrouri si plasarea unor copii ale codurilor direct in functia de apelare. Functia este valabila doar pentru funtii apelate in acelasi fisier C ca sicel de definitie.

-fomit-frame-pointer = eliberarea unui registru sumplimentar pentru a putea fi folosit in programul nostru.

-funroll-loops = extinderea buclei pentru a dispune de multiple copii ale codului pentru multiple iteratii ale unei bucle. Daca aceasta optiune era des utilizata in arhitecurile cu procesoare mai vechi, procesoarele moderne o neglijeaza.

Pe de alta parte, PellesC se remarca printr-o flexibilitate redusa, el nepermitand controlul optimizarii la fel ca in cazul gcc. Totusi, dispune de optimizari predefinite in ceea ce priveste viteza sau spatiul.

3. Asamblarea

Aceasta etapa realizeaza traducerea intr-un limbaj cat mai apropiat de codul masina. Anumite aranjari ale instructiunilor sunt realizate

4. Link-editarea

Editarea legaturilor se poate executa in 3 maniere : statica, dinamica si runtime. In primul rand, maniera statica realizeaza asamblarea inclusa in fisierul executabil pentru fiecare functie apelata de programul cu pricina. Functiile se apeleaza direct prin adresa codului.

Legaturile realizate in maniera dinamica se refera la maparea librariilor folosind memoria virtuala in spatiul de adresa a programului cand acesta se executa. Librariile sunt intr-un singur loc din sistem.

Maniera runtime de realizare a legaturilor si de editare este folosita in momentul in care un program apeleaza o librarie care nu a fost inclusa la compilare. dlopen() (gcc) si GetProcAddress() (Windows) sunt locurile in care sunt mapate librariile. Rezultatul este trecut in dlsym() respective GetProcAddress() de unde vor fi apelate direct din program folosind pointeri.

3.2. Compilatorul GCC-Robitu Paul

Compilatorul GNU Compiler Collection este cel mai utilizat compilator pentru C/C++ (se foloseste de asemenea G++). Este un compilator open source, si desi a fost proiectat pentru C, au aparut totusi extensii pentru limbaje precum C++ sau Fortran.

Sintaxa este relativ simpla, formata din gcc <optiuni> <nume fisier>. Asadarm, pentru un fisier C putem executa comanda gcc –o nume_fisier numefisier.c. Numele executabilului este mentionat intre –o si nume_fisier.c . Comanda gcc de fapt executa mai multe operatii : preprocesarea cpp (includerea fisierelor header si expandarea macro-urilor), generarea codului obiect, editarea legaturilor si crearea programului executabil. Desigur, compilatorul ne ofera posibilitatea sa executam aceste operatii manual (folosind atribute precum –e –c sau –o) pentru un control mai bun (putem folosi de asemenea atributul –Wall pentru generarea unui numar maxim de avertismente.

In prezent, gcc a ajuns la versiunea 4.7.0 (Martie 2012) disponibila pe situl oficial gcc.gnu.org.

3.2.1. Exemple

1. GCC

Vom compila o sursa simpla de C. Programul va defini doua variabile si va calcula media aritmetica a lor. Fisierul sursa este medie_aritmetica.c.

```
paul@PaulLaptop:~/Desktop/SO$ ls -la
drwxr-xr-x 2 paul paul 4096 2012-05-05 19:17 .
drwxr-xr-x 6 paul paul 4096 2012-05-05 19:12 ..
-rw-r--r-- 1 paul paul  344 2012-05-05 19:11 medie_aritmetica.c
```

Putem afisa continutul fisierului folosind comanda cat din linux.

```
paul@PaulLaptop:~/Desktop/SO$ cat medie_aritmetica.c
#include <stdio.h>
int main() {
int a,b,medie;
printf("\n Acesta este un exemplu de program pentru compilarea folosind
gcc.\n");
printf("Programul va calcula media aritmetica a doua numere.\n");
a=10;
b=15;
medie = (a+b)/2;
printf("Rezultatul : %d \n",medie);
printf(" Apasati o tasta pentru a continua.\n ");
getchar();
}
```

In mod evident, este necesara includerea bibliotecii stdio.h . Vom trece in continuare la compilarea propriu-zis. Gcc ne ofera atat posibilitatea de a afisa programele implicate in compilare (atributul “-v”) , cat si cea de a nu realiza linkarea (atributul “-c”).

```
paul@PaulLaptop:~/Desktop/SO$ gcc -v -c medie_aritmetica.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/lto-wrapper
```

Target: x86_64-linux-gnu

Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.5.2-8ubuntu4' --with-bugurl=file:///usr/share/doc/gcc-4.5/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.5 --enable-shared --enable-multiarch --with-multiarch-defaults=x86_64-linux-gnu --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib/x86_64-linux-gnu --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.5 --libdir=/usr/lib/x86_64-linux-gnu --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-plugin --enable-gold --enable-ld=default --with-plugin-ld=ld.gold --enable-objc-gc --disable-werror --with-arch-32=i686 --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu

Thread model: posix
gcc version 4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4)
COLLECT_GCC_OPTIONS='-v' '-c' '-mtune=generic' '-march=x86-64'
 /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/cc1 -quiet -v medie_aritmetica.c -D_FORTIFY_SOURCE=2 -quiet -dumpbase medie_aritmetica.c -mtune=generic -march=x86-64 -auxbase medie_aritmetica -version -fstack-protector -o /tmp/cchJwn99.s
GNU C (Ubuntu/Linaro 4.5.2-8ubuntu4) version 4.5.2 (x86_64-linux-gnu)
	compiled by GNU C version 4.5.2, GMP version 4.3.2, MPFR version 3.0.0-p8, MPC version 0.9
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/../../../../../x86_64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/include
 /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.

```
GNU C (Ubuntu/Linaro 4.5.2-8ubuntu4) version 4.5.2 (x86_64-linux-gnu)
        compiled by GNU C version 4.5.2, GMP version 4.3.2, MPFR version
3.0.0-p8, MPC version 0.9
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-
heapsize=131072
Compiler executable checksum: 9755ab75799195519479ef699703b13b
COLLECT_GCC_OPTIONS='-v' '-c' '-mtune=generic' '-march=x86-64'
 as -V -Qy --64 -o medie_aritmetica.o /tmp/cchJwn99.s
GNU assembler version 2.21.0 (x86_64-linux-gnu) using BFD version (GNU
Binutils for Ubuntu) 2.21.0.20110327
COMPILER_PATH=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/../../../:/lib/:/usr/lib/:/usr/lib/x86_64-linux-gnu/
COLLECT_GCC_OPTIONS='-v' '-c' '-mtune=generic' '-march=x86-64'
```

Dupa acest pas, compilatorul a creat fisierul obiect cu extensia “.o”.

```
paul@PaulLaptop:~/Desktop/SO$ ls -la
total 16
drwxr-xr-x 2 paul paul 4096 2012-05-05 19:17 .
drwxr-xr-x 6 paul paul 4096 2012-05-05 19:49 ..
-rw-r--r-- 1 paul paul  344 2012-05-05 19:11 medie_aritmetica.c
-rw-r--r-- 1 paul paul 2088 2012-05-05 19:17 medie_aritmetica.o
paul@PaulLaptop:~/Desktop/SO$ gcc -v medie_aritmetica.o -o
medie_aritmetica
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/lto-wrapper
Target: x86_64-linux-gnu

Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.5.2-
8ubuntu4' --with-bugurl=file:///usr/share/doc/gcc-4.5/README.Bugs --
enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-
```

```
4.5 --enable-shared --enable-multiarch --with-multiarch-defaults=x86_64-
linux-gnu --enable-linker-build-id --with-system-zlib --
libexecdir=/usr/lib/x86_64-linux-gnu --without-included-gettext --enable-
threads=posix --with-gxx-include-dir=/usr/include/c++/4.5 --
libdir=/usr/lib/x86_64-linux-gnu --enable-nls --with-sysroot=/ --enable-
clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-
plugin --enable-gold --enable-ld=default --with-plugin-ld=ld.gold --enable-
objc-gc --disable-werror --with-arch-32=i686 --with-tune=generic --enable-
checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --
target=x86_64-linux-gnu
Thread model: posix
gcc version 4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4)
COMPILER_PATH=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/:/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-
gnu/4.5.2/../../../:/lib/:/usr/lib/:/usr/lib/x86_64-linux-gnu/

COLLECT_GCC_OPTIONS='-v' '-o' 'medie_aritmetica' '-mtune=generic' '-
march=x86-64'
 /usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/collect2 --build-id --
eh-frame-hdr -m elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-
x86-64.so.2 -z relro -o medie_aritmetica /usr/lib/x86_64-linux-
gnu/gcc/x86_64-linux-gnu/4.5.2/../../../crt1.o /usr/lib/x86_64-linux-
gnu/gcc/x86_64-linux-gnu/4.5.2/../../../crti.o /usr/lib/x86_64-linux-
gnu/gcc/x86_64-linux-gnu/4.5.2/crtbegin.o -L/usr/lib/x86_64-linux-
gnu/gcc/x86_64-linux-gnu/4.5.2 -L/usr/lib/x86_64-linux-gnu/gcc/x86_64-
linux-gnu/4.5.2/../../.. -L/usr/lib/x86_64-linux-gnu medie_aritmetica.o -lgcc --
as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/crtend.o
/usr/lib/x86_64-linux-gnu/gcc/x86_64-linux-gnu/4.5.2/../../../crtn.o


paul@PaulLaptop:~/Desktop/SO$ ls -la
total 28
drwxr-xr-x 2 paul paul 4096 2012-05-05 20:07 .
```

```
drwxr-xr-x 6 paul paul 4096 2012-05-05 19:49 ..
-rwxr-xr-x 1 paul paul 8541 2012-05-05 20:02 medie_aritmetica
-rw-r--r-- 1 paul paul  324 2012-05-05 20:01 medie_aritmetica.c
-rw-r--r-- 1 paul paul 2000 2012-05-05 20:01 medie_aritmetica.o
paul@PaulLaptop:~/Desktop/SO$ ./medie_aritmetica
```

Acesta este un exemplu de program pentru compilarea folosind gcc.

Programul va calcula media aritmetica a doua numere.

```
Rezultatul : 12
Apasati o tasta pentru a continua.
```

Am afisat rezultatul trunchiat folosind “%d”.

2. Pelles C

Intrucat acest compilator (similar cu CL.EXE) este adaptat pentru sistemele pe 64 de biti, ne vom rezuma studiul la un exemplu simplu. Vom compila un program C folosind compilatorul Pelles C. Programul va afisa in consola un mesaj. Codul sursa este :

```
#include <stdio.h>
int main(int argc,char **argv) {
  printf("Acesta este un program compilat cu Pelles C");
}
```

Trecem la compilare :

```
C:\Program Files\PellesC>cc /Ze program.c
```

Observam ca sunt generate fisierele program.exe, program.ppx, program.obj, program.tag. Pentru a rula aplicatia tastam program in command prompt. Acum va aparea “este un program compilat cu Pelles C“.

Bibliografie:

1.”Compilers-Principles,Techniques&Tools”,Second Edition,Alfred V.Aho,Monica S.Lam,Ravi Sethi
2.”Modern Operanting System”,Second Edition,Tanenbaum
3. “The Theory and Practice of Compiler Writing”, J. Tremblay, P. Sorenson
4. “ Compiler Design and Construction”, A. Pyster
5. http://www.wikipedia.org
6.http://facultate.regielive.ro/download-9977.html
7.”BNF and EBNF:What are they and how do they work?”-L.M.Garshol