Cuprins

Nichifor Andrei

Achiriloaie Constantin

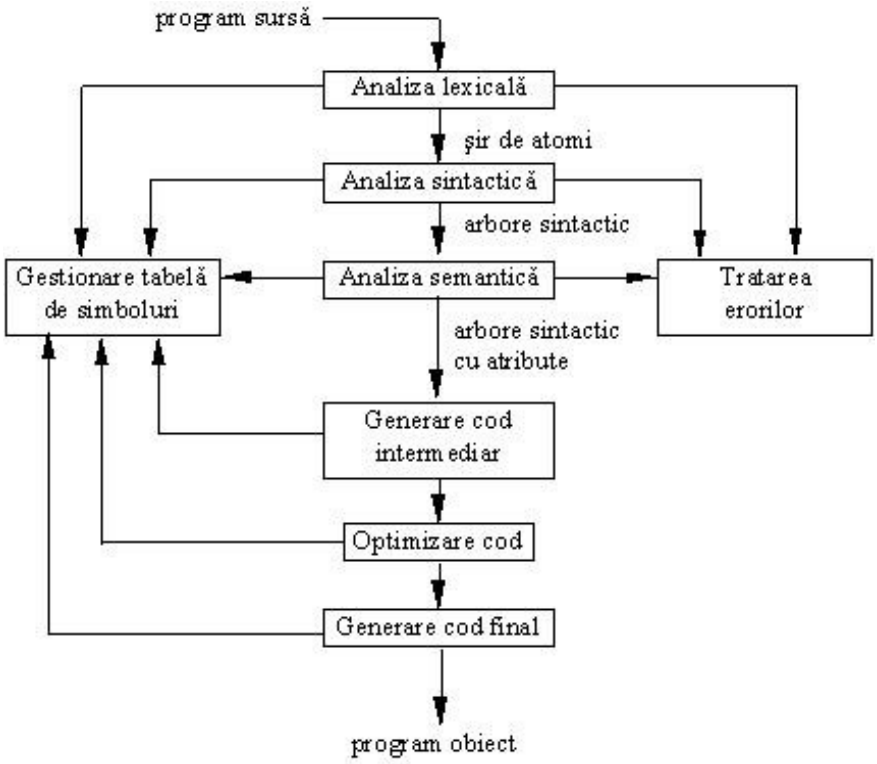
Sisu Liviu

Mistrapau Bogdan

1.Concepte Generale

Un translator este un program care primeste la intrare un text scris intr-un limbaj de programare,numit limbaj sursa si produce la iesire un text echivalent scris in alt limbaj de programare,intitulat limbaj obiect.
Daca limbajul sursa este un limbaj de nivel inalt, iar limbajul obiect este un limbaj de nivel inferior (limbaj de asamblare sau cod masina), atunci translatorul respectiv se numeste compilator.

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:



Analiza lexicala**:** textul sursa este preluat sub forma unei secvente de caractere care sunt grupate apoi in entitati numite atomi. Atomilor li se atribuie coduri lexicale, astfel ca , la iesirea acestei faze, programul sursa apare ca o secventa de asemenea coduri. Exemple de atomi: cuvinte cheie, identificatori, constante numerice, semne de punctuatie etc.

Analiza sintactica**:** are ca scop gruparea atomilor rezultati in urma analizei lexicale in structuri sintactice. O structura sintactica poate fi vazuta ca un arbore ale carui noduri terminale reprezinta atomi, în timp ce nodurile interioare reprezinta siruri de atomi care formeaza o entitate logica . Exemple de structuri sintactice: expresii, instructiuni, declaratii etc.

Generarea de cod intermediar**:** in aceasta faza are loc transformarea arborelui sintactic intr-o secventa de instructiuni simple, similare macroinstructiunilor unui limbaj de asamblare. Diferenta dintre codul intermediar si un limbaj de asamblare este in principal aceea ca, in codul intermediar nu se specifica registrele utilizate in operatii. Exemple de reprezentari pentru codul intermediar: notatia postfix, instructiunile cu trei adrese etc. Codul intermediar prezinta avantajul de a fi mai usor de optimizat decat codul masina .

Optimizarea de cod**:** este o faza optionala , al carei rol este modificarea unor portiuni din codul intermediar generat, astfel incat programul rezultat sa satisfaca anumite criterii de performanta vizand timpul de executie si/sau spatiul de memorie ocupat.

Generarea codului final**:** presupune transformarea instructiunilor codului intermediar (eventual optimizat) în instructiuni masina (sau de asamblare) pentru calculatorul tinta (cel pe care se va executa programul compilat).
In afara de actiunile enumerate mai sus, procesul de compilare mai include urmatoarele:
Gestionarea tabelei de simboluri: tabela de simboluri (TS) este o structura de date destinata pastrarii de informatii despre simbolurile (numele) care apar in programul sursa; compilatorul face referire la aceasta tabela aproape in toate fazele compilarii.

Tratarea erorilor**:** un compilator trebuie sa fie capabil sa recunoasca anumite categorii de erori care pot sa apara in programul sursa; tratarea unei erori presupune detectarea ei, emiterea unui mesaj corespunzator si revenirea din eroare, adica, pe cat posibil, continuarea procesului de compilare pana la epuizarea textului sursa, astfel incat numarul de compilari necesare eliminarii tuturor erorilor dintr-un program sa fie cat mai mic. Practic, exista erori specifice fiecarei faze de compilare.

1.1 Derularea procesului de compilare

Fazele unui proces de compilare se pot înlantui, în principiu, în doua moduri:

- ☐ La iesirea fiecarei faze se va genera un fisier intermediar continand forma de reprezentare a programului sursa rezultata in faza respectiva, fisier care va constitui intrare pentru faza urmatoare. In acest caz, in fiecare faza va avea loc cel putin o parcurgere a programului sursa, de la inceput la sfarsit. O asemenea parcurgere se numeste trecere.

- ☐ Doua sau mai multe faze de compilare se interclaseaza astfel incat ele sa se execute printr-o singura trecere.

Aplicarea uneia sau alteia dintre cele doua modalitati depinde de natura limbajului compilat, precum si de mediul în care urmeaza sa ruleze compilatorul.

1.2 Clasificare

Putem face o clasificare a compilatoarelor in functie de platforma pe care se va putea executa codul produs de acestea.

Astfel compilatoarele native, care ruleaza pe un anumit tip de calculator si pe un anumit sistem de operare, vor genera un cod care va rula pe acelasi tip de calculator si pe acelasi sistem de operare. Compilatoarele incrucisate(cross) vor rula pe o anumita platforma, dar vor produce cod obiect pentru alt tip de calculator.

Acestea sunt folosite pentru a genera programe care vor rula pe calculatoare cu o noua arhitectura sau pe dispozitive speciale care nu pot gazdui propriile lor compilatoare.

Iesirea unor compilatoare poate viza hardware-ul la un nivel foarte jos(ASIC-circuit integrat pentru o anumita aplicatie). Astfel de compilatoare sunt denumite

compilatoare hardware sau elemente de sinteza deoarece programele pe care le compileaza controleaza configuratia finala a hardware-ului si modul in care acesta functioneaza. In acest caz nu vom avea instructiuni care sa se execute in ordine, ci doar interconectare de tranzistoare si tabele de cautare.

Un compilator pentu un limbaj relativ simplu scris de o anumita persoana poate fi un simplu program monolitic. In momentul in care limbajul sursa este unul complex, iar rezultatul dorit este unul de calitate ridicata, proiectarea poate fi impartita in mai multe faze sau pasi. Avand aceste faze separate, vom putea imparti dezvoltarea proiectului in parti mai mici, care vor putea fi dezvoltate de diversi programatori. Modularitatea permite inlocuirea unei anumite faze cu una mai noua sau inserarea de noi pasi.

Divizarea procesului de compilare in faze a introdus urmatorii termeni:prima parte, partea de mijloc si partea finala.

Chiar si cele mai mici compilatoare au mai mult de doua faze, care vor apartine primei si ultimei parti desi nu putem face o impartire exacta a acestor doua parti. Partea din fata este in general considerata a fi partea in care se desfasoara analiza sintactica si cea semantica, impreuna cu transformarea la un nivel mai jos de reprezentare.

Partea centrala este proiectata pentru a realiza optimizarea intr-o forma diferita de cod sursa sau cod masina.

Independenta cod sursa/cod masina are rolul de a imparti optimizarea intre diferite versiuni ale compilatorului.

Partea finala preia iesirea partii din mijloc si poate face mai multe analize, transformari si optimizari pentru un anumit tip de calculator. Apoi el va genera cod pentru un procesor particular sau pentru un anumit sistem de operare.

O clasificare a compilatoarelor in functie de numarul de pasi isi are fondul in resursele limitate ale calculatoarelor. Compilarea inseamna efectuarea unui munci intense, iar la inceput calculatoarele nici nu aveau destula memorie pentru a contine un program care sa faca aceasta treaba. De aceea compilatoarele au fost impartite in programe

mai mici, fiecare executand un anumit pas asupra sursei si realizand o parte din analiza si transformarea necesara.

Abilitatea de a compila intr-un singur pas este de multe ori vazuta ca un beneficiu deoarece simplifica munca scrierii acestuia si este in acelasi timp o operatie mai rapida decat compilarea in mai multi pasi. Cateva limbaje au fost proiectate pentru a se putea compila intr-un singur pas(Pascal).

In unele cazuri vom avea nevoie de un compilator care sa efectueze mai multi pasi asupra sursei. Dezavantajul compilarii intr-un singur pas este reprezentat de faptul ca nu putem efectua multe din optimizarile sofisticate necesare pentru a genera cod de inalta calitate. Poate fi dificil sa numaram exact cati pasi va efectua un compilator optimizat. De exemplu, diferite faze de optimizare pot analiza o expresie de mai multe ori in timp ce o alta expresie va fi analizata doar o singura data.

Impartirea compilatorului in mai multe parti ofera o buna posibilitate pentru depanarea acestuia, deoarece corectarea mai multor programe mai mici e mult mai simpla decat corectarea unuia foarte complex.

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

2.1 Elemente generale: propozitii, instructiuni, lexeme, token

Limbajele, formal vorbind, sunt multimi de şiruri de caractere dintr-un alfabet fixat.

Acest lucru este valabil atât pentru limbajele naturale, cât şi pentru cele artificiale. Şirurile de caractere ale limbajului se numesc *propozitii* (*sentences*) sau *instructiuni*, *afirmatii* (*statements*). Orice limbaj are un numar de reguli sintactice care stabilesc care din şirurile de caractere ce se pot forma cu simboluri din alfabet fac parte din limbaj. Daca pentru limbajele naturale regulile sintactice sunt in general complicate, limbajele de programare sunt relativ simple din punct de vedere sintactic. Exista in

fiecare limbaj de programare unitatile sintactice de nivelul cel mai mic – numite *lexeme* – care nu sunt cuprinse in descrierea formala a sintaxei limbajului.

Aceste unitati sunt cuprinse intr-o specificare lexicala a limbajului care precede descrierea sintaxei. Lexemele unui limbaj de programare cuprind identificatorii, literalii, operatorii, cuvintele rezervate, semnele de punctuatie. Un program poate fi privit ca un şir de lexeme şi nu un şir de caractere; obtinerea şirului de lexeme din şirul de caractere (programul obiect) este sarcina *analizorului lexical*. Un *token* al unui limbaj este o categorie de lexeme.

De exemplu, un identificator este un token care are ca instante lexeme ca: suma, total, i, medie, etc.

Tokenul PLUS pentru operatorul aritmetic "+" are o singura instanta.

Instructiunea:

suma = a + b

este compusa din lexemii şi tokenurile specificate in tabelul urmator:

LEXEM	**TOKEN**
suma	IDENTIFICATOR
=	ASIGNARE
a	IDENTIFICATOR
+	PLUS
b	IDENTIFICATOR

2.2 Descrierea BNF

Un limbaj de programare poate fi descris, formal, prin aşa–zisele mecanisme de generare a şirurilor din limbaj. Aceste mecanisme se numesc gramatici şi au fost descrise de Chomsky in anii 1956 – 1959. Nu mult dupa ce Chomsky a descoperit cele patru clase de limbaje formale, grupul ACM – GAMM a proiectat limbajul Algol 58 iar la o conferinta internationala John Backus a prezentat din partea grupului acest limbaj folosind o metoda formala noua de descriere a sintaxei.

Aceasta noua notatie a fost modificata de Peter Naur la descrierea limbajului Algol 60. Metoda de descriere a sintaxei limbajelor de programare aşa cum a fost folosita de Naur poarta numele de **forma Backus - Naur** sau simplu **BNF**.

De remarcat faptul ca BNF este aproape identica cu mecanismul generativ al lui Chomsky – gramatica independenta de context. BNF este un metalimbaj.

Acesta foloseşte abstractiile pentru a descrie structurilem sintactice.

O abstractie este scrisa intre paranteze unghiulare, ca de exemplu:

<asignare>, <variabila>, <expresie>, <if_stm>, <while_stm> etc.

Definitia unei abstractii este data printr-o *regula* sau *productie* care este formata din:

- partea stânga a regulii ce contine abstractia care se defineşte.
- o sageata → (sau caracterele ::=) care desparte partea stânga a regulii de partea

  dreapta.
- partea dreapta a regulii care este un şir format din abstractii, tokenuri, lexeme.

De exemplu, regula:

<asignare> → <variabila> = <expresie>

defineşte sintaxa unei asignari: abstractia <asignare> este o instanta a abstractiei <variabila> urmata de lexemul = urmat de o instanta a abstractiei <expresie>.

O propozitie care are structura sintactica descrisa de aceasta regula este:

total = suma1 + suma2

Intr-o regula BNF abstractiile se numesc ***simboluri neterminale*** sau simplu ***neterminali***, lexemii şi token-urile se numesc ***simboluri terminale*** sau simplu ***terminali***.

O descriere BNF sau o gramatica (independenta de context) este o colectie de reguli.

Daca se intâmpla ca un neterminal sa aiba mai multe definitii acestea se pot insera intr-o singura regula in care partea dreapta contine partile drepte ale definitiilor sale despartite

prin simbolul |.

2.3 Exemplu descriere BNF:

<S> ::= '-' <FN> | <FN>

<FN >::= <DL> |<DL >'.' <DL>

<DL>: := <D> | <D>< DL>

<D> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9

Succesiunea valida de simboluri in limbajul descris astfel apartine unei multimi de numere , posibil negative si/sau posibil fractionare

Pentru a produce un numar se incepe cu simbolul de start S:

<S>

Folosind regulile BNF corespunzatoare simbolul S este inlocuit.

In cazul acesta alegem un numar pozitiv, deci inlocuim cu <FN>:

<FN>

Urmatorul pas este inlocuirea lui <FN> folosind regulile BNF.

Alegem un numar fractionar. Algoritmul de inlocuire este repetat pana cand se ajunge la rezultatul dorit(iteratiile sunt prezentate mai jos):

<DL>.<DL>

<D>.<DL>

2. <DL>

2. <D> <DL

2. <D> <D>

2. 3 <D>

2.3 4

2.4 EBNF – o forma usor extinsa a BNF

BNF a fost extinsa in diverse moduri mai ales din ratiuni de implementare a unui generator de analizor sintactic. Orice extensie a BNF este numita EBNF (Extended BNF).

Extensiile BNF nu maresc puterea descriptiva a mecanismului ci reprezinta facilitati

privind citirea sau scrierea unitatilor sintactice.

Extensiile ce apar frecvent sunt:

- includerea, in partea dreapta a unei reguli, a unei parti optionale ce se scrie intre

  paranteze drepte. De exemplu, instructiunea if ar putea fi descrisa astfel:

  <if_stm> → if <expr_logica> then <stm> [else <stm>]

- includerea in partea dreapta a unei reguli a unei parti ce se repeta de zero sau

  mai multe ori. Aceasta parte se scrie intre acolade şi poate fi folosita pentru a

  descrie recursia:

  <lista_de_instructiuni> → <instructiune> {; <instructiune>}

- includerea in partea dreapta a unei reguli a optiunii de a alege o varianta dintr-un grup. Acest lucru se face punând variante in paranteze rotunde, despartite prin |. De exemplu, o instructiune for se poate defini astfel:

<for_stm> → for <var> = <expr> (to | downto) <expr> do <stm>

Trebuie precizat ca parantezele de orice fel din descrierea EBNF fac parte din metalimbaj, ele nu sunt terminali ci doar instrumente de a nota o anume conventie. In cazul in care unele din aceste simboluri sunt simboluri terminale in limbajul descris, acestea vor fi puse intre apostrof : '[' sau '{' etc.

EBNF nu face altceva decât sa simplifice intr-un anume fel definitiile sintactice.

2.5 Exemplu EBNF:

In cazul prezentat mai sus pentru generarea rezultatului am fost nevoiti sa folosim un algoritm recursiv. Acesta face BNF-ul mai greu de citit si inteles.

<S> := '-'? <D>+ ('.' <D>+)?

<D> := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Folosind o descrire extinsa BNF gramtica(BNF) prezentata mai sus devine mai accesibila. Orice converisie EBNF – BNF este posibila – forma extinsa este doar mai usor de folosit, toti operatorii putant fi descrisi cu ajutorul BNF.

3. Analiza lexicala si semantica

3.1 Analiza lexicala

Un compilator este un program complex care realizeaza traducerea unui program sursa intr-un program obiect. De obicei programul sursa este scris intr-un limbaj de nivel superior celui in care este scris programul obiect.

Fazele unui compilator sunt descrise in figura de mai jos :

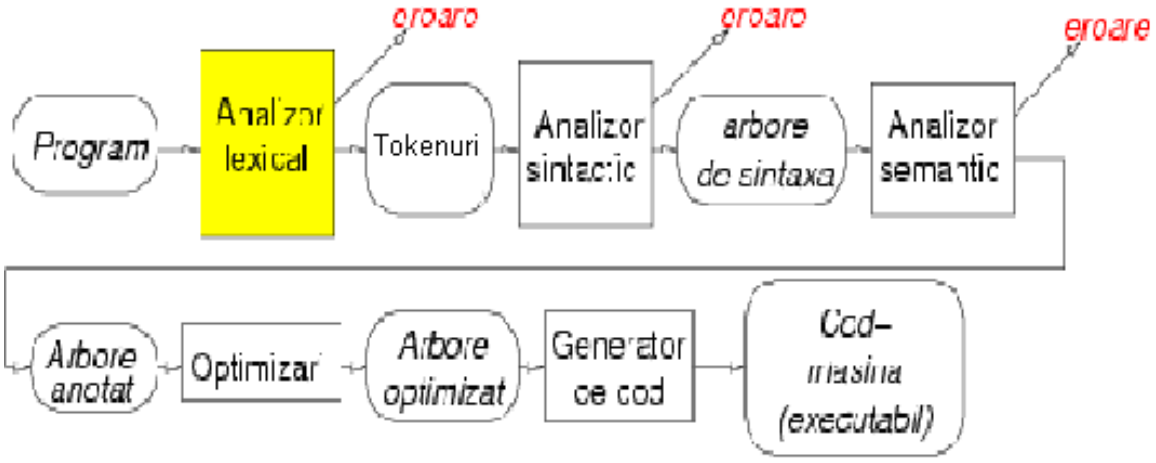


Figura nr 1[1]

Analiza lexicala realizeaza traducerea textului programului intr-o forma mai usor de prelucrat de catre celelalte componente. Analizorul lexical(scaner-ul) considera textul primit la intrare ca fiind format din unitati lexicale(simboluri de baza) pe care

[1] http://www.cs.cmu.edu/~mihaib/articole/regex/regex-html.html

le recunoaste producand atomi lexicali (token-uri). Un atom lexical poate sa fie de exemplu, un cuvant cheie al limbajului (for, while, etc) dar si un numar sau un nume. Nu exista o corespondenta biunivoca intre sirurile de intrare si atomii lexicali. De exemplu, daca cineva introduce la intrare sirul "23 + 44 * 12", analizorul lexical preia caracterele '2', '3', '+' samd unul cate unul, ignora spatiile si furnizeaza la iesire un sir de elemente constand din "23", "+", "44", "*" si "12"

Daca pentru atomul lexical corespunzator cuvantului cheie **case** exista un singur sir de intrare, pentru atomul lexical corespunzator unui numar intreg pot sa existe foarte multe siruri de intrare.

Un atom lexical(token) este reprezentat printr-un cod numeric care specifica clasa acestuia si o serie de atribute care sunt specifice fiecarei clase. Astfel, poate sa existe clasa operatorilor relationali pentru care un atribut trebuie sa se specifice tipul concret al operatorului. Tipul atomului lexical este necesar pentru analiza sintactica in timp ce valoarea atributului este semnificativa pentru analiza semantica si generarea de cod. Pentru un atom lexical de tip numar atributele vor descrie tipul numarului si valoarea acestuia.

Una dintre deciziile ce trebuie luate la inceputul proiectarii unui compilator consta din stabilirea atomilor lexicali. De exemplu, se pune problema daca sa existe cate un atom lexical pentru fiecare operator de comparatie (<, <=, >, >=) sau sa existe un unic atom lexical - corespunzator operatiei de comparatie. In primul caz generarea de cod poate sa fie mai simpla. Pe de alta parte existenta unui numar mare de atomi lexicali poate complica in mod exagerat analiza sintactica. In general, operatorii care au aceeasi prioritate si asociativitate pot sa fie grupati impreuna. Prioritatea operatorilor reprezinta ordinea in care acestia se aplica asupra unei valori. Asociativitatea indica ordinea de aplicare a operatorilor : de la stanga la dreapta sau de la dreapta la stânga.

Un scaner apare in general ca o functie care interactioneaza cu restul compilatorului printr-o interfata simpla : ori de cate ori analizorul sintactic(parser-ul) are nevoie de un nou atom lexical va apela scaner-ul care ii va da atomul lexical urmator

Sa luam un exemplu :

1.identifier→ letter(letter|digit) .

2.letter→ ‘A’| … | ‘Z’.

3.digit→’0’| … | ‘9’.

Scanerul sau analizatorul lexical functioneaza pe principiul automatelor finite.

Din nefericire, aceste automate presupun cunoasterea sfarsitului sirului.Sarcina unui analizor lexical este sa extraga urmatorul simbol de baza din textul de intrare, determinand sfarsitul simbolului in timpul cautarii. Deci, automatul finit rezolva doar partial problema analizei lexicale. Pentru a mari eficienta analizorului lexical, vom folosi, din multimea automatelor care accepta limbajul dat, automatul cu cele mai putine stari.

Delimitatorii(cuvintele cheie, caracterele speciale precum si combinatiile de caractere speciale) impreuna cu indetificatorii si constantele reprezinta simbilorile de baza ale unui program sursa.Pentru a clasifica simbolurile de baza, vom imparti starile finale ale automatelor in clase.

Reprezentarea textuala a constantelor si a sirurilor utilizata pentru a interoga tabela de simboluri este obtinuta din sirul de intrare. Din aceste motive, automatul este extins la un traducator cu stari finite care emite un caracter la fiecare tranzitie de stare.Din punct de vedere al teoriei combinationale, acest translator este un caz particular de masina Mealy. Caracterele de iesire sunt colectate impreuna intr-un sir de caractere, care formeaza aparitia atomului lexical.

Caracterul special punct si virgula este pentru limbajul Pascal un simbol de baza, nefiind inceputul unui alt simbol de baza.In momentul in care un analizor Pascal ajunge in starea finala corespunzatoare acestuia, se poate opri si poate accepta simbolul. In acest caz s-a depistat sfarsitul sirului iar referinta de la intrare este pozitionata pe urmatorul caracter.

Caracterul doua puncte este de asemenea un simbol de baza Pascal, dar este si inceputul simbolului :=. Din acest motiv analizorul lexical Pascal trebuie sa

avanseze dincolo de caracterul : pentru a vedea ce urmeaza.

Un exemplu de automat finit pentru exemplul de mai sus :

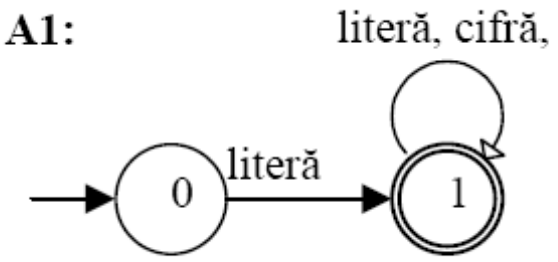


Figura nr 2[2]

Cuvintele cheie : Cea mai simpla modalitate de scriere a cuvintelor cheie este cea realizata prin intermediul cuvintelor rezervate. Cuvintele rezervate sunt identificatori pe care utilizatorul nu-i poate folosi in alt scop. Aceasta abordare necesita scrierea identificatorilor fara goluri, pentru a se folosi spatiile ca separatori, intre identificatori sau intre un identificator si o constanta. In interiorul numerelor pot aparea litere si de aceea nu trebuie sa fie separate de partea cealalta a numarului prin spatii. Avantajul principal al acestei reprezentari este claritatea, precum si posibilitatea redusa de erori de scriere. Dezavantajul principal este acela ca programatorul poate uita unele dintre cuvintele cheie si sa se foloseasca drept identificatori utilizator.

Cele mai frecvent folosite reguli de scriere, in acest mod a cuvintelor cheie sunt:

- sublinierea cuvintelor cheie;
- delimitarea la stanga si la dreapta cu ajutorul unor caractere speciale;
- prefixarea cuvintelor cheie cu un caracter special, sfarsitul fiind marcat de primul spatiu sau de primul caracter diferit de o litera sau o cifra;

[2] A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988

- scrierea cuvintelor cheie cu ajutorul literelor mari iar a identificatorilor cu litere mici sau viceversa.

Tratarea erorilor

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

- apariia unui caracter ilegal;
- nerespectarea regulilor gramaticale.

In cazul nostru, aparitia unui caracter ilegal determina eliminarea acestuia, analizorul lexical intrerupand verificarea atomului lexical. In acest caz, analizorul lexical va semnala eroarea, dar va furniza analizorului sintactic un rezultat posibil. Nerespectarea regulilor gramaticale duce la intreruperea analizei atomului lexical curent, recuperarea ultimei stari finale parcurse, a contextului analizei din acel moment si construirea atomului lexical curent.

Iata in cele din urma exemplul[3] unui analizor lexical ce are ca rol recunoasterea unui caracter alfanumeric sau a unei cifre :

```
function IsAlNum(c: char): boolean;
begin
IsAlNum := IsAlpha(c) or IsDigit(c);
end;

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

function IsDigit(c: char): boolean;
```

[3] J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985..

```
begin
IsDigit := c in ['0'..'9'];
end;
```

3.2 Analiza semantica

Aceasta faza este de obicei incorporata in faza de analiza sintactica. Rolul acestei faze consta din verificarea din punct de vedere semantic a structurilor recunoscute drept corecte din punct de vedere sintactic. Majoritatea verificarilor realizate de catre aceasta faza se refera la tipurile constructiilor din cuvinte. Pentru ca un program sa fie corect din punct de vedere semantic, toate variabilele, functiile, clasele, etc. trebuie sa fie definite corespunzator, expresiile si variabilele sa fie folosite astfel incat sa se incadreze in tipul in care au fost definite.

O mare parte a analizei semantice o reprezinta verificarea variabilelor/functiilor si a tipului acestora. In unele limbaje de programare, identificatorii trebuie declarati inainte de a fi folositi. In momentul in care compilatorul intalneste o noua declarare, inregistreaza tipul datei asociate acestuia. Apoi, pe parcursul examinarii programului, verifica daca tipul identificatorului este respectat in termenii operatiei care se efectueaza. De exemplu tipul expresiei din partea dreapta a unei expresii de atribuire trebuie sa fie acelasi cu tipul expresiei din partea stanga . De asemenea, identificatorul din partea stanga trebuie sa fie declarat corect. Parametrii unei functii trebuie sa corespunda cu argumentele unui apel al functiei in numar si tip. Limbajul de programare poate impune ca identificatorii sa fie unici, asadar interzicerea a doua declaratii globale sa aiba acelasi nume. Operanzii aritmetici vor trebui sa fie numerici, de acelasi tip. Acestea au fost exemple de conditii verificate de analizorul semantic.

Analiza semantica poate fi inclusa in analiza sintactica, de exemplu, pentru o operatie de adunare, parser-ul poate verifica daca operanzii sunt de tip numeric si compatibili pentru aceasta operatie.

Cea mai consistenta parte a analizei semantice o constituie **verificarea tipului-** orice operand din orice expresie respecta tipul cu care a fost definit.Verificarea tipului poate fi facuta in timpul compilarii, in timpul executiei, sau impartita de ambele procese.Verificarea statica e facuta in timpul compilarii. Informatia folosita de analizorul static e obtinuta in urma declaratiilor si stocata intr-un tabel de simboluri. Dupa colectarea informatiei, tipurile implicate in fiecare operatie sunt verificate. E destul de dificil pentru un limbaj de programare care face numai verificare statica sa corecteze toate erorile. Chiar si vechiul Pascal nu putea gasi toate erorile de tip in timpul compilarii, deoarece multe dintre acestea nu pot fi corectate de analizorul de tip. De exemplu, daca a si b sunt de tipul int, si le initializam cu valori mari, a*b poate depasi limita maxima a intS. Aceste tipuri de erori nu pot fi de obicei detectate in timpul compilarii.[4]

Analiza dinamica presupune includerea tipului pentru fiecare locatie a datelor in timpul rularii. De exemplu, o variabila de tipul "double" va contine impreuna cu valoarea actuala si o eticheta ce indica "double type". Executia oricarei operatii incepe cu verificarea acestor etichete de tip. Cand o operatie de adunare este apelata, se va verifica in prealabil tipul variabilelor folosite si daca sunt compatibile.Analiza dinamica a tipului da o performanta mai scazuta timpului de rulare, dar poate raporta erori care nu se pot detecta in timpul compilarii. În principiu, orice verificare de tip poate fi facuta dinamic, daca in codul obiect se pastreaza tipul unui element impreună cu valoarea acelui element.

Conversia automata a tipului apare cand compilatorul detecteaza o eroare de tip si o corecteaza automat la tipul corespunzator. In cazul limbajului C, adunarea unui integer cu o variabila float va genera o eroare, pe care analizorul va incerca sa o elimine prin inserarea unei operatii de conversie in codul compilat. Daca nu se poate efectua nici o conversie, atunci se va raporta o eroare de compilare. In limbajul C++ conversiile automate pot genera un timp de compilare mare si de asemenea pot aparea rezultate diferite de realitate.

[4] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA:

Addison-Wesley, 1986.

4. Arborele sintactic

Rolul analizei sintactice(parser) consta in transformarea sirului de atomi lexicali intr-o descriere structurala a acestora(arborele sintactic), verificand totodata daca sirul este corect sau nu. Descrierea structurala a sirului de atomi lexicali reprezinta de fapt un echivalent al arborelui de derivare.

Analizorul sintactic poate realiza o construire explicita a arborelui sintactic, caz in care acesta este accesibil la sfarsitul analizei sintactice, sau poate declansa anumite actiuni asupra bazei de date prin intermediul unor proceduri semantice.

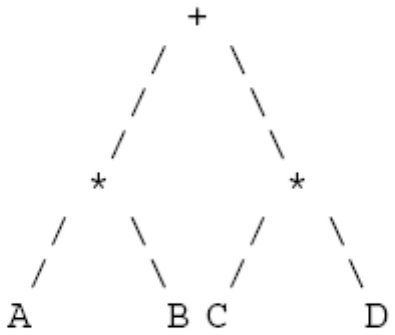
In principiu exista doua strategii de construire a arborelui sintactic:strategia descendenta in care arborele este construit de la radacina spre nodurile terminal si strategia ascendenta in care arborele este construit de la nodurile terminale spre radacina[5].

Sa consideram de exemplu expresia :

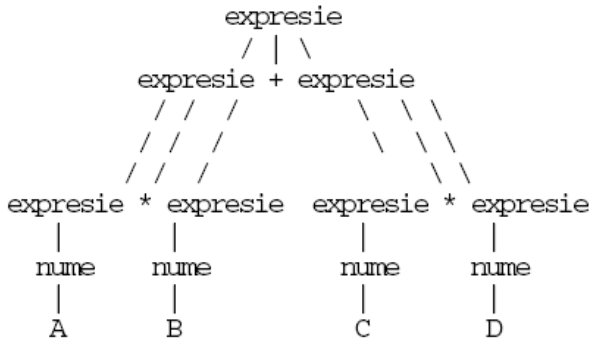
A * B + C * D

Aceasta expresie poate sa fie descrisa de urmatorul tip de arbore numit arbore sintactic:

[5] Addison Wesley - Compiler Design - Formal Syntax And Semant



In acest arbore au fost evidentiate relatiile (din punctul de vedere al modului de evaluare) intre componentele expresiei. Daca se doreste insa sa se evidentieze structura expresiei din punctul de vedere al unitatilor sintactice din care este formata, atunci se va utiliza pentru reprezentarea expresiei un arbore de derivare (parse tree). Pentru exemplul considerat un arbore de derivare ar putea sa fie de forma:



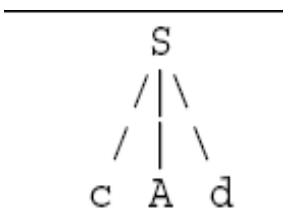
4.1 Analiza sintactica top - down[6]

Analiza sintactica top-down poate sa fie interpretata ca operatia de construire a arborilor de derivare pornind de la radacina si adaugand subarbori de derivare intr-o ordine prestabilita.

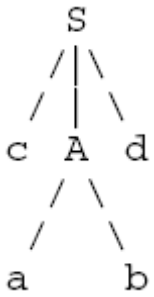
[6]Irina Athanasiu – Limbaje formale si automate

Sa consideram de exemplu gramatica S □□cAd, A□□ab | a si sirul de intrare cad. Construirea

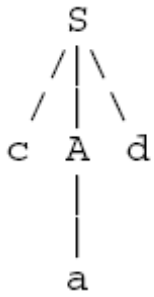
arborelui de derivare pentru acest sir porneste de la simbolul de start al gramaticii S cu capul de citire pe sirul de intrare pozitionat pe caracterul c. Pentru S se utilizeaza productia S -> cAd si se obtine arborele de derivare :



In acest moment se observa ca frunza cea mai din stanga corespunde cu primul caracter din sirul de intrare. In acest caz se avanseaza cu o pozitie pe sirul de intrare si in arborele de derivare. Frunza curenta din arborele de derivare este acum A si se poate aplica una dintre productiile corespunzatoare acestui neterminal :



Se poate avansa pe sirul de intrare si in arbore deoarece avem din nou coincidenta simboliilor terminali. In acest moment se ajunge la compararea simbolului d din sirul de intrare cu simbolul b din arbore, corespunzator trebuie sa se revina cautandu-se o noua varianta pentru A. Rezulta ca capul de citire trebuie sa fie readus In pozitia simbolului a. Daca acum se utilizeaza productia A->a se obtine arborele :



In continuare se va ajunge la acceptarea sirului de intrare.Din cele prezentate anterior rezulta doua observatii :

- In general implementarea presupune utilizarea unor tehnici cu revenire (backtracking);
- daca gramatica este recursiva stanga algoritmul poate conduce la aparitia unui ciclu infinit.

4.2 Analiza sintactica buttom-up

Analiza sintactica de tip bottom-up încearca sa construiasca un arbore de derivare pentru un sir de intrare dat pornind de la frunze spre radacina aplicand metoda "handle pruning". Adica trebuie sa se construiasca în stiva partea dreapta a unei productii. Selectia productiei construite se face pe baza "începuturilor" care reprezinta începuturi posibile de siruri derivate conform productiei respective.

4.2.1 Analiza sintactica de tip LR(k)

LR(k) este o metoda de analiza ascendenta al carui nume are urmatoarea semnificatie : primul L indica sensul parcurgerii - stânga, dreapta, litera R indica

faptul ca derivarea este dreapta iar k indica numarul de atomi lexicali necesari pentru o alegere corecta a unui început stiva. Avantajele metodei LR(k) sunt :

- se pot construi analizoare de tip LR pentru toate constructiile din limbajele de programare care pot sa fie descrise de gramatici independente de context;
- clasa limbajelor care pot sa fie analizate descendent în mod determinist este o submultime proprie a
  limbajelor care pot sa fie analizate prin tehnici LR;
- detectarea erorilor sintactice se face foarte aproape de atomul lexical în legatura cu care a aparut eroarea.

Dezavantajul major al acestei metode - volumul mare de calcul necesar pentru implementarea unui astfel de analizor fara aportul unui generator automat de analizoare sintactice.

4.2.2.Analiza sintactica predictiva (descendent recursiva)

Dezavantajul abordarii anterioare consta în necesitatea revenirilor pe sirul de intrare ceea ce conduce la complicarea deosebita a algoritmilor si mai ales a structurilor de date implicate deoarece automatul cu stiva corespunzator cazului general trebuie sa fie nedeterminist. Exista însa gramatici pentru care daca se utilizeaza transformari, prin eliminarea recursivitatii stanga si prin factorizare seobtin gramatici care pot sa fie utilizate pentru analiza top-down fara reveniri. În acest caz dandu-se un simbol de intrare curent si un neterminal exista o singura alternativa de productie prin care din neterminalul respectiv sa se deriveze un sir care începe cu simbolul de intrare care urmeaza.

Pentru proiectarea analizoarelor predictive se utilizeaza diagrame de tranzitie. O diagrama de tranzitie este un graf care prezinta productiile corespunzatoare unui neterminal. Etichetele arcelor reprezinta atomi lexicali sau simboli neterminali. Fiecare arc etichetat cu un atom lexical indica tranzitia care urmeaza sa se execute daca se

recunoaste la intrare atomul lexical respectiv. Pentru arcele corespunzatoare neterminalelor se vor activa procedurile corespunzatoare neterminalelor respective. Pentru a construi o diagrama de tranzitie pentru un neterminal A într-o gramatica în care nu exista recursivitate stanga si în care s-a facut factorizare stanga se procedeaza în modul urmator :

1. se creaza doua stari: initiala si finala;

2. pentru fiecare productie A ->X1 X2 ... Xn se va crea o cale formata din arce între starea initiala si starea finala cu etichetele X1 X2 ... Xn.

Rolul analizei lexicale este foarte asemanator cu cel al analizei sintactice: identificarea, conform unor reguli, a unitatilor distincte in cadrul programului, semnalarea de erori in cazul abaterii de la reguli, clasificarea si codificarea unitatilor identificate etc. Din aceasta cauza, functiile analizorului lexical pot fi preluate de analizorul sintactic. Cu toate acestea, in marea majoritate a cazurilor, se prefera separarea celor doua activitati in faze distincte. In favoarea acestei decizii se pot aduce urmatoarele argumente:

- Proiectarea separata este mai simpla, mai clara, permite lucrul in echipa si corespunde organizarii modulare a compilatorului.
- Procesul de analiza lexicala desi este mai simplu, este relativ lung, in timp. El presupune: citirea textului sursa de pe suportulextern, accesul la fiecare caracter, numeroase comparatii cu elementele unor multimi de caractere sau de grupuri de caractere in vederea identificarii si clasificarii atomilor, cautari si/sau introduceri in tabele etc. Separarea va conduce in acest caz la organizarea mai judicioasa a programului, la cresterea eficientei analizei lexicale permitand chiarscrierea ANLEX intr-un limbaj cu facilitati speciale pentru aceste operatii in timp cu urmatoarele faze pot fi realizate in alt limbaj de programare. Un alt procedeu pentru cresterea vitezei analizorului lexical este utilizarea unor buffere de intrare, in memoria interna, pentru inmagazinarea textului sursa. In timpul desfasurarii analizei lexicale avand la intrare un buffer, poate fi pregatit, prin citirea de pe suportul extern, bufferul urmator.

- Sintaxa atomilor lexicali este simpla in comparatie cu sintaxa instructiunilor limbajului si poate fi exprimata printr-ogramatica regulata (GR). Regulile specifice unei GR si procesul de analiza aferent pot fi modelate cu ajutorul expresiilor regulate sau a automatelor finite ceea ce simplifica mult procesul de proiectareprogramare al analizorului lexical.[7]

5. Compilatorul Microsoft Cl.exe si compilatoarele Gcc din Linux

In capitolul ce urmeaza vom prezenta doua din cele mai cunoscute compilatoare: CL.EXE si GCC.

In ciuda apropierii disperate intre mediile de dezvoltare, atat sistemele UNIX cat si Microsoft Windows impart o arhitectura comuna de tip back-end cand vine vorba de compilatoare. Generarea de executabile este in esenta realizata pe ambele sisteme de un singur program, compilatorul. In cazul sistemelor Microsoft cel mai cunoscut compilator este CL.EXE, iar in cazul sistemelor UNIX/Linux acesta este GCC.

Compilarea, fie ca e facuta sub windows, fie ca e sub UNIX/Linux, este formata din 5 etape: preprocesare C, analiza, translatarea, asamblarea si link-editarea.

Preprocesarea C

Preprocesarea se ocupa cu partea logica din spatele directivelor din C. Aceasta lucreaza intr-o singura cale si in esenta este doar un motor de substitutie.

Gcc -E

Aceasta comanda ruleaza doar etapa de preprocesare. Aceste etape includ fisiere de tip .c si de asemenea traduc macro-urile in linie de cod de tip C. Se poate adauga –o pentru a face redirectarea spre un fisier.

Cl -E

[7] J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

La fel ca si mai sus, aceasta comanda va rula doar in etapa de preprocesare, trimitand catre iesirile standard rezultatele.

5.1 Analiza si translatarea

Etapele de analiza si translatare sunt cele mai folositoare etape ale compilatorului. Din pacate, lumea UNIX/Linux si cea Microsoft Windows nu sunt la fel in alegerea sintaxei pentru asamblare. Dar multe dintre uneltele GNU au posibilitatea flexibila de a alege sintaxa Intel.

Gcc -S

Aceasta comanda va folosi ca intrare fisiere de tip .c si ca iesire .s in sintaxa de tip AT&T. Pentru a avea sintaxa Intel trebuie sa adaugam optiunea –masm=intel. Iar pentru a face asocieri intre variabilele si stivele folosite folosim –fverbose -asm.

Gcc poate fi apelat cu multiple optiuni de optimizare care pot face lucruri interesante codului de asamblare de iesire. Sunt intre 4 si 7 optimizari generale ale claselor care pot fi specificate cu –ON, unde 0<=N<=6. 0 este fara optimizare (default) si 6 este de obicei maxim, chiar daca in multe cazuri nici o optimizare nu este facuta peste 4, depinzand de arhitectura si versiunea gcc.

Sunt de asemenea multe setari de finete in optiunile de asamblare care sunt specificate cu flag-ul –f. Cele mai interesante sunt –funroll-loops, -finline-functions si –fomit-frame-pointer. "-funroll-loops" inseamna a extinde bucla astfel incat sa avem n copii ale codului pentru n iteratii ale buclei. La procesoarele moderne aceasta optimizare este neglijabila. "-finline-functions" inseamna convertirea efectiva a tuturor functiilor dintr-un fisier in macrouri si plasarea copiilor codurilor acestora direct in functia de apelare. Aceasta functie este valabila doar pentru functii apelate in acelasi fisiser C ca si cel de definitie. "-fomit-frame-pointer" elibereaza un registru suplimentar pentru a fi folosit in programul nostru. Daca aveam mai mult de 4 variabile locale mari aceasta setare poate fi un mare avantaj, altfel nu este folositoare.

Cl -S

cl.exe are si el de asemenea o optiune –S care genereaza asamblarea si are mai multe optiuni de optmizare. Din pacate cl nu permite ca optimizarea sa fie controlata la fel de fin ca in cazul gcc. Principalele optiuni de optimizare pe are le ofera cl.exe sunt predefinite pentru viteza sau spatiu.

5.2 Asamblare

In etapa de asamblare codul este tradus aproape direct in cod masina. Totusi apare un minim de preprocesare, analiza si rearanjare de instructiuni.

GNU

Acesta este asamblorul GNU si are la intrare o sintaxa AT&T sau Intel, .asm file si genereaza un fisiser obiect de tip .o.

MASM

Acesta este asemblorul Microsoft. Se exeuta ruland fisierul ml.

5.3 Link-editarea (editarea legaturilor)

Atat Microsoft Windows cat si Unix au proceduri de link-editare similare, chiar daca baza este putin diferita. Ambele sisteme au 3 tipuri de link-editare si ambele le implementeaza in moduri similare.

Link-editarea statica

Aceasta inseamna ca pentru fiecare functie apelata de program, asamblarea este inclusa in fisiserul executabil. Apelarea functiilor se face apeland direct adresa codului, in acelasi fel in care sunt apelate functiile in program.

Link-editarea dinamica

Aceasta inseamna ca librariile exista intr-un singur loc din intreg sistemul si memoria virtuala a sistemului de operare va mapa aceasta locatie in spatiul de adresa a programului cand se ruleaza programul. Adresa la care apare maparea nu este mereu garantata, chiar daca ramane constanta o data ce executabilul a fost creat.

Link-editarea runtime

Aceasta editare apare cand un program apeleaza o functie dintr-o librarie care nu a fost inclusa in momentul compilarii. Libraria este mapata in dlopen() sub UNIX/Linux si in LoadLibrary() in cazul Microsoft Windows, ambele intorc un rezultat care este apoi trecut intr-o functie simbol de rezolutie(dlsym() si GetProcAddress()). Acestea din urma la randul lor intor o functie pointer care poate fi apelata direct din program ca si cum ar fi o functie normala. Aceasta abordare este des folosita de aplicatii pentru a incarca librarii specifice care definesc initializarea functiilor. Astfel de functii de initializare de obicei dau adrese de functii ce prezinta programul care le-a incarcat.

Ld

Ld este linkerul GNU. Acesta genereaza un fisier executabil valid.

Link.exe

Este lin-editorul Microsoft Visual C++ . In mod normal putem trece optiunea direct prin intermediul comenzii cl -link . Oricum il putem folosi direct pentru a face legatura intre fisierele obiect si cele .dll intr-un fisier executabil. Microsoft Windows de asemenea are nevoie si de un fisier .lib sau .def aditional .dll-urilor pentru a face legatura intre ele.

5.4 Compilatorul CL.EXE

Compilatorul primeste primele indicatii de la user in format text,in linie de comanda, dar de asemenea si de la fisiere si variabilele de sistem. Acest text este impartit in semne care ne sunt prezentate ca si semne de linie de comanda, chiar daca sunt de la alte surse. Semnele sunt separate de spatii albe in general, dar analizatorul de linie de comanda furnizeaza o serie de cazuri speciale. Un mic sumar ar fi ca spatiile albe sunt permise intr-un semn daca sunt imprejmuite de cote duble.

Tipuri

Fiecare linie de comnada este formata din unul sau mai multe token-uri.

Sunt cunoscute insa 3 categorii generale:

- o optiune incepe cu o cratima sau slash. Pentru unele comenzi unul sau mai multe token-uri din linia de comanda care urmeaza poate continua optiunea (ca si argument al acestei optiuni, in loc sa inceapa noi comenzi)
- sau, un semn de linie de comanda care incepe cu @ nume de fisier de comanda, este de asemenea numit un fisier de raspuns, al carui text mai multe token-uri de linie de comanda.
- Orice alte nume de token-uri ale liniei de comanda din fisierul de imput

Exista niste amestecari ale diferitelor tipuri de directive. Cum am sugerat deja, o comanda care numeste un fisier de comanda poate sa contina mai multe comenzi de mai multe feluri, incluzand si mai multe nume de fisiere. Este de asemenea stiut ca fisiserul de input poate fi numit de optiuni. (ex: /tc, /to si /tp)

Surse

Compilatorul isi gaseste comenzile efectiv in linia de comanda compuse din urmatoarele surse in ordinea precizate:

- valaorea variabilei cl
- linia de comanda in sine
- valoarea variabilei cl

Stiind ca in fiecare sursa, fiecare semn care numeste un fisier de comanda este inlocuit de textul din fisier. Fiecare sursa, si fiecare linie de comanda este analizata independent. Aceasta inseamna in particular ca nici un token nu poate fi transportat catre sursa urmatoare sau catre alta linie si nici o alta optiune care este altfel permisa nu se poate transmite in subtoken-uri.

Doar in putine cazuri conteaza daca o instructiune spre compilator a fost trimisa dintr-o sursa sau alta. Este o tehnica standard al acestor notite ca tot ce se refera la

linia de comanda, doar calificata ca fiind linie de comanda, inseamna efectiv linia de comanda compusa mai sus.

Linia de comanda care apare mai sus este ce poate fi oferit de user. Este de asemenea stiut ca CL poate extrage 2 linii de comanda de codate. Una precede comenzile oferite de user si este interpretata pentru a seta optiunile initiale pe care userul este liber sa le modifice. Totusi daca optiunile rescrise de user sau sunt incompatibile cu optiunile initiale, apoi optiunile initiale sunt aruncate. Altfel, optiunile inititale persista ca si default. Celelalte cazuri se aplica dupa ce userul aloca linia de comanda si lucreaza in ocnsecinta pentru a asigura optiunile obligatorii si sa le setam chiar daca sunt neglijate de user.

In timp ce nimic exceptional se intampla, este bine sa mentionam ca linia de comanda primeste atat o procesare obisnuita, ca cea descrisa mai sus si ca si cale timpurie. Mai tarziu apare o scanare rapida, mai ales pentru a decide cateva puncte de purtare pentru procesele obisnuite si pentru lucruri ca logo de startup care sunt deschise inainte ca orice alt lucru subtantial sa inceapa. Trecerea timpurie afecteaza doar optiunile /be, /clr,/nologo,/noover, si /zx si in unele cazuri foarte usoare.

Ai dat un manual de folosire CL. Cum funcţionează înăuntru, ce soluţii constructive foloseşte, asta este de scris la cap compilatoare, nu folosirea lui din linia de comanda.

Exemplu de compilare folosind Compilatorul Microsoft Cl.exe

Creem directorul **c:\test**.

Creem fisierul test.cpp iar in el intoducem uramtorul cod::

```
#include <iostream>

using namespace std;
```

```
int main() {
  cout << "Acesta este un test! \n";
  return 0;
}
```

Compilam aplicatia:

```
cl /EHsc test.cpp
```

Acum ar treui sa apara:

Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86

Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

test.cpp

Microsoft (R) Incremental Linker Version 7.10.3077

Copyright (C) Microsoft Corporation. All rights reserved.

/out:test.exe

test.obj

Pentru a rula aplicaia tastam:

test

Acum va aparea:

Acesta este un test!

Compilatorul GCC

Compilatorul **GCC** sau GNU Compiler Collection este unul din cele mai utilizate şi performante compilatoare pentru C/C++(în special C, pentru C++ folosindu-se des compilatorul G++). Şi aceasta aparţine colecţiei de software GNU, adică este un program open source. Este un compilator extensibil, deşi este dedicat pentru C totuşi are extesii şi pentru alte limbaje de programare cum ar fi: C++, Pascal, Fortran, Objective-C etc. Este un program open source cu toate acestea executabilele generate de acesta nu trebuie să fie neapărat open source, chiar dacă include librăriile standard C sau C++.

Prima versiune GCC îi aparţine lui Richard Stallman (iniţiatorul proiectului GNU), şi este datată în anul 1986. Pe parcursul anilor au tot apărut versiuni din ce în ce mai evoluate, la început necesitând instalarea separată a compilatorului, mai apoi să fie incluse în distribuţiile GNU/LINUX (în acest caz pentru doritori se poat face doar upragadarea la o versiune mai nouă a compilatorului GCC).

Sintaxa generală a compilatorului pentru apelarea acestuia este:

gcc opţiuni nume_fişier

Pentru C fişierele sursă au extensia **.c**. Pentru a compila un program **nume.c**, se executa comanda:

gcc -o nume nume.c

Argumentul nume al opţiunii -o indică numele dat executabilului (de obicei, acelaşi nume de bază ca şi pentru sursă). Fără opţiunea -o, executabilul e denumit implicit a.out.

Un alt mod de a folosi opţiunea -o este:

gcc nume.c -o nume

Puţin ne vom ocupa în continuare de compilarea şi execuţia unui program C, pentru a înţelege cum putem sau cum mai putem folosi compilatorul gcc.

Deci ştiind care este comanda pentru compilare vom şi executa un program **nume.c**, şi anume:

gcc nume.c -o nume

./nume

Prima comandă semnifică compilarea şi link-editarea (rezolvarea apelurilor de funcţii) fişierului sursă **nume.c**, generându-se un executabil al cărui nume este specificat prin opţiunea -o. De fapt, se execută în mod transparent următoarea secvenţă de operaţii:

- preprocesorul cpp expandează macrourile şi include fişierele header;
- se generează cod obiect prin compilare;
- editorul de legături ld rezolvă apelurile de funcţii şi creează executabilul.

Această secvenţă poate fi parcursă şi manual. Mai întâi, se execută preprocesarea (-E), generându-se fişierul intermediar **nume.cpp**:

gcc -E nume.c -o nume.cpp

Se generează un fişier obiect utilizând fişierul sursă anterior preprocesat:

gcc -x cpp-output -c nume.cpp -o nume.o

Fişierul obiect este link-editat şi se creează executabilul:

gcc nume.o -o nume

Opţiuni utile ale lui gcc:

-Iname	Fişierele .h sunt căutate şi în directorul numit name; implicit, fişierele .h sunt căutate în directorul curent

	şi în directoarele ce conţin headerele pentru librăriile standard (/usr/include şi /usr/include/sys).
-Lname	Librăriile sunt căutate şi în directorul numit name; implicit, librăriile sunt căutate în /lib şi /usr/lib.
-lname	Link-editează librăria cu numele libname
-static	Link-editează static o librărie
-D name	Definirea macroului numit name
-D name=valoare	Definirea macroului cu valoarea specificată de valoare
-U name	Anularea macroului numit name
-g	Include informaţii de depanare în fişierul-obiect generat

Pentru a putea compila programe cu funcţii matematice standard e nevoie de biblioteca libm. În acest caz, se foloseşte opţiunea -l urmată, fără spaţiu, de numele bibliotecii (fără prefixul lib folosit convenţional pentru toate bibliotecile) astfel:

gcc -lm -o nume nume.c

La compilare, e utilă generarea cât mai multor avertismente. Aceasta se specifică cu opţiunea -Wall (warnings: all).

Optiuni de compilare mai des utilizate:

Intelegand modul in care compilatorul face conversia unui program de la codul sursa C la codul executabil vom fi capabili sa intelegem si utilitatea urmatoarelor optiuni de compilare, astfel:

-c

gcc file1.c file2.c ..... -c executable

Obliga compilatorul sa nu scoata formatul executabil ci doar fisierele intermediare, cu extensia .o, urmand ca “asamblarea” lor in fisierul executabil sa se faca prin comanda:

gcc file1.o file2.o ..... -o executable

-llibrary

Linkediteaza sursa utilizand libraria specificata in aceasta optiune. Optiunea trebuie sa urmeze dupa fisierele sursa C. Librariile pot fi cele sistem sau chiar create de catre utilizator. In general atunci cand utilizam functiile matematice va trebui sa includem declaratiile lor printr-o directiva de preprocesare de genul: #include<math.h> si trebuie sa includem biblioteca matematica la linkeditare de genul:

gcc calc.c -o calc -lm

-Ipathname

Adauga o cale unde compilatorul va cauta fisierele ce apar in directivele: #include<...> Implicit preprocesorul va cauta fisierele in directorul curent, apoi in directoarele specificate prin optiunea –I si abia in final in directorul /usr/include.

gcc prog.c -I/home/myname/myheaders

Ar mai fi de spus în legătură cu utilizarea gcc ca şi compilator, numai că posibil intrând în mai multe detalii vom omite unele caracteristici care tot mai apar cu apariţia versiunilor mai noi de gcc (la ora actuală gcc se află undeva la versiunea 4.0.2, inclusă în distribuţia SuSE 10.0), aşa că soluţia cea mai bună este să parcurgem manualele versiunilor recente pentru a afla mai multe despre gcc.

Exemplu de compilare folosind complatorul Linux Gcc:

[nexusonline@localhost prog]$ ls -rtl

-rw-rw-r-- 1 nexusonline nexusonline 157 Jan 8 17:33 program.c

[nexusonline@localhost prog]$ cat program.c

#include <stdio.h>

```
int main(void)
{
  printf("\nAcesta este primul program C.\n");
  printf("Apasati ENTER pentru a continua.");
  getchar();
  return 0;}
```

[nexusonline@localhost prog]$ gcc -v -c program.c

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --disable-dssi --with-java-home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre --with-cpu=generic --host=i386-redhat-linux

Thread model: posix

gcc version 4.1.0 20060304 (Red Hat 4.1.0-3)

/usr/libexec/gcc/i386-redhat-linux/4.1.0/cc1 -quiet -v program.c -quiet -dumpbase program.c -mtune=generic -auxbase program -version -o /tmp/ccnpguim.s

ignoring nonexistent directory "/usr/lib/gcc/i386-redhat-linux/4.1.0/../../../../i386-redhat-linux/include"

#include "..." search starts here:

#include <...> search starts here:

/usr/local/include

/usr/lib/gcc/i386-redhat-linux/4.1.0/include

/usr/include

End of search list.

GNU C version 4.1.0 20060304 (Red Hat 4.1.0-3) (i386-redhat-linux)

compiled by GNU C version 4.1.0 20060304 (Red Hat 4.1.0-3).

GGC heuristics: --param ggc-min-expand=81 --param ggc-min-heapsize=96980

Compiler executable checksum: bba44d5df49c85f0bc824786061245c8

as -V -Qy -o program.o /tmp/ccnpguim.s

GNU assembler version 2.16.91.0.6 (i386-redhat-linux) using BFD version 2.16.91.0.6 20060212

[nexusonline@localhost prog]$ ls -rtl

-rw-rw-r-- 1 nexusonline nexusonline 157 Jan 8 17:33 program.c

-rw-rw-r-- 1 nexusonline nexusonline 1012 Jan 8 17:34 program.o

[nexusonline@localhost prog]$ gcc -v program.o -o program

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --disable-dssi --with-java-

home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre --with-cpu=generic --host=i386-redhat-linux

Thread model: posix

gcc version 4.1.0 20060304 (Red Hat 4.1.0-3)

/usr/libexec/gcc/i386-redhat-linux/4.1.0/collect2 --eh-frame-hdr -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o program /usr/lib/gcc/i386-redhat-linux/4.1.0/../../../crt1.o /usr/lib/gcc/i386-redhat-linux/4.1.0/../../../crti.o /usr/lib/gcc/i386-redhat-linux/4.1.0/crtbegin.o -L/usr/lib/gcc/i386-redhat-linux/4.1.0 -L/usr/lib/gcc/i386-redhat-linux/4.1.0 -L/usr/lib/gcc/i386-redhat-linux/4.1.0/../../.. program.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.0/crtend.o /usr/lib/gcc/i386-redhat-linux/4.1.0/../../../crtn.o

[nexusonline@localhost prog]$ ls -rtl

-rw-rw-r-- 1 nexusonline nexusonline 157 Jan 8 17:33 program.c

-rw-rw-r-- 1 nexusonline nexusonline 1012 Jan 8 17:34 program.o

-rwxrwxr-x 1 nexusonline nexusonline 4958 Jan 8 17:34 program

[nexusonline@localhost prog]$ ./program

Acesta este primul program C.

Apasati ENTER pentru a continua.

BIBLIOGRAFIE:

1. Niklaus Wirth - Compiler Construction
http://www.wikipedia.org

2. BNF and EBNF: What are they and how do they work?" - Lars Marius Garshol(www.garshol.priv.no/download/text/bnf.html)

"Conceptele fundamentale ale limbajelor de programare" - Horia Ciocarlie, Ed. Orizonturi Universitare, 2006

3. http://www.cs.cmu.edu/~mihaib/articole/regex/regex-html.html
A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988

J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985..

Addison Wesley - Compiler Design - Formal Syntax And Semant

4. Irina Athanasiu – Limbaje formale si automate

5. Niklaus Wirth - Compiler Construction
http://www.wikipedia.org