1/8/2008

Compilatoare

Continut

1.ISTORIE....3

2.COMPILATOARELE IN EDUCATIE....4

3.IESIREA COMPILATORULUI....4

4.COMPILARE VS INTERPRETARE....5

5.DESIGNUL COMPILATORULUI....5

6.COMPILATOARELE MONO-FAZATE VERSUS MULTI-FAZATE....6

7.FRONT-END....7

8.BACK-END....7

9.DESCRIEREA BNF A GRAMATICII UNUI LIMBAJ....8

MODALITATI DE PARSARE....10

Parsarea top-down (LL)....10

Parsarea bottom-up (LR)....10

10.ANALIZA LEXICALA....11

ROLUL ANALIZEI LEXICALE....12

PROBLEME IN ANALIZATOARELE LEXICALE....12

TOKEN, TIPARE, CUVINTE....13

11.ANALIZA SINTAXEI....14

ROLUL PARSERULUI....15

GRAMATICI FARA CONTEXT....16

12.COMPILATORUL GCC....17

INTERFETE....17

OPTIMIZARE....17

NUCLEUL..18

13.Bibliografie..19

14.PARTICIPARE PE CAPITOLE...19

Compilatoare

Compilatorul este un program (sau set de programe) ce interpreteaza textul scris in limbajul sursa in alt limbaj, numit limbajul “target”. Succesiunea originala este numita, de obicei, codul sursa, iar output-ul este codul obiect. In mod obisnuit, iesirea are o forma potrivita pentru procesarea de catre alte programe (de exemplu, linker), dar poate fi un fisier text human-readable.

Motivul pentru care dorim sa interpretam codul sursa este de a crea un program executabil. Numele “compilator” este folosit in principal, la programele care interpreteaza codul sursa dintr-un limbaj de programare de nivel superior intr-un limbaj de nivel inferior (de exemplu, limbaj de asamblare sau limbaj masina). Programul care interpreteaza un limbaj de un nivel inferior intr-unul de un nivel superior, se numeste decompilator. Programul care interpreteaza limbaje de nivel superior se numeste, de obicei, traducator de limbaj, traducator sursa la sursa sau convertor de limbaj. Limbajul de rescriere este un program care interpreteaza forma expresiilor fara sa schimbe limbajul. Un compilator executa majoritatea operatiunilor urmatoare: analiza lexicala, preprocesare, analiza unui text, analiza semantica, generarea codului si optimizarea codului.

1. ISTORIE

Software-ul calculatoarelor pioniere a fost scris timp de multi ani numai in limbaj de asamblare. Limbajele de programare de nivel superior nu au fost concepute pana cand beneficiile refolosirii software-ului pe diferite tipuri de procesoare au inceput sa fie apreciate, in comparatie cu costurile scrierii unui compilator. De asemenea, capacitatea limitata a calculatoarelor pioniere a creat dificultati in implementarea unui compilator.

La sfarsitul anilor 1950, au fost propuse limbajele de programare independente, care ruleaza pe diferite tipuri de procesoare si microarhitecturi.

Ulterior, au fost concepute mai multe compilatoare experimentale. Primul compilator a fost realizat de Grace Hooper in 1952 pentru limbajul programare A-0. Echipa FORTRAN, de la IBM, condusa de John Backus este cunoscuta ca fiind cea care

a introdus primul compilator, in 1957. COBOL a fost in 1960 un limbaj timpuriu compilat pe multiple arhitecturi.

In multe domenii de aplicatie, ideea de a folosi limbaj de nivel superior a fost preferata. Datorita functionalitatii in dezvoltare suportate de noile limbaje de programare si complexitatii in crestere a arhitecturii calculatoarelor, compilatoarele au devenit din ce in ce mai complexe.

Primele compilatoare au fost scrise in limbaj de asamblare. Primul compilator self-hosting – capabil sa-si compileze propriul cod sursa intr-un limbaj de nivel superior – a fost creat pentru LISP de actre Hart si Levin de la Universitatea de Informatica de la Massachusetts in 1962. Din anii 1970, a devenit o procedura uzuala implementarea compilatorului in limbajul de compilare, desi Pascal si C au fost populare pentru limbajul de implementare. Construirea compilatorului self-hosting reprezinta o problema intriganta – primul compilator de acest fel trebuie sa fie compilat de un compilator intr-un limbaj diferit sau compilat, ruland compilatorul intr-un interpretor.

2. COMPILATOARELE IN EDUCATIE

Constructia compilatoarelor si optimizarea lor sunt studiate in universitati ca parte a stiintei calculatoarelor. Aceste cursuri sunt completate de implementarea compilatorului pentru un limbaj de programare educational. Un exemplu educational este compilatorul PL/0 al lui Niklaus Wirth, pe care Wirth l-a folosit la predarea construirii compilatoarelor. In ciuda simplitatii sale, compilatorul PL/0 a introdus cateva concepte in domeniu:

1. Program development by stepwise refinement (de asemenea, titlul eseului lui Wirth din anul 1971)
2. Utilizarea parserului descendent recursiv
3. Utilizarea EBNF pentru specificarea sintaxei limbajului.
4. Un generator de cod care produce cod-P portabil
5. Utilizarea diagramelor T in descrierea formala a problemei bootstrapping.

3. IESIREA COMPILATORULUI

Un mod de clasificare a compilatoarelor este dupa platforma pe care se executa codul generat de aceastea. Aceasta este cunoascuta ca platforma target. Un compilator nativ sau gazduit este acela al carui output este conceput sa functioneze pe acelasi tip de calculator si sistem de operare ca si compilatorul. Iesirea unui compilator incrucisat este conceput sa lucreze pe o platforma diferita. Compilatoarele incrucisate sunt

folosite, de obicei, in dezvoltarea software-ului pentru sistemele integrate care nu sunt concepute sa suporte un mediu de dezvoltare software.

Programul generat de catre un compilator care produce codul unei masini virtuale poate sau nu fi executat pe aceeasi platforma ca si compilatorul care l-a produs. De aceea, aceste compilatoare nu sunt clasificate ca fiind native sau incrucisate.

4. COMPILARE VS INTERPRETARE

Limbajele de programare de nivel inalt sunt impartite in limbaje de compilare si de interpretare. Totusi, sunt putine limbaje care cer sa fie exclusiv compilate sau exclusiv interpretate. Categorizarea reflecta cele mai populare implementari ale limbajului – de exemplu, BASIC este considerat ca un limbaj de interpretare si de compilare in C, in ciuda existentei compilatoarelor BASIC si interpretoarelor C.

Intr-un fel, toate limbajele sunt interpretate, "executia" fiind un caz special de interpretare, realizat de tranzistoare care comuta pe un processor. Modernizarea tinde spre compilarea just-in-time si interpretarea bytecode.

Exista unele exceptii. Unele specificatii ale limbajului indica faptul ca implementarile trebuie sa contina si serviciul de compilare; de exemplu, Common Lisp. Alte limbaje au caracteristici care sunt usor de implementat intr-un interpretor, dar fac scrierea unui compilator mai dificila; de exemplu, APL, SNOBOL4 si numeroase limbaje de script ce permit programelor sa conceapa codul sursa arbitrar in timpul de lucru, cu operatii pe string-uri si apoi executa codul prin trecerea lui intr-o functie de evaluare. Pentru a implementa aceste caracteristici intr-un limbaj de compilare, programele trebuie sa fie lansate cu o librarie care include o versiune a compilatorului.

5. DESIGNUL COMPILATORULUI

Modul de concepere a compilatorului depinde de complexitatea proceselor care vor fi indeplinite, experienta celui care concepe si de resursele disponibile (de exemplu, persoane si unelte).

Un compilator cu un limbaj relativ simplu scris de o persoana poate fi o singura, monolitica piesa software. Atunci cand limbajul sursa este complex si de dimensiune mare si este ceruta o iesire de calitate superioara, schita poate fi impartita intr-un numar relativ independent de faze/treceri. Existenta fazelor separate face posibila divizarea in parti mici a dezvoltarii, distribuite la mai multe persoane. De asemenea, este mai usor sa se inlocuiasca o singura faza de una imbunatatita, sau sa se introduca o noua faza mai tarziu (de exemplu, optimizarea aditionala).

Divizarea proceselor de compilare in faza a fost consacrata de Production Quality Compiler-Compiler Project (PQCC) de la Universitatea Carnegie Mellon. Acesta a introdus termenii ca front end, middle end (foarte putin folosit) si back end.

Cele mai mici compilatoare au cel putin doua faze. Totusi, aceste faze sunt considerate ca facand parte din front end sau back end. Punctul de intersectie al acestor doua end-uri este deschis pentru orice dezbatere. Front end-ul se considera a fi acolo unde se realizeaza procesarea sintactica si semantica, impreuna cu traducerea la un nivel inferior a reprezentarii.

Middle end-ul este conceput pentru optimizarea unei forme, alta decat codul sursa sau codul masina. Acest cod sursa/cod masina este creat pentru permiterea optimizarilor generice de a fi impartite intre versiunile compilatorului care suporta limbaje diferite si procesoare target.

Back end-ul ia iesirea de la mijloc. Poate sa realizeze mai multe analize, transformari si optimizari pentru un calculator anume. Apoi, genereaza cod pentru un anumit procesor si sistem de operare.

Aceasta abordare front-end/middle/back-end face posibil sa combinam front end-uri pentru limbaje diferite cu back end-uri pentru procesoare diferite. Exemple practice ale acestei abordari sunt Colectia Compilatorului GNU, LLVM si Kitul Compilatorului Amsterdam, care au multiple front end-uri, analiza imparita si multiple back end-uri.

6. COMPILATOARELE MONO-FAZATE VERSUS MULTI-FAZATE

Clasificarea compilatoarelor dupa numarul de faze isi are baza in resursele limitate hardware ale calculatoarelor. Compilarea implica executarea intensa si calculatoarele pioniere nu dispuneau de suficienta memorie care sa contina un program care facea toata aceasta treaba. Asadar, compilatoarele au fost impartite in programe mai mici, fiecare facand o trecere prin sursa (sau o reprezentarea a acesteia) executand ceva din analiza si traducerile cerute.

Abilitatea de a compila intr-o singura trecere este, de obicei, considerata ca un avantaj deoarece simplifica sarcina de scriere a compilatorului si compilatoarele mono-fazate sunt mai rapide decat cele multi-fazate. Multe limbaje au fost create pentru a putea fi compilate mono-fazat (exemplu, Pascal).

In unele cazuri, modul de realizare a unei caracteristici de limbaj poate sa solicite compilatorului sa execute mai mult de o trecere prin sursa. De exemplu, se considera o declaratie aparuta pe linia 20 a sursei care afecteaza traducerea unei declaratii de pe linia 10. In acest caz, prima trecere trebuie sa culeaga informatii despre declaratiile aparute dupa cele pe care le afecteaza, in acelasi timp realizandu-se traducerea respectiva intr-o trecere ulterioara.

Dezavantajul compilarii intr-o singura trecere este acela ca nu poate sa execute multe din optimizarile complexe necesare sa genereze cod de calitate superioara. Etapa de optimizare poate analiza o expresie de mai multe ori, iar pe alta expresie doar o singura data.

7. FRONT-END

Front end-ul analizeaza codul sursa pentru a crea o reprezentare interna a programului, numita reprezentare intermediara sau RI. De asemenea, serveste si tabelului de simboluri, o structura de date legand fiecare simbol in codul sursa cu informatia asociata, ca de exemplu, locatie, tip. Aceasta se realizeaza in timpul a mai multor treceri, care includ urmatoarele:

1. Reconstructia liniei. Limbajele care permit spatii arbitrare printre indentificatori solicita o faza inainte de parsare, care converteste secventa de caractere de la intrare intr-o forma canonica pregatita pentru parser.

2. Analiza lexicala imparte textul codului sursa in parti mici numite simboluri(token). Fiecare token este o unitate mono-atomica a limbajului, de exemplu un cuvant cheie, un identificator sau un nume simbol. Sintaxa tokenului se numeste lexing sau scanare, iar software-ul care face analiza lexicala se numeste analizator lexical sau scaner.

3. Preprocesarea. Unele limbaje, precum C, solicita o faza de preprocesare care suporta macro substitutie si compilare conditionata. Astfel, preprocesorul manipuleaza tokenii lexicali, fata de formele sintactice. Totusi, unele limbaje precum Scheme suporta macro substitutie bazata pe formele sintactice.

4. Analiza sintaxei implica parsarea secventei de token pentru indentificarea structurii sintactice a programului. Aceasta faza construieste arborele de parsare, care inlocuieste secventa liniara de token cu o structura de arbore creata in conformitate cu regulile gramaticii formale, care defineste sintaxa limbajului. Acest arbore de parsare este adeseori analizat si transformat ulterior de faze ulterioare in compilator.

5. Analiza semantica este faza in care compilatorul adauga informatie semantica arborelui de parsare si construieste tabelul de simboluri. Aceasta faza executa controale semantice precum controlul de scriere sau legatura obiect (asociind referinte ale variabilelor si ale functiilor cu definitiile lor), sau asignare definitiva (solicitand initializarea tuturor variabilelor locale inainte de folosinta), respingand programele incorecte sau emitand avertismentele.

8. BACK-END

Termenul back end este uneori confundat cu generatorul de cod din cauza suprapunerii cu functionalitatea de generarea a codului de asamblare. Principalele faze ale back end-ului cuprind urmatoarele:

1. Analiza: Aici se culeg informatii ale programului de la reprezentarea intermediara derivata din intrare. Analizele tipice sunt analiza fluxului de date pentru crearea legaturilor use-define, analiza dependentelor, analiza alias, analiza pointerilor etc. Analiza exacta este baza oricarei optimizari a compilatorului. Graful de apelare si graful fluxului de control sunt create in faza de analiza.

2. Optimizarea: reprezentarea limbajului intermediar este transformata in forme echivalent functionabile,dar mai rapide (si mai mici). Optimizari populare sunt inline expansion, dead code elimination, constant propagation, loop transformation, register allocation si chiar automatic parallelization.

3. Generarea codului: limbajul intermediar transformat este tradus in limbajul tinta, de obicei limbajul masina al sistemului. Aceasta implica resurse si decizii de stocare, precum hotararea care variabila sa incapa in registru si memorie si selectarea si programarea instructiunilor de masina corespunzatoare impreuna cu modurile de adresare asociate.

Existenta analizei si optimizarii interprocedurale este comuna in compilatoarele moderne de la HP, IBM, SGI, Intel, Microsoft si Sun Microsystems. GCC-ul a fost criticat pentru mult timp pentru lipsa puternicelor optimizari interprocedurale, dar se produc schimbari pe aceasta tema. Un alt compilator open source cu structura de analiza si optimizare este Open64, care este folosit de multe organizatii pentru cercetare si scopuri comerciale.

Din cauza timpului si spatiului cerut suplimentar pentru analiza si optimizarea compilatorului, unele dintre ele le omit. Utilizatorii trebuie sa foloseasca optiunile de compilare pentru a spune explicit compilatorului ce optimizare sa fie activata.

9. DESCRIEREA BNF A GRAMATICII UNUI LIMBAJ

BNF este prescurtarea de la Backus-Naur Form, notatia Backus-Naur. Aceasta este notatia traditionala folosita de informaticieni pentru a reprezenta o gramatica fara context. BNF reprezinta o modalitate formala de a descrie un limbaj formal.

Notatia Backus-Naur este o notatie raspandita pentru a descrie gramatica limbajelor de programare, seturi de instructiuni sau protocoale de comunicatii, dar poate fi folosita si pentru a reprezenta parti ale gramaticii limbii naturale.

John Backus a creat notatia pentru a defini gramatica limbajului ALGOL, iar Peter Naur a simplificat-o pentru a micsora setul de caractere folosit.

BNF este un fel de joc matematic: pornesti cu un simbol (care se numeste simbol de pornire si care este numit prin conventie S in exemple) si apoi ai un set de reguli care iti spun cu ce poti inlocui acel simbol. Limbajul definit prin gramatica BNF este de fapt doar setul cu toate sirurile pe care le poti produce urmand aceste reguli.

Regulile se numesc reguli de productie si arata ca in exemplul urmator:

```
<symbol> := <alternative1> | alternative2 ...
```

O regula de productie stabileste doar ca simbolul din stanga semnului := trebuie sa fie inlocuit cu una dintre alternativele din partea dreapta. Alternativele sunt separate de simbolul |, care are sensul de SAU. (O variatie ar fi folosirea simbolului ::== in loc de := cu acelasi sens). Alternativele sunt simboluri terminale si simboluri non-terminale. Simbolurile non-terminale sunt de obicei puse intre < si >. Simbolurile terminale sunt pur si simplu piese din sirul final. Ele poarta numele de simboluri terminale pentru ca nu mai exista reguli pentru ele: termina procesul de productie.

O alta variatie o reprezinta inchiderea intre ghilimele a caracterelor terminale pentru a le deosebi de cele non-terminale.

In continuare voi ilustra un exemplu de gramatica BNF:

```
<S>  := '-' <FN> |
        <FN>
<FN> := <DL> |
        <DL> '.' <DL>
<DL> := <D> |
        <D> <DL>
 <D>  := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
'8' | '9'
```

Simbolurile prezentate aici sunt toate prescurtari: <S> este simbolul de start, <FN> este un numar fractionar, <DL> este un sir de cifre si <D> este o cifra.

Propozitiile valide in limbajul definit de aceasta gramatica sunt toate numere, posibil fractionare si posibil negative. Pentru a produce un numar vom incepe cu simbolul de start <S> :

```
<S>
```

Apoi se inlocuieste <S> cu unul dintre rezultatele sale. In cazul de fata aleg <FN> :

```
<FN>
```

Urmatorul pas este sa se inlocuiasca simbolul <FN> cu unul dintre rezultatele sale. Cautam sa obtinem un numar fractionar, deci vom alege varianta cu sirurile de cifre separate prin '.', apoi vom continua inlocuirile pana vom obtine un numar fractionar cu doua zecimale.

```
<DL> . <DL>
<D> . <DL>
3 . <DL>
3 . <D> <DL>
3 . <D> <D>
3 . 1 <D>
3 . 1 4
```

Rezultatul este numarul fractionar 3.14.

MODALITATI DE PARSARE

Parsarea top-down (LL)

Aceasta este cea mai usoara metoda de a parsa ceva conform cu o gramatica. Functioneaza in felul urmator: pentru fiecare rezultat cauta sa afle cu ce simbolul non-terminale a fost generat.

Apoi, in timpul parsarii, se incepe cu caracterul de start si se compara seturile de pornire ale diferitelor rezultate produse cu primul element de intrare pentru a determina care dintre regulile de productie a fost folosita. Desigur, asta se poate realiza doar daca nu exista doua seturi de pornire pentru un simbol care sa aiba acelasi simbol terminal. Daca exista, atunci nu se poate determina ce regula sa se aleaga doar uitandu-ne la primul simbul terminal de la intrare.

Gramaticile LL sunt deseori clasificate dupa numere: LL(0), LL(1), LL(2) etc. Numarul dintre paranteze comunica numarul maxim de simboluri terminale care trebuie verificate simultan pentru a determina regula de productie in orice moment de timp. Deci pentru LL(0) nu ar trebui sa verifici nici un simbol terminal, ai alege mereu regula de productie corecta. Acest lucru este posibil doar daca toate variabilele ar avea un singur simbol terminal, iar daca au un singur rezultat atunci gramatica ar avea un singur sir. Pe scurt: gramaticile de tipul LL(0) nu prezinta interes.

Cel mai util tip de gramatica este LL(1), pentru ca poti sa alegi regula de productie folosindu-te doar de primul terminal al intrarii de fiecare data.

Parsarea bottom-up (LR)

O metoda mai dificila de parsare este parsarea bottom-up. Aceasta tehnica aduna informatii de la intrare pana cand observa ca poate reduce o secvent de intrare cu o variabila. Pentru a demonstra felul in care functioneaza voi folosi gramatica definita anterior si voi incerca parsarea sirului '3.14' pentru a vedea cum a fost generat din gramatica.

Algoritmul incepe prin citirea lui 3 de la intrare si apoi se verifica daca poate fi redus la simbolul din care a fost produs. Se pote, a fost produs de <D>, cu care inlocuim 3. <D> provine din <DL>, asa ca inlocuim <D> cu <DL>. Se observa ca aceasta gramatica este ambigua: se poate reduce in continuare <DL> cu <FN>, ceea ce ar fi incorect. Dupa ce am terminat acesti pasi citim urmatorul simbol de la intrare, '.', pe care nu-l putem reduce.

```
<D>

<DL>

<DL>.
```

Cum nu putem reduce, mai aducem un simbol de la intrare: 1. Il putem reduce la <D> si apoi citim urmatorul simbol, 4. 4 poate fi redus la <D>, apoi <DL>, apoi grupul <D> <DL> poate fi redus la <DL>.

```
    <DL> .
<DL> . 1
<DL> . <D>
<DL> . <D> 4
<DL> . <D> <D>
<DL> . <D> <DL>
<DL> . <DL>
```

Uitandu-ne la gramatica observam ca numai <FN> poate produce <DL> . <DL> si o reducem, iar apoi pe <FN> il reducem la <S>. Aici ne oprim, deoarece am ajuns la sfarsitul parsarii.

```
<DL> . <DL>
<FN>
<S>
```

Mai sus am demostrat cum poate fi folosita notatia Backus-Naur pentru a genera gramatica unui limbaj si am dat exemple de modalitati prin care se poate ajunge de la rezultatul final inapoi la simbolul care l-a produs.

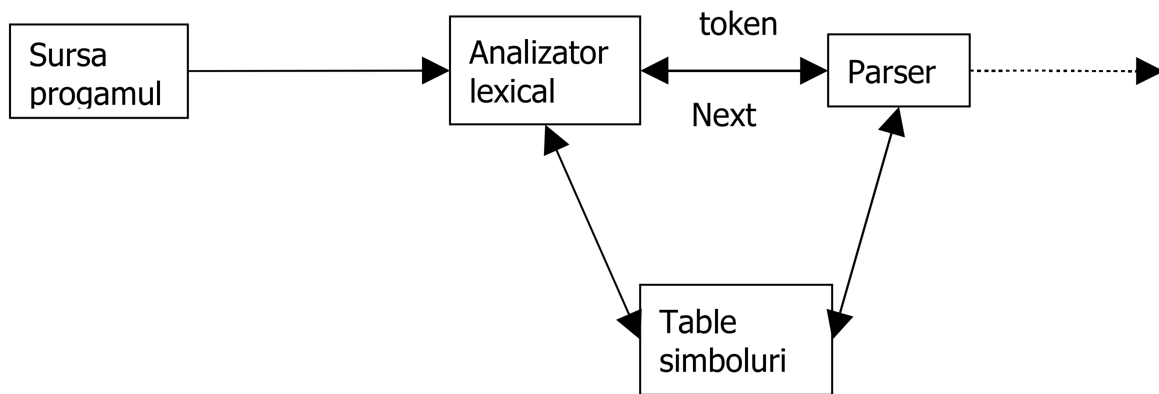
10.ANALIZA LEXICALA

O metoda simpla de a construi un analizator lexical este de a construi o diagrama ce ilustreaza structura cuvintelor rezervate limbajului, urmata de transformarea acesteia intr-un program ce recunoaste cuvintele rezervate(tokens). Tehnica implementarii analizatoarelor lexicale poate fi aplicata de asemenea si in alte arii, ca de exemplu limbaje de querying sau sisteme de informatii(information retrieval systems). In fiecare

din aceste aplicatii problema reala o reprezinta specificatiile si designul unor programe ce executa actiuni in functie de anumite tipare in siruri de caractere.

ROLUL ANALIZEI LEXICALE

Analiza lexicala este prima etapa a compilarii unui program, iar scopul ei este generarea unei secvente de token-uri(cuvinte cheie ale limbajului, simboluri) pornind de la codul sursa, acestea fiind folosite de catre parser pentru analiza sintaxei. Aceasta relatie este implementata de obicei prin scrierea analizei lexicale ca o subrutina a parserului. La primirea mesajului "next token" de la parser, analizatorul lexical citeste caractere de la intrare pana poate identifica urmatorul token.



O data ce analizatorul lexical este partea din compilator ce citeste codul sursa, acesta poate avea si roluri secundare, cum ar fi stergerea comentariilor din cod sau a spatiilor albe. Un alt rol este corelarea mesajelor de eroare de la compilator cu programul sursa. De exemplu, analizatorul lexical(AL) poate tine numarul caracterelor de linie noua(\n) ce le citeste, pentru a afisa o anumita eroare raportata de compilator la linia corespunzatoare. Daca limbajul sursa suporta functii macro preprocesare, atunci aceste functii destinate preprocesarii sunt de asemenea implementate in timpul analizei lexicale. Uneori din cauza multitudinii de taskuri ce le poate rula, AL-urile sunt impartite in doua faze ce se succed: *scanarea* si *analiza*. Scanarea este responsabila cu taskurile simple, in timp ce analiza se ocupa cu operatiile mai complexe. De exemplu, in cazul limbajului Fortran, compilatorul foloseste scanerul pentru a elimina spatiile albe de la intrare.

PROBLEME IN ANALIZATOARELE LEXICALE

Exista cateva motive pentru care s-a facut impartirea fazei de analiza in analiza lexical si parsare.

1. Designul simplu este probabil cel mai important factor. Separarea analizei lexicale de cea a sintaxei ofera de cele mai multe ori simplitate in implementarea

lor. De exemplu, un parser ce include conventiile pentru comentarii in cod este mult mai complex decat unul ce presupune comentariile eliminate de catre AL.

2. Eficienta compilatorului este imbunatatita. Un AL separat de permite sa construim un procesor pentru aceasta faza mult mai specializat si eventual mai eficient. Foarte mult timp este alocat citirii programului sursa and partitionarii acestuia in token-uri. Tehnicile specializate pentru buffering aplicate citirii caracterelor si procesarii tokenurilor pot imbunatati semnificativ performantele unui compilator.

3. Portabilitatea compilatorului este imbunatatita. Caractere ce nu sunt standard sau anomalii specifice anumitor device-uri pot fi eliminate din prima etapa.

TOKEN, TIPARE, CUVINTE

In general, in textul introdus exista un set de stringuri(siruri de caractere) pentru care se genereaza acelasi token. Acest set de stringuri este descris de catre o regula denumita tipar(pattern) asociata unui token. Tiparul trebuie sa se potriveasca fiecarui string din set. Un cuvint(lexeme - denumire engleza) este o secventa de caractere din programul sursa ce se potriveste tiparului pentru a rezulta un token. De exemplu, in codul Pascal: "const pi = 3.14;", secventa de caractere "pi" este un cuvant pentru tokenul "identificator".

Exemple de token:

Token	Exemplu de cuvinte	Descriere
const	const	const
if	if	if
id	pi, count, D2	litera urmata de litere sau cifre
literal	"example"	orice caracter intre " si ", exceptand "

In majoritatea limbajelor de programare, urmatoarele structuri sunt tratate ca token-uri: cuvinte cheie, operatori, identificatori, constante, siruri de caractere si semne de punctuatie(paranteze, ',', ';'). In exemplul de mai sus, cand secventa de caractere "pi" apare in codul programului sursa este returnat un token catre parser. Returnarea unui token este deseori implementata ca transmitere a unui numar intreg(corespunzator tokenului) mai departe. Un tipar este o regula ce descrie un set de cuvinte ce pot reprezenta un token in programul sursa. Tiparul pentru token-ul *const* este doar stringul "const". Pentru a descrie precis tiparul pentru tokenuri mai complexe se folosesc expresii regulate.

Unele limbaje cresc complexitatea analizatoarelor lexicale, ca de exemplu Fortranul. Acesta necesita prezenta anumitor comenzi in pozitii fixe pe linii, rezultand in analiza suplimentara pentru stabilirea corectitudinii sursei. Limbajele moderne insa tind sa omita acest aspect, lasand liber formatul sub care este redactat codul, iar acest aspect devenind din ce in ce mai putin important in cadrul analizatoarelor lexicale.

Tratarea spatiilor variaza de la limbaj la limbaj. In unele, ca de exemplu Fortran sau Algol68, spatiile nu sunt semnificative decat in cadrul stringurilor. Acestea pot fi adaugate la libera alegere pentru a face codul mai citibil. Acest gen de conventii poate creste complexitatea unui AL foarte mult.

Un exemplu ce ilustreaza dificultatea recunoasterii de tokenuri este functia DO din Fortran. In cazul:

DO 5 I = 1.25

nu ne putem da seama, pana vedem punctul, ca DO nu este un cuvant cheie, ci parte dintr-un identificator DO5I. Pe de alta parte, in:

DO 5 I = 1,25

vedem toate token-urile, corespunzatoarea cuvantului cheie DO, label-ului 5, identificatorului I, operatorului =, constantei 1, virgulei si constantei 25. Aici nu putem fi siguri pana nu vedem virgula, ca DO este un cuvant cheie.

In multe limbaje, anumite secvente de caractere sunt rezervate, predefinite si nu pot fi schimbate de catre utilizator. Daca cuvintele cheie nu sunt rezervate, AL-ul trebuie sa faca distinctie intre cuvinte cheie si cele definite de catre programator. In PL/I, cuvintele cheie nu sunt rezervate, deci regulile pentru a face diferenta sunt complicate, ca in exemplul urmator:

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;

11.ANALIZA SINTAXEI

Fiecare limbaj de programare are reguli care prescriu structura sintactica a unor programe bine definite. Gramaticile ofera avantaje importante atat programatorilor cat si scriitorilor de compilatoare.

O gramatica ofera o precizie, in acelasi timp usor de inteles, a specificatie sintactica a unui limbaj de programre.

Din anumite clase de gramatici putem automat construi un parser eficient care determina daca programul sursa este bun din punct de vedere sintactic. Procesul de constructie a parser-ului poate descoperii ambiguitati sintactice si constructii dificile de

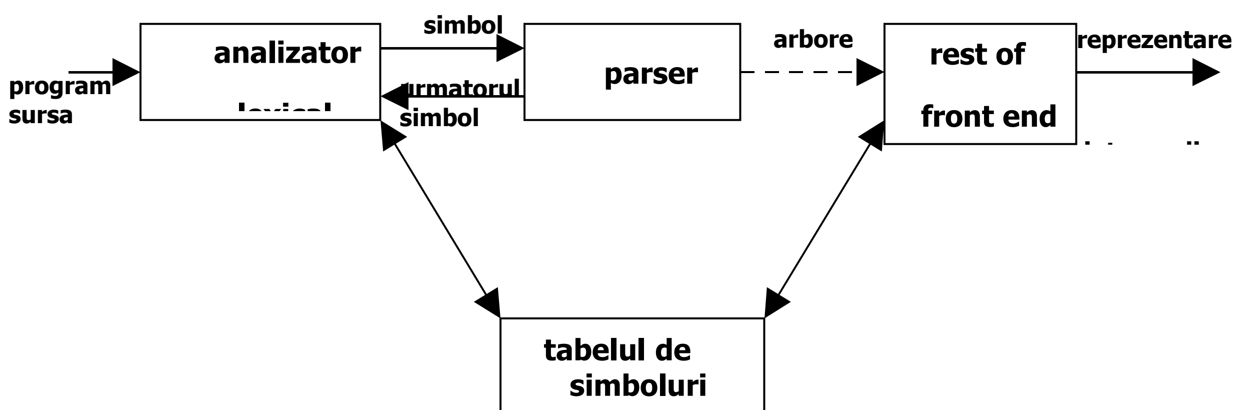
parsat care ar putea trece neobservate in faza initiala a proiectarii limbajului si de compilatorul sau.

O gramatica bine proiectata leaga o structura la un limbaj de programare care este folosit pentru traducerea programelor sursa in cod corect si pentru detectarea erorilor. Sunt disponibile unelte pentru convertirea descriptiei traducerii bazate pe gramatica in programe care functioneaza.

Limbajele evolueaza dupa o perioada de timp, achizitionand noi constructii si realizand noi task-uri. Aceste noi constructii pot fi adaugate mult mai usor la un limbaj cand exista o implementare bazata pe o descriere gramaticala a limbajului.

ROLUL PARSERULUI

In modelul de mai jos de compilator, parser-ul obtine un sir de simboluri de la analizatorul lexical si verifica daca sirul poate fi generat de catre gramatica pentru limbajul sursa. Ne asteptam ca parser-ul sa raporteze orice eroare de sintaxa intr-un mod inteligibil. Ar trebui de asemenea sa revina din erorile frecvent intalnite pentru a putea procesa restul datelor de intrare.



Exista trei tipuri generale de parsere pentru gramatici. Metode universale de parsing ca algoritmul Cocke-Younger-Kasami si algoritmul lui Earley pot parsa orice gramatica. Totusi aceste metode sunt ineficiente in folosirea producerii compilatoarelor. Metodele frecvent folosite in compilatoare sunt clasificate ca fiind top-down ori bottom-

up. Parseri top-down construiesc arbori parseri de sus(root) in jos(leaves), in timp ce parseri bottom-up pornesc de la “leaves” si urca pana la “root”. In ambele cazuri, datele de intrare in parser sunt scante de la stanga la dreapta, cate un simbol odata.

Cele mai eficiente metode top-bottom si bottom-up functioneaza doar asupra claselor de gramatici, dar cateva dintre aceste subclase, cum ar fi gramticile LL si LR, sunt in deajuns de expresive pentru a descrie cele mai sintactice constructii din limbajele de programare. Parserele implementate manual de obicei functioneaza cu gramatici LL. Parserele pentru mai vasta clasa a gramaticilor LR sunt de obicei construite de unelte automate.

In acest capitol, presupunem ca iesirea parserului este o reprezentare a arborului parser pentru valul de simboluri produse de analizatorul lexical, in practica, exista un numar de task-uri care ruleaza in timpul parsingului, cum ar fi colectarea de informatii despre anumite simboluri din tabelul de simboluri, efectuand diverse analize semantice, si generand un cod intermediar. Am inclus toate aceste activitati in casuta “rest of front end”.

GRAMATICI FARA CONTEXT

Multe constructii din limbaje de programare au o structura recursiva fireasca care poate fi definita de catre gramatici fara context. Ca exemplu, putem sa avem o declaratie conditionala definiata astfel : daca S1 si S2 sunt declaratii si E este o expresie, atunci “**if** E **then** S1 **else** S2” este o declaratie.

Aceasta forma de declarare conditionala nu poate fi specificata folosind notatia expresiilor obisnuite. Pe de alta parte, folosind variabila sintactica ‘decl’ pentru a indica clasa de declaratii si ‘expr’ pentru clasa de expresii, putem rescrie declaratia de mai sus folosind producerea gramaticii : decl -> **if** expr **then** decl **else** decl

O gramatica fara context consta in terminale, nonterminale, un simbol de start, si productie.

Terminalele sunt simbolurile de baza din care sunt formate sirurile. In exemplu de mai sus fiecare din cuvintele cheie **if**,**then** si **else** este un terminal.

Nonterminalele sunt variabile sintactice care denota seturi de siruri. Pentru exemplu decl si expr sunt nonterminale. Nonterminalele definesc un seturi de siruri care ajuta la definirea limbajului generat de gramatica.

Intr-o gramatica, un nonterminal este identificat ca simbol de start, si setul de siruri il indica in limbajul definit de gramatica.

Productiile gramaticii specifica modul in care terminalele si nonterminalele pot fi combinate pentru a forma un sir. Fiecare productie consta in nonterminal, urmat de o sageta, urmat de un sir de nonterminale si terminale.

12.COMPILATORUL GCC

Compilatorul GCC (GNU Compiler Collection) este un set de compilatoare dezvoltate pentru mai mai multe limbaje de programare de catre proiectul GNU. GCC este un element cheie al pachetului GNU. Pe langa faptul ca este compilatorul oficial al sistemului GNU, GCC a fost adoptat ca si compilator standard de majoritatea sistemelor moderne bazate pe UNIX, printre ele numarandu-se Linux, familia BSD si Mac OS X.

GCC a fost portat pe o larga varietate de arhitecturi si este distribuit ca si componenta esentila in programele software dezvoltate in GCC. De asemenea GCC este inglobat si in Symbian, Playstation si SegaDreamcast.

Initial a fost numit GNU C Compiler, deoarece suporta doar limbajul de programare C. GCC 1.0 a fost lansat in 1987, si a fost extins la C++ in luna decembrie a aceluias an. Au fost ulterior dezvoltate interfete pentru Fortran, Pascal, Java, Ada, etc.

GCC este distribuit de Free Software Foundation (FSF) sub licenta GNU General Public License (GNU GPL) si sub GNU Lesser General Public License (GNU LGPL). GCC este un software gratuit.

INTERFETE

Interfetele variaza intern. Ele trebuie sa produca arborii care vor fi manipulati de nucleu.

Pana de curand, reprezentarea prin arbori nu a fost prefect independenta de procesorul caruia i-a fost dedicata. S-a creat o confuzie in modul de reprezentare al arnborilor, care diferea de la un limbaj de programare la altul, iar fiecare interfata putea sa ofere proprile reprezentari a arborilor.

In 2005 doua noi fore de arbori independenti de limbaj au fost introdusi. Aceste noi formate de arbori sunt numite GENERIC si GIMPLE.

Interpretarea este acum facuta creand initial arbori dependenti de limbaj si dupa acestia sunt convertiti la GENERIC. Apoi acestia sunt transformati intr-o forma mai putin complexa, bazata pe SSA: GIMPLE, care este limbaj comun pentru majoritatea noilor versiuni ale limbajelor puternice de programare dar si este independenta de arhitectura.

OPTIMIZARE

Optimizarea arborilor nu intra in ceea ce majoritatea programatorilor considera o sarcina a interfetei, deoarece nu este dependenta de limbajul de programare si nu implica intepretarea. Dezvoltatorii GCC-ului i-au dat acestei parti ceva contradictoriu in denumire : middle-end. Acest proces include eliminarea codului mort, eliminarea redundantei partiale, numerotarea globala si inlocuira scalara a agregatelor. Dependenta de vectori si automatizarea vectorizarii sunt dezvoltate in momentul de fata.

NUCLEUL

Comportarea nucleului GCC este specificata in parte de macro-urile pre-procesor si de functiile specifice pentru fiecare arhitectura, careia ii este adresata. Interfata nucelului foloseste acestea pentru a alege in generarea RTL-ului. Chiar daca GCC RTS este independent de procesor, secventa initiala a instructiunilor abstracte este deja adaptata procesorului caruia ii este adresata.

Setul exact de optimizari, variaza de la veriune la versiune, pe masura ce apar tehnici noi care sunt introduse, dar include algoritmii standard: Optimizarea buclelor, a JUMP-urilor, eliminarea subexpresilor, introduce programarea.

O versiune re-incarcata schimba registrii abstracti in registrii reali, specifici masinii, folosindu-se de datele stranse de setul de instructiuni ale masinii considerate. Aceasta este cea mai grea sarcina, deoarece trebuie facuta pentru diferitele tipuri de arhitecturi.

Ultima faza este mai usoara, deoarece sabloanele au fost deja generate in timpul reincarcarii si codul de asamblat este deja scris.

13.Bibliografie

- Addison Wesley - Compilers: Principles, Techniques and Tools – Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
- Addison Wesley - Compiler Design - Formal Syntax And Semantics Of Programming Language (1995)
- Introduction to Parsing - Dick Grune
- Morgan Kaufmann - Engineering A Compiler.pdf
- http://en.wikipedia.org/wiki/GNU_Compiler_Collection
- http://en.wikipedia.org/wiki/Backus-Naur_form

14.PARTICIPARE PE CAPITOLE

Sef de echipa: Florin Diaconeasa

Stefan Poschina – cap. 9

Florin Diaconeasa – cap. 10

Adrian Grigoras – cap. 12

Narcisa Stefu – cap. 1,2,3,4,5

Alexandru Gava – cap. 6,7,8

Alexandru Lupu – cap. 11