Noțiuni generale de funcționare a unui compilator

Modelele, teoria și algoritmii asociați unui compilator, explicate în detaliu de Alfred Aho, Monica Lam, Ravi Sethi, și Jeffrey Ullman, pot fi aplicate unei game largi de probleme în design-ul și dezvoltarea de software. Prin urmare, prezentarea sumară a etapelor parcurse de-a lungul întregului proces de compilare poate furniza o perspectivă de ansamblu asupra conceptelor și contextului în care au fost utilizate și a modului de funcționare al acestora. Analiza sintactică, generarea de cod intermediar și tabela de simboluri constituie pilonii procesării de text și generării de cod pentru întrebările axate pe înțelegerea fluxului programelor din prezentul proiect.[[1]](#footnote-1)

Etapele procesului de compilare

**Faza de analiză** a unui compilator constă în descompunerea codului sursă în părțile constitutive ale limbajului de programare, impunerea unei structuri gramaticale și crearea unei reprezentări intermediare asociate codului.

O structură gramaticală implică prezența unor reguli de sintaxă specifice limbajului. **Sintaxa** unui limbaj de programare descrie forma adecvată a programelor sale, în timp ce **semantica** limbajului definește sensul acestora, mai precis, ce face fiecare program atunci când este executat. Pentru specificarea sintaxei se folosește o notație utilizată pe scară largă, numită gramatică independentă de context[[2]](#footnote-2).

Sarcina principală a **analizatorului lexical** este să citească caracterele codului sursă, să le grupeze în lexeme (șiruri de caractere) și să producă o secvență de tokeni formați din tipul lexemului și opțional, un pointer către intrarea din tabela de simboluri. Atunci când analizatorul lexical descoperă un lexem care constituie un identificator (nume de constante, variabile sau funcții), introduce acel lexem în tabelă. Tokenii sunt trimiși către parsator pentru analiza de sintaxă[[3]](#footnote-3).

**Parsatorul** obține un șir de tokeni de la analizatorul lexical și verifică faptul că șirul de tokeni poate fi generat de gramatica limbajului. Parsatorul va raporta orice erori de sintaxă într-un mod inteligibil. Pentru programele bine formate, analizatorul construiește un arbore de parsare și îl transmite restului compilatorului pentru procesare ulterioară.

Fazele ulterioare ale compilatorului trebuie să analizeze rezultatul parsatorului, pentru a asigura conformitatea programului cu regulile care nu sunt verificate de parsator[[4]](#footnote-4).

O gramatică pentru C sau Java nu distinge între identificatorii care sunt șiruri de caractere diferite. În schimb, toți identificatorii sunt reprezentați printr-un simbol precum id în gramatică. Într-un compilator pentru un astfel de limbaj, faza de **analiză semantică** verifică dacă identificatorii sunt declarați înainte de a fi utilizați[[5]](#footnote-5).

Dacă partea de analiză detectează pe parcursul **verificării statice** (verificare făcută de compilator) incorectitudinea codului sursă din punct de vedere sintactic sau semantic (nu sunt urmate regulile sintactice și semantice ale limbajului în care a fost scris), atunci trebuie să furnizeze mesaje informative, astfel încât utilizatorul să poată lua măsuri corective.

**Verificarea sintactică** nu verifică doar dacă programul respectă gramatica limbajului. De exemplu, constrângerile precum declararea unui identificator cel mult o dată într-un domeniu de valabilitate, sau încadrarea obligatorie a instrucțiunii break într-o buclă sunt de natură sintactică, deși nu sunt impuse de o gramatică utilizată pentru parsare.

**Verificările de tip** ale unui limbaj asigură că operatorii sau funcțiile sunt aplicate numărului și tipului corect de operanzi. Dacă este necesară conversia între tipuri (de exemplu, atunci când un număr de tip întreg este adăugat la un număr de tip float), atunci verificatorul de tip poate insera un operator în arborele de sintaxă pentru a reprezenta acea conversie[[6]](#footnote-6).

În detectarea erorilor, au un rol esențial atât gramatica limbajului cât și tabela de simboluri[[7]](#footnote-7).

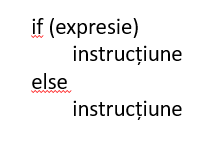
Pe parcursul etapei de analiză sunt colectate și stocate informații despre programul sursă într-o structură de date numită **tabelă de simboluri**, care este transmisă împreună cu reprezentarea intermediară către etapa de sinteză[[8]](#footnote-8).

**Faza de sinteză** translatează programul intermediar în programul țintă, utilizând informațiile din tabela de simboluri.

Partea de analiză este adesea numită partea de frontend a compilatorului, iar partea de sinteză este numită partea de backend.

Analiza sintactică

Implicit, prin design, fiecare limbaj de programare are reguli precise care descriu structura sintactică a programelor corecte (de exemplu, în C, un program este format din funcții, o funcție din declarații și instrucțiuni, o instrucțiune din expresii și așa mai departe)[[9]](#footnote-9). Structură ierarhică a elementelor constitutive ale unui limbaj de programare este descrisă de o gramatică. De exemplu, o instrucțiune if-else poate avea forma



Această instrucțiune este reprezentată de concatenarea cuvântului cheie if, o paranteză deschisă, o expresie, o paranteză închisă, o instrucțiune, cuvântul cheie else și o altă instrucțiune. Folosind variabila expr pentru a desemna o expresie și variabila stmt pentru a desemna o instrucțiune, această regulă de poate fi exprimată sub forma



O astfel de regulă se numește o regulă de producție. Într-o regulă de producție, elementele lexicale precum cuvântul cheie if și parantezele sunt numite terminale. Variabilele precum expr și stmt sunt numite neterminale[[10]](#footnote-10).

O **gramatică independentă de context** are patru componente:

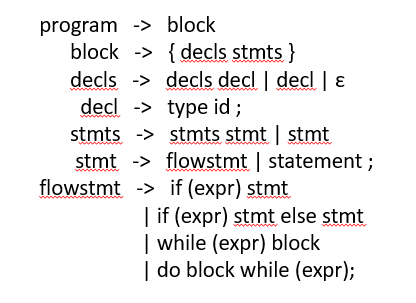
1. Un set de simboluri terminale, reprezentate de simbolurile elementare ale limbajului definit de gramatică

2. Un set de neterminale, unde un neterminal reprezintă un șir de terminale

3. Un set de reguli de producție, unde fiecare regulă de producție constă dintr-un neterminal (partea stângă a regulii de producție), o săgeată și o secvență de terminale și/sau neterminale (partea dreaptă a regulii de producție)

4. Un simbol de start desemnat din rândul neterminalelor

O specificare a unei gramatici se face prin enumerarea regulilor de producție, începând cu regula de producție a simbolului de start. Cifrele, semnele (de exemplu, <, >, =) și cuvintele cheie (de exemplu, while) sunt terminale. Regulile de producție asociate aceluiași neterminal pot fi grupate prin simbolul |, cunoscut în mod convențional drept ,,sau’’[[11]](#footnote-11).



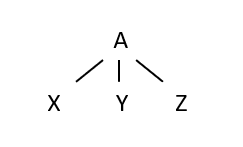
//poza cu gramatica aia simpla pe care am definit-o, gramatica unui limbaj primitiv de programare gramatica din fig genereaza un set limitat de instr

**Derivarea** unor șiruri de caractere pe baza unei gramatici se face începând cu simbolul de start și înlocuind în mod repetat un neterminal din regula de producție a simbolului cu corpul unei reguli de producție asociate acelui neterminal (un neterminal poate avea mai multe reguli de producție asociate). Șirurile terminale care pot fi derivate din simbolul de start formează limbajul definit de gramatică[[12]](#footnote-12).

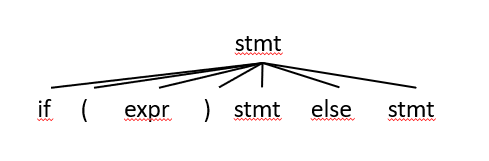
**Parsarea (analiza sintactică)** este definită de aflarea modului în care un șir de terminale este derivat din simbolul de start al gramaticii, iar în cazul în care șirul de terminale nu poate fi derivat din simbolul de start, de raportarea erorilor de sintaxă.

Un **arbore de parsare (arbore de sintaxă abstract)** este o imagine a unei derivări, aratând cum simbolul de start al unei gramatici derivă un șir în limbajul definit de gramatică. Fiecare nod dintr-un arbore de sintaxă reprezintă un element constitutiv al limbajului[[13]](#footnote-13).

Dacă neterminalul A are asociată o regulă de producție A -> XYZ, arborele de parsare are un nod interior etichetat cu A cu trei copii etichetați X, Y și Z, de la stânga la dreapta:



sau as putea face cu if-else ca sa am consistenta un arbore de sintaxa pentru gramatica



Având în vedere o gramatică independentă de context, un arbore de parsare este un arbore cu următoarele proprietăți:

1. Rădăcina este simbolul de start;

2. Fiecare frunză este un terminal;

3. Fiecare nod interior este un neterminal[[14]](#footnote-14).

Construcția de sus în jos a unui arbore de parsare ca cel din Fig. 2.17, se face pornind de la rădăcină, și efectuând în mod repetat următorii doi pași:

1. Pentru nodul N se va selecta una dintre regulile de producție asociate și vor fi construiți copiii pentru simbolurile din corpul regulii de producție;

2. Se va găsi următorul nod la care urmează să fie construit un subarbore (de obicei, cel mai din stânga neterminal neexpandat al arborelui)[[15]](#footnote-15).

Generarea de cod intermediar

Ideea de a asocia diverse proprietăți cu părțile constitutive ale unui limbaj de programare (de exemplu, valori și tipuri cu expresii) poate fi exprimată în cadrul gramaticilor. Neterminalele și terminalele pot avea asociate **atribute sintetizate**, iar regulile de producție pot avea asociate **reguli semantice** care descriu modul în care atributele sunt calculate la nivelul nodurilor arborelui de parsare unde acea regulă de producție este folosită.

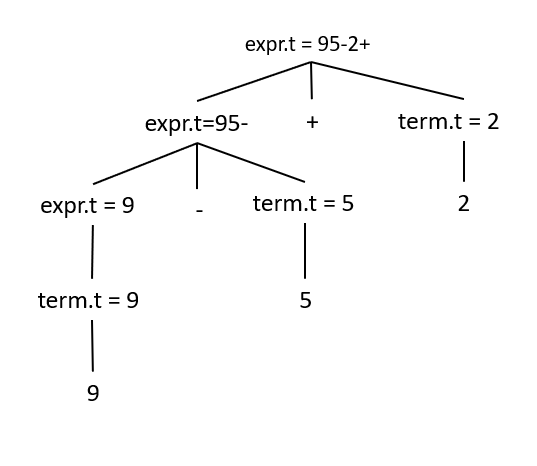
Reprezentarea intermediară a codului se realizează cu ajutorul arborilor de sintaxă abstractă ce modelează structura sintactică ierarhică a codului sursă[[16]](#footnote-16). Odată ce arborele de sintaxă este construit, codul intermediar poate fi generat prin evaluarea atributelor și executarea fragmentelor de cod asociate nodurilor din arbore[[17]](#footnote-17).

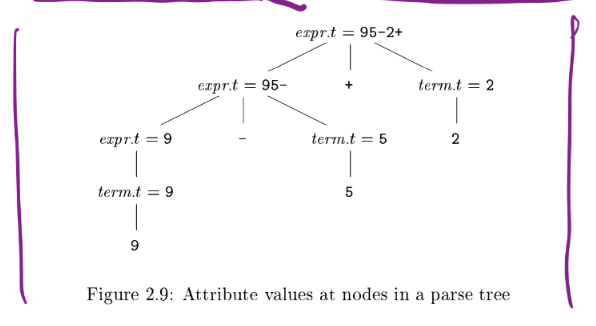
Se spune că un atribut este sintetizat dacă valoarea lui la un nod al arborelui analizat N este determinată de valorile atributelor copiilor lui N și N însuși. Atributele sintetizate au proprietatea că pot fi evaluate în timpul unei singure traversări de jos în sus a unui arbore de parsare[[18]](#footnote-18).

Fragmentele de cod aferente regulilor semantice sunt executate atunci când regula de producție este utilizată în timpul analizei de sintaxă și realizează **translatarea direcționată de sintaxă**[[19]](#footnote-19). Procesul de evaluare a valorii atributului, se numește **definiție direcționată de sintaxă**.

Pentru un șir de intrare dat, este construit arborele de parsare, iar apoi sunt aplicate regulile semantice pentru a evalua atributele fiecărui nod din arbore.

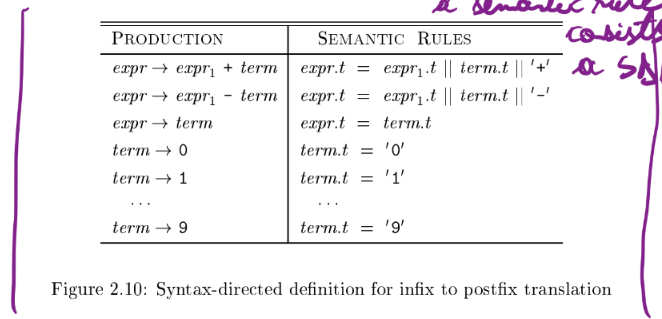
De exemplu, Fig. 2.9 prezintă un arbore de parsare adnotat pentru 9-5+2. Valoarea 95-2+ a atributului de la rădăcină este notația postfixă pentru 9-5+2.





Fiecare neterminal (expr, term) are un atribut t sub forma unui șir de caractere care reprezintă notația postfixă pentru expresia generată de acel neterminal. Simbolul || din regula semantică este operatorul utilizat în mod convențional pentru concatenarea șirurilor.

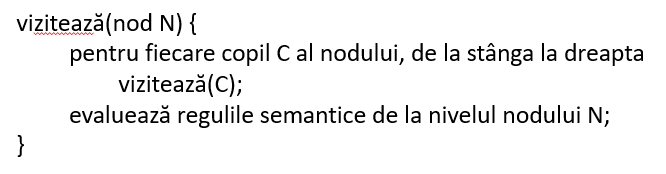
|  |  |
| --- | --- |
| REGULA DE PRODUCȚIE | REGULA SEMANTICĂ |
| expr -> expr1 + term | expr.t = expr1.t || term.t || ’+’ |
| expr -> expr1 - term | expr.t = expr1.t || term.t || ’-’ |
| expr -> term | expr.t = term.t |
| term -> 0 | term.t = ’0’ |
| term -> 1 | term.t = ’1’ |
| ... | ... |
| term -> 9 | term.t = ’9’ |

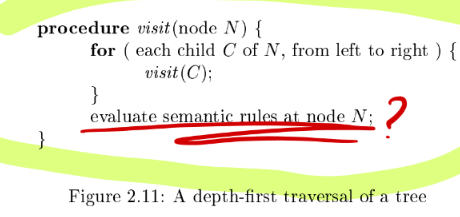


Forma postfixă a unei cifre este cifra însăși. De exemplu, regula semantică asociată cu regula de producție term -> 9 face ca term->t să fie 9 ori de câte ori această regulă de producție este utilizată la nivelul unui nod din arborele de parsare. Celelalte cifre sunt translatate similar. Când regula de producție expr -> expr1 + term este aplicată, valoarea term->t preia valoarea expr->t[[20]](#footnote-20).

Definiția direcționată de sintaxă din exemplul precedent are următoarea proprietate importantă: șirul care reprezintă translația neterminalului de la începutul fiecărei reguli de producție este concatenarea translațiilor neterminalelor din corpul regulii de producție, în aceeași ordine ca și în regulă. O definiție direcționată de sintaxă cu această proprietate este denumită simplă.

Translatarea expr->t este concatenarea translatărilor lui expr1 și term1, urmată de simbolul +. expr1 și term apar în aceeași ordine atât în ​​corpul regulii de producție, cât și în regula semantică. Nu există simboluri suplimentare înainte sau între translatări. În acest exemplu, singurul simbol suplimentar apare la sfârșit[[21]](#footnote-21).

Parcurgerile de arbori sunt folosite pentru specificarea execuției fragmentelor de cod în cadrul translatării și evaluarea atributelor. O traversare a unui arbore începe de la rădăcină și vizitează fiecare nod al arborelui într-o anumită ordine. Procedura visit(N) din Fig. 2.11 reprezintă o traversare în adâncime care vizitează copiii unui nod de la stânga la dreapta, așa cum se arată în Fig. 2.12. Pe parcursul traversării fiecărui nod, este evaluat la final atributul (după ce cu siguranță au fost evaluate atributele copiilor). 



O definiție direcționată de sintaxă nu impune nicio ordine specifică pentru evaluarea atributelor dintr-un arbore de parsare. Orice ordine de evaluare care calculează un atribut după toate celelalte atribute de care depinde este acceptată. Atributele sintetizate pot fi evaluate în timpul oricărei traversări de jos în sus, adică traversarea care evaluează atributele unui nod după ce au fost evaluate atributele copiilor săi.[[22]](#footnote-22)

Un element constitutiv poate avea cel mult un atribut asociat. În cazul identificatorilor (numelor de constante, variabile sau funcții) valoarea atributului este un pointer către intrarea din tabela de simboluri aferentă acestuia care permite accesul la mai multe informații[[23]](#footnote-23).

Tabela de simboluri

Tabelele de simboluri sunt structuri de date care sunt utilizate de compilatoare pentru a păstra informații despre identificatori (nume de constante, variabile și funcții). Informațiile sunt colectate în mod incremental de fazele de analiză ale unui compilator și utilizate de fazele de sinteză pentru a genera codul țintă. Intrările din tabela de simboluri conțin informații despre identificatori, cum ar fi șirul de caractere asociat(lexemul), tipul lor, adresa lor din memorie și orice alte informații relevante. În cazul procedurilor, sunt stocate informații precum numărul și tipurile argumentelor, metoda de transmitere a fiecărui argument (de exemplu, prin valoare sau prin referință) și tipul returnat.\*aia de la inceput

Tabelele de simboluri trebuie de obicei să accepte mai multe declarații ale aceluiași identificator în cadrul unui program (din domenii de valabilitate diferite)[[24]](#footnote-24).

În consecință, rolul unei tabele de simboluri este de a transmite informații de la declarațiile variabilelor către utilizarea acestora. O acțiune semantică pune informații despre identificator în tabela de simboluri, atunci când este analizată declarația acestuia. Ulterior, o acțiune semantică asociată cu o regulă de producție obține informații despre identificator din tabela de simboluri[[25]](#footnote-25).

1. Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, Boston, USA, 2006 [↑](#footnote-ref-1)
2. Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, Boston, USA, 2006, p. 40 [↑](#footnote-ref-2)
3. Ibidem, p. 109 [↑](#footnote-ref-3)
4. Ibidem, p. 209 [↑](#footnote-ref-4)
5. Ibidem, p. 216 [↑](#footnote-ref-5)
6. Ibidem, p. 97 [↑](#footnote-ref-6)
7. Ibidem, p. 92 [↑](#footnote-ref-7)
8. Ibidem, p. 4 [↑](#footnote-ref-8)
9. Ibidem, p. 191 [↑](#footnote-ref-9)
10. Ibidem, p. 42 [↑](#footnote-ref-10)
11. Ibidem, p. 43 [↑](#footnote-ref-11)
12. Ibidem, p. 44 [↑](#footnote-ref-12)
13. Ibidem, p. 353 [↑](#footnote-ref-13)
14. Ibidem, p. 45 [↑](#footnote-ref-14)
15. Ibidem, p. 61 [↑](#footnote-ref-15)
16. Ibidem, p. 41 [↑](#footnote-ref-16)
17. Ibidem, p. 99 [↑](#footnote-ref-17)
18. Ibidem, p. 54 [↑](#footnote-ref-18)
19. Ibidem, p. 53 [↑](#footnote-ref-19)
20. Ibidem, p. 55 [↑](#footnote-ref-20)
21. Ibidem, p. 56 [↑](#footnote-ref-21)
22. Ibidem, p. 57 [↑](#footnote-ref-22)
23. Ibidem, p. 112 [↑](#footnote-ref-23)
24. Ibidem, p. 85 [↑](#footnote-ref-24)
25. Ibidem, p. 89 [↑](#footnote-ref-25)