Compilatoare

Rolul unui compilator este de a traduce un program (sursa) scris intr-un limbaj sursa intr-un program obiect scris intr-un limbaj obiect.

Programul sursa este de obicei scris intr-un limbaj de nivel inalt (mai apropiat de limbajul natural) si el este tradus intr-un limbaj mai apropiat de limbajul masina. De exemplu, se traduce din C in limbaj de asamblare sau din limbaj de asamblare in limbaj masina. Inainte de a se face traducerea efectiva se face o analiza (sintactica si lexicala) a codului sursa (daca acesta este corect scris), rezultand eventual erori sau mesaje de atentionare.

Limbaje de nivel inalt:

1. **Limbaje imperative**: Algol (60, 68), Fortran, COBOL, Pascal, Ada, Modula 2, C etc. Aceste limbaje sunt concepute pentru computere clasice caracterizate prin: **CPU** (procesor), **memorie** si **bus**, care face legatura intre CPU si memorie.
2. **Limbaje pentru programare functionala**: LISP, HOPE, Miranda, Haskell si FP. Acestea sunt caracterizate prin:
   - nu se face distinctie intre declaratii si expresii
   - denumirile sunt folosite pentru a identifica expresii si functii, ele nefiind folosite pentru a identifica locatii de memorie
   - functiile sunt folosite numai ca argumente sau ca valori returnate de functii.

   Principiul ce sta la baza acestor limbaje este **reducerea** (evaluarea unui program functional se face inlocuind succesiv expresii cu expresii mai simple pana cand se obtine forma normala).
3. **Limbaje pentru programare logica**: Prolog. Principiul ce sta la baza lor este calculul predicatelor, iar mecanismul de executie este **rezolutia** (demonstrarea implicatiilor din calculul predicatelor de ordin 1).
4. **Limbaje orientate pe obiecte**: Pascal, C++, Java, C# etc. Sunt limbaje evoluate de tip imperativ.

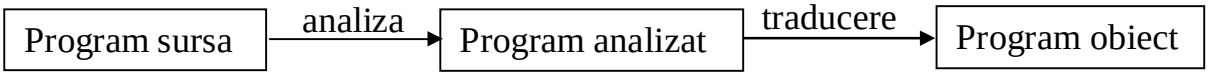
Pe langa aceste 4 clase principale de limbaje de nivel inalt mai exista altele pentru aplicatii particulare:

- limbaje folosite pentru descrierea hardware-ului
- limbaje scrise pentru sisteme de operare (comenzi)
- limbaje folosite pentru texte formatate sau pentru formate grafice

Activitatea desfasurata de orice compilator este impartita in mai multe faze (etape) de compilare. Programul sursa este divizat de catre compilator intr-o multime de fragmente. Fiecare fragment de cod sursa este analizat si tradus intr-un fragment intr-un alt limbaj. Daca analiza si traducerea codului sursa se face printr-o singura

parcurgere a codului sursa, se spune ca avem un compilator **single-pass** (fiecare fragment de cod sursa este trecut prin toate etapele compilarii). Exista compilatoare care necesita mai multe parcurgeri ale codului sursa. Acestea se numesc compilatoare **multi-pass**. Orice compilator este **multi-phase**, dar poate fi single-pass sau multi-pass. De exemplu, pentru limbajul ALGOL 60 este dificil de scris compilatoare single-pass. Un compilator multi-pass nu este neaparat mai lent decat unul single-pass, pentru ca amandoua trebuie sa treaca prin (in general) aceleasi faze de compilare. Totusi, un compilator multi-pass foloseste de obicei mai multa memorie, dar nici acest lucru nu mai reprezinta in prezent un dezavantaj.

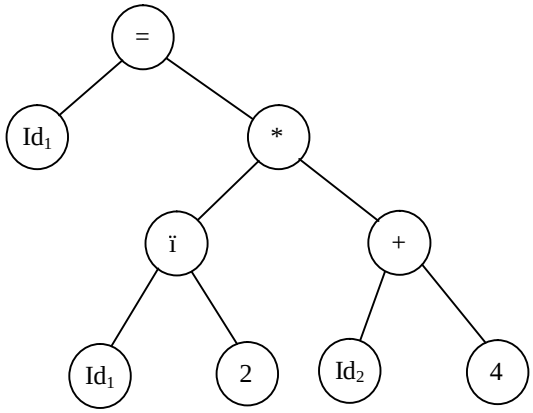
Orice compilator trece un program prin urmatoarele etape principale:



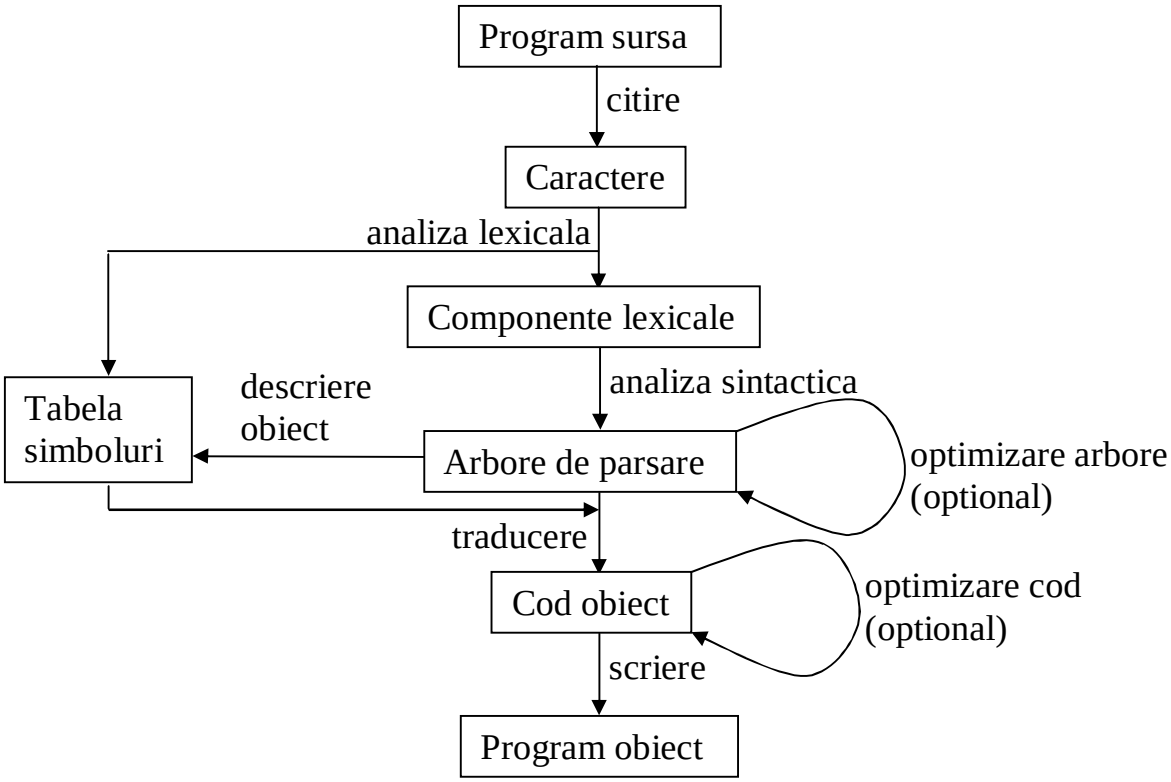
Compilatorul construieste:

1. Un arbore rezultat in urma analizarii gramaticale, care indica structura programului, modul in care se vor imbina fragmentele de program pentru a rezulta un fragment mai mare
2. O tabela de simboluri (token-i) ce apar in program

Exemplu: x = (x-2)*(y+4)



In continuare vom detalia etapele compilarii:



Analiza lexicala, tabela de simboluri

In urma analizei lexicale sirul de caractere citit este divizat intr-o multime de componente lexicale. O parte dintre aceste componente sunt introduse in tabela de simboluri (de exemplu: denumiri de variabile, functii, proceduri, tipuri de date, etichete etc.). De obicei, fiecarui token din tabela i se asociaza un cod si o descriere. Referirea in arborele de parsare la un token din tabela de simboluri se face pe baza acestui cod sau pe baza unui pointer la tabela.

Analiza sintactica, arbore de parsare

Secventa de token-i obtinuti din faza analiza lexicala este analizata si se determina modul de asociere a acestora in vederea obtinerii de fragmente de fraze. De asemenea, se determina modul in care aceste fragmente de fraze se combina pentru a rezulta fragmente mai mari de fraze sau fraze. Se construieste astfel un asa numit arbore de parsare care, ulterior, parcurs, conduce la programul obiect.

Traducerea

Este etapa in care arborele de parsare este parcurs si este tradus in limbajul obiect. In urma parcurgerii arborelui, frazele rezultate sunt scrise in limbajul obiect.

Traducerea se efectueaza in momentul in care programul sursa ña fost intelesò, ideea programului este practic exprimata intr-un alt limbaj (obiect).

Optimizari

Optional, anumite compilatoare au implementate si faze de optimizare.

Exemple:

- se elimina sevente de cod la care nu se poate ajunge cu executia programului
- expresii care nu influenteaza contextul programului (de ex. x = x;)
- variabile care nu mai sunt utilizate sunt eliminate.

Cu toate ca optimizarile sunt optionale in procesul de compilare, ele sunt foarte importante pentru ca duc la marirea eficientei codului obiect si, implicit, la marirea vitezei de executie si micsorarea spatiului de memorare (program obiect mai scurt si memorie mai putina necesara in momentul executiei).

Erori de compilare

Cand scriem programe (sursa) este inevitabil sa facem greseli. De aceea este foarte important ca sa fim ajutati de catre compilator pentru a le elimina.

Exista doua categorii de erori pe care le genereaza un compilator:

1. **Erori din timpul compilarii**. Compilatorul trebuie sa furnizeze o lista de erori programatorului in momentul in care un program sursa nu poate fi compilat. Erorile trebuie sa fie insotite de explicatii, din care programatorul sa inteleaga ce a gresit, precum si sa fie date sugestii care ar putea duce la corectarea lor. Unele compilatoare furnizeaza si asa numitele mesaje de **warning** (atentionare) care atrag atentia programatorului asupra unei secvente de cod care, desi este corecta d.p.d.v. al compilarii, ar putea conduce la rezultate neasteptate de catre programator. De exemplu, in C, cand se face o atribuire intr-o conditie, adesea programatorul de fapt uita sa dubleze semnul = (egal), care inseamna o comparatie a doi termeni daca sunt egali si nu atribuire.

2. **Erori din timpul executiei (run-time errors)**. Desi unele programe sunt compilate (sunt corecte d.p.d.v. lexical si sintactic) rezultand codul obiect, in timpul executiei se pot detecta erori. De exemplu, se ajunge in timpul executiei la operatii matematice imposibile (impartire prin zero, logaritm cu baza sau argument nepozitiv etc.), sau se refera incorect un element dintr-un vector (rezultand accesari eronate de casute de memorie) etc. Pentru acest tip de erori, in

timpul compilarii, se genereaza in codul obiect secvente care ajuta la tratarea lor (mesaje, parasire program etc.).

Pentru a implementa etapele de analiza lexicala si sintactica avem nevoie de cunostiinte teoretice legate de limbaje formale si automate. De obicei, etapa de analiza lexicala se descrie cu ajutorul **expresiilor regulate**, iar etapa de analiza sintactica cu ajutorul **gramaticilor** (independente de context).