Obiectul şi utilitatea disciplinei

Tehnicile de compilare sunt tehnici de programare specializate utilizate, în primul rând, la scrierea programelor de translatare dar, în acelaşi timp, aplicabile la realizarea unei game de programe similare translatoarelor. Se ştie că **translatorul** este un program care verifică programele scrise de utilizator într-un anumit limbaj de programare şi le „traduce” în programe echivalente exprimate într-un limbaj accesibil calculatorului (de exemplu cod maşină). Exponentul cel mai cunoscut al familiei programelor de translatare este **compilatorul**, ceea ce justifică şi denumirea disciplinei.

Istoria scrierii compilatoarelor a debutat în 1957, când a fost realizat primul compilator pentru limbajul de programare Fortran. Performanța deosebită pentru vremea aceea, aparține unei echipe de cercetători americani, de la firma I.B.M, condusă de John Backus. Implementarea compilatorului a durat circa 18 luni ceea ce poate să pară mult astăzi când compilatoare pentru limbaje de programare comparabile cu Fortran sau chiar mai complexe (Pascal, C, Ada, Java) se pot scrie într-un timp mult mai scurt. Principalii factori care au contribuit la această evoluție sunt:

- Definirea ştiinţifică a noilor limbaje de programare prin includerea unor concepte de bază comune şi respectarea riguroasă a anumitor principii care uşurează verificarea şi translatarea programelor.

- Elaborarea, între timp, a unei teorii a compilării bine fundamentată matematic prin Algebra superioară, Logica Matematică şi în special prin **Teoria limbajelor formale**.

- Apariția şi perfecționarea unor unelte software specializate pentru proiectarea şi scrierea compilatoarelor.

Astăzi nu este posibil să se scrie programe de translatare competitive fără cunoaşterea teoriei compilării şi ignorând aplicarea tehnicilor, algoritmilor şi metodelor specifice. Mai mult, cel care se ocupă de implementarea unui limbaj de programare trebuie să aibă solide cunoştințe atât din domeniul general al ştiinței calculatoarelor (arhitectură, cod maşină, echipamente periferice) cât şi o privire corectă şi completă asupra întregului domeniu al limbajelor de programare. Din acest motiv, disciplina Tehnici de compilare este logic legată de disciplina *Conceptele fundamentale ale limbajelor de programar*e în care s-au introdus deja o serie de cunoştințe, concepte şi noțiuni care vor fi utilizate şi dezvoltate în continuare: limbaje de programare, simbolisme pentru reprezentarea lor, gramatica unui limbaj, arbore sintactic, probleme legate de implementarea diferitelor concepte şi mecanisme specifice limbajelor de programare etc.

Conținutul disciplinei Tehnici de compilare este util şi pentru realizarea altor tipuri de programe decât cele din familia compilatoarelor: Sisteme de operare, Sisteme de gestiune a bazelor de date, Editoare de texte, toată gama de Programe utilitare sau chiar simple aplicații care presupun, ca interfață cu utilizatorul, un limbaj de comunicare (comandă). Cu cât acest limbaj este mai complicat, cu atât este mai util să stăpânim astfel de tehnici şi metode.

Aceste cunoştințe sunt importante chiar din faza definirii şi proiectării limbajului. Aplicate corect, ele pot conduce la simplificarea limbajului de comunicare şi, implicit, la eficientizarea procesului de traducere.

- **Se poate concluziona că** teoria compilării oferă tehnici şi metode avansate de programare utile într-o gamă largă de aplicații în care comunică două sisteme, cazul cel mai frecvent fiind acela în care cele două sisteme sunt omul şi respectiv calculatorul.

Funcțiile unui compilator

Simplu exprimat, un **compilator** este un translator care citeşte un program scris într-un limbaj de nivel înalt (limbajul sursă) şi îl traduce într-un limbaj echivalent în alt limbaj, de nivel mai scăzut (limbajul obiect sau destinație).

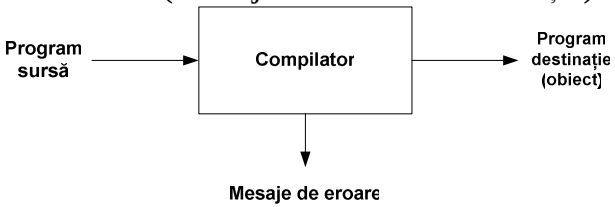

Fig. 1.1 Compilatorul

În mod frecvent limbajul destinație este limbaj maşină sau un limbaj apropiat de acesta. Ca o parte importantă a procesului de traducere, compilatorul semnalează utilizatorului prezența erorilor în programul sursă.

Sarcinile de bază pe care trebuie să le îndeplinească orice compilator sunt în esență aceleaşi. Înțelegând aceste sarcini se pot construi compilatoare pentru o mare varietate de limbaje sursă şi calculatoare destinație folosind aceleaşi tehnici de bază.

Modelul analiză-sinteză al compilării

Funcțiile de principiu ale unui compilator sunt:

- Analiza programului sursă;
- Sinteza programului destinație (obiect);

Corespunzător acestor funcții, compilatoarele se compun din două părți..

- Partea de analiză desface programul sursă în componenetele sale de bază şi crează o reprezentare intermediară a programului sursă.
- Partea de sinteză construieşte programul destinație din această reprezentare intermediară.

Structura unui compilator

Datorită complexității sale deosebite, procesul de compilare se descompune în mai multe **faze**, fiecare fază transformând programul sursă dintr-o reprezentare în alta.

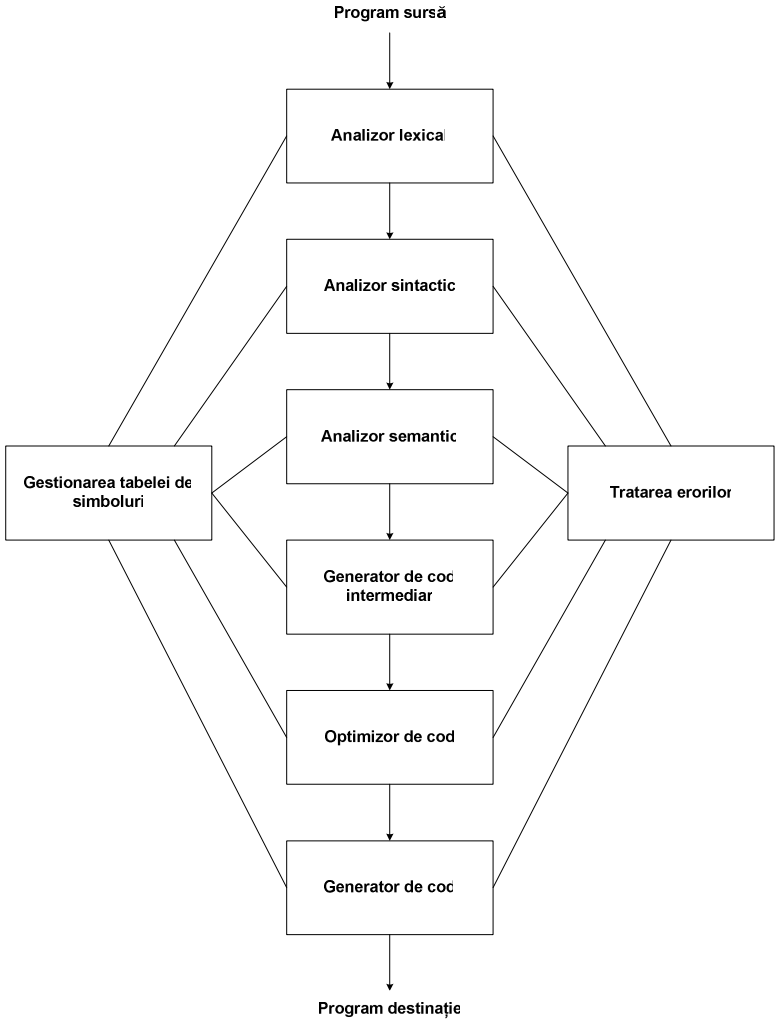

Fig. 1.2. Fazele unui compilator

Gestionarea tabelei de simboluri

O funcție esențială a compilatorului este să înregistreze identificatorii utilizați în programul sursă şi să colecționeze informații despre diferitele atribute ale fiecărui identificator. Aceste atribute se pot referi la memoria alocată pentru un identificator, tipul său, domeniul de valabilitate şi, în cazul numelor de proceduri, informații ca numărul şi tipul argumentelor, metoda de transmitere a fiecărui argument iar, pentru funcții, şi tipul valorii returnate.

O **tabelă de simboluri** (TS) este o structură de date care conține o înregistrare pentru fiecare identificator, cu câmpuri pentru atributele identificatorului. Structura de date trebuie să permită regăsirea rapidă a înregistrării pentru fiecare identificator. Crearea înregistrării unui identificator în TS se face la analiza lexicală sau la cea sintactică. Informațiile corespunzătoare atributelor se introduc ulterior, pe măsură ce se avansează în procesul de compilare. Aceste informații se utilizează pentru a efectua verificările semantice (de exemplu verificările de tip) precum şi pentru generarea corectă a codului obiect.

Detectarea şi semnalarea erorilor

Fiecare fază de compilare poate descoperi erori. Reacția compilatorelor la erori poate fi diferită:

a) Oprirea compilării la prima eroare, semnalarea şi corectarea ei în programul sursă, urmată de reluarea compilării de la început.
b) „Tratarea” erorilor detectate astfel încât compilarea să poată continua, permițând să se detecteze şi alte erori care vor fi corectate în mod global, la sfârşit. Acest mod de ”tratare” se numeşte *revenirea din eroare*.

Cea mai mare parte dintre erorile de compilare sunt detectate în fazele de analiză sintactică şi semantică.

Obiectul fazelor de analiză a programului sursă

Partea de analiză a procesului de compilare constă, la rândul ei din 3 faze:

1. **Analiza lexicală** sau **liniară**

Şirul de caractere formând programul sursă este citit de la stânga la dreapta şi grupat în **atomi** sau *simboluri lexicale* care sunt secvențe de caractere având o semnificație comună. De exemplu, în instrucțiunea de atribuire *a*:=*b*+*c**10* sunt localizați următorii atomi:

- identificatorii *a*,*b* şi *c*;
- simbolurile *:=* , + şi *;
- numărul întreg *10*.

Tot pe parcursul acestei faze se elimină spațiile dintre atomi iar atomii sunt codificați numeric pentru a uşura prelucrarea lor în continuare.

2. **Analiza sintactică** sau **ierarhică**

Atomii sunt grupați ierarhic în colecții mai mari (expresii, declarații, instrucțiuni) cu o semnificație comună, reprezentând *propozițiile gramaticale* ale programului sursă.

În timpul analizei sintactice, operațiile indicate în programul sursă sunt determinate şi înregistrate într-o structură ierarhică numită arbore. Deseori se utilizează un tip special de arbore numit **arbore sintactic**, în care fiecare nod reprezintă o operație iar fiii unui nod sunt argumentele operației (fig. 1.3).

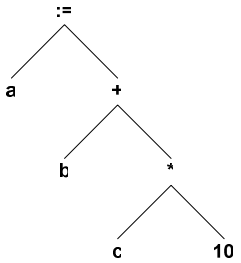

Fig. 1.3. Arbore sintactic pentru a:=b+c*10

Structura ierarhică a unui program este, în multe cazuri, exprimată prin reguli recursive. De exemplu, sunt binecunoscute regulile recursive de definire a expresiilor.

Similar, într-o mare categorie de limbaje, şi instrucțiunile sunt definite prin reguli recursive.

Împărțirea în analiză lexicală şi sintactică este oarecum arbitrară. De obicei se alege o împărțire care simplifică sarcina totală a analizei. Astfel, numerele, sirurile de caractere, cuvintele (identificatorii) semnele de punctuație sunt considerate simboluri lexicale în timp ce expresiile, instrucțiunile, declarațiile sunt construcții sintactice.

3. Analiza semantică sau **contextuală**

Efectuează verificările care țin de sensul (înțelesul) componentelor programului şi culege informații de tip pentru faza următoare: generarea de cod. Ea utilizează structura ierarhică determinată de faza de analiză sintactică pentru a identifica operatorii şi operanzii expresiilor şi instrucțiunilor.

O componentă importantă a analizei semantice este **verificarea tipurilor**. Această componentă verifică dacă fiecare operator are operanzi permişi de specificarea limbajului sursă. De exemplu, definițiile multor limbaje de programare consideră că este eroare atunci când un număr real este utilizat pentru a indexa un tablou.

O altă parte distinctă o reprezintă **analiza de domeniu** adică verificarea utilizării fiecărui identificator strict numai în domeniul său de vizibilitate.

Obiectul fazelor de sinteză a programului destinație

4. Generarea codului intermediar

După analizele sintactică şi semantică, unele compilatoare generează o reprezentare intermediară explicită a textului sursă. Se poate concepe această reprezentare intermediară ca un program pentru un calculator abstract.

Există diferite forme de reprezentaări intermediare. O formă des utilizată este aşa numitul **cod cu 3 adrese** care seamănă cu limbajul de asamblare pentru un calculator în care fiecare locație de memorie poate juca rol de registru. Codul cu 3 adrese constă dintr-o secvență de instrucțiuni, fiecare având cel mult trei operanzi. Această formă intermediară are câteva proprietăți:

a) Fiecare instrucțiune cu 3 adrese are cel mult un operator pe lângă cel de atribuire; în prealabil, compilatorul trebuie să decidă asupra ordinei în care se execută operațiile (prioritatea operatorilor);
b) Pentru a păstra valorile calculate în fiecare instrucțiune, compilatorul trebuie să genereze variabile temporare (variabile create de compilator fără corespondență directă în textul sursă);
c) Pot exista instrucțiuni cu 3 adrese cu mai puțin de 3 operanzi.

5. Optimizarea codului

Scopul acestei faze este acela de a îmbunătății codul intermediar, astfel încât să rezulte cod maşină mai rapid. În acest sens se acționează pentru eliminarea redundanțelor, a calculelor şi variabilelor inutile. Există mari diferențe între optimizarea de cod care se face în diferite compilatoare. În aşa numitele „compilatoare cu optimizare”, în care acestei faze i se acordă o importanță deosebită, fracțiunea din timpul de compilare cheltuită pentru optimizare este foarte mare. Există însă şi optimizări simple care îmbunătățesc considerabil eficiența programului obiect fără a încetini prea mult compilarea.

6. Generarea de cod

Generarea codului destinație (obiect) este faza finală a compilatorului. Codul obiect generat poate să fie, de exemplu, cod maşină relocabil sau un cod virtual. Pe lângă transformarea instrucțiunilor intermediare în secvențe de instrucțiuni maşină (virtuale) mai trebuie rezolvate următoarele probleme:

- selecționarea şi alocarea de celule de memorie pentru variabilele din program;

- alegerea şi implementarea celor mai eficiente tehnici de acces la fiecare variabilă în parte (inclusiv la componentele variabilei) utilizând toate posibilitățile de adresare ale calculatorului: indexare, indirectare etc.;

- alocarea registrelor pentru calcule şi pentru reținerea temporară a rezultatelor intermediare.

În fig. 1.4 se prezintă, pe faze, întregul proces de traducere al instrucțiunii *a:=b+c*10* şi principalele structuri de date asociate compilării.

Observație: Se consideră că *a, b şi c* sunt variabile reale.

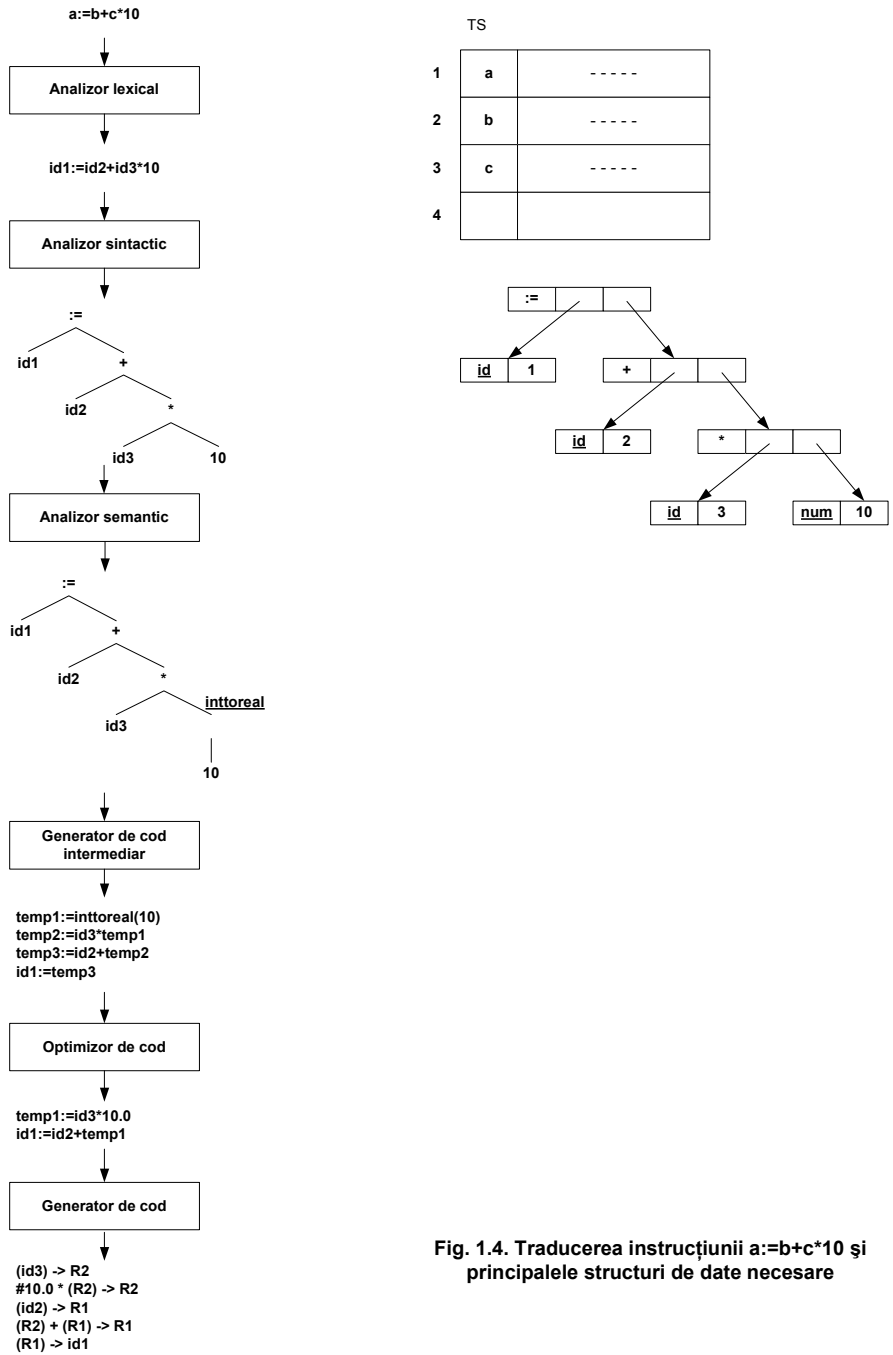

Fig. 1.4. Traducerea instrucțiunii a:=b+c*10 şi principalele structuri de date necesare

Gruparea fazelor

Delimitarea fazelor de compilare ca în paragrafele precedente urmăreşte organizarea logică a unui compilator. În diferite implementări concrete apar abateri de la această delimitare, în special datorită grupării împreună a unor activități din mai multe faze. O primă grupare este aceea în **front-end** şi **back-end.**

Front end este prima parte a unui compilator, adică acele faze sau părți din faze care depind în primul rând de limbajul sursă. Ea include de obicei: analiza lexicală şi sintactică, crearea TS, analiza semantică şi generarea codului intermediar. În *front end* se poate, de asemenea, face o superficială optimizare de cod şi trebuie obligatoriu inclusă tratarea erorilor specifică fazelor componente.

Back end include acele poțiuni din compilator care depind de calculatorul destinație. Nu depinde direct de limbajul sursă ci numai de limbajul intermediar. Include optimizarea codului şi generarea de cod împreună cu operațiile necesare de tratare a erorilor şi cele de manipulare a TS.

Gruparea fazelor în acest fel ar permite, de exemplu, să se realizeze un compilator pentru un limbaj pe un anumit calculator refăcând partea de *back end* a unui compilator pentru acelaşi limbaj, operațional pe alt calculator. Este de asemenea tentant ca mai multe compilatoare pentru limbaje diferite să genereze acelaşi cod intermediar şi să utilizeze acelaşi *back end*. *Front end*-urile fiind diferite, se obțin mai multe compilatoare pentru acelaşi calculator. Datorită diferențelor între limbajele sursă, această a doua modalitate n-a fost prea mult exploatată.

În practica realizării compilatoarelor, mai multe faze ale compilării sunt, de obicei, implementate împreună în aşa numitele treceri. O *trecere* constă din citirea unui fişier de intrare

reprezentând textul sursă al programului sau codul intermediar generat de trecerea anterioară, efectuarea operațiilor specifice fazelor componente şi crearea unui fişier de ieşire conținând cod intermediar pentru trecerea următoare sau codul obiect final. În practică există o mare varietate de grupări ale fazelor în treceri.

Din acest motiv, dar şi din motive de claritate, se preferă studierea tehnicilor de compilare pe faze.

Activitățile fazelor componente ale unei treceri sunt întrețesute. De exemplu, pentru cazul în care analiza lexicală, analiza sintactică, analiza semantică şi generarea codului obiect sunt grupate într-o singură trecere, şirul de atomi generat de analiza lexicală poate fi tradus direct în cod intermediar. Programul principal (coordonatorul) unei astfel de treceri poate fi analizorul sintactic. El încearcă să identifice structura gramaticală a şirului de atomi, apelând analizorul lexical, ca funcție, ori de câte ori are nevoie de un atom. Pe măsură ce se avansează în identificarea structurii gramaticale, analizorul sintactic apelează generatorul de cod intermediar, care efectuează şi verificările semantice, organizate, de asemenea, ca funcții distincte.

Citirea şi scrierea fişierelor intermediare durează un timp relativ mare, acesta fiind un argument pentru un număr mic de treceri. Pe de altă parte, gruparea într-o singură trecere a unui număr mare de faze poate să ducă la limitarea dimensiunii programelor compilate, datorită volumului mare de date care trebuie reținute, incluzând şi forma internă a programului.

Pentru unele faze, gruparea într-o trecere unică este destul de simplă (de exemplu analiza lexicală şi sintactică). În schimb generarea de cod este mai greu de efectuat până când nu s-a generat în întregime reprezentarea intermediară (ex. variabilele utilizate *înainte* de a fi declarate în PL/I, Algol 68 şi salturile înainte în codul obiect corespunzătoare implementării unor instrucțiuni condiționale sau ciclice).

Unelte pentru construcția compilatoarelor

La scrierea de compilatoare se utilizează, evident, uneltele software generale disponibile pe calculatorul respectiv (exemplu: depanatoare, editoare etc.). În plus au fost dezvoltate şi unelte specializate care ajută la implementarea diferitelor faze ale unui compilator.

La scurt timp după ce s-au scris primele compilatoare au apărut sisteme care să ajute în procesul de scriere a compilatoarelor (compiler-compiler, generatoare de compilatoare, sisteme de scriere a translatoarelor). În general, ele sunt orientate spre un anumit model de limbaj şi sunt potrivite pentru a genera compilatoare pentru limbaje similare cu modelul.

De exemplu, pornind de la faptul că pentru multe limbaje de programare analiza lexicală este în esență aceeaşi (diferă doar cuvintele cheie şi semnele care trebuie recunoscute), astfel de sisteme conțin rutine de analiză lexicală fixă. Acestor rutine li se furnizează lista cuvintelor cheie. Metoda nu poate fi aplicată pentru acele limbaje care conțin atomi nestandard (exemplu identificatori incluzând şi alte caractere decât litere sau cifre).

Alte unelte generale au fost create pentru proiectarea automată a unor componente ale compilatorului. Aceste unelte utilizează limbaje specializate pentru specificarea şi implementarea componentei respective, pe baza unor algoritmi complicați. Ele sunt cu atât mai utile cu cât ascund detaliile algoritmilor de generare şi produc componente care se pot uşor integra cu restul compilatorului.

Exemple:

1. **Generatoare de analizoare lexicale.** Specificarea de intrare a generatorului se bazează pe expresii regulate. Analizorul lexical generat în mod automat este un automat finit.

2. **Generatoare de analizoare sintactice.** Generarea analizorului sintactic se face pe baza gramaticii limbajului, furnizată la intrare. Importanța acestei metode a scăzut pe măsură ce s-a simplificat implementarea directă a analizei sintactice.

3. **Maşini de traducere dirijate de sintaxă.** Acestea produc colecții de rutine care parcurg arborele sintactic, generând cod intermediar.

4. **Generatoare de cod automate.** O astfel de unealtă primeşte o colecție de reguli care definesc traducerea fiecărei operații din limbajul intermediar în limbajul maşină pentru calculatorul destinație. Regulile trebuie să includă suficiente detalii pentru a putea trata diferitele metode de acces posibile pentru date; de exemplu variabilele pot fi în registre, într-o locație fixă (statică) în memorie sau li se poate aloca o poziție într-o stivă. Tehnica de bază este ”potrivirea tiparelor”: instrucțiunile din codul intermediar sunt înlocuite cu ”tipare” care reprezintă secvențe de instrucțiuni maşină.

5. **Maşini pentru fluxul datelor.** Sunt utilizate la optimizarea de cod pentru culegerea de informații despre modul de transmitere a valorilor între diferitele părți ale programului. Maşina primeşte la intrare detalii despre relația dintre instrucțiunile de cod intermediar şi informația culeasă.