Compilatoare

Litera A

abstract stack machine = maşina virtuală stivă

Maşină virtuală bazată pe stivă. Memoria acesteia este împărţită în trei zone: memoria pentru codul programului, memoria pentru datele globale şi stiva, care este folosită pentru apeluri de proceduri şi date locale. Nu există decât registre indicatoare ale stivei (vârf, bază), iar mulţimea instrucţiunilor cuprinde instrucţiuni aritmetice şi logice (de exemplu add, sub) pentru elementele din vârful stivei, instrucţiuni pentru gestiunea stivei (de tip push, pop, top) şi instrucţiuni pentru modificarea fluxului de control (salturi condiţionate şi necondiţionate, apeluri şi întoarceri din subprograme).

activation record = înregistrare de activare

Bloc de memorie ce conţine informaţiile necesare pentru execuţia unei proceduri, cum sunt: adresa de revenire, parametrii actuali, datele locale procedurii, starea registrelor înainte de execuţia procedurii, indicator către înregistrarea de activare a procedurii apelante, valoarea rezultatului etc. Structura înregistrării de activare depinde de convenţiile de apel pentru subprograme, de tipul domeniului de vizibilitate şi de modul de legare a numelor.

automaton = automat

Obiect matematic care descrie un dispozitiv utilizat pentru acceptarea limbajelor. Pentru fiecare clasă de limbaje corespunzătoare ierarhiei Chomsky există câte un tip de automat care poate să accepte limbaje din clasa respectivă. Cel mai simplu tip de automat este automatul finit. Clasa limbajelor care sunt acceptate de către automatele finite sunt limbajele regulate (generate de către gramatici sau expresii regulate). Clasa limbajelor care sunt acceptate de către automate cu stivă (push down) sunt limbajele independente de context. Limbajele dependente de context sunt acceptate de către automatele liniar mărginite. Cel mai general automat este Maşina Turing care acceptă limbaje care pot să fie generate de modelul general de gramatici.

Litera B

back end = partea dintr-un compilator, dependentă de maşina ţintă

Parte a compilatorului (vezi compiler) ce efectuează prelucrări legate de limbajul ţintă (de exemplu generarea şi optimizarea de cod obiect (vezi code generation). Împărţirea unui compilator în cele două părţi (front end şi back end) este utilă pentru a obţine implementări portabile. Astfel, pentru a obţine compilatoare pentru n limbaje care să genereze cod pentru m maşini, utilizarea unui limbaj intermediar (limbaj pentru o maşină virtuală) şi separarea strictă a celor două părţi ale compilatorului reduce la n + m numărul de programe ce trebuie scrise (în loc de n x m câte sunt necesare dacă nu se utilizează această abordare).

backpatching = întoarcerea pentru completarea etichetelor

Tehnică utilizată pentru generarea de cod intermediar într-o singură parcurgere a arborelui sintactic sau de derivare. În absenţa acestei tehnici pentru generarea codului intermediar sunt necesare două parcurgeri ale arborelui de derivare sau sintactic datorită existenţei unor referinţe "înainte" la instrucţiuni care nu au fost încă generate, deci pentru care nu se cunosc încă etichetele sau adresele corespunzătoare. Conform acestei tehnici se lasă ne completate în instrucţiuni referinţele care nu se pot rezolva şi se construieşte o listă a referinţelor nerezolvate. În momentul în care se pot rezolva referinţele, se face întoarcerea şi completarea conform listei.

basic block = bloc de bază

Secvenţă de instrucţiuni consecutive, astfel încât fluxul de control parcurge secvenţa de la început la sfârşit, fără posibilitatea ramificării sau opririi în interiorul secvenţei. Instrucţiunile de control al fluxului determină împărţirea codului în blocuri de bază.

Litera C

cod optimization = optimizare de cod

Tehnică prin care pentru un program dat se obține un program scris în acelaşi limbaj, echivalent din punct de vedere al funcționalității dar cu performanțe mai bune din punct de vedere al timpului de execuție şi / sau a spațiului de memorie utilizat.

common subexpression = subexpresie comună

Subexpresie care apare în aceeaşi formă în două sau mai multe locuri diferite din program, astfel încât valorile referite de ea nu sunt modificate între utilizări. Codul ce conține subexpresii comune poate fi optimizat astfel: subexpresia este calculată la prima apariție şi valoarea sa este memorată într-o variabilă tempoarară care va înlocui subexpresia în următoarele utilizări.

compiler = compilator

Program care transformă un program scris într-un limbaj sursă într-un program echivalent scris într-un limbaj țintă. Limbajul destinație poate să fie un alt limbaj de programare sau să fie limbajul unei maşini reale sau virtuale. În cazul în care este vorba de o maşină virtuală execuția programului rezultat se poate face prin interpretare. Compilarea presupune câteva etape (faze) specifice cum sunt: analiza lexicală (vezi lexical analysis), analiza sintactică (vezi sintax analysis), analiza semantică (vezi **semantic analysis**), generarea şi optimizarea de cod intermediar (vezi **intermediate code generation, intermediate code optimization**), generarea şi optimizarea de cod obiect (vezi **object code generation**, **object code optimization**). În cazul în care se generează cod obiect pentru o maşină reală, în codul generat se include şi codul necesar pentru utilizarea unor servicii specifice fazei de execuție: gestionarea automată a memoriei dinamice, utilizarea firelor de execuție multiple, etc. Unele compilatoare oferă suport pentru depanarea programelor, modificarea unor componente ale programului fără recompilarea întregului program, paralelizare etc.

context free grammar = gramatică independentă de context

Un caz particular de gramatică (vezi grammar) care poate genera structuri recursive şi poate descrie sintaxa majorității construcțiilor din limbajele de programare existente. Fiecare regulă indică o posibilă rescriere a unui simbol neterminal, indiferent de contextul în care apare (de aici şi numele gramaticii). Acest tip de gramatică este utilizat pentru specificarea construcțiilor considerate în faza de analiză sintactică (vezi syntax analysis).

copy propagation = propagarea copiei

Optimizare a codului intermediar generat de compilator (vezi compiler) ce constă în utilizarea expresiei care apare în membrul drept al unei instrucțiuni de atribuire în locul referinței din membrul stâng, în instrucțiunile pentru care nici unul dintre cei doi membrii nu este redefinit. Aceasta optimizare permite eventuala eliminare a instrucțiunii de atribuire.

Litera D

dead code elimination = eliminarea codului neutilizat

Optimizare a codului intermediar sau obiect generat de către compilator (vezi compiler), constând în eliminarea instrucțiunilor ce calculează valori care nu vor fi niciodată folosite ulterior sau a instrucțiunilor care nu sunt parcurse niciodată, indiferent de combinația datelor de intrare ale programului.

Litera E

environment = context

Funcție care realizează corespondența dintre un nume şi locația de memorie unde se află valoarea corespunzătoare numelui. În practică această corespondență se realizează menținând pentru fiecare bloc (adică o funcție sau un număr de instrucțiuni cuprinse între acolade ca în C, sau begin end ca în Pascal) o listă de perechi nume - locație de memorie şi a unui indicator către structura de date corespunzătoare blocului care îl conține pe cel curent.

Litera F

finite automaton = automat finit

Dispozitiv format dintr-o bandă de intrare (care conține un şir de simboli) şi o unitate de control. Un cap de citire permite citirea simbolurilor de pe banda de intrare şi se poate deplasa numai spre dreapta cu câte un simbol. Funcționarea unității de control pleacă dintr-o stare inițială, cu capul de citire poziționat pe primul simbol din şirul de intrare. Pe baza stării curente şi a simbolului curent de pe banda de intrare automatul trece într-o nouă stare, avansând eventual pe banda de intrare pe următorul simbol. Se spune că automatul a executat o tranziție. Automatul **acceptă** şirul de intrare dacă ajunge într-o stare finală având capul de citire după ultimul simbol de pe banda de intrare. Mulțimea şirurilor acceptate de către un automat finit este limbajul formal acceptat de către automat. Automatele finite pot să fie deterministe sau nedeterministe. În cazul unui utomat determinist pentru fiecare combinație stare curentă, simbol curent pe banda de intrare există cel mult o singură stare următoare şi automatul avansează pentru fiecare tranziție. Clasa limbajelor acceptate de către automatele finite deterministe este identică cu cea acceptată de automatele nedeterministe. Analizoarele lexicale (vezi lexical analysis) sunt implementate ca simulatoare de automate finite deterministe.

flow graph = graful fluxului controlului

Graf orientat, ale cărui noduri sunt blocuri de bază (vezi basic block). Există un arc între blocurile B_1 şi B_2, dacă prima instrucțiune a lui B_2 poate apare imediat după ultima instrucțiune a lui B_1 pentru o execuție particulară a programului. Folosind acest graf, se pot obține informații despre variabilele programului, ca de exemplu: durata valabilității unei definiții de variabilă (până la următoarea definiție a sa), durata de viață a unei variabile, porțiunea din program în care variabilele ce apar într-o expresie rămân neschimbate etc. Acest tip de analiză se numeşte analiză a fluxului de date (engl. data-flow analysis).

formal language = limbaj formal

Mulțime de şiruri generate utilizând simboli dintr-un alfabet. Pentru mulțimile infinite sunt interesante mecanismele de descriere (specificare) finite (gramatici (vezi **gramatica**), automate, etc.). Nu orice limbaj poate să fie descris într-o formă finită.

front end = partea dintr-un compilator, independentă de maşina țintă

Acea parte a unui compilator (vezi compiler) care prelucrează codul sursă, fără a ține seama de particularitățile limbajului țintă. Macro substituția, analiza lexicală, analiza sintactică, cea semantică, generarea şi optimizarea de cod intermediar, (vezi intermediate code generation) sunt etape ale compilării realizate de această componentă.

Litera G

garbage collector = colector de memorie disponibilă

Componentă a sistemului de execuție (vezi run-time system) care implementează colectarea automată a spațiului devenit disponibil din memoria dinamică. Colectorul caută obiectele care nu mai sunt folosite şi adaugă spațiul ocupat de acestea la memoria disponibilă. Este utilizat în special pentru limbaje care nu admit aritmetică pe pointeri (limbaje funcționale, Java).

grammar = gramatică

O mulțime de reguli (numite producții) utilizate pentru generarea propozițiilor corecte dintr-un limbaj formal. Are semnificație de specificație finită a limbajului. O regulă indică modul în care se poate rescrie un subşir care constituie partea stângă a producției utilizând subşirul care reprezintă partea dreaptă. O regulă conține două categorii de simboluri: simboluri terminale care fac parte din alfabetul utilizat pentru construirea propozițiilor şi simboluri neterminale care sunt folosite în procesul generării. Pornind de la un simbol de start al gramaticii (un simbol neterminal special), prin rescrieri succesive (adică prin aplicări succesive ale producțiilor gramaticii) se pot obține toate propozițiile din limbajul generat de gramatică. Pe baza formei producțiilor, gramaticile sunt clasificate conform ierarhiei Chomsky în gramatici regulate (tipul 3), independente de context (tipul 2), dependente de context (tipul 1), general (tipul 0). Pentru limbajele de programare alfabetul este format din litere, cifre, operatori, semne de punctuație iar simbolurile neterminale reprezintă noțiuni ca: program, instrucțiune, expresie, declarație, lista, etc. De exemplu, regulile pentru generarea expresiilor aritmetice având ca operanzi identificatorii a şi b şi ca operatori adunarea şi înmulțirea sunt E -> a, E -> b, E -> E + E, E -> E * E. Simbolul de start al gramaticii în acest caz este E.

granularity = granulariate (a unei instrucțiuni)

Complexitatea acțiunii codificate de instrucțiunea respectivă.

instruction cost = costul instrucțiunii

Funcție care asociază fiecărei instrucțiuni o valoare calculabilă, astfel încât să se poată calcula şi optimiza funcția cost pentru codul generat de către un compilator. Dacă economia de spațiu este importantă, costul instrucțiunii poate fi lungimea sa, în general se alege un cost care să reflecte timpul de execuție al instrucțiunilor.

Litera I

intermediate code generation = generarea de cod intermediar

Etapă a procesului de compilare ce constă în parcurgerea arborelui de derivare sau sintactic (produs de către analiza sintactică (vezi **syntax analysis**), "ornat" cu atribute de către analiza semantică (vezi **semantic analysis**) şi generarea de cod într-un limbaj intermediar (vezi compiler, intermediate language).

intermediate language = limbaj intermediar

Limbaj cu complexitatea între cea a limbajului sursă şi cea a limbajului țintă, în care este tradus codul sursă ca o etapă înainte de generarea programului în limbajul țintă. Dacă diferențele dintre limbajul sursă şi cel țintă sunt mari, se pot utiliza mai multe limbaje intermediare, din ce în ce mai apropiate de limbajul țintă, ca etape ale transformării. Prin utilizarea limbajelor intermediare, acelaşi compilator poate fi folosit pentru generarea de cod pentru mai multe maşini țintă, modificând numai partea dependentă de limbajul destinație (vezi back end). De asemenea, se pot utiliza algoritmi de optimizare a limbajului intermediar, independenți de caracteristicile limbajelor sursă şi destinație. Ca exemple de limbaje intermediare se pot considera: arborii de derivare, arborii sintactici, codul cu trei adrese (vezi three address code) şi limbajul pentru maşina stivă (vezi virtual machine).

interpreter = interpretor

Program care citeşte un program scris într-un limbaj de programare şi îl execută instrucțiune cu instrucțiune. Spre deosebire de un compilator, un interpretor nu generează cod. La limită, un procesor este un interpretor implementat hardware pentru programe scrise în limbaj maşină. Un interpretor analizează fiecare instrucțiune a programului pentru a-i determina tipul şi, în funcție de acesta execută o anumită acțiune. Timpul necesar pentru selecția acțiunii ce trebuie executată, precum şi diferențele dintre limbajul interpretat şi cel în care este scris interpretorul sunt factori care influențează performanțele acestuia. Un interpretor este un program mai simplu de scris decât un compilator. Se poate considera că un intrepretor simulează o maşină virtuală pe un calculator real.

Litera L

lexeme = lexemă

Unul din şirurile de caractere ce corespund unui atom lexical (vezi token). De exemplu atomul lexical IDENTIFICATOR corespunde şirurilor de caractere formate din litere şi cifre, care încep cu o literă şi nu sunt cuvinte cheie. În schimb, atomul lexical IF corespunde numai lexemului "if".

lexical analysis (scanning sau tokenizing) = analiză lexicală

Etapă a procesului de compilare (vezi compiler) care transformă şirul de caractere ce reprezintă programul scris în limbaj sursă, într-un şir de atomi lexicali (vezi token). Gramatica folosită pentru recunoaşterea atomilor lexicali este regulată (vezi regular grammar). Exemple de atomi lexicali utilizați în limbajele de programare: identificator, număr, operator, delimitator, terminator, etc. Un analizor lexical simulează de fapt funcționarea unui automat finit determinist (vezi **finite automaton**). Deoarece algoritmul de analiză este acelaşi indiferent de limbaj, există generatoare de analizoare lexicale care primesc la intrare specificarea atomilor lexicali şi a modului în care se calculează atributele acestora un analizor lexical corespunzător. Un exemplu de astfel de generator este lex, care generează analizoare lexicale scrise în C.

LL(1)

Clasă de analizoare sintactice descendente (top down) pentru care decizia referitoare la regula (producția) pe baza căreia se "explică" noțiunea curentă (neterminalul curent) se ia pe baza unui singur simbol "vizibil" din şirul de intrare. Primul L din nume indică faptul că parcurgerea şirului de intrare se face de la stânga la dreapta. Al doilea L indică faptul că algoritmul este echivalent cu o derivare stânga. Clasa gramaticilor independente de context pentru care se pot construi analizoare LL(1) se numeşte clasa gramaticilor LL(1), similar clasa limbajelor pentru care se pot construi gramatici de tip LL(1) se numesc limbaje LL(1). Deoarece algoritmul de analiză este acelaşi indiferent de gramatică, există generatoare de analizoare LL(1) care primesc la intrare specificarea gramaticii şi produc un analizor sintactic LL(1). Un exemplu de astfel de generator este JavaCC, care generează analizoare sintactice de tip LL(1) scrise în Java.

loop optimization = optimizarea ciclurilor

Optimizare a codului intermediar generat de către compilator (vezi compiler) care are ca scop mărirea vitezei de execuție a codului conținut în cicluri (de exemplu prin identificarea unor invarianți şi plasarea lor înaintea ciclurilor sau prin înlocuirea ciclurilor cu număr finit mic de paşi cu o secvență obținută prin repetarea corpului ciclului, eliminându-se astfel instrucțiunile pentru controlul ciclului).

LR(1)

Clasă de analizoare sintactice ascendente (bottom up) pentru care decizia referitoare la regula (producția) pe baza căreia se face abstractizarea de la un şir de simboli terminali şi neterminali care formează partea dreaptă a unei reguli (producții) la neterminalul din partea stânga se ia pe baza unui singur simbol "vizibil" din şirul de intrare. Primul L din nume indică faptul că parcurgerea şirului de intrare se face de la stânga la dreapta. Caracterul R indică faptul că algoritmul este echivalent cu o derivare dreapta. Clasa gramaticilor independente de context pentru care se pot construi analizoare LR(1) se numeşte clasa gramaticilor LR(1), similar clasa limbajelor pentru care se pot construi gramatici de tip LR(1) se numesc limbaje LR(1). Clasa limbajelor LL(1) este un subset propriu al clasei LR(1). Deoarece algoritmul de analiză este acelaşi indiferent de gramatică, există generatoare de analizoare LR(1) care primesc la intrare specificarea gramaticii şi produc un analizor sintactic LR(1). Un exemplu de astfel de generator este yacc, care generează analizoare sintactice de tip LR(1) scrise în limbajul C.

Litera O

object code generation = generare de cod obiect

Etapă a compilării care produce un program scris în limbaj destinație, echivalent cu programul sursă (vezi compiler). De obicei sursa generatorului de cod este constituită de codul scris în limbaj intermediar care a fost eventual optimizat.

overloaded function (operator) = funcție (operator) supraîncărcată (supradefinită)

Funcție al cărei nume are semnificații diferite în contexte diferite. Corpul funcției sau numărul de argumente sunt diferite de la un context la altul. Mai general se vorbeşte de simboluri supraîncărcate, acestea având semnificații diferite în contexte diferite. De exemplu operatorul "*" are comportări diferite după cum se aplică unor valori întregi, reale sau unor şiruri de caractere.

Litera P

parse tree = arbore de derivare

Reprezentarea sub forma unui arbore a secvenței de derivări (rescrieri), folosind producțiile unei gramatici independente de context, ce conduc la obținerea unei propoziții în limbajul generat de gramatică (vezi grammar, context free grammar). Tuturor simbolurilor din partea dreaptă a unei producții le corespund în arbore noduri descendenți ai nodului corespunzător simbolului neterminal din stânga producției. Frunzele arborelui corespund simbolurilor terminale ce compun propoziția generată. Construirea arborelui se poate face într-o manieră descendentă (top-down) sau ascendentă (bottom-up). În primul caz noțiunea corespunzătoare simbolului de start al gramaticii (în cazul limbajelor de programare este vorba de program sau de instrucțiune) este "explicată" prin rescrieri utilizând noțiuni mai simple până se ajunge la un şir de simboli din alfabetul peste care este construit limbajul (în cazul limbajelor de programare este vorba de atomi lexicali). În al doilea caz se face o abstractizare de noțiuni pornind de la un şir de simboli din alfabet până când se obține cea mai generală noțiune (de exemplu

program sau instrucțiune în cazul limbajelor de programare).

pass = trecere

În contextul procesului de compilare are semnificația de parcurgere integrală a datelor de intrare (adică a programului sursă sau a unei forme intermediare în care acesta a fost transformat), prelucrarea lor şi memorarea rezultatului într-un fişier sau în memorie. În general se încearcă realizarea mai multor etape (faze) ale compilării (vezi compiler) în aceeaşi trecere.

pattern = tipar

Şir format, după anumite reguli, din caractere obişnuite şi metacaractere, astfel încât reprezintă un model pentru o mulțime de propoziții ale unui limbaj. Termenul se referă în general la elementele din care se compun expresiile regulate (vezi regular expressions).

pattern matching = identificarea tiparului

Găsirea unui subşir al şirului primit la intrare, subşir care corespunde unui anumit tipar (vezi pattern). Se poate utiliza pentru analiza lexicală pentru identificarea lexemelor care corespund atomilor lexicali din limbaj.

peephole optimization = optimizare ce ia în considerare o porțiune mică de cod

Îmbunătățire ce constă în înlocuirea unei secvențe de instrucțiuni din codul obiect cu o secvență de instrucțiuni ce produce acelaşi rezultat, dar produce performanțe mai bune ale codului, fie din punct de vedere al spațiului ocupat, fie, mai ales din punct de vedere al timpului de execuție.

polymorfic function (operator) = funcție (operator) polimorfică

Funcție (operator) care se poate evalua în contexte diferite cu argumente de tipuri diferite. Corpul funcției şi numărul de argumente sunt aceleaşi în toate cazurile. De exemplu operatorul "&" de luare a adresei în limbajul C are aceeaşi funcționalitate indiferent de tipul obiectului (variabilei sau funcției căruia i se aplică).

push down automaton = automat cu stivă

Dispozitiv format dintr-o bandă de intrare (care conține un şir de simboli), o stivă şi o unitate de control. Un cap de citire permite citirea simbolurilor de pe banda de intrare şi se poate deplasa numai spre dreapta cu câte un simbol. Funcționarea unității de control pleacă dintr-o stare inițială, cu un simbol special de inițializare în vârful stivei şi cu capul de citire poziționat pe primul simbol din şirul de intrare. Pe baza stării curente, a simbolului curent de pe banda de intrare şi a simbolului aflat în vârful stivei automatul trece într-o nouă stare, înlocuind simbolul din vârful stivei cu un şir de caractere şi avansând eventual pe banda de intrare pe următorul simbol. Se spune că automatul a executat o tranziție. Automatul **acceptă** şirul de intrare dacă ajunge într-o stare finală având capul de citire după ultimul simbol de pe banda de intrare. Mulțimea şirurilor acceptate de către un automat cu stivă este limbajul formal acceptat de către automat. Automatele cu stivă pot să fie deterministe sau nedeterministe. În cazul unui automat determinist pentru fiecare combinație stare curentă, simbol în vârful stivei, simbol curent pe banda de intrare există cel mult o singură stare următoare, şir memorat în stivă, automatul avansează pentru fiecare tranziție. Clasa limbajelor acceptate de către automatele cu stivă deterministe este o submulțime a celei acceptate de către automatele cu stivă nedeterministe. Analizoarele sintactice (vezi sintax analysis) sunt implementate ca simulatoare de automate cu stivă.

Litera R

register allocation = alocarea registrelor

Stabilirea modului de utilizare a registrelor pentru variabile. Alegerea valorilor care vor fi păstrate în registre. Este o componentă foarte importantă a generării de cod (vezi code generation), deoarece instrucțiunile ce adresează registre sunt mult mai rapide decât cele care adresează locații de memorie, astfel că o bună alegere a valorilor ce vor fi alocate în registre asigură generarea de cod performant (vezi şi register spill). Problema alocării registrelor este echivalentă cu problema colorării grafurilor.

register assignment = atribuirea registrului

Selectarea unui registru particular în care se va aloca o anumită valoare (vezi şi register allocation).

register spill = eliberarea forțată registrului

Acțiune ce presupune salvarea conținutului unui registru în memorie în vederea memorării unei noi valori în registrul eliberat. Este necesară în situația în care trebuie să se execute o instrucțiune care presupune utilizarea unui anumit registru, sau trebuie să fie calculată o nouă valoare şi numai există nici un registru disponibil. Eliberarea forțată a registrului trebuie să fie făcută cât mai rar cu putință, deoarece afectează performanțele codului compilat. Nu există o regulă generală pentru alegerea unui registru ce va fi eliberat, diferite euristici fiind utilizate, după caz.

regular expression = expresie regulată

O metodă alternativă de specificare a clasei de limbaje care pot să fie specificate şi prin gramatici regulate. O formulă care se construieşte recursiv astfel: şirul vid şi elementele unui alfabet finit constituie o expresie regulată peste alfabetul respectiv; reuniunea, intersecția şi închiderea tranzitivă a expresiilor regulate constituie expresii regulate; orice expresie regulată se obține numai prin operațiile de mai sus. Pentru specificarea limbajului care corespunde analizei lexicale se folosesc expresii regulate.

regular grammar = gramatică regulată

Un caz particular de gramatică (vezi grammar) pentru care producțiile sunt de forma A -> b sau A -> aB, unde A şi B sunt simboluri neterminale, iar a,b sunt simboluri terminale. Acest tip de gramatică corespunde nivelului construcțiilor considerate de către analiza lexicală (descrierea atomilor lexicali) (vezi **lexical analysis**)

run-time system = sistem de execuție

Cod utilizat pentru suportul execuției programului, realizând legătura cu sistemul de operare şi controlând diferite aspecte ca de exemplu: controlul alocării şi eliberării memoriei dinamice, dispunerea stivei, a heap-ului în raport cu codul programului, structura înregistrăriilor de activare, etc.

Litera S

scope = domeniu de vizibilitate (al declarației unui nume)

Porțiunea dintr-un program în care este valabilă declarația unui nume. Vizibilitatea poate fi statică (sau lexicală), însemnând că declarația numelui poate fi găsită prin examinarea textului programului, sau dinamică, când această declarație este identificată luându-se în considerație pe baza lanțului de apeluri, contextul curent din timpul execuției.

semantic analysis = analiză semantică

Etapă (fază) a compilării ce constă în detectarea erorilor semantice, pornindu-se de la rezultatele analizei sintactice. O componentă fundamentală a analizei semantice este stabilirea şi verificarea tipului diferitelor construcții din program (vezi type checking, type inference). Ca rezultat al analizei semantice se obține un arbore sintactic sau un arbore de derivare cu atribute calculate. Faza de analiză semantică se poate realiza concomitent cu analiza sintactică dacă se utilizează scheme de translatare.

stack allocation = alocarea pe stivă

Strategie de alocare a înregistrărilor de activare (vezi activation record) corespunzătoare apelurilor de subprograme, care utilizează o stivă. Dacă considerăm arborele ce reprezintă ierarhia de apeluri de subprograme, la un anumit moment dat stiva va conține înregistrările de activare corespunzătoare subprogramelor în ordinea în care au fost apelate. În vârful stivei se găseşte înregistrarea de activare pentru sub programul curent. La baza stivei se găseşte înregistrarea de activare pentru programul principal.

symbol table = tabelă de simboli

Tabelă ce conține toți identificatorii (numele) întâlniți în program şi informații despre aceştia: identificator / cuvânt cheie, expresie de tip (dacă este cazul), adresa locației de memorie asociată (în faza de generare de cod sau de interpretare), nivelul de imbricare la care a fost definit identificatorul, eventual o legătură la un alt identificator definit la acelaşi nivel. În cazul numelui unei funcții, în tabela de simboli se poate memora şi lista parametrilor formali şi legătura către contextul definiției funcției (vezi environment). Este inițializată de către faza de analiză lexicală (vezi **lexical anaysis**) şi completată cu informații în timpul următoarelor faze. Fiind o structură de date foarte des utilizată, accesul al tabelele de simboli trebuie să fie cât mai rapid şi în general acestea se organizează sub formă de tabele de dispersie (engl. hash tables). Informațiile care se mențin în tabela de simboli depind de tipul limbajului şi de deciziile de proiectare a compilatorului. Tabela de simboli se utilizează pentru compilarea sau interpretarea unui program. În cazul codului executabil compilat tabela de simboli se poate păstra şi în timpul execuției, pentru depanarea programelor. În cazul interpretării, tabela de simboli poate să joace în anumite condiții şi rol de memorie pentru variabilele din program.

syntax analysis (parsing) = analiză sintactică

Etapă a compilării (vezi compiler) ce recunoaşte propoziții corecte scrise în limbajul sursă (formate din atomi lexicali). Ca rezultat al fazei de analiza sintactică se construieşte un arbore sintactic sau de derivare, ca formă intermediară de reprezentare a programului (vezi syntax tree).

syntax directed definition = definiție orientată pe sintaxă

Gramatică independentă de context (vezi **context free grammar**), căreia i se asociază o mulțime de atribute şi reguli de calcul al atributelor (acțiuni). Atributele sunt utilizate pentru a asocia simbolilor terminali şi neterminali informații necesare pentru stabilirea tipului sau pregătirea generării de cod. De exemplu o constantă de tip NUMAR sau un identificator pot avea ca atribut asociat tipul INTREG, o instrucțiune poate avea tipul VOID, INTEGER, FLOAT, etc. Regulile de calcul sunt asociate producțiilor, considerându-se că de fiecare dată când se utilizează o producție se vor aplica şi regulile de calcul asociate.

syntax tree = arbore sintactic

Reprezentare a unei expresii sau a unui întreg program sub forma unui arbore, astfel încât fiecare nod reprezintă o operație, iar descendenții nodului sunt argumentele operației. În general este rezultatul analizei sintactice (vezi syntax analysis).

Litera T

temporaries = variabile temporare

Variabile introduse de compilator pentru memorarea rezultatelor intermediare ale unor expresii. De exemplu, în cazul codului cu trei adrese (vezi three adress code), dacă o expresie conține doi operatori, pentru evaluarea expresiei se vor genera două instrucțiuni, fiind necesară o variabilă temporară pentru memorarea rezultatului intermediar generat.

three adress code = cod cu trei adrese

Limbaj intermediar (vezi intermediate cod generation), ale cărui instrucțiuni au forma generică

```
x = y op z.
```

unde op este un operator aritmetic sau logic, iar *x, y* şi *z* sunt nume de variabile ale programului sau nume de variabile temorare (vezi temporaries), generate de compilator. *y* şi *z* pot fi şi constante. Limbajul conține şi instrucțiuni condiționale de tip

```
if x then eticheta
```

unde *x* este o variabilă logică, iar *eticheta* este eticheta unei instrucțiuni din codul generat, instrucțiuni de salt de forma *goto eticheta* şi instrucțiuni de apel de procedură. Numele limbajului provine de la faptul că, de regulă, o instrucțiune conține trei adrese: două pentru operanzi şi una pentru rezultat. Desigur, există instrucțiuni care conțin mai puține adrese (de exemplu

instrucțiunea de atribuire, instrucțiunea de salt, instrucțiunea condițională). Instrucțiunile pot fi etichetate, permițând astfel referirea lor din alte instrucțiuni (de exemplu pentru instrucțiunile de salt).

token = atom lexical

Unitate lexicală indivizibilă, recunoscută de analizorul sintactic. Unui atom lexical îi pot corespunde unul sau mai multe şiruri diferite (vezi lexem). Exemple de atomi lexicali caracteristici limbajelor de programare: nume (identificatori), numere, cuvintele cheie dintr-un limbaj, operatori (ca de exemplu < , + sau mod). Un atom lexical este reprezentat de un tip (cod reprezentat de obicei ca un număr întreg) şi de o colecție de atribute. Astfel şirul 123 este un atom lexical de tip NUMAR care are atributele: întreg şi valoarea asociată, şirul abcd este un atom lexical de tip IDENTIFICATOR care poate avea asociat ca atribut adresa din tabela de simboli unde sunt păstrate informații despre el. Alegerea mulțimii de atomi lexicali este o opțiune a proiectanților de compilatoare.

translation scheme = schema de translatare

Definiție orientată pe sintaxă (vezi syntax directed definition) care conține şi informații despre modul de implementare a acțiunilor (de exemplu poziția lor, adică momentul execuției, în raport cu parcurgerea simbolilor care formează partea dreaptă a producției aplicate).

Turing machine = maşina Turing

Cel mai general automat. Dispozitiv format dintr-o bandă (care conține un şir de simboli) şi o unitate de control. Un cap de citire / scriere permite citirea / scrierea simbolurilor de pe / pe bandă. Se poate deplasa spre stânga sau spre dreapta cu o singură poziție. Funcționarea unității de control pleacă dintr-o stare inițială. Pe baza stării curente şi a simbolului curent de pe bandă, maşina trece într-o nouă stare, poate să scrie pe bandă sau să se deplaseze cu o poziție spre stânga sau spre dreapta. Se spune că maşina a executat o tranziție. Spre deosebire de automatele finite sau de cele cu stivă banda maşinii Turing este infinită la dreapta. Datorită faptului că maşina poate să scrie şi să citească pe / de pe bandă se poate considera că maşina dispune de o memorie infinită. În cazul maşinii Turing există noțiunea de acceptare, care se aseamănă cu cea corespunzătoare automatelor finite sau a celor cu stivă. şi anume se spune că maşina **acceptă** un limbaj dacă pentru orice şir din limbaj pornind cu capul de citire poziționat după şirul de intrare maşina se opreşte. În aceleaşi condiții pentru orice şir care nu face parte din limbaj maşina nu se opreşte. Datorită faptului că maşina este în stare să scrie pe bandă, înseamnă că ea este în stare să semnaleze apartenența sau neapartenența unui şir la limbaj şi prin intermediul unui răspuns scris pe bandă. Apare astfel noțiunea de decidabilitate. şi anume se spune că o maşină Turing **decide** un limbaj dacă şi numai dacă pentru orice şir de intrare maşina se opreşte, în cazul în care şirul face parte din limbaj maşina scrie un răspuns afirmativ pe bandă, în caz contrar răspunsul este negativezi În ambele cazuri răspunsul înlocuieşte şirul de intrare. Noțiunea de decidabilitate este mai restrictivă decât cea de acceptabilitate, adică există limbaje pentru care se pot construi maşini Turing care să le accepte, dar pentru care nu se pot construi maşini Turing care să le decidă, în timp ce pentru orice limbaj decidabil se poate construi o maşină Turing care să îl accepte prin transformarea maşinii care îl decide. Maşina Turing este cel mai general model de automat, este echivalentă din punct de vedere a cea ce poate să calculeze cu un calculator (evident echivalența nu se păstrează din punctul de vedere al timpului de calcul). Maşina Turing nu se utilizează direct în compilatoare, ea este utilizată însă ca model de calcul în calculabilitate.

type checking = verificarea tipurilor

Acțiune care asigură că tipul unei construcții din program are semnificație în contextul în care aceasta se găseşte. De exemplu, în majoritatea limbajelor de programare, se va semnala o eroare dacă se încearcă adunarea unei valori întregi cu un vector. În funcție de tipul limbajului, această verificare se poate face în timpul compilării, caz în care se spune că verificarea tipurilor este statică sau în timpul execuției, caz în care se spune că verificarea este dinamică.

type conversion = conversie de tip

Schimbarea tipului unei valori. Această conversie poate fi făcută implicit de către compilator, sau explicit, la cererea programatorului. Pentru fiecare limbaj de programare există o listă de tipuri compatibile între care se pot face conversii.

type expression = expresie de tip

Expresie ce reprezintă tipul unei construcții din limbaj. Cele mai simple expresii de tip sunt: un tip de bază al limbajului (ex. integer, real, boolean), un tip definit de programator sau o variabilă care poate lua ca valori expresii de tip. Utilizând astfel de

expresii de tip se pot construi expresii de tip mai complicate prin utilizarea constructorilor de tip (operatori ce primesc ca argumente expresii de tip şi construiesc o nouă expresie de tip, de exemplu produsul cartezian, funcția, pointer, lista, etc.) asupra uneia sau mai multor expresii de tip. De exemplu

$$(\forall \alpha.)\alpha X \text{ int} \rightarrow \text{int}$$

este o expresie de tip care reprezintă funcții de două variabile pentru care al doilea argument este de tip întreg. Tipul rezultatului funcției este acelaşi cu tipul primului argument. Este utilizată în faza de analiză semantică (vezi **syntax analysis**) pentru verificarea utilizării corecte a tipurilor.

type inference = inferența tipurilor

Determinarea expresiei de tip asociată unei construcții din limbaj luând în considerație şi contextul în care construcția este folosită. Este caracteristică limbajelor de programare pentru care nu există declarații de tip (de exemplu ML). În aceste cazuri, se consideră că utilizarea tipurilor este corectă, dacă pentru fiecare construcție din program se poate infera o expresie de tip.

type system = sistem de tipuri

O mulțime de reguli ce permite ataşarea unor expresii de tip (vezi type expression) diferitelor construcții dintr-un program.

Litera V

virtual machine = maşină virtuală

Calculator imaginar care ştie să "execute" programe scrise într-un limbaj intermediar (vezi intermediate language). Arhitectura unei maşini virtuale include mulțimea instrucțiunilor, registrele maşinii, organizarea memoriei. Exemple clasice de organizări pentru maşinile virtuale sunt: maşina stivă (vezi abstract stack machine), sau maşina cu trei adrese (vezi three adress code). Un interpretor este un simulator al unei maşini virtuale.