Teoria traducerii

Limbaje formale și translatoare (Compilatoare)

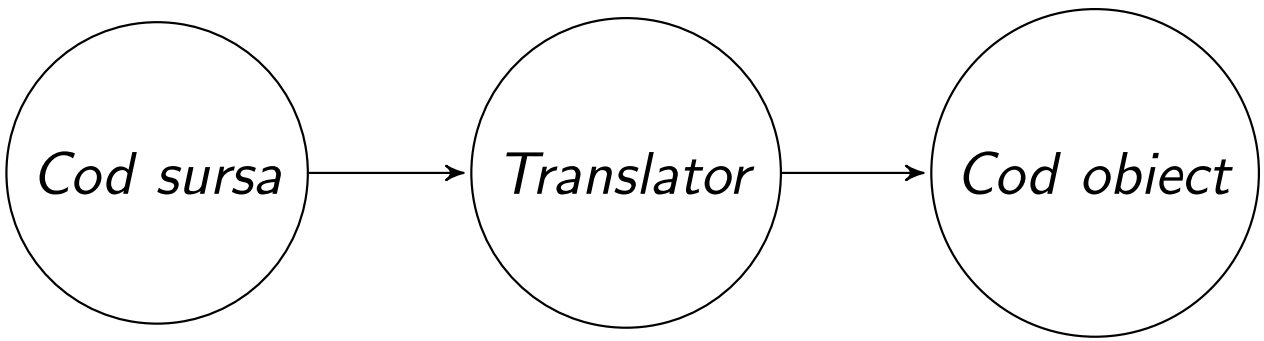
November 6, 2020

Mihai-Lica Pura

Cuprins

Definiții

- **Compilator/Translator** - programul sau setul de programe care traduce propoziții scrise într-un limbaj (**limbajul sursă**) într-un alt limbaj (**limbajul țintă**)



  - propoziția de intrare - **cod sursă**
  - propoziția de ieșire - **cod obiect**

Definiții

- motivul cel mai des întâlnit pentru care este nevoie de traducerea codului sursă constă în nevoia de a crea un **program executabil**.
- termenul **compilator** se folosește în special pentru programele care traduc:
  - cod sursă scris într-un **limbaj de programare de nivel înalt**
  - în cod obiect într-un **limbaj de programare nivel scăzut** (limbaj mașină)

Definiții

- **decompilator (decompiler)** - programul care poate să traducă dintr-un limbaj de nivel scăzut într-un limbaj de nivel înalt
- **translator de limbaje, translator sursă la sursă sau convertor de limbaje** - program capabil să facă traduceri între două limbaje de programare de nivel înalt
- **rescriitor de limbaj (rewriter)** - program care poate schimba forma de exprimare a codului sursă, fără a schimba limbajul în care a fost scris

Clasificarea compilatoarelor

după **platforma țintă** (platforma pe care va fi executat codul obiect produs de către compilator):

- **Compilatoare native (native/hosted compilers)**
  - codul obiect va fi rulat pe același tip de mașină și pe același tip de sistem de operare ca și cele pe care a fost rulat compilatorul
    Ex: compilatorul din Microsoft Visual Studio 2017

Clasificarea compilatoarelor

- **Cross compiler**
  - codul obiect generat de către compilator va fi rulat pe un alt tip de platformă
  - acest tip de compilatoare se utilizează în special pentru dezvoltarea de programe pentru sistemele embedded, care nu au fost proiectate astfel încât să suporte un mediu de dezvoltare
    Ex: programele pentru microprocesoare

- Ieșirea compilatorului care produce cod obiect pentru o mașină virtuală poate fi executat atât pe același tip de platformă ca și cea pe care a rulat compilatorul, cât și pe un alt tip. De aceea, aceste compilatoare nu sunt clasificate ca și native sau cross.

Etapele traducerii

- **1) Analiza lexicală**
  - pe baza: gramaticii regulare care definește sintaxa tipurilor de atomi lexicali
  - intrare - codul sursa (secvența de caractere)
  - ieșire - secvența de atomi lexicali
- **2) Analiza sintactică**
  - pe baza: gramaticii independente de context care definește sintaxa propozițiilor limbajului
  - intrare - secvența de atomi lexicali
  - ieșire - arborele sintactic (și tabela de simboli)
- **3) Analiza semantică**

Etapele traducerii

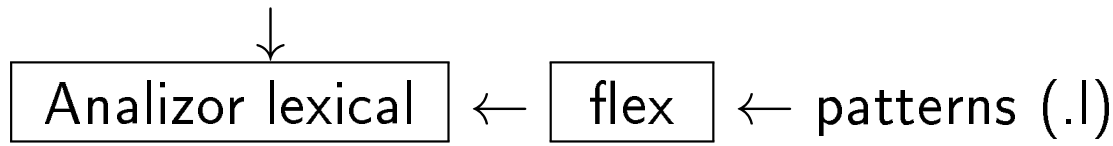
- **4) Translatare in cod intermediar**
  - intrare - arborele sintactic
  - ieșire - codul intermediar
- **5) Optimizarea codului intermediar**
  - intrare - codul intermediar
  - ieșire - codul intermediar optimizat
-
  - Optiune 1) **Generarea de cod**
    - ingtrare - codul intermediar optimizat
    - ieșire - fișierul executabil (cod mașină)
  - Optiune 2) **Interpretarea**
    - simularea execuției codului intermediar de catre un interpretor

Etapele traducerii exemplificate pentru o propoziție în limbaj natural

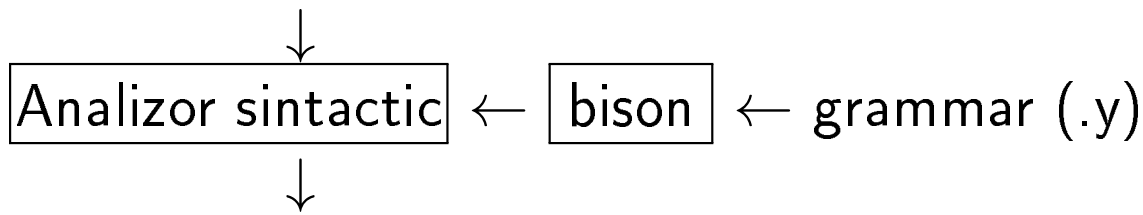
	Propozitia este valida din punct de vedere lexical? (**atomii lexicali care compun propozitia apartin vocabularului limbajului**?)	Propozitia este corecta din punct de vedere sintactic? (**atomii lexicali sunt in ordinea corecta**?)	Propozitia este corecta din punct de vedere semnantic? (**propozitia are inteles**?)
Eminescu este un scriicotr.	NU		
Eminescu scriitor un este.	DA	NU	
Scriitor este un Eminescu.	DA	DA	NU
Eminescu este un scriitor.	DA	DA	DA

Etapele traducerii exemplificate pentru o propoziție în limbaj de programare

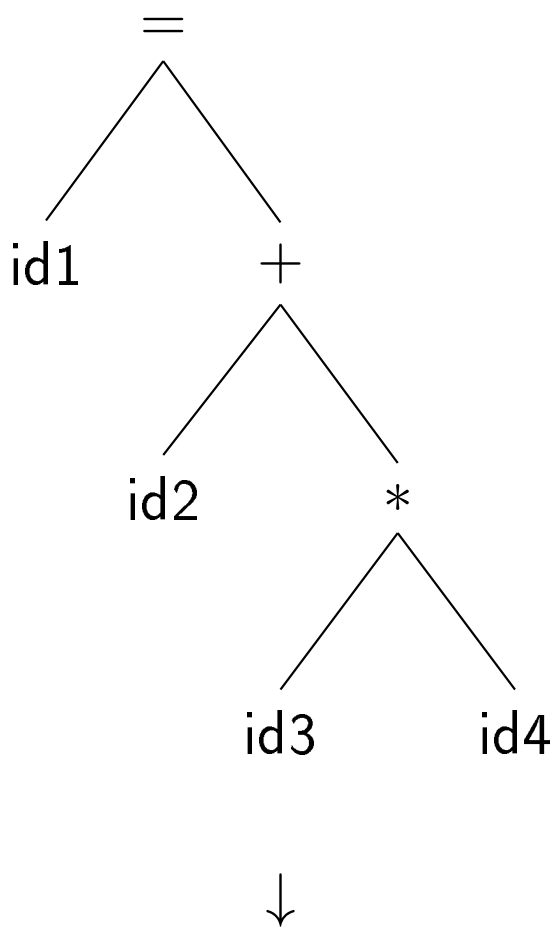
$$a = b + c * d$$



$$id1 = id2 + id3 * id4$$



Etapele traducerii exemplificate pentru o propoziție în limbaj de programare



Etapele traducerii exemplificate pentru o propoziție în limbaj de programare

```
load    id3
mul     id4
add     id2
store   id1
```

Proiectarea compilatoarelor

- **Abodarea monolitică**
  - compilator care traduce un limbaj relativ simplu
  - este scris de catre o singura persoana
- **Abordarea modulară**
  - limbajul de tradus este complex
  - calitatea codului obiect generat trebuie sa fie maximă
  - dezvoltarea compilatorului va fi împărțită la mai multe persoane care lucreaza în paralel
  - este mult mai ușor să se înlocuiasca un modul component cu o implementare mai bună
  - sau sa se adauge ulterior alte module
  - subiectul unui concurs lansat de Production Quality Compiler-Compiler Project (PQCC) al Universității Carnegie Mellon

Abordarea modulară

- **front-end** (partea "din față" a unui compilator)
  - analizează codul sursă cu scopul de a construi o reprezentare internă a acestuia **reprezentarea intermediară** - (de nivel mai scazut decat cel al limbajului in care a fost scris codul sursă)
  - crează, populează și gestionează **tabela de simboli**
    - o structura de date care memorează pentru fiecare atom lexical informațiile semantice asociate
- **back-end** (partea "din spate" a compilatorului)
  - analize, transformări și optimizari specifice pentru o anumită platformă
  - generarea de cod pentru un anumit sistem de operare și pentru un anumit procesor

Front-end

Etape:

- 1. **reconstruirea liniilor**
- 2. **analiza lexicală**
- 3. **preprocesarea**
- 4. **analiza sintactică**
- 5. **analiza semantică**
- 6. **generearea codului intermediar**

Back-end

Etape:

- 1. **analiza codului intermediar**
- 2. **optimizarea codului intermediar**
- 3. **generarea codului final**

Reconstruirea liniilor

- pregătește codul sursă pentru analiza gramaticală, convertind secvența de caractere într-o formă canonică
- necesară în cazul limbajelor în care
  - cuvintele cheie sunt marcate prin simboli speciali (**stropping**)
  - se permite prezența în cadrul identificatorilor a unui număr arbitrar de spații

Reconstruirea liniilor

- **stropping** refers to the method used to mark letter sequences as having a special property (most often being a keyword or certain type of variable/storage location)
  - în limbajul Algol68 cuvintele cheie sunt marcate prin prefixarea cu un apostrof: 'BEGIN
  - în limbajul Algol60 cuvintele cheie sunt marcate prin scrierea lor între apostrofi: 'BEGIN'
  - limbajele Ruby și Pearl utilizează caracterere sigil pentru a identifica caracteristicile unei variabile/constante
- stropping oferă posibilitatea de a utiliza **aceeași secvență de caractere pentru a reprezenta atomi lexicali cu semnificație diferită**

Reconstruirea liniilor

- **PHP** (largely inspired by Perl)
  - any name preceded by $ is a variable name
  - names not prefixed by this are considered constants or functions
- **Ruby**
  - ordinary variables lack sigils
  - $ is prefixed to global variables
  - @ is prefixed to instance variables
  - @@ is prefixed to class variables
- **Windows PowerShell**
  - variable names are prefixed by the $ sigil

Reconstruirea liniilor

- **Transact-SQL**
  - @ precedes a local variable or parameter name
  - @@ prefix distinguishes system variables (known as global variables)
- **C#**
  - any variable names may be prefixed with @ to allow the use of variable names that would otherwise conflict with keywords
  - when @ sigil is applied to string literals it changes the way they are interpreted (character escapes are not used and strings can extend over multiple lines)

Analiza lexicală

- **împarte codul sursă în atomi lexicali** - pe baza gramaticii regulare care definește sintaxa tipurilor de atomi lexicali
- **extrage informații privind locația în codul sursă** a fiecărui atom lexical (linie și coloană)
- detectează **atomii lexicali care nu aparțin limbajului** respectiv

Analiza lexicală

- un atom lexical (token) este o componentă singulară indivizibilă a unui limbaj - unitate lexicală
- scopul analizei lexicale este **transformarea șirului de caractere care formează codul sursă într-o secvență de atomi lexicali**
- dacă lipsește etapa de reconstrucție a liniilor, analizorul lexical trebuie să elimine și separatorii din codul sursă
- poate **elimina comentariile** din codul sursă, caz în care operația respectivă nu va mai trebui realizată în etapa de preprocesare

Tipuri de atomi lexicali

1. **Cuvintele rezervate**
   Ex:
   a) C: int, long, float, double, unsigned, void, if, else, for, while, do, switch, default, case, struct
   b) C++: class, inline, virtual
2. **Identificatorii**
   a) nume de variabile, de constante, de funcții, de structuri/clase, de namespace-uri, etc.
3. **Constante**
   Ex:
   a) intregi: (12, +3, 0)
   b) reale (12.0, +3.2, 0.0)
   c) caracter ('a', '\n')
   d) șir de caractere ("", "la facultate")
   e) bool (true, false)
   f) hexazecimale (0x45, 0xFA)
   g) ș.a.

Tipuri de atomi lexicali

4. **Operatori**
   A. aritmetici
      Ex: C: +,-,*,/,%
   B. logici
      Ex: C: &,|,!
5. **Semne de punctuație** (simboli sau grupuri de simboli cu rol delimitativ)
   Ex: C: spațiu, punct și virgulă
6. **Comentariile**
   Ex: C: liniile care încep cu //, sau șirurile de caractere dintre "/*" și "*/"

Analiza lexicală

- fiecare atom lexical este caracterizat prin doua elemente
  - **tipul**
    - necesar în analiza sintactică
    - este dat de clasa de atomi lexicali din care face parte
  - **valoarea semantică**
    - necesară în analiza semantică
    - este reprezentata de succesiunea de caractere care formeaza atomul lexical respectiv
- Ex: identificator, NumeVariabila

Analiza lexicală

- împărțirea se bazeaza pe faptul că în **analiza sintactică nu este important care atom lexical urmează, ci mai ales ce tip de atom lexical urmează**
- **exact care atom lexical a urmat va fi important în analiza semantică**
- Ex: pentru instrucțiunea "int a1;"
  - la analiza sintactică este important că după cuvântul cheie "int" urmează un identificator
  - nu are importanță că acest identificator este "a1"
  - iar la analiza semnatică va fi nevoie și de informația cum că identificatorul respectiv este chiar "a1"

Implementarea analizorului lexical

- analiza lexicală este **partea cea mai critică pentru un compilator, din punct de vedere al performanțelor**
- de exemplu, dacă analizorul lexical este implementat în forma comandată de către analizorul sintactic, atunci el va fi apelat pentru fiecare atom lexical din codul sursă
- rezultă deci că această componentă trebuie să fie foarte eficientă
- căutarea liniară sau utilizarea unei succesiuni foarte lungi de instrucțiuni *if* la procesarea unui atom lexical trebuie evitate

Implementarea analizorului lexical

- analizorul lexical ar trebui să ofere și
  - o bună **diagnosticare a erorilor** care survin la analiză
  - un **mecanism eficient de tratare** a acestora
- Ex:
  - dacă a fost folosită o constantă numerică cu valoarea mai mare decât valoarea maximă acceptată de către limbaj, atunci:
    - analizorul lexical ar trebui să emită un mesaj în acest sens
    - și să înlocuiască valoarea constantei cu valoarea maximă
  - dacă a fost folosit un șir de caractere cu o lungime mai mare decât lungimea maximă acceptată, atunci:
    - analizorul lexical trebuie să emită un mesaj în acest sens
    - și să trunchieze șirul la lungimea maximă acceptată

Implementarea analizorului lexical

- în vederea generării de mesaje de eroare sau de atenționare cât mai utile:
  - analizorul lexical trebuie **să extragă și informații privind locația în codul sursă** a fiecărui atom lexical
  - acestea vor fi folosite nu numai pentru mesajele rezultate în urma analizei lexicale, ci și pentru cele de la analiza sintactică și semantică

Implementarea analizorului lexical

- a scrie un analizor lexical înseamnă a scrie un program care este capabil să recunoască și să extragă toate cuvintele (secvențele de caractere) care aparțin limbajului supus analizei
- fiecare tip de atomi lexicali constituie un **limbaj regular**
- sintaxa atomilor lexicali este definită, de obicei, de către o **gramatică regulară**
- pentru recunoașterea lor se va utiliza un **automat finit determinist** (conform celor anterior studiate la **Limbaje regulare**)

Tipuri de analizoare lexicale

După relația dintre analizorul lexical și cel sintactic:

Analizor lexical independent de analizorul sintactic

- analizorul lexical este lansat pentru a analiza întregul cod sursă
- generează întreaga secvență de atomi lexicali
- actiunea analizorului sintactic este succesivă acțiunii analizorului lexical

Tipuri de analizoare lexicale

Analizor lexical comandat de către analizorul sintactic

- analizorul lexical este implementat sub forma unei proceduri (funcții) care, la fiecare apel, întoarce un atom lexical
- poate fi implementat pentru a fi utilizat în două modalități:
  - **prin apel direct**: analizorul sintactit cere **următorul atom lexical** care poate fi extras din codul sursa
  - **prin apel indirect**: analizorul sintactic cere un anumit atom lexical

Tipuri de analizoare lexicale

După structura analizorului lexical:

- analizor lexical **monobloc (unipas)**:
  - primește ca și intrare secvența de caractere care formează codul sursă și returnează secvența de atomi lexicali, fără a trece prin etape intermediare
- analizor lexical **structurat (multipas)**:
  - împarte codul sursă în atomi lexicali prin parcurgerea mai multor etape:
    - 1. transliterarea
    - 2. explorarea
    - 3. selectarea

Exemplu multipass

while a-b>0 do b:=b+7

1. Șirul transliterat

c.l c.l c.l c.l c.l c.b c.l c.o c.l c.o c.c c.b c.l c.l c.l c.o c.o c.l c.o c.c

w h i l e a − b > 0 d o b := b + 7

2. Șirul explorat

↓while ↓a ↓- ↓b ↓> ↓do ↓b ↓:= ↓b ↓+ ↓7

3. Șirul selectat

↓while ↓a ↓- ↓b ↓> ↓do ↓b ↓:= ↓b ↓+ ↓7

cuv.cheie id op id op cuv.cheie id attrib. id op const.intreaga

Preprocesarea

- este necesară în cazul limbajelor care utilizează:
  - **macrourile** (pentru înlocuirea acestora)
    - #define min(X,Y) ((X) < (Y) ? (X):(Y))
    - x = min(a,b); $\Longrightarrow$ x = ((a) < (b) ? (a):(b));
    - y = min(1,2); $\Longrightarrow$ y = ((1) < (2) ? (1):(2));
    - z= min(a+28,*p); $\Longrightarrow$ z = ((a+28) < (*p) ? (a+28):(*p));
  - **compilarea condiționată**
    - permite ca o anumita parte din codul sursa să fie ignorată în timpul compilării, în funcție de anumite condiții
    - se realizează prin intermediul directivelor condiționale: #if,#ifdef,#ifndef
  - **pentru eliminarea comentariilor**
  - etc.
- se realizează prin manipularea atomilor lexicali

Analiza sintactică

- verifică dacă **atomii lexicali sunt în ordinea corectă** - pe baza gramaticii independente de context care definește sintaxa propozițiilor limbajului sursă
- **crează și populează tabela de simboli** (structură auxiliară arborelui sintactic)
- detectează **dacă atomii lexicali sunt scriși într-o ordine eronată**

Analiza sintactică

- implică analiza gramaticală a secvenței de atomi lexicali pentru a identifica structura sintactică a programului
- în această fază se construiește **arborele de derivare**
  - secvența liniară a atomilor lexicali este înlocuită cu o structură arborescentă
- poate fi abordată în două feluri (conform celor anterior studiate la **Limbaje independente de context**):
  1. ascendent (bottom-up)
  2. descendent (top-down)

Analiza semantică

- Domenii de cercetare ale semanticii – sensurile cuvântului semantică
  1. Semantica – studiul semnificației termenilor în afara oricărui context - **lexicografia**
  2. Semantica – studiul sistemelor conținutului -**semantica structurală**
  3. Semantica – studiul raporturilor dintre termen (enunț) și referent – **studiul referinței**
  4. Semantica – **studiu al condițiilor de adevăr ale enunțurilor**
  5. Semantica – studiul **sensului** particular pe care termenii sau enunțurile le capătă **în context, în ansamblul textual**

Analiza semantică

- **Verificarea tipului (type checking)** – se verifică dacă există erori privind tipurile de date folosite
  - operatorul modulo - ambii operanzi trebuie să fie de tip întreg: $a\%b$
  - operatorii logici - operanzii trebuie să fie de tip boolean: !a
  - accesarea valorii unui element dintr-un vector - primul identificator trebuie să reprezinte un vector, iar al doilea trebuie să fie un întreg: a[b]
- **Object binding** – asocierea dintre definițiile funcțiilor și claselor și utilizarea acestora
- **Definite assigment** – verificarea faptului că toate variabilele locale au fost inițializate înainte de a fi utilizate

Analiza semantică

- pentru a putea realiza analiza semantică, compilatorul adaugă informații semantice arborelui de derivare completând în mod corespunzător **tabela de simboli**
- **popularea tabelei de simboli este în general incorporată în etapa de analiză sintactică**
- tabela de simboli este implementată prin **tabele de dispersie** (tabele hash) sau prin **tabele indexate**

Analiza semantică

În tabela de simboli, fiecare atom lexical este reprezentat prin:

- un identificator unic
- un cod numeric care specifică tipul acestuia
- o serie de informații semantice specifice tipului
  - pentru un atom lexical de tip nume de variabilă:
    - numele
    - tipul
    - domeniul de valabilitate (scope)
    - dacă a fost sau nu inițializată
    - etc.
  - pentru un atom lexical de tip nume de funcție:
    - numele
    - tipul valorii returnate
    - numărul de parametrii
    - tipul și modul de transfer al fiecărui parametru
  - etc.

Domeniul de valabilitate (scope)

- **reprezintă porțiunea din program în care un identificator dat este accesibil**
- același identificator (același nume) se poate referi la entități semantice diferite, în porțiuni diferite din program
- singura condiție este ca domeniile de valabilitate ale acestor entități identificate cu același nume să nu se suprapună
- ex: utilizarea aceluiași nume:
  - pentru variabile de tip diferit
  - pentru variabile și funcții sau metode
  - pentru variabile și nume de clase
  - ș.a.

Domeniul de valabilitate (scope)

- un identificator poate avea un domeniu de valabilitate restrâns
  - variabilele locale unei funcții pot fi accesate numai în cadrul codului funcției
  - variabilele globale pot fi accesate de oriunde din cadrul programului în care au fost declarate
  - atributele și metodele definite într-o clasă sunt accesibile numai pentru instanțele clasei respective

Domeniul de valabilitate (scope)

- **static**
  - depinde numai de textul programului și nu este influențat de comportamentul din momentul rulării
  - legătura dintre date și identificator (în cazul variabilelor), și dintre cod și identificator (în cazul metodelor și funcțiilor) este statică
  - ex: majoritatea limbajelor de programare
- **dinamic**
  - depinde de execuția programului
  - o variabilă cu domeniu de valabilitate dinamic se referă la cea mai apropiată legătură din execuția programului
  - ex: Lisp, SNOBOL

Domeniul de valabilitate (scope)

- legăturile dintre identificatori și date sau cod sunt definite de către:
  - declarațiile claselor (rezultă numele claselor)
  - definițiile metodelor (rezultă numele metodelor)
  - declarațiile de variabile (rezultă identificatorii variabilelor)
  - parametrii formali din definițiile funcțiilor (rezultă identificatorii variabilelor locale)
  - definițiile atributelor unei clase/structuri (rezultă identificatorii membrilor clasei/structurii)

Contexte

- contextele **păstrează definițiile și declarațiile curente necesare analizei semantice**
- contextele sunt implementate prin intermediul tabelelor de simboli
- cu alte cuvinte, tabela de simboli este cea care memorează legăturile curente existente între identificatori și date, respectiv cod

Contexte

- în general un context este o tabelă de dispersie asupra căreia se pot realiza următoarele operații:
  - **deschidere** context nou
  - **adăugarea** unui nou identificator împreună cu informațiile de rigoare
  - **căutarea** unui identificator (verificarea dacă un identificator a fost definit în contextul curent)
  - **extragerea** informațiilor privind un identificator existent
  - închiderea unui context inclusiv cu **ștergerea** identificatorilor din contextul respectiv

Contexte

- în cadrul **analizei sintactice**, pe măsură ce se întâlnesc:
  - instrucțiunile de declarare a variabilelor/funcțiilor
  - instrucțiunile de definiție a tipurilor
  - ș.a.

  se adaugă **contextului curent** informațiile despre fiecare în parte

- în cadrul **analizei sintactice**, la întâlnirea:
  - instrucțiunilor care folosesc variabile/tipuri/funcții

  se face **analiza semantică** a acestora pe baza informațiilor memorate în contextul curent

Tipuri de date

- tipul de date este un set de valori împreună cu operațiile care se pot efectua asupra acestora
- clasificare:
  - **tipuri simple, de bază** (int, char, float, double, bool, union)
    - sunt suportate chiar de către procesor
    - nu au corespondență în lumea reală
  - **tipuri compuse** (variabile multidimensionale, pointeri, structuri, clase)
    - încearcă să ofere o corespondență cu lumea reală
  - **tipuri complexe** (liste, arbori)
    - nu sunt suportate direct de către limbaj, dar pot fi implementate

Tipuri de bază

- nu au informație semantică suplimentară
  - cu excepția tipului enum
  - în acest caz vor trebui memorate în tabela de simboli numele membrilor enum-ului și valoarea asociată fiecăruia în parte
- deoarece sunt deja implementate în hardware, ele nu trebuie create
- variabilele declarate pentru un tip de bază trebuie să aibă o referință către tipul lor

Tipuri compuse - variabile multidimensionale

- în tabela de simboli, pentru un atom lexical de tip nume de **variabilă multidimensională**, trebuie memorate:
  - numele variabilei
  - tipul variabilei (de bază sau compus)
  - numărul de dimensiuni
  - dimensiunea/dimensiunile
  - dacă a fost sau nu inițializată

Tipuri compuse - pointeri

- în tabela de simboli, pentru un atom lexical de tip nume de **pointer**, trebuie memorate:
  - numele variabilei
  - tipul variabilei
  - dacă a fost sau nu inițializată

Tipuri compuse - structuri, clase

- trebuie salvată o listă de perechi *nume-tip* care să memoreze informațiile despre membrii tipului (atribute și metode)
- aceste informații sunt salvate ca și context, în tabela de simboli

Type-checking

- **verifică dacă operațiile folosesc ca și operanzi tipuri potrivite**
- această validitate a tipului versus operația curentă depinde de fiecare tip în parte și de limbajul sursă
- orice nerespectare în această privință va ridica o eroare de tip
- dacă toate erorile de tip sunt verificate de către compilator, atunci limbajul este **strongly typed**
- altfel, el se numește **weakly typed**

Type-checking

- **strongly typed**
  - o variabilă poate fi folosită numai în conformitate cu tipul cu care a fost definită
  - ex: C#
  - int a; Console.Writeline("{0}", a);
- **weakly typed**
  - o variabilă poate fi folosită și altfel decât în conformitate cu tipul cu care a fost definită
  - ex: C, C++
  - int a; printf("%c", a);

Analiza semantică

Type-checking

- **statică**, adică la compilare (C, C++)
  - descoperă foarte multe erori încă în etapa de compilare
  - elimină execuția de cod suplimentar la rulare
- **dinamică**, adică la rularea programului (variabilele anonime din C#, Perl, Phyton, Ruby)
  - compilatorul va adăuga în codul obiect generat cod suplimentar pentru verificarea de tip
  - tipurile statice sunt restrictive
- **fără verificare de tip** (limbajul de asamblare)

Type-checking

- **Verificarea de tip** – verificarea programelor în care toate elementele au atribuit un tip
- **Inferența de tip** – completarea informației privind tipul, informație care lipsește, pornind de la context
- **Sinteza de tip** – determinarea tipului unei construcții (expresii) pe baza tipurilor membrilor ei

- concepte diferite, dar denumirile lor sunt adesea folosite interschimbabil

Type-checking

- construcțiile limbajelor care au asociat un tip sunt:
  - constantele
  - variabilele
  - funcțiile
  - expresiile
  - instrucțiunile
    - condiția lui *if*, *while*, *do-while* în C# trebuie să aibă tipul bool
    - valoarea pe care se face *switch* trebuie să aibă un tip întreg (C/C++), sau inclusiv string (C#)

Type-checking

- formalismul verificării de tip este reprezentat de către **regulile logice ale inferenței**
- regulile de inferență au forma:
  Dacă Ipoteza este adevărată, atunci și Concluzia este adevărată.
- raționamentul pentru verificarea de tip este:
  Dacă E1 și E2 au anumite tipuri, atunci E3 are un anumit tip.
- ex: int a,b=3; a = b + 3.4;

Type-checking

- există **tipuri compatibile**
- compatibilitatea în ambele direcții vs. compatibilitatea într-o singură direcție
- ex:
  - int este compatibil cu float, dar float nu este compatibil cu int
- **conversii de tip (cast)**
  - **implicit cast**
    float a = 3;
  - **explicit cast**
    int a = (int)3/4;

Type-checking - subtipuri

- un subtip poate fi folosit totdeauna în locul tipului părinte
- ex:
  - enumerările – tipul părinte este tipul întreg
  - moștenirea
  - polimorfismul

Type-checking - non typed variables

- C#
  - ex: var x = 10;
  - după atribuire, x primește tipul int – tip stabilit **static, la compilare**
- JavaScript
  - ex: var x = 10;
  - variabila x rămâne fără tip chiar și după atribuire, în continuare putând fi folosită pentru a i se atribui orice alt tip de valoare – tip **stabilit dinamic, la rulare**
- tipul se deduce astfel din context (inferență de tip)

Analiza semantică

Acțiuni

- pentru **declarații de variabile**:
  - adăugarea de informații în tabela de simboli
  - verificarea redefinirii unui identificator
  - dacă este o declarare cu inițializare, atunci se verifică tipul valorii atribuite
  - ș.a.
- pentru **declarații de funcții**:
  - adăugarea de informații în tabela de simboli
  - verificarea redefinirii unui identificator
  - relațiile de moștenire sunt corecte
  - ș.a.

Acțiuni

- pentru **declarații de structuri, clase**:
  - adăugarea de informații în tabela de simboli
  - verificarea redefinirii unui identificator
  - metodele din cadrul claselor au fost definite o singură dată
  - relațiile de moștenire sunt corecte
  - ș.a.

Analiza semantică

Acțiuni

- pentru **instrucțiuni**:
  - verifică faptul că nu se folosesc identificatori care să nu fi fost definiți
  - identificatorii sunt folosiți în conformitate cu structura lor
    - a.b (a este un tip compus și are ca și membru pe b)
    - v[10] (v este o variabilă multidimensională)
    - f(5,"ATM") (f este o funcție care acceptă doi parametrii, primul de tip întreg, iar al doilea de tip șir de caractere)
  - identificatorii sunt accesați în conformitate cu modul de acces declarat (public/private/static/const, etc.)
  - verificarea de tip
- pentru **expresii**:
  - verificarea de tip
  - eventual sinteza și/sau inferența de tip

Generarea codului intermediar

- Ca și cod intermediar pot fi folosite mai multe variante:
  1. Chiar **arborele de derivare** construit anterior
  2. forma **postfix** (reverse Polish)
  3. **three address code**
  4. etc.

Generarea codului intermediar

- **în forma postfix (forma Poloneză inversă)** fiecare operator urmează după toți operanzii săi
  - $3 + 4$ se scrie $3\ 4\ +$
- **codul cu trei adrese** poate fi scris în două forme:
  $A := B\ op\ C$
  $op\ A, B, C$
  - translatarea expresiei $A/B^*C + D$ în cod cu trei adrese (prin introducerea și utilizarea a trei variabile temporare) este:
    - $T1 := A/B$ DIV T1,A,B
    - $T2 := T1^*C$ MUL T2,T1,C
    - $T3 := T2 + D$ ADD T3,T2,D

Analiza codului intermediar

Constă în adunarea de informații despre program, pornind de la codul intermediar construit pe baza codului sursă

- Analiza fluxului de date cu scopul de a construi use-define chaines
- Analiza dependențelor
- Analiza alias-urilor
- Analiza pointerilor
- Analiza escape
- Construirea grafului de apeluri
- Construirea grafului fluxului de control

Informațiile obținute în această etapă vor sta la baza optimizării ulterioare

Optimizarea codului intermediar

- codul intermediar este transpus într-o formă echivalentă care este:
  - **mai rapidă** - optimizarea timpului de execuție
  - și/sau de **dimensiuni mai mici** - optimizarea memoriei consumate
- cuprinde:
  - Inline Expansion
  - Dead code elimination
  - Constant propagation
  - Loop transformation
  - Register allocation
  - Automatic parallelization

Optimizarea codului intermediar

- optimizarea se poate face și:
  - direct asupra codului sursă înainte de compilare
  - sau asupra codului obiect generat (în limbaj de asamblare sau în limbaj mașină)
- efectuarea optimizării duce la încetinirea procesului de compilare, dar codul rezultat va rula mai rapid și/sau va necesita mai puțină memorie

Optimizarea globală

Necesită analiza și modificarea unor părți mari de cod

- ex: **optimizarea buclelor**
  - pentru codul sursă:

```
for(int r = 0; r < 100; r++ )
{
    // ... alte calcule
    float pi = 4 * arctan(1);
    aria = pi * r * r;
}
```

  - execuția ar fi mai eficientă dacă instrucțiunea *float pi=4*arctan(1);* ar fi mutată în afara buclei *for*
  - mutarea ei se poate face mult în fața instrucțiunii *for*, de unde și clasificarea ca optimizare globală

Optimizarea locală

- **optimizarea peephole** - instructiunile sunt analizate in grupuri de cateva (de exemplu câte două), iar **modificările se fac numai in interiorul unui grup**
- ex: pentru un procesor cu un singur registru
  - codul intermediar in trei adrese:
    A:=B+C <-> ADD A,B,C
    X:=A*Z <-> MUL X,A,Z
  - va fi translatat in codul obiect:
    LOAD B
    ADD C
    STORE A
    LOAD A
    MUL Z
    STORE X
  - instrucțiunile scrise cu roșu ar putea fi eliminate

Generarea codului final

- codul intermediar optimizat este translatat în limbajul de ieșire, și anume limbajul mașină al procesorului țintă

- spre deosebire de restul etapelor compilării, **generarea codului final este dependentă de mașina țintă**:
  - arhitectura diferă de la un tip de mașină la altul
  - fiecare tip de procesor are propriul set de instrucțiuni
  - fiecare tip de arhitectură are particularități care trebuie luate în considerare
- ex: pentru un procesor care are un singur registru
  - din codul intermediar:
    A:=B+C (scris ADD A,B,C în formatul codului cu trei adrese)
  - se generează:
    LOAD B
    ADD C
    STORE A

Generarea codului final

- trebuie să ducă la obținerea unui **fișier executabil**
- regulile pe care trebuie să le respecte un fișier executabil pe o anumită arhitectură (relativ la procesor și la sistemul de operare) sunt specificate în ceea ce se numește **application binary interface (ABI)**

Generarea codului final

- ABI descrie:
  - instrucțiunile procesorului țintă
  - modul de utilizare al regiștrilor
  - convențiile de apel: modul în care funcțiile primesc parametrii de la apelant, precum și modul în care ele pot returna valori apelantului
  - programarea instrucțiunilor (pentru instrucțiunile care pot fi executate în paralel)
  - organizarea memoriei
  - formatul fișierului executabil
  - ș.a.

Generarea codului final

- Exemple de ABI:
  - **System V Application Binary Interface**
    `http://www.cs.albany.edu/~sdc/csi402/ELFdocs/gabi3.1.pdf`
  - **System V Application Binary Interface - Intel386 Architecture Processor Supplement**
    `http://math-atlas.sourceforge.net/devel/assembly/abi386-4.pdf`
  - **System V Application Binary Interface - AMD64 Architecture Processor Supplement**
    `https://courses.cs.washington.edu/courses/cse351/12wi/supp-docs/abi.pdf`
  - **System V Application Binary Interface - MIPS RISC Processor Supplement**
    `http://math-atlas.sourceforge.net/devel/assembly/mipsabi32.pdf`
  - **System V Application Binary Interface - SPARC Processor Supplement**
    `http://math-atlas.sourceforge.net/devel/assembly/abi_sysV_sparc.pdf`

Detectarea erorilor

- **în fiecare etapă a compilarii pot fi identificate erori specifice**
- Exemple:
  - în analiza lexicală: un șir de caractere din codul sursa nu corespunde niciunui atom lexical
  - în analiza sintactică: șirul de atomi lexicali identificați nu formează o propoziție corectă din punct de vedere sintactic
  - în analiza semantică: se gasesc erori la verificarea tipurilor

Detectarea erorilor

- cum se face tratarea erorilor: după identificarea unei erori analiza continuă sau se oprește?
- raportarile unei erori trebuie sortate în ordinea de apariție în cadrul codului sursă și afișate în mod corespunzator utilizatorului
- erorile depistate în urma verificarilor efectuate pot duce la:
  - la afișarea unor mesaje de eroare și rejectarea programului (**nu se poate genera cod obiect**)
  - la afisarea de mesaje de atenționare (warnings) și **generarea de cod obiect**
  - **generarea de cod obiect**

A Compiler With a Sense of Humor

These are some of the **error messages produced by Apple's MPW C compiler**. They are all real. (If you must know I was bored one afternoon and decompiled the String resources for the compiler.)

- "String literal too long (I let you have 512 characters; that's 3 more than ANSI said I should)"
- "...And the lord said, 'lo, there shall only be case or default labels inside a switch statement'"
- "A typedef name was a complete surprise to me at this point in your program"
- "You can't modify a constant, float upstream, win an argument with the IRS, or satisfy this compiler"
- "This struct already has a perfectly good definition"
- "type in (cast) must be scalar; ANSI 3.3.4; page 39, lines 10-11 (I know you don't care, I'm just trying to annoy you)"

A Compiler With a Sense of Humor

- "Can‘t cast a void type to type void (because the ANSI spec. says so, that‘s why)"
- "Huh?"
- "Can‘t go mucking with a ’void*’"
- "We already did this function"
- "This label is the target of a goto from outside of the block containing this label AND this block has an automatic variable with an initializer AND your windows wasn‘t wide enough to read this whole error message"
- "Call me paranoid but finding ’/*’ inde this comment makes me suspicious"
- "Too many error on one line (make fewer)"
- "Symbol table full - fatal heap error; please go buy a RAM upgrade from your local Apple dealer"

Teoria traducerii - Intrebări recapitulative

- Arătați care tipuri de analiză semantică au loc după generarea codului final.
- Explicați de ce este nevoie ca tabelele de simboli să fie implementate prin tabele de dispersie.
- Explicați de ce propozițiile ambigue nu pot fi procesate de către un compilator bazat pe algoritmii studiați la acest curs.
- Explicați de ce un cod interpretat rulează mai încet decât un cod compilat.

Teoria traducerii - Intrebări recapitulative

- Explicați utilitatea ABI pentru scrierea unui compilator.
- Explicați diferența dintre un mesaj de tip warning și unul de tip error în funcționarea unui compilator.
- Explicați cum sunt codate regulile de analiză semantică de către analizorul semantic.
- Explicați de ce este nevoie de tabela de simboli în cadrul procesului de compilare.

Teoria traducerii - Intrebări recapitulative

- Explicați diferența dintre verificarea de tip statică și cea dinamică, din cadrul analizei semantice.
- Explicați motivația practică a împărțirii în cele două componente (front-end și back-end) a procesărilor executate de către un compilator.
- La compilarea unui program se afișează mesajul de eroare "too many characters in character constant". Având în vedere etapele compilării, scrieți ce tip de eroare este aceasta și explicați.

Bibliografie

- Paul N. Hilfinger, **CS 164 Reader #1**
  `http://inst.eecs.berkeley.edu/~cs164/sp10/notes/notes.pdf`

- CPS 343/543, 444/544 **Lecture notes: Formal languages and grammars**
  `http://goo.gl/A1ZnJZ`

- R.B. Yehezkael, **Course Notes on Formal Languages and Compilers**
  `http://goo.gl/TpuoAT`

- Umberto Eco, **Cinci sensuri ale cuvântului semantică**, în **De la arbore la labirint**, editura POLIROM, 2009

Bibliografie

- T. K. Prasad, **Code Generation**
  `http://cecs.wright.edu/~tkprasad/courses/cs781/L23CG.pdf`
- **ATLAS Assembly Homepage**
  `http://math-atlas.sourceforge.net/devel/assembly/`