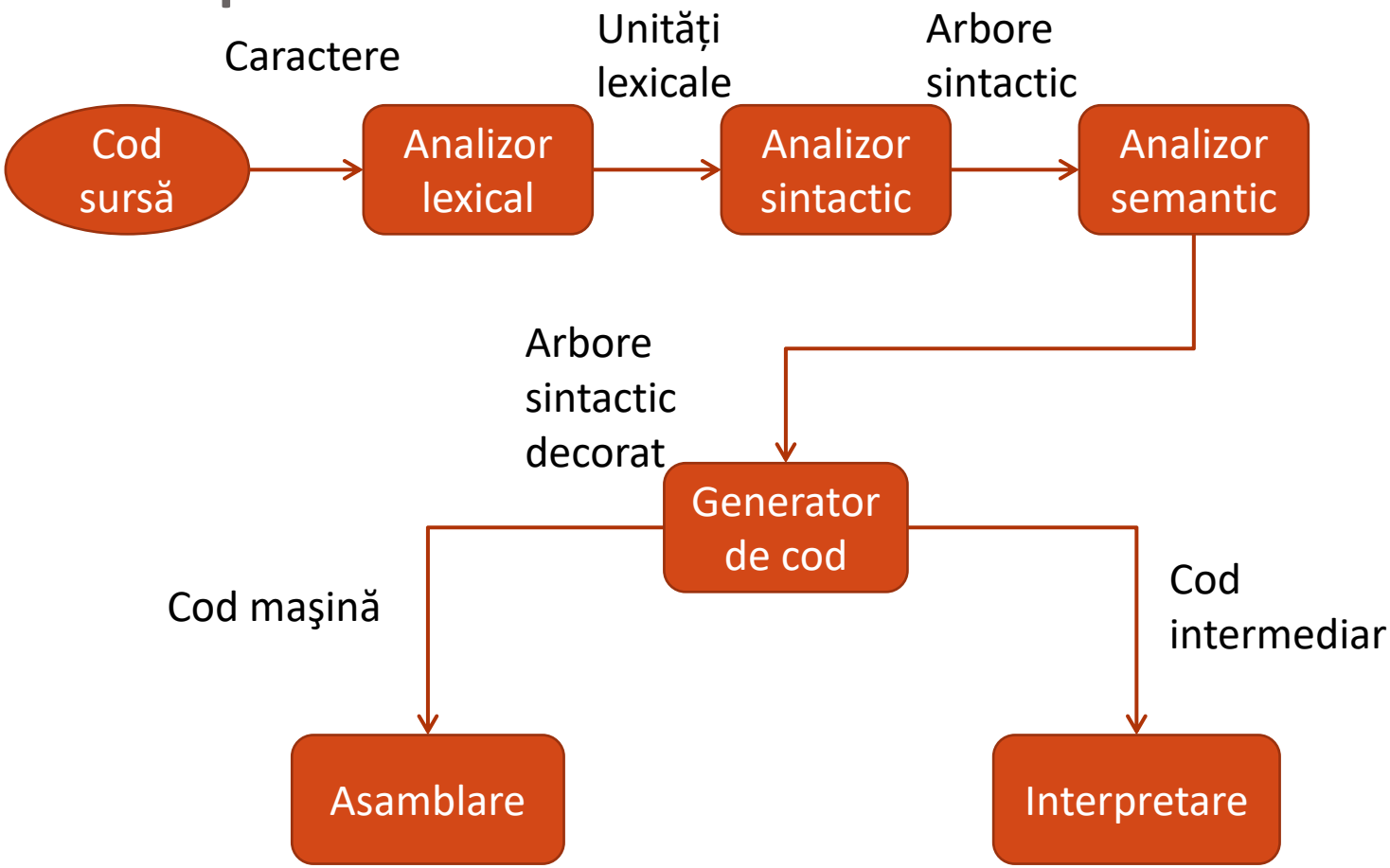
Limbaje formale, automate şi compilatoare

Curs 12

Compilare


Caractere
Unități lexicale
Arbore sintactic
Cod sursă
Analizor lexical
Analizor sintactic
Analizor semantic
Arbore sintactic decorat
Generator de cod
Cod maşină
Cod intermediar
Asamblare
Interpretare

Recapitulare

- Analiza lexicală
  - Validează tokeni
- Analiza sintactică
  - Validează arborele sintactic
- Analiza semantică
  - Detectează toate celelalte erori
  - Ultimul pas de analiză

Tipuri şi declarații (1)

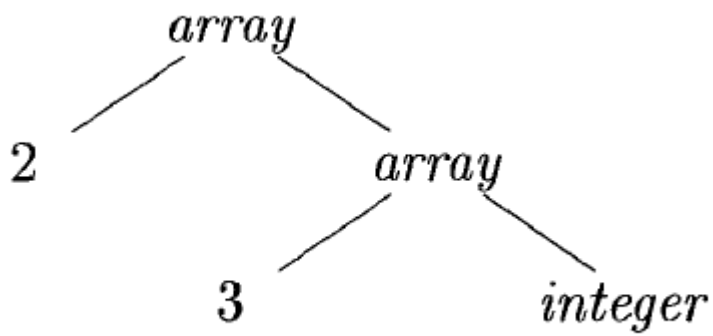
- Verificări şi traduceri
  - Verificarea are în vedere compatibilitatea de tipuri
  - Traducerea are în vedere spatiul necesar pentru memorarea unei variabile
- Expresii pentru tipuri.
  - Un tip de bază
  - Un nume de tip
  - O înregistrare este o structură de date cu câmpuri cu nume
  - Expresii pentru tipuri pot fi formate cu constructorul pentru funcții. $s \rightarrow t$ semnifică o funcție de la tipul s la tipul t

Tipuri şi declaraţii (2)

- Expresii pentru tipuri
  - Dacă *s* și *t* sunt expresii pentru tipuri, atunci *sxt* este expresie pentru tip
  - Expresiile pentru tip pot conține variabile ale căror valori sunt expresii pentru tip
- Expresiile pentru tip pot fi reprezentate ca un graf aciclic orientat
  - Nodurile interne sunt constructori
  - Frunzele sunt expresii de tip elementare

Tipuri şi declaraţii (3)

- Exemplu
  - *int[2][3]*
  - Este interpretată ca un tablou de tablouri de int
  - Expresia pentru tip echivalentă este array(2, array(3, integer))



Echivalența de tipuri (1)

- Calculată cu reguli de potrivire
  - Dacă două expresii sunt egale, atunci returnează tipul, altfel eroare
  - Pot să apară ambiguități atunci cînd tipurile sunt redenumite, iar redenumirile sunt folosite în expresii pentru tip

Echivalența de tipuri (2)

- În cazul reprezentării cu grafuri aciclice orientate, două tipuri sunt echivalente structural dacă și numai dacă
  - Sunt același tip de bază
  - Sunt obținute prin aplicarea aceluiași constructor peste tipuri echivalente structural
  - Unul este o redenumire al celuilalt

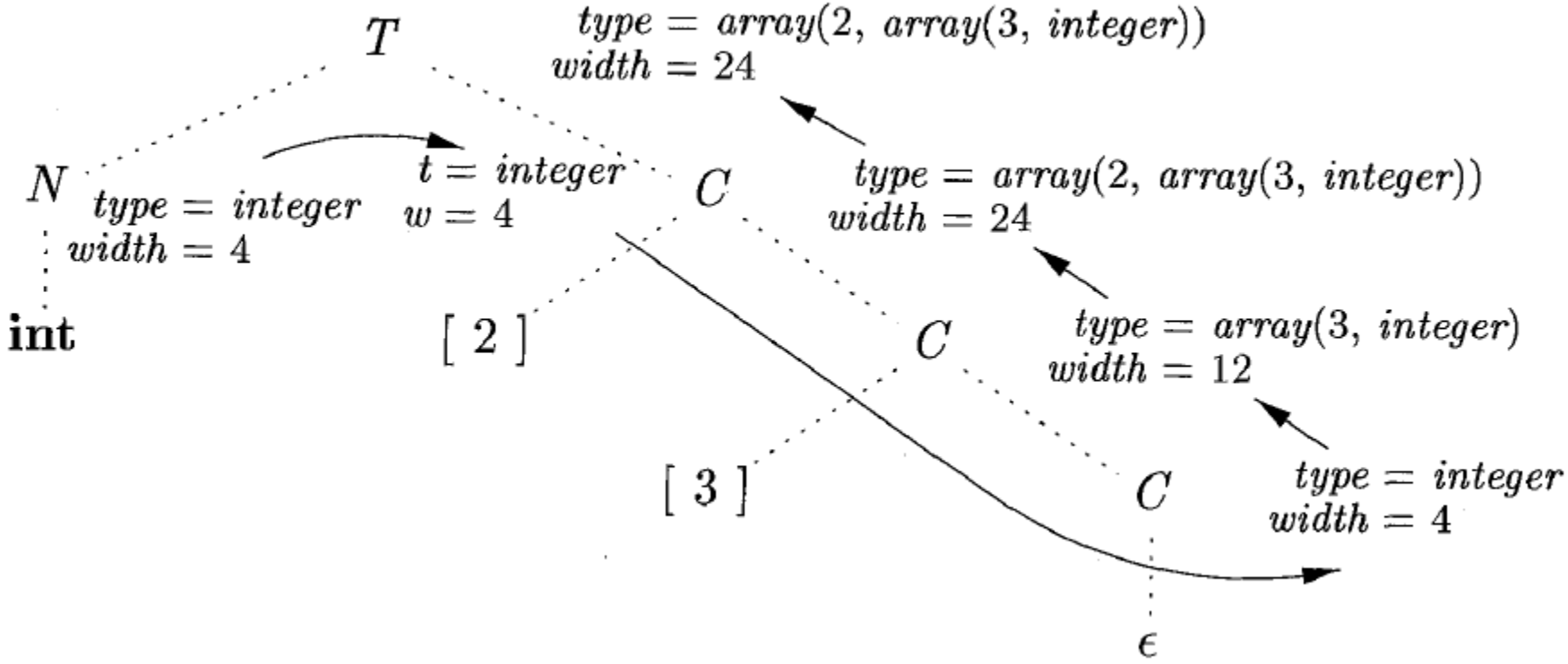
Stocarea locală de tipuri (1)

- Știind tipul unui identificator, putem determina cantitatea de memorie necesară acelui identificator
  - La compilare sunt folosite pentru a asocia fiecărui identificator o adresă relativă (memorate în tabelul de valori)
  - Pentru tipuri de date cu dimensiune variabilă (ex. string) sau a căror dimensiune nu poate fi cunoscută până la runtime, este alocată o cantitate de memorie pornind de la o adresă

Stocarea locală de tipuri (2)

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

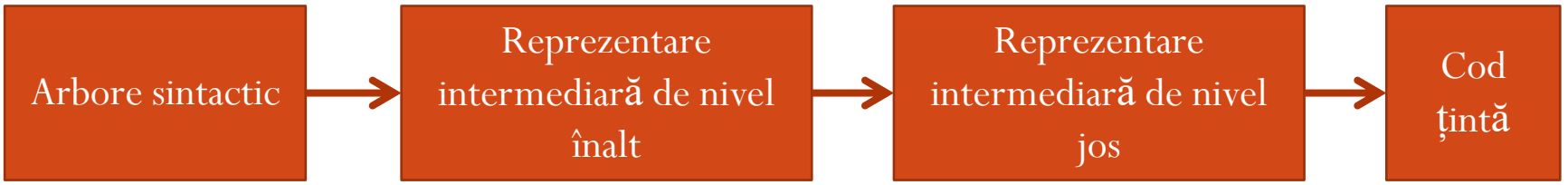
Stocarea locală de tipuri (3)



Generare de cod

- Transformă analiza semantică în cod executabil
- Limbajul trebuie transformat diferit pentru orice sistem
  - Generare de cod intermediar – detașează partea de analiză și interpretare de generarea de cod mașină

Generare de cod



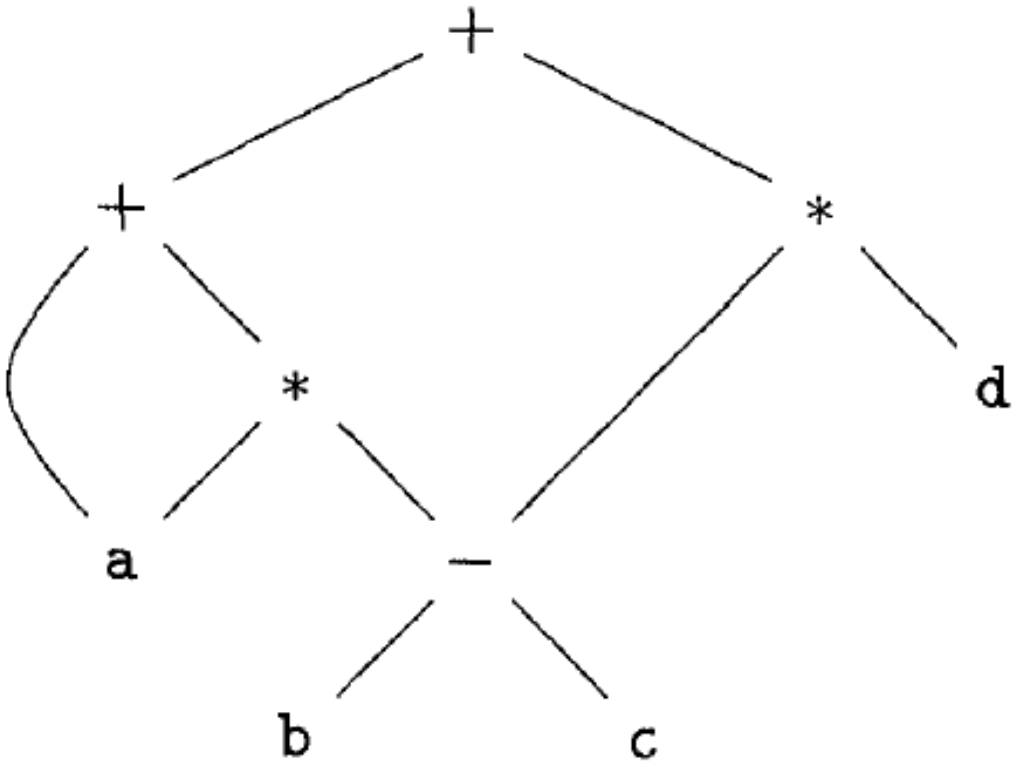
Compilatoarele pot folosi multiple straturi de reprezentări intermediare.

Cod intermediar

- Arbori sintactici decorați
- Cod cu trei adrese
  - x = y op z
- Cod intermediar
  - De nivel înalt
    - apropiați de limbajul sursă (arbori sintactici)
    - buni pentru sarcini de nivel înalt (ex. verificare de tip)
  - De nivel jos
    - alocarea de memorie și regiștri
    - selecția de instrucțiuni

Arbori sintactici

- Grafuri orientate aciclice
  - a + a * (b - c) + (b - c) * d



Grafuri aciclice orientate

- Gramatica de mai jos poate construi arbori sintactici sau arbori aciclici orientați
- Funcțiile *Leaf* și *Node* vor crea noduri noi dacă nu există deja noduri egale
- Dacă nodul există deja, va fi returnat în locul unui nod nou

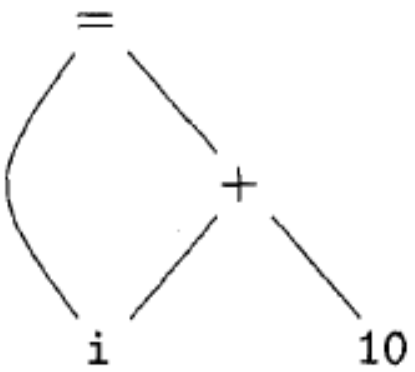
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

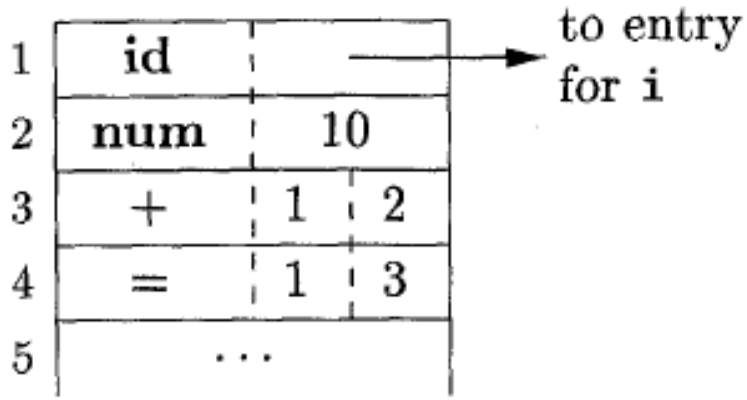
Grafuri aciclice orientate

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Grafuri aciclice orientate – reprezentare tabelară

- Nodurile arborelui sunt reprezentate într-un tablou
  - Fiecare rând reprezintă un nod
  - Prima celulă reprezintă operatorul
  - Fiecare nod are o valoare asociată (pointer sau constantă)





Grafuri aciclice orientate – reprezentare cu tablouri

- Nodurile sunt referite prin indicele rândului
  - Indicii nodului sau expresiei sunt numiți *value number*
  - În practică sunt folosiți pointeri
  - Valorile numerice pot fi folosite pentru a construi grafuri aciclice orientate

Metoda value number de construcție de arbori orientați

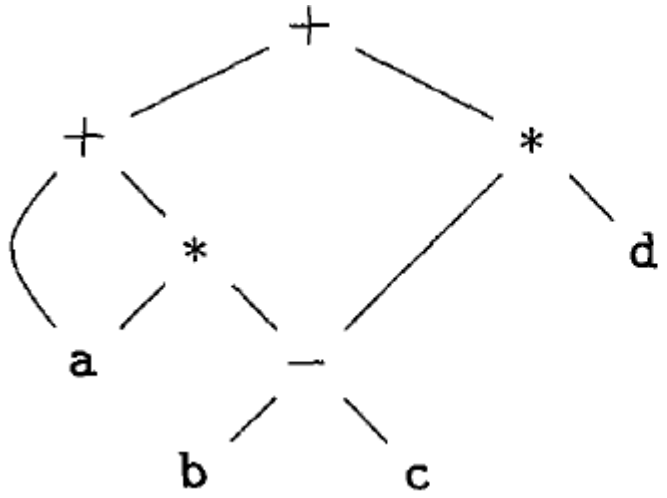
- **Intrare**: eticheta operatorului și membrii operației
- **Ieșire**: Valoarea numerică pentru nodul cu signatura <op, l, r>
- Algoritm
  - Caută în tablou nodul M cu eticheta op, și fiii stânga – dreapta l și r
  - Dacă există, return M
  - Altfel, creează un nou nod N, cu eticheta op și fiii l și r; return N

Coduri cu trei adrese

- Reprezentat ca o secvență de maxim trei elemente
  - Operator
  - Operand stânga
  - Operand dreapta
  - Ex: a + b * c
    - t1 = b * c
    - t2 = a + t1
- Poate fi folosit pentru descompunerea de operații aritmetice, instrucțiuni de control etc.
- Util pentru generare de cod și optimizare

Coduri cu trei adrese

- Este o descriere liniară a unui arbore sintactic sau a unui arbore orientat
- Pentru graful discutat anterior, traducerea în cod cu trei adrese este dată mai jos



$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

Adrese și instrucțiuni

- Codul cu trei adrese este construit pe baza a două concepte: adrese și instrucțiuni
- O adresă poate fi:
  - Nume (va fi înlocuit cu o adresă de memorie la generare)
  - Constante (sunt necesare conversii de tip)
  - Variabile temporare generate de compilator (folosite în general pentru optimizări)

Instrucțiuni pentru cod cu trei adrese

- Atribuiri de forma $c = a\ op\ b$
- Atribuiri de forma a = *op b*
- Copiere: $a = b$
- Salt: *goto L*; instrucțiunea cu adresa L va fi următoarea executată
- Salturi condiționale: *if a goto L* sau *if false a goto L*
- Salturi condiționale: *if a relop b goto L*; dacă a este în relația *relop* cu b, execută instructiunea L, dacă nu execută următoarea adresă după instrucțiunea curentă

Instrucțiuni pentru cod cu trei adrese

- Apeluri de funcții:
  - param x1
  - param x2
  - …
  - param xn
  - call p, n
- Copiere indexată: *a = b[i]* sau *a[i] = b*
  - *a = b[i]* modifică valoarea adresei care vine la i poziții dupa adresa lui b
- Asignare de adrese sau pointeri: *a=&b*, *a=*b*, **a=b*

Exemplu

- Fie instrucțiunea
  - do i = i + 1; while (a[i]<v);

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a [ t2 ]
104: if t3 < v goto 100
```

Bibliografie

- O parte dintre exemple au fost preluate din A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Second Edition. Addison-Wesley, 2007