Logică și structuri discrete

Gramatici

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

8 decembrie 2014

Reamintim: limbaje, expresii regulate și automate

Automatele pot descrie comportamentul unui sistem simplu.
din fiecare stare s, intrarea σ determină starea următoare s'

Un automat *recunoaște* un limbaj (un șir face parte din limbaj?)
ex. text cu comentarii încheiate corect

Din automate, putem obține *traductoare* (fiecare intrare poate produce o ieșire)
putem *prelucra* limbaje (ex. elimina comentarii)

Expresii regulate reprezintă concis automate, deci *limbaje regulate*
putem recunoaște șiruri cu o anumită structură (ex. număr real)

În general, dorim să *recunoaștem* dacă un șir aparține unui limbaj, să *generăm* șiruri dintr-un limbaj, sau să le *transformăm*.

Limbaje care nu sunt regulate

Există limbaje foarte simple care nu sunt regulate:

$\{a^n b^n \mid n \geq 0\}$ paranteze echilibrate

$\{ww \mid w \in \{a, b\}^*\}$ cuvânt, apoi repetat

$\{ww^R \mid w \in \{a, b\}^*\}$ cuvânt, apoi inversat

Limbajele regulate au proprietatea că orice cuvânt suficient de lung conține un subșir care poate fi repetat (*pumping lemma*):

$\exists p \in \mathbb{N}$ astfel ca fiecare cuvânt cu $|w| \geq p$ are forma $w = xyz$ cu
- $|y| \geq 1$
- $|xy| \leq p$
- $\forall k \geq 0 \; . \; xy^k z \in L$

$\Rightarrow$ demonstrăm prin reducere la absurd că o expresie nu e regulată

Limbajele de programare trebuie descrise precis

Din standardul C:

(6.8.4) *selection-statement:*

if **(** *expression* **)** *statement*
if **(** *expression* **)** *statement* **else** *statement*
switch **(** *expression* **)** *statement*

(6.8.5) *iteration-statement:*

while **(** *expression* **)** *statement*
do *statement* **while** **(** *expression* **)** **;**
for **(** $expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for **(** *declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

Sintaxa limbajelor de programare e descrisă prin *gramatici*.

Gramatică

Un limbaj e descris prin mulțimea *simbolurilor* sale, și prin *regulile* prin care pot fi combinate acele simboluri: *sintaxa*.

Sintaxa unei limbi: modul în care pot fi formate *fraze* cu *cuvintele* limbii (cuvintele sunt aici simbolurile)

O *gramatică* descrie cum se obțin șirurile dintr-un limbaj prin *reguli de producție* (*reguli de rescriere*) pornind de la un *simbol de start*

Exemplu: propoziții în limba engleză (mult simplificat)

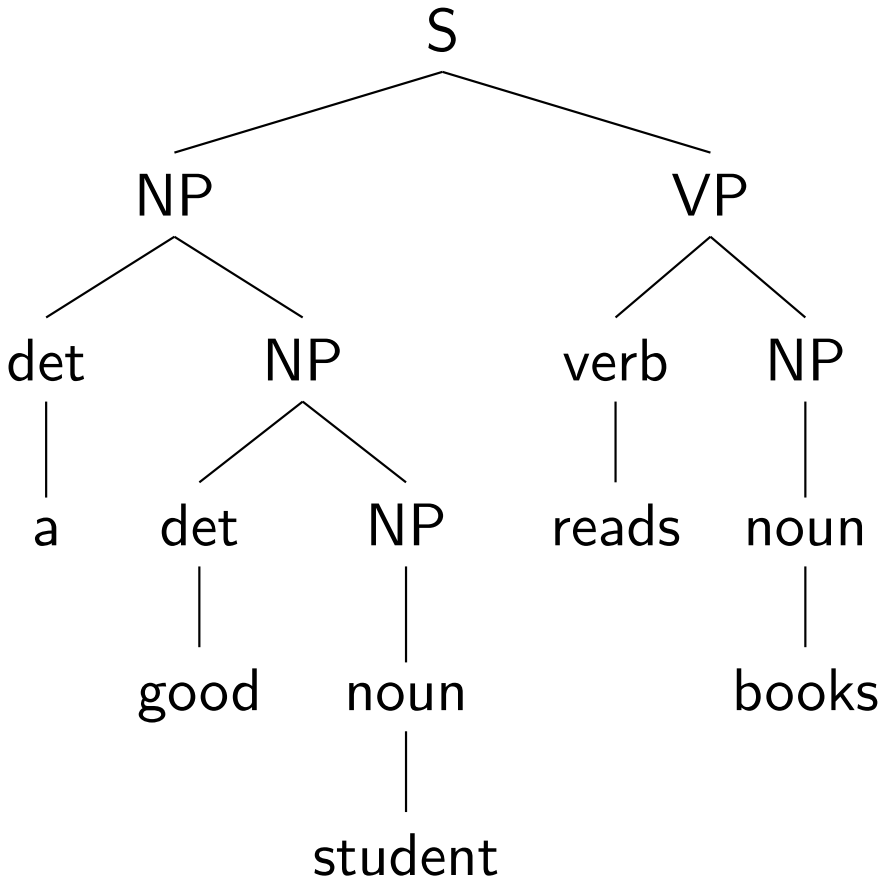
$S \rightarrow NP\ VP$	parte substantivală + predicat
$NP \rightarrow subst$	substantiv
$NP \rightarrow det\ NP$	cu parte determinantă (art/adj)
$VP \rightarrow verb$	predicat: doar verb
$VP \rightarrow verb\ NP$	predicat: cu complement direct

Remarcăm:
simboluri care apar în stânga (sunt înlocuite): *neterminale*
simboluri care apar numai în dreapta: *terminale*

Arborele de derivare

O *derivare* a unui șir dintr-o gramatică e aplicarea unui *șir de reguli* care transformă simbolul de start al gramaticii în șirul dat.
(indicând la fiecare pas și simbolul transformat)

Arborele de derivare e o reprezentare ierarhică a unei derivări, scriind partea dreaptă a fiecărei reguli sub partea stângă:



Exemple de limbaje

șirurile de paranteze echilibrate

$S \to \epsilon$

$S \to (S)S$

$\{ww^R \mid w \in \{a, b\}^*\}$ cuvânt, apoi inversat

$S \to \epsilon$

$S \to aSa$

$S \to bSb$

Ierarhia Chomsky [după Noam Chomsky]

Notăm:
litere mari: neterminale; mici: terminale; grecești: șiruri arbitrare

0) gramatici nerestricționate (orice reguli de rescriere)
   limbaje *recursiv enumerabile* (recunoscute de o mașină Turing)

1) gramatici *dependente de context*
   reguli: $\alpha A \beta \to \alpha \gamma \beta$
      A rescris doar când e între α și β
      $\gamma \neq \epsilon$ (nevid), sau $S \to \epsilon$ doar dacă S nu apare în dreapta

2) gramatici *independente de context*
   reguli: $A \to \gamma$ stânga: neterminal; dreapta: orice

3) gramatici *regulate*: generează *limbajele regulate*:
   reguli de forma $A \to a$ și $A \to aB$ (regulate la dreapta)
   alternativ: $A \to a$ și $A \to Ba$ (regulate la stânga)
      dar nu amândouă combinat!

Gramatică formală

O gramatică formală G e formată din:

Σ: o mulțime de simboluri *terminale*

N: o mulțime de simboluri *neterminale*, $N \cap \Sigma = \emptyset$

P: o mulțime de *reguli de producție*, de forma

$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)$

un neterminal N, eventual într-un context (stg/dr) e rescris cu un șir de terminale și neterminale

S: un *simbol de start*

Limbajul definit de G e format din toate șirurile de *terminale* care se pot obține din S aplicând oricâte reguli

Forma Backus-Naur (BNF)

dupa John Backus (dezvoltatorul limbajului FORTRAN)
și Peter Naur (ALGOL 60) (fiecare: premiul *Turing*)

Notație frecvent folosită pentru gramatici independente de context

folosește `::=` pentru definiție și `|` pentru alternativă

Neterminal `::=` rescriere1 | rescriere2 | ... | rescriereN

uneori folosite cu extensii:

- [*element-opțional*]
- *simbol** (steaua Kleene) pentru repetiție
- *simbol*\+ (plus) pentru repetiție cel puțin odată
- paranteze pentru gruparea elementelor

Exemple: instrucțiuni în C (simplificat)

Stmt ::= ExpStmt | IfStmt | WhileStmt | Block

ExpStmt ::= expr `;`

IfStmt ::= `if (` expr `)` Stmt `else` Stmt
| `if (` expr `)` Stmt

WhileStmt ::= `while (` expr `)` Stmt

Block ::= `{` Stmt* `}`

Problema: cu care `if` se potrivește `else` ?

```
if (x > 0) if (y > 0) x = 0; else y = 0;
```

Gramatica e *ambiguă*: există șiruri cu mai mulți *arbori de derivare* (arbori sintactici)

Dezambiguarea gramaticii

Pentru a dezambigua gramatica, trebuie rescrisă
în acest caz, "echilibrând" ramurile unui `if`

Stmt ::= BalancedStmt | UnBalancedIf

BalancedStmt ::= ExpStmt | WhileStmt | Block | BalancedIf

ExpStmt ::= expr `;`

WhileStmt ::= `while` `(` expr `)` Stmt

Block ::= `{` Stmt* `}`

BalancedIf ::= `if` `(` expr `)` BalancedStmt `else` Stmt

UnBalancedIf ::= `if` `(` expr `)` Stmt

Expresii aritmetice

E ::= num | E + E | E - E | E * E | E / E | (E)

și aici avem ambiguitate: nu e precizată precedența operatorilor
rescriem pe 3 nivele de precedență:

E ::= T | E + T | E - T

T ::= F | T * F | T / F

F ::= int | (E)

Eliminăm și recursivitatea la stânga
(E pe prima poziție în producțiile lui E, la fel T)

E ::= T RestE

RestE ::= ϵ | + T RestE | - T RestE

T ::= F RestT

RestT ::= ϵ | * F RestT | / F RestT

F ::= int | (E)