Languages and Machines (Module 7 TCS+IAM)

L&M 4: Parallel Composition of Automata & Grammars

Extra Material Blackboard + Ch 3:1-5

Alexander Skopalik

Discrete Mathematics and Mathematical Programming, Applied Mathematics, University of Twente

Lecture 4

Contents

Contents

1. Composition of Automata in Software Engineering
   - Product of Automata
   - Parallel composition
   - Projection and Hiding
2. Grammars and Derivations
   - Derivations
   - Derivation trees
   - Generated Language
3. Regular grammars
4. Syntax of programming languages
   - Ambiguity
   - Naive Parsing

Real-Life Applications

Applications of regular languages:

1. Searching in text (pattern matching, cf. C 2.4)
2. Lexical analysis of software programs (keywords, identifiers)
3. Input validation for software security
4. System specification in software engineering

Typical approach:

1. Identify a number of patterns
2. Specify the language by a regular expression
3. Transform the expression to an NFA-λ
4. Transform the NFA-λ into a DFA, using the subset construction
5. Minimise the DFA
6. Translate the minimal DFA directly into code (C, Java, Python)

A regular expression for E-mail addresses

according to standard RFC 5322, cf. http://emailregex.com

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-
\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-
9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|
[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Note: this regexp uses extended notation, which you find in many practical examples, like ? for optional, [a-z] for ranges of symbols.

Application in Software Engineering

Automata are used in Software Engineering:

- Specification of properties and models of systems
- Verification of properties for models

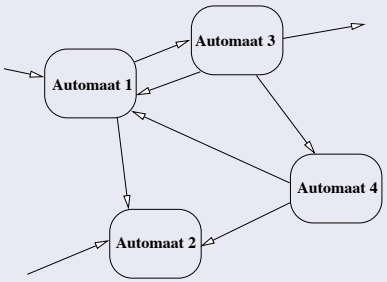
example: $\mathcal{L}(\textit{efficient elevator}) \subseteq \mathcal{L}(\textit{safe elevator})$

Prerequisites for methods in Software Engineering:

- Divide-and-conquer: not monolithic, but compositional
  - product and parallel composition of languages and automata
- Information-hiding: irrelevant details/internal behavior must be able to be suppressed (hidden)
  - Projection and hiding of languages and automata

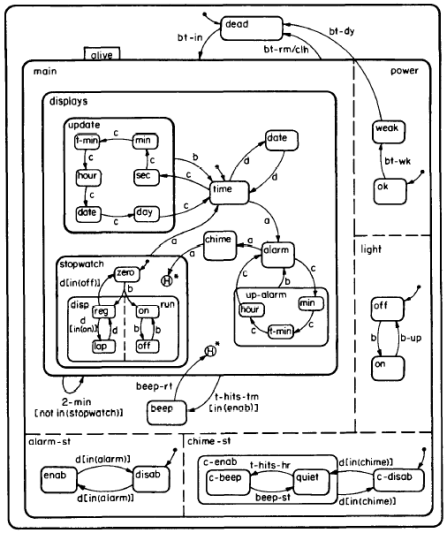
Composition of Automata

- A system often consists of a number of components
- Components mutually exchange messages
- Now we want to compose automata (DFA/NFA)



- Communication = simultaneously reading the same symbol

Remember: Smart Watch by David Harel



Using parallel components, but also hierarchical states

Product of automata

$M_1 \times M_2$: Productautomata of automata M_1 and M_2

- We want $M_1 \times M_2$ to accept a word, if it is simultaneously accepted by both M_1 and M_2.
- So we require: $\mathcal{L}(M_1 \times M_2) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$

States and transitions

- The states of the automata $M_1 \times M_2$ will consist of pairs of states of M_1 and M_2.
- The transitions of $M_1 \times M_2$ will consist of the simultaneous transitions of M_1 and M_2.

Cartesian product:

- $P \times Q = \{(p, q) \mid p \in P \wedge q \in Q\}$
- E.g.: $\{a, b, c\} \times \{1, 2\} = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$

Productautomata for DFA's (mathematical)

- Let the following (possibly incomplete) DFA's be given:

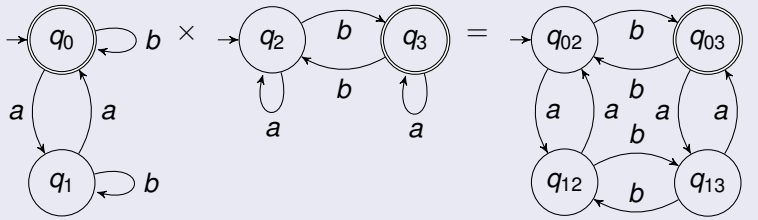
$$M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1) \text{ and } M_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2)$$

- We now define the product automata (another DFA):

$$M_1 \times M_2 := (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (q_1^0, q_2^0), F_1 \times F_2)$$

where $\delta((q_1, q_2), a) := (\delta_1(q_1, a), \delta_2(q_2, a))$

- If q_1 or q_2 can't do a a, then (q_1, q_2) can't either
- Example: even number of a's, odd number of b's:



Product automata for NFA-λ

For NFA-λ it's almost the same:

- δ is now a relation, not a function
- We require no synchronisation of λ-steps
- These λ-steps thus occur independently (=interleaving).

Product of two NFA-λ's (formally)

- Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2)$.
- Define $M_1 \times M_2 := (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (q_1^0, q_2^0), F_1 \times F_2)$
- where for $a \neq \lambda$:
  - $(q_1', q_2') \in \delta((q_1, q_2), a)$ if $q_1' \in \delta_1(q_1, a)$ and $q_2' \in \delta_2(q_2, a)$
- but for $a = \lambda$:
  - $(q_1', q_2') \in \delta((q_1, q_2), \lambda)$ if $\begin{cases} q_1' \in \delta_1(q_1, \lambda) \text{ and } q_2' = q_2; \text{ or} \\ q_2' \in \delta_2(q_2, \lambda) \text{ and } q_1' = q_1. \end{cases}$

Better Notation

Original notation:

- $(q_1', q_2') \in \delta((q_1, q_2), a)$ if $q_1' \in \delta_1(q_1, a)$ and $q_2' \in \delta_2(q_2, a)$
- $(q_1', q_2') \in \delta((q_1, q_2), \lambda)$ if $\begin{cases} q_1' \in \delta_1(q_1, a) \text{ and } q_2' = q_2; \text{ or} \\ q_2' \in \delta_2(q_2, a) \text{ and } q_1' = q_1. \end{cases}$

Better notation: $p \xrightarrow{a}_1 q$ for $q \in \delta_1(p, a)$

$$\frac{q_1 \xrightarrow{a}_1 q_1' \quad q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2')} \qquad \frac{q_1 \xrightarrow{\lambda}_1 q_1'}{(q_1, q_2) \xrightarrow{\lambda} (q_1', q_2)} \qquad \frac{q_2 \xrightarrow{\lambda}_2 q_2'}{(q_1, q_2) \xrightarrow{\lambda} (q_1, q_2')}$$

Parallel composition of automata

- Parallel composition of automata is similar to product
- The difference is mostly in the alphabet of actions

With the product of automata, the following applied:

- $\Sigma_{M_1 \times M_2} = \Sigma_{M_1} \cap \Sigma_{M_2}$.
- So M_1 must provide all symbols of M_2.
- So this is not a good separation of concerns.

With parallel composition we separate four types of actions:

- On $\Sigma_1 \cap \Sigma_2$ we still require synchronization, as with product
- On $\Sigma_1 \setminus \Sigma_2$, M_1 decides what happens (and M_2 does nothing)
- On $\Sigma_2 \setminus \Sigma_1$, M_2 decides what happens (and M_1 does nothing)
- λ is never synchronized on, instead it is interleaved

Parallel Composition of NFA-λ (mathematical)

- Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2)$
- Then $M_1 || M_2 := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_1^0, q_2^0), F_1 \times F_2)$,
- where $\delta((q_1, q_2), a)$ is defined by:

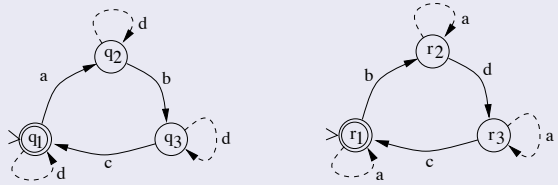
$$\frac{q_1 \xrightarrow{a}_1 q_1' \quad q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2')} \quad a \in \Sigma_1 \cap \Sigma_2$$

$$\frac{q_1 \xrightarrow{a}_1 q_1'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2)} \quad a \in \Sigma_1 \setminus \Sigma_2 \text{ or } a = \lambda$$

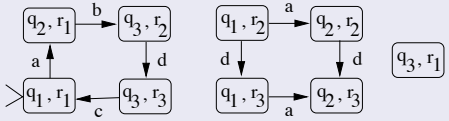
$$\frac{q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1, q_2')} \quad a \in \Sigma_2 \setminus \Sigma_1 \text{ or } a = \lambda$$

Example parallel composition

- We now take the parallel composition of the automata:



- This results in:



Note:

- only the 1st four states are reachable
- parallel composition and product are identical with “self-loops”

Projection and hiding

Information Hiding

- In software engineering information hiding is essential
- In our case we want to hide some actions
- Hiding can be done by simply replacing actions by λ

In the previous example:

- we might be only interested in the actions a and d;
- the actions b and c are meant only as internal synchronization.

We can write this in two ways: $(\Sigma = \{a, b, c, d\})$

- $M \restriction \{a, d\}$: project M on the actions a and d
- $M \setminus \{b, c\}$: hide the actions b and c in M (hiding)
- The effect of both operations is the same:
  b and c are replaced by a λ (that can be removed)

Projection for Languages

Projection for symbols $a \in \Sigma$:

- $a \upharpoonright \Sigma_1 = \begin{cases} a, \text{als } a \in \Sigma_1 \\ \lambda, \text{als } a \notin \Sigma_1 \end{cases}$

Projection for words $w \in \Sigma^*$:

- $\lambda \upharpoonright \Sigma_1 = \lambda$
- $(aw) \upharpoonright \Sigma_1 = (a \upharpoonright \Sigma_1)(w \upharpoonright \Sigma_1)$
- Example: $abbccdaacadbba \upharpoonright \{a, c\} = accaacaa$

Projection for languages $L \subseteq \Sigma^*$:

- $L \upharpoonright \Sigma_1 = \{w \upharpoonright \Sigma_1 \mid w \in L\}$

Projection for automata M (NFA-λ):

- $M \upharpoonright \Sigma_1$: Replace every occurrence of $a \notin \Sigma_1$ by λ

Language of parallel composition and projection

- Let NFA-λ M_1 and M_2 be given, with alphabet Σ_1 and Σ_2
- We already saw for the product automata: $\mathcal{L}(M_1 \times M_2) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$
- So:
  - $w \in \mathcal{L}(M_1 \times M_2)$ if and only if
  - $w \in \mathcal{L}(M_1)$ and $w \in \mathcal{L}(M_2)$
- What is the language of parallel composition? $\mathcal{L}(M_1 || M_2)$
- The following applies:
  - $w \in \mathcal{L}(M_1 || M_2)$ if and only if
  - $w \upharpoonright \Sigma_1 \in \mathcal{L}(M_1)$ and $w \upharpoonright \Sigma_2 \in \mathcal{L}(M_2)$

System Verification by Model Checking

Modeling and Specification

- Let *Ext* be the alphabet of external symbols; *Int* is the alphabet of internal communications
- Let *Spec* be a RE over alphabet *Ext*
- Let $Comp_i$ be components over alphabet $Ext \cup Int$.

Correctness: The external behaviour of the composed system conforms to the specification,

$$\mathcal{L}((Comp_1 \,||\, \cdots \,||\, Comp_n) \backslash Int) \subseteq \mathcal{L}(Spec)$$

Practical challenge: the intermediate state space grows exponentially with the number of components.
After hiding and minimisation, the result is typically small, but it is not (always) possible to compute the small automaton directly.

Contents

Example: calculator expressions

Alphabet: $\{1, 2, 3, +, \times\}$. Variables: $\{S, T\}$. Expression: $1 + 2 \times 3$.
Let us specify the language of all correct expressions:

Grammar G_1

$$S \rightarrow 1 \mid 2 \mid 3 \mid S + S \mid S \times S$$

Grammar G_2

$$\begin{aligned} S &\rightarrow T \times S \mid T \\ T &\rightarrow 1 \mid 2 \mid 3 \mid T + T \end{aligned}$$

Grammar G_3

$$\begin{aligned} S &\rightarrow T + S \mid T \\ T &\rightarrow 1 \mid 2 \mid 3 \mid T \times T \end{aligned}$$

Only G_3 reflects the "correct" arithmetic interpretation
Note: All grammars are still ambiguous (check!)

Basic Concepts of Context-Free Grammars

Definition

Context-Free Grammar (CFG): 4-tuple $\langle V, \Sigma, P, S \rangle$ with

- V a set of *variables* (help symbols, *non-terminals*)
- Σ (again) the *alphabet*, disjunct with V (so $V \cap \Sigma = \emptyset$)
- $P \subseteq V \times (V \cup \Sigma)^*$ a set of *rules* (production)
- $S \in V$ the *start symbol*

Additional remarks

- A rule is a pair (A, w) with $A \in V$ and $w \in (V \cup \Sigma)^*$
  - Notation: $A \rightarrow w_1 \mid w_2 \mid \cdots$ if $(A, w_1), (A, w_2), \ldots \in P$.
  - Special case: *null-rule* $A \rightarrow \lambda$
- Convention:
  - Variables with upper case letters ($A, B, \ldots$)
  - Alphabet with lower case letters ($a, b, \ldots$)

Examples of CFG's

- $V = \{S, A\}$ and $\Sigma = \{a, b\}$:

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow AAA \mid bA \mid Ab \mid a \end{aligned}$$

- $V = \{\mathit{Sentence}, \mathit{NounPhrase}, \ldots\}$ and $\Sigma = \{a, \ldots, z\}^*$:

$$\begin{aligned} \mathit{Sentence} &\rightarrow \mathit{NounPhrase}\ \mathit{VerbPhrase} \mid \\ &\quad\ \mathit{NounPhrase}\ \mathit{Verb}\ \mathit{D.O.Phrase} \\ \mathit{NounPhrase} &\rightarrow \mathit{ProperNoun} \mid \\ &\quad\ \mathit{Determiner}\ \mathit{CommonNoun} \\ \mathit{Determiner} &\rightarrow a \mid \mathit{the} \\ \mathit{CommonNoun} &\rightarrow \mathit{car} \mid \mathit{hamburger} \mid \cdots \\ &\cdots \end{aligned}$$

Rule application

- A rule of the form

$$A \rightarrow w$$

can be *applied* to words of the form

$$uAv$$

and results in a *derivation* of the form

$$uAv \Rightarrow uwv$$

- Examples: rules (i) $A \rightarrow Aa$ and (ii) $A \rightarrow b$:

$$bA \overset{\text{(i)}}{\Longrightarrow} bAa \overset{\text{(i)}}{\Longrightarrow} bAaa \overset{\text{(ii)}}{\Longrightarrow} bbaa$$
$$bA \overset{\text{(i)}}{\Longrightarrow} bAa \overset{\text{(ii)}}{\Longrightarrow} bba$$
$$bA \overset{\text{(ii)}}{\Longrightarrow} bb$$

Derivability

Definition

Take a CFG $\langle V, \Sigma, P, S\rangle$ and a string $v \in (V \cup \Sigma)^*$.
The set of *derivable strings* $Der(v)$ is the *smallest* set such that

- $v \in Der(v)$;
- If $x\,A\,y \in Der(v)$, and $(A \to w) \in P$, then $x\,w\,y \in Der(v)$.

So: $w \in Der(v)$ if a (possibly empty) series of derivations exists:

$$v = w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n = w$$

Notation: $v \overset{*}{\Longrightarrow} w$ for "w is derivable from v"
Example: $b\,A \overset{*}{\Longrightarrow} b\,b\,a\,a$ and $b\,A \overset{*}{\Longrightarrow} b\,b$.
So $b\,b \in Der(b\,A)$.

Examples of derivations

Rules: $S \to A\,A$ and $A \to A\,A\,A \mid b\,A \mid A\,b \mid a$

(i)	(ii)	(iii)	(iv)
$S \Rightarrow A\,A$	$S \Rightarrow A\,A$	$S \Rightarrow A\,A$	$S \Rightarrow A\,A$
$\Rightarrow a\,A$	$\Rightarrow A\,A\,A\,A$	$\Rightarrow A\,a$	$\Rightarrow a\,A$
$\Rightarrow a\,A\,A\,A$	$\Rightarrow a\,A\,A\,A$	$\Rightarrow A\,A\,A\,a$	$\Rightarrow a\,A\,A\,A$
$\Rightarrow a\,b\,A\,A\,A$	$\Rightarrow a\,b\,A\,A\,A$	$\Rightarrow A\,A\,b\,A\,a$	$\Rightarrow a\,A\,A\,a$
$\Rightarrow a\,b\,a\,A\,A$	$\Rightarrow a\,b\,a\,A\,A$	$\Rightarrow A\,A\,b\,a\,a$	$\Rightarrow a\,b\,A\,A\,a$
$\Rightarrow a\,b\,a\,b\,A\,A$	$\Rightarrow a\,b\,a\,b\,A\,A$	$\Rightarrow A\,b\,A\,b\,a\,a$	$\Rightarrow a\,b\,A\,b\,A\,a$
$\Rightarrow a\,b\,a\,b\,a\,A$	$\Rightarrow a\,b\,a\,b\,a\,A$	$\Rightarrow A\,b\,a\,b\,a\,a$	$\Rightarrow a\,b\,a\,b\,A\,a$
$\Rightarrow a\,b\,a\,b\,a\,a$	$\Rightarrow a\,b\,a\,b\,a\,a$	$\Rightarrow a\,b\,a\,b\,a\,a$	$\Rightarrow a\,b\,a\,b\,a\,a$

- Leftmost derivations: (i) and (ii) (left variable first)
- Rightmost derivations: (iii) (right variable first)

Derivation trees

Definition derivation tree

Take a CFG $G = \langle V, \Sigma, P, S\rangle$ and a derivation $S \overset{*}{\Longrightarrow} w$.
The (ordered) derivation tree of $S \overset{*}{\Longrightarrow} w$ is constructed as follows:

- Select S as root
- At the application of a rule $A \to x_1\,x_2 \cdots x_n$ (all $x_i \in V \cup \Sigma$): add $x_1, x_2, \ldots, x_n$ as children of A
- At the application of a rule $A \to \lambda$: add λ as (only) child of A

Remark

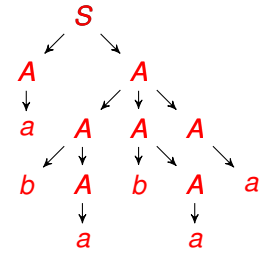
At any time during the construction, the derived word is equal to the symbols on the leaves, in the order of the tree

Examples of derivation trees (1)

Leftmost derivation

$S \Rightarrow A\,A$
$\Rightarrow a\,A$
$\Rightarrow a\,A\,A\,A$
$\Rightarrow a\,b\,A\,A\,A$
$\Rightarrow a\,b\,a\,A\,A$
$\Rightarrow a\,b\,a\,b\,A\,A$
$\Rightarrow a\,b\,a\,b\,a\,A$
$\Rightarrow a\,b\,a\,b\,a\,a$

Derivation tree



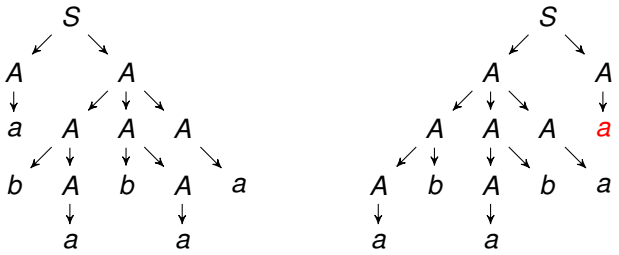
Rightmost derivation

$S \Rightarrow A\,A$
$\Rightarrow A\,A\,A\,A$
$\Rightarrow A\,A\,A\,a$
$\Rightarrow A\,A\,b\,A\,a$
$\Rightarrow A\,A\,b\,a\,a$
$\Rightarrow A\,b\,A\,b\,a\,a$
$\Rightarrow A\,b\,a\,b\,a\,a$
$\Rightarrow a\,b\,a\,b\,a\,a$

- The derivation tree makes no distinction based on the *order*
- One-to-one relation between leftmost derivations and derivation trees

Examples of derivation trees (2)

- Derivations of a single sentence can result in different trees
- Rules: $S \to A\,A$ and $A \to A\,A\,A \mid b\,A \mid A\,b \mid a$
- Multiple derivation trees exist for $S \overset{*}{\Longrightarrow} a\,b\,a\,b\,a\,a$
  (so also different leftmost derivations)



- Such a grammar is called *ambiguous*

Sentential forms, sentences and language

Definition: language of a context-free grammar

Given a CFG $G = \langle V, \Sigma, P, S \rangle$:

- If $S \overset{*}{\Longrightarrow} w$, then w is called a *sentential form* (in general: $w \in (V \cup \Sigma)^*$)
- If $w \in \Sigma^*$, then w is called a *sentence*.
- The *language* of G is defined as $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \overset{*}{\Longrightarrow} w\}$.

Definition: Context-free language

A set of words $X \subseteq \Sigma^*$ is a *context-free language* if a CFG G exists, such that $X = \mathcal{L}(G)$.

We sometimes write **CFG** for the class of context-free languages.

Examples of context-free languages

1. Number of a's followed by the same number of b's?
   $(\{a^n b^n \mid n \geq 0\})$
   $$S \to a\,S\,b \mid \lambda$$
2. Number of a's, then b's, then the same number of a's?
   $(\{a^n b^m a^n \mid n, m > 0\})$
   $$S \to a\,S\,a \mid a\,A\,a \qquad A \to b\,A \mid b$$
3. Correctly nested brackets? (a is left bracket, b right bracket)
   $$S \to S\,S \mid a\,S\,b \mid \lambda$$
4. Palindromes over a, b?
   $$S \to a\,S\,a \mid b\,S\,b \mid a \mid b \mid \lambda$$

These languages are all *non*-regular!

Considerations about context-free languages

- Which languages are generated by the following grammars?
  G_1 : $S \to A\,B$ $A \to a\,A \mid a$ $B \to b\,B \mid \lambda$
  G_2 : $S \to a\,S \mid a\,B$ $B \to b\,B \mid \lambda$
  Answer: both generate $a^+ b^*$ (a regular language!).
- Are all regular languages context-free? How can you prove something like that?
- Can you give an unambiguous grammar for every context-free language?
- G_1 and G_2 are called *equivalent* if $L(G_1) = L(G_2)$.
  Can we always check equivalence of CFGs? How?

Contents

Regular grammars

Definition: Regular grammar

A grammar is called *regular* if every rule has one of the following forms (with $A, B \in V$ and $a \in \Sigma$):

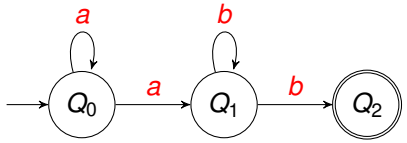
1. $A \to a$
2. $A \to a\,B$
3. $A \to \lambda$

Remarks

- If G_1 and G_2 are equivalent, and G_2 is regular, then G_1 does not have to be regular
- Example: G_1 : $S \to A\,B$ $\quad A \to a\,A \mid a$ $\quad B \to b\,B \mid \lambda$
  G_2 : $S \to a\,S \mid a\,B$ $\quad B \to b\,B \mid \lambda$
  (see previous sheet; both have language a^+b^*)

Finite automaton to grammar: Example

- Automaton M:



- Grammar G: $Q_0 \to a\,Q_0 \mid a\,Q_1$
  $Q_1 \to b\,Q_1 \mid b\,Q_2$
  $Q_2 \to \lambda$
- Common word $aabb \in L(M) \cap L(G)$:

Computation	Derivation	Partial word
$[Q_0, aabb] \vdash [Q_0, abb]$	$Q_0 \Rightarrow a\,Q_0$	a
$\vdash [Q_1, bb]$	$\Rightarrow a\,a\,Q_1$	aa
$\vdash [Q_1, b]$	$\Rightarrow a\,a\,b\,Q_1$	aab
$\vdash [Q_2, \lambda]$	$\Rightarrow a\,a\,b\,b$	$aabb$

From finite automata to regular grammars

Theorem (Note: Material from §6.3!)

Given an NFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$
Define $G = \langle Q, \Sigma, P, q_0 \rangle$ with

- $(q, a\,q') \in P$ for all $q \in Q$ and $q' \in \delta(q, a)$
- $(q, \lambda) \in P$ for all $q \in F$

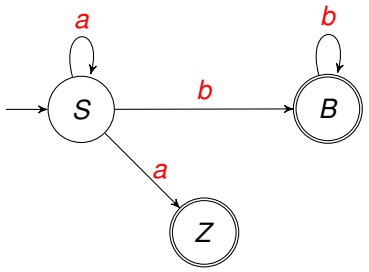
Then $L(G) = L(M)$.

Intuition

- Every state in the automaton becomes a variable
- Every transition $q \xrightarrow{a} q'$ becomes a rule $q \to a\,q'$
- Every accepting state q becomes a rule $q \to \lambda$
- Paths in the automaton correspond to derivations
- Rules of the form $A \to a$ are not used

Regular grammar to automaton: Example

- Grammar: $S \rightarrow a\,S \mid b\,B \mid a$
  $B \rightarrow b\,B \mid \lambda$
- Automaton: $Q = \{S, B, Z\}$
  $q_0 = S$
  $F = \{B, Z\}$



From regular grammars to finite automata

Theorem

Given a regular CFG $G = \langle V, \Sigma, P, S \rangle$
Define $M = \langle V \cup \{Z\}, \Sigma, \delta, S, F \rangle$ with

- Z a special new variable, not in V
- $B \in \delta(A, a)$ for all $A \rightarrow a\,B$, and $Z \in \delta(A, a)$ for all $A \rightarrow a$
- $A \in F$ for all $A \rightarrow \lambda$, and $Z \in F$

Then $L(M) = L(G)$

Intuition

- Every variable of the grammer becomes a state
- Special (accepting) state z for terminating rules
- Rules $A \rightarrow a\,B$ become transitions $A \xrightarrow{a} B$
- Rules $A \rightarrow a$ become transitions $A \xrightarrow{a} Z$ (with $Z \in F$)
- Rules $A \rightarrow \lambda$ result in $A \in F$

Contents

Syntax of programming languages

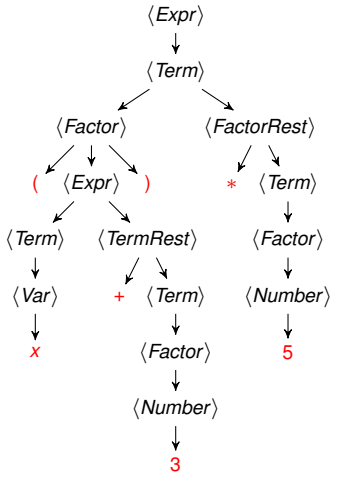
- Completely modeled by non-ambiguous context-free grammars
- Example: CFG for syntax of expressions

$$\begin{aligned}
\langle Expr \rangle &\rightarrow \langle Term \rangle\ \langle TermRest \rangle_{opt} \\
\langle TermRest \rangle &\rightarrow + \langle Expr \rangle \mid - \langle Expr \rangle \\
\langle Term \rangle &\rightarrow \langle Factor \rangle\ \langle FactorRest \rangle_{opt} \\
\langle FactorRest \rangle &\rightarrow * \langle Term \rangle \mid / \langle Term \rangle \\
\langle Factor \rangle &\rightarrow \langle Var \rangle \mid \langle Number \rangle \mid (\langle Expr \rangle)
\end{aligned}$$

- *Backus-Naur-Vorm* (BNF): notation-agreements
  - Non-terminals in angular brackets, for example $\langle Number \rangle$
  - Subscript $_{opt}$ (optional) for "λ-choice:"
    - $\langle A \rangle \rightarrow \langle B \rangle_{opt}$ short for $\langle A \rangle \rightarrow \langle B \rangle | \lambda$
    - $\langle A \rangle \rightarrow \langle B \rangle_{opt} \langle C \rangle$ short for $\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle | \langle C \rangle$
  - Many variations and extensions exist
- See book (appendix IV) for complete BNF grammar for Java programs

Syntax of programming languages: example

Parse (i.e., find a derivation for): $(x + 3) * 5$

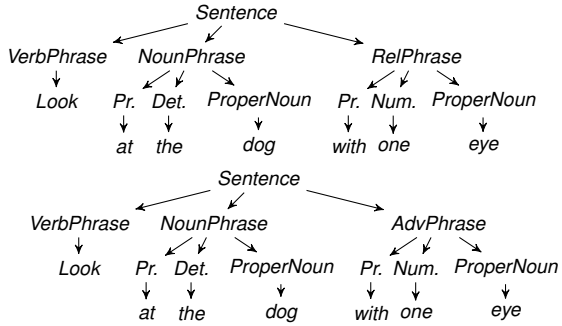


Example of ambiguity

- Sentence: *Look at the dog with one eye*
- Grammar:

$$\begin{aligned} \textit{Sentence} &\rightarrow \textit{VerbPhrase NounPhrase RelPhrase} \mid \\ &\quad\ \textit{VerbPhrase NounPhrase AdvPhrase} \\ \textit{NounPhrase} &\rightarrow \textit{Pr. Det. PropoerNoun} \\ &\ldots \end{aligned}$$

- Derivation trees:



- Different derivations result in different *interpretations*

Definition of (un)ambiguity

Definition

A CFG is *ambiguous* if a word exists in $L(G)$ with two distinct derivation trees.

Remarks

- Ambiguity results from the grammar, not from the language
  The following language has an ambiguous ánd an unambiguous grammar:
  $G_1: S \rightarrow aS \mid Sa \mid a \qquad G_2: S \rightarrow aS \mid a$
  $L(G_1) = L(G_2) = a^+$ holds; but G_1 is ambiguous, G_2 is not
- Fact: There are also languages with *only* ambiguous grammars
  Example: $L = \{a^m b^m c^n \mid m, n \geq 0\} \cup \{a^m b^n c^n \mid m, n \geq 0\}$
- Unambiguous grammars exist for regular languages
- Programming languages have unambiguous grammars

The principle of parsing

Definition

- A *parsing* of a sentence is a derivation tree for that sentence
- A *parsing algorithm* for a grammar is an algorithm that produces a parsing for each sentence

Naive parsing algorithm: Search and backtrack

1. If the sentential form consists of only S, we are done
2. Search right side of a rule that matches part of the sentence
   Success?
   - Replace the right side by the corresponding left side
   - Repeat step 1 (next phase)

   Failure?
   - Go back to the previous phase
   - Try the next possibility (backtracking)

Example of naive parsing

- Grammar: $S \to a\,S\,b \mid a\,A \qquad A \to a\,A \mid a$
- Language: $\{a^m b^n \mid n \geq 0, m > n+1\}$
- Parsing:

$$
\begin{array}{lllllll}
aaab & \Leftarrow & Aaab & \Leftarrow & AAab & \Leftarrow & AAAb \;\lightning \\
 & & & \Leftarrow & AaAb & \Leftarrow & AAAb \;\lightning \\
 & & & & & \Leftarrow & AAb \;\lightning \\
 & & & & & \Leftarrow & ASb \;\lightning \\
 & \Leftarrow & aAab & \Leftarrow & AAab & \lightning & \\
 & & & \Leftarrow & aAAb & \Leftarrow & \cdots \;\lightning \\
 & \Leftarrow & aaAb & \Leftarrow & AaAb & \lightning & \\
 & & & \Leftarrow & aAAb & \lightning & \\
 & & & \Leftarrow & aAb & \Leftarrow & AAb \;\lightning \\
 & & & & & \Leftarrow & Ab \;\lightning \\
 & & & & & \Leftarrow & Sb \;\lightning \\
 & & & \Leftarrow & aSb & \Leftarrow & S \;\checkmark
\end{array}
$$

- The complexity of this algorithm is not acceptable! Why?
  - It becomes even worse if there is also λ on the right side