Models for languages

Models suitable to recognize/accept, translate, compute languages
– They “receive” an input string and process it

• Operational models (Automata)

Models suitable to describe how to generate a language
– Sets of rules to build phrases of a language

• Generative models (Grammars)
Grammars (1)

• Generative models produce strings
  – grammar (or syntax)

• A grammar is a set of rules to build the phrases of a language
  – It applies to any notion of language

• A formal grammar generates strings of a language through a rewriting process
Rewriting

• Rewriting relevant to many fields
  – Mathematics
  – Computer science
  – Logic

• It consists of a wide range of methods for replacing subterms of a “formula” with other terms
  – Potentially nondeterministic
Examples

• Semantically equivalent formulae in propositional logic
  – $A \land B$ can be replaced with $\neg(\neg A \lor \neg B)$
  – $\neg A \lor B$ can be replaced with $A \Rightarrow B$
  – ...

• Examples of tautologies in FOL
  – We can rewrite the tautology $\neg A \lor A$ by replacing $A$ with a w.f.f. of propositional or FOL logic
Linguistic rules (1)

• Natural languages are explained through rules such as:
  – A phrase is made of a subject followed by a predicate
  – A subject can be a noun or a pronoun or ...
  – A predicate can be a verb followed by a complement

• Programming languages are expressed similarly:
  – A program consists of a declarative part and an executable part
  – The declarative part ...
  – The executable part consists of a statement sequence
  – A statement can be ...
Linguistic rules (2)

• In general, a linguistic rule describes a “main object”
  – Examples: a book, a program, a message, ...
as a sequence of “composing objects”

• Each “composing object” is “refined” by replacing it with more detailed objects and so on... until a sequence of base elements is obtained
Grammars (2)

• A grammar is a linguistic rule
• It is composed by
  – a main object: initial symbol
  – composing objects: nonterminal symbols
  – base elements: terminal symbols
  – refinement rules: productions
• Formally?
Quotes

“A grammar can be regarded as a device that enumerates the sentences of a language”

“A grammar of $L$ can be regarded as a function whose range is exactly $L$”

Definition

• A grammar is a tuple \(<V_N, V_T, P, S>\) where
  – \(V_N\) is the nonterminal alphabet (or vocabulary)
  – \(V_T\) is the terminal alphabet (or vocabulary)
  – \(V = V_N \cup V_T\)
  – \(S \in V_N\) is a particular element of \(V_N\) called axiom or initial symbol
  – \(P \subseteq V^* \cdot V_N \cdot V^* \times V^*\) is the (finite) set of rewriting rules or productions

• A grammar \(G = <V_N, V_T, P, S>\) generates a language on the alphabet \(V_T\)
Productions

• A production is an element of $V^* \cdot V_N \cdot V^* \times V^*$
  – This is usually denoted as $\langle \alpha, \beta \rangle$, where
    \[ \alpha \in V^* \cdot V_N \cdot V^* \text{ and } \beta \in V^* \]
• We generally indicate a production as $\alpha \rightarrow \beta$
  – $\alpha$ is a sequence of symbols including at least one nonterminal symbol
  – $\beta$ is a (potentially empty) sequence of (terminal or non terminal) symbols
Example

- $V_N = \{S, A, B, C, D\}$
- $V_T = \{a, b, c\}$
- $S$ is the initial symbol
  - It is not mandatory to call it $S$
- $P = \{
  S \rightarrow AB,
  BA \rightarrow cCD,
  CBS \rightarrow ab,
  A \rightarrow \varepsilon
\}$
- The generated language is on the alphabet $\{a, b, c\}$
Chomsky hierarchy (1)

- Grammars are classified according to the form of their productions
- Chomsky classified grammars in four types

![Diagram showing the Chomsky hierarchy with types 0, 1, 2, and 3 nested inside each other.]

Grammars
Chomsky hierarchy (2)

• Type 3 grammars restrict productions to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal
  – The rule $S \rightarrow \varepsilon$ is also allowed here if $S$ does not appear on the right side of any rule

• Type-2 grammars are defined by rules of the form $A \rightarrow \gamma$ where $A$ is a nonterminal and $\gamma$ is a string of terminals and nonterminals
Chomsky hierarchy (3)

• **Type-1 grammars** have rules of the form $\alpha A\beta \rightarrow \alpha \gamma \beta$, where $A$ is a nonterminal and $\alpha$, $\beta$ and $\gamma$ are strings of terminals and nonterminals.
  
  – $\gamma$ must be non-empty
  
  – The rule $S \rightarrow \epsilon$ is allowed if $S$ does not appear on the right side of any rule

• **Type-0 grammars** include all formal grammars
Immediate derivation relation

\[ \alpha \Rightarrow \beta \ (\beta \text{ is obtained by immediate derivation from } \alpha) \]

- \( \alpha \in V^* \cdot V_N \cdot V^* \) and \( \beta \in V^* \)

if and only if

\[ \alpha = \alpha_1 \alpha_2 \alpha_3, \quad \beta = \alpha_1 \beta_2 \alpha_3 \text{ and } \alpha_2 \rightarrow \beta_2 \in P \]

\( \rightarrow \) \( \alpha_2 \) is rewritten as \( \beta_2 \) in the context \( \langle \alpha_1, \alpha_3 \rangle \)
Example

In the grammar G

- $V_N = \{S, A, B, C, D\}$
- $V_T = \{a,b,c\}$
- S is the initial symbol
- $P = \{S \rightarrow AB, BA \rightarrow cCD, CBS \rightarrow ab, A \rightarrow \varepsilon\}$

- $aaBAS \Rightarrow aacCDS$
- $bcCBSAdd \Rightarrow bcabAdd$
Language generated by a grammar

• Given a grammar \( G = \langle V_N, V_T, P, S \rangle \)

\[ \forall x \ ( x \in L(G) \iff x \in V_T^* \land S \Rightarrow^+ x ) \]

• Informally the language generated by a grammar \( G \) is the set of all strings
  – Consisting only of terminal symbols that can be derived from \( S \)
  – In any number of steps
Example 1

- $G_1 = \langle \{S, A, B\}, \{a, b, 0\}, P, S \rangle$
  - $P = \{S \rightarrow aA, A \rightarrow aS, S \rightarrow bB, B \rightarrow bS, S \rightarrow 0\}$

- Some derivations
  - $S \Rightarrow 0$
  - $S \Rightarrow aA \Rightarrow aaS \Rightarrow aa0$
  - $S \Rightarrow bB \Rightarrow bbS \Rightarrow bb0$
  - $S \Rightarrow aA \Rightarrow aaS \Rightarrow aabB \Rightarrow aabbS \Rightarrow aabb0$

- An easy generalization $L(G_1) = \{aa, bb\}^* . 0$
Example 2

• $G_2=\langle\{S\}, \{a,b\}, \{S\rightarrow aSb \mid ab\}, S\rangle$
  – $\{S\rightarrow aSb \mid ab\}$ is an abbreviation for $\{S\rightarrow aSb, S\rightarrow ab\}$

• Some derivations
  – $S \Rightarrow ab$
  – $S \Rightarrow aSb \Rightarrow aabb$
  – $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

• An easy generalization $L(G_2)={a^n b^n \mid n>0}$
  – $L(G_2)={a^n b^n \mid n\geq0}$ if we substitute $S\rightarrow ab$ with $S\rightarrow \epsilon$
Example 3

• $G_3 = \langle \{S, A, B, C, D\}, \{a, b, c\}, P, S \rangle$
  – $P = \{ S \rightarrow aACD, A \rightarrow aAC | \varepsilon, B \rightarrow b, CD \rightarrow BDc, CB \rightarrow BC, D \rightarrow \varepsilon \}$

• Some derivations
  – $S \Rightarrow aACD \Rightarrow aCD \Rightarrow aBDc \Rightarrow^* abc$
  – $S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaCBDc \Rightarrow aaBCDc \Rightarrow aabCDc \Rightarrow aabBDcc \Rightarrow aabbDcc \Rightarrow aabbcc$
  – $S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaCCD \Rightarrow aaCC$
Some natural questions

• What is the practical use of grammars?
• What languages can be obtained through grammars?
• What is the relationship between automata and grammars?
  – And between languages generated by grammars and languages accepted by automata?
  – And the Chomsky hierarchy?
Some answers

• Chomsky hierarchy can be “renamed”
  – Type 3 grammars: regular
  – Type 2 grammars: context-free
  – Type 1 grammars: context-sensitive
  – Type 0 grammars: unrestricted

• Correlations
  – Regular grammars – regular languages - FSAs
  – Context-free grammars – context-free languages - NDPDAs
  – Unrestricted grammars – recursively enumerable languages - MTs
## Automata, languages, and grammars

<table>
<thead>
<tr>
<th>Chomsky hierarchy</th>
<th>Grammars</th>
<th>Languages</th>
<th>Minimal automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>Unrestricted</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
</tr>
<tr>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Context-sensitive</td>
<td>(Linear bounded automaton)</td>
</tr>
<tr>
<td>Type-2</td>
<td>Context-free</td>
<td>Context-free</td>
<td>NDPDA</td>
</tr>
<tr>
<td>Type-3</td>
<td>Regular</td>
<td>Regular</td>
<td>FSA</td>
</tr>
</tbody>
</table>
Definition

• If for each $\alpha \rightarrow \beta \in P$ we have $|\alpha| = 1$ and
  \[ \beta \in V_N. V_T \cup V_T \cup \{\epsilon\}, \] the grammar is left regular
• If for each $\alpha \rightarrow \beta \in P$ we have $|\alpha| = 1$ and
  \[ \beta \in V_T. V_N \cup V_T \cup \{\epsilon\}, \] the grammar is right regular
• A grammar is regular (RG) iff it is either left regular or right regular
• A language is regular iff it is generated by some regular grammar
  – There must be at least one grammar that generates it
RGs and FSAs

Let $A$ be a FSA. An equivalent RG $G$ can be found constructively. Equivalent means that $G$ generates exactly the same language that is recognized by $A$ (and vice versa).

Regular grammars, finite state automata and regular expressions are different models to describe the same class of languages.
Building a RG from a FSA

- If A=<Q, I, δ, q₀, F>, then it is possible to build G=<VN, VT, S, P> such that
  - VN = Q,
  - VT = I,
  - S = q₀
  - For all δ(q, i) = q’
    - q → i q’ ∈ P
    - If q’ ∈ F then q’ → ε ∈ P

- δ*(q, x) = q’ if and only if q ⇒* xq’
Building a FSA from a RG

If $G=\langle V_N, V_T, S, P \rangle$ then it is possible to build $A=\langle Q, I, \delta, q_0, F \rangle$ such that

- $Q = V_N \cup \{q_F\}$
- $I = V_T$
- $q_0 = S$
- $F = \{q_F\}$
- For all $A \rightarrow bC$, $\delta(A,b) = C$
- For all $A \rightarrow b$, $\delta(A,b) = q_F$
Example (1)

- Build an automaton that recognizes the language $L$ on the alphabet \{a, b, 0, 1\}, such that a string $x$ is in $L$ if $x$ has the following properties:
  
  - If $x$ starts with ‘a’ then $x$ has an even number of ‘1’s and an odd number of ‘0’s
  - If $x$ starts with ‘b’ then $x$ has an even number of ‘0’s and an odd number of ‘1’s
Example (2)

• Equivalent grammar

- \( S \to aS_1 \mid bS_5 \mid 0S_9 \mid 1S_9 \mid \varepsilon \)
- \( S_1 \to aS_1 \mid bS_1 \mid 1S_2 \mid 0S_4 \)
- \( S_2 \to aS_2 \mid bS_2 \mid 1S_1 \mid 0S_3 \)
- \( S_3 \to aS_3 \mid bS_3 \mid 0S_2 \mid 1S_4 \)
- \( S_4 \to aS_4 \mid bS_4 \mid 0S_1 \mid 1S_3 \mid \varepsilon \)
- \( S_5 \to aS_5 \mid bS_5 \mid 0S_6 \mid 1S_8 \)
- \( S_6 \to aS_6 \mid bS_6 \mid 0S_5 \mid 1S_7 \)
- \( S_7 \to aS_7 \mid bS_7 \mid 0S_8 \mid 1S_6 \)
- \( S_8 \to aS_8 \mid bS_8 \mid 0S_7 \mid 1S_5 \mid \varepsilon \)
- \( S_9 \to aS_9 \mid bS_9 \mid 0S_9 \mid 1S_9 \mid \varepsilon \)
Definition

• A grammar is called **context-free (CFG)** if
  – for each $\alpha \rightarrow \beta \in P$, we have $|\alpha| = 1$, i.e., $\alpha$ is an element of $V_N$.

• They are called context-free because the rewriting of $\alpha$ does not depend on its context
  – context = part of the string surrounding it
Context-free grammars

• CFGs are the same as the BNFs (BNF = Backus-Naur Form) used for defining the syntax of programming languages
  – they are well fit to define typical features of programming and natural languages, ... but not all
• Regular grammars are also context-free grammars
  – But not vice versa
Example of BNF

<if_statement> ::= 
  if <boolean_expression> then
  <statement_sequence>
  [ else <statement_sequence> ]
end if ;

<statement_sequence> ::= <statement> [ ;
  <statement_sequence> ]

• Terminals in red bold
• Nonterminals surrounded by angular brackets
• Optional items enclosed in square brackets
CFGs are equivalent to NDPDAs

intuitive justification (no proof: the proof is the “core” of compiler construction)
$S \Rightarrow aSb \Rightarrow aabb$
Definition

• General (also called unrestricted) grammars are grammars without any limitation on productions
  – They correspond to type 0 in the Chomsky hierarchy
• Both context-free grammars and regular grammars are non-restricted
General grammars and TMs

• General grammars (GGs) and TMs are equivalent formalisms
  – Given a GG it is possible to build a TM that recognizes the language generated by the grammar
  – Given a TM it is possible to define a GG that generates the language accepted by the TM

• How?
From a GG to a TM (1)

Given a general grammar \( G = \langle V_N, V_T, P, S \rangle \), let us construct a NDTM \( M \) such that \( L(M) = L(G) \):

- \( M \) has one memory tape
- The input string \( x \) is on the input tape
- The memory tape is initialized with \( S \) (better: \( Z_0S \))
- The memory tape in general will contain a string \( \alpha (\in V^*) \)
  - It is scanned searching the left part of some production of \( P \)
  - When one is found, (not necessarily the first one) \( M \) operates a nondeterministic choice and the chosen part is replaced by the corresponding right part (if there are many right parts, again, \( M \) operates nondeterministically)
From a GG to a TM (2)

• In this way, whenever $\alpha \Rightarrow \beta$ we have
  
  \[ c_s = \langle q_s, Z_0 \alpha \rangle \vdash^* \langle q_s, Z_0 \beta \rangle \]
  
  for some state $q_s$

• If and when the tape contains a string $y \in V_T^*$, it is compared with $x$

  • If they coincide, $x$ is accepted

  • otherwise this particular sequence of moves does not lead to acceptance
Remarks

• Using a NDTM facilitates the construction but it is not necessary

• Note that, if $x \notin L(G)$, M might even try infinitely many computations, none of which leads to acceptance
  – some of these might never terminate, thus (correctly) being unable to conclude that $x \in L(G)$
  – *and* being unable to conclude $x \notin L(G)$

Indeed the definition of acceptance requires that M reaches an accepting configuration if and only if $x \in L$
  – It does not require that M terminates its computation in a non-final state if $x \notin L$
  – Again, we have the complement problem and the asymmetry between solving a problem in the positive or negative sense
From a TM to a GG (1)

Given a single-tape TM M, let us build a general grammar G, that generates L(M):

– First, G generates all strings of the type x$X, x ∈ V_T^*, X being a “copy of x” composed of nonterminal symbols (e.g., for x = aba, x$X = aba$ABA)
  • This can be done with a finite number of productions
– G simulates the configurations of M using the string on the right of $
– G has a derivation x$X ⇒* x iff x is accepted by M
  • The basic idea is to simulate each move of M by an immediate derivation of G
From a TM to a GG (2)

G has therefore derivations of the form $x$X$⇒x$q,X$ (initial configuration of M)

Furthermore

- If, in M, $\delta(q,A) = <q', A', R>$ then G includes the production $qA \rightarrow A'q'$
- If, in M, $\delta(q,A) = <q', A', S>$ then G includes the production $qA \rightarrow q'A'$
- If, in M, $\delta(q,A) = <q', A', L>$ then G includes the production $BqA \rightarrow q' BA'$

∀ B in the alphabet of M (recall that M is single-tape, hence it has a unique alphabet for input, memory, and output)
If and only if: \( x\$\alpha\text{BqAC}\beta \Rightarrow x\$\alpha\text{BA'}q'C\beta \), etc.

– We finally add productions allowing \( G \) to derive from \( x\$\alpha\text{BqFAC}\beta \) the string \( x \) alone iff \( M \) reaches an accepting configuration \( (\alpha\text{BqFAC}\beta) \), by deleting whatever is at the right of $ (including the $)