REGular and Context-Free Grammars

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo .felicioni @ polimi . it

March 30, 2021

[1]Mostly based on Alessandro Barenghi and (especially) Nicholas Mainardi's material

Grammars

What are grammars?

- Another formalism to define a language
- Generative approach: the grammar points out how a sentence (i.e. an element of the language) is generated
- For some grammar classes, automated algorithms to derive the recogniser automaton are available (and pretty widely used!)
- Possible to define grammar classes corresponding to a precise computing model (FSA, [N|D]PDA, TM)

Formalization

A formal definition

- A grammar is defined by a 4-tuple $(\mathbf{V_n}, \mathbf{V_t}, \mathbf{P}, S)$ where:
  - $\mathbf{V_n}$ is the non-terminal symbol alphabet
  - $\mathbf{V_t}$ is the terminal symbol alphabet
  - $\mathbf{P} \subseteq \mathbf{V}^*\mathbf{V_n}^+\mathbf{V}^* \times \mathbf{V}^*$ is the set of syntactic productions, where $\mathbf{V} = (\mathbf{V_t} \cup \mathbf{V_n})$
  - $S \in \mathbf{V_n}$ is the starting symbol, known as the *axiom*
- A derivation $\alpha \Rightarrow \beta$, with $\alpha \in \mathbf{V}^+, \alpha = \alpha_1\alpha_2\alpha_3$ and $\beta \in \mathbf{V}^*, \beta = \alpha_1\beta_2\alpha_3$ exists if and only if there is a $p \in \mathbf{P}$ such that $p = \alpha_2 \rightarrow \beta_2$
- $\Rightarrow^*$ indicates the reflexive and transitive closure of $\Rightarrow$
- A grammar generates the language $L_G = \{x \mid x \in \mathbf{V_t}^* \wedge S \Rightarrow^* x\}$

Notation

Common Conventions

- Non terminal symbols are UPPERCASE, terminal symbols are lowercase
- S is the axiom of the grammar
- Single character symbols are used (no tokenization needed)
- Concatenation . mark is omitted
- Regular expressions in the RHS of the rule are not used except for the $|$ symbol employed to shorten the notation

Recognizer automaton

Equivalences

Grammar Type	Language Class	Generic Rule Form	Recognizer Automaton
3	Regular	$A \to aB \mid a \mid \epsilon$	FSA
2	Context Free	$A \to \beta$	NPDA
1	Context-sensitive	$\alpha \to \beta \vert \epsilon, \vert\alpha\vert \leq \vert\beta\vert$	lin-bounded TM
0	Recursively enumerable	any $\alpha \to \beta$	TM

First examples

Warming up

- Regular: $L = (aa)^*$
- Sample grammar generating the language
  - $S \to \epsilon$ (zero is even)
  - $S \to aA$ (when the first a is produced...)
  - $A \to aS$ (make a pair and continue, or ...)
  - $A \to a$ (make a pair and stop)
- Context Free: $L = a^n b^n c^m a^m; n \geq 0, m \geq 1$
- Sample grammar generating the language
  - $S \to S_1 S_2$ (concatenation is easy)
  - $S_1 \to aS_1 b | \epsilon$ (grow the pairs from within)
  - $S_2 \to cS_2 a | ca$ (avoid generating no ca pairs)

Union With Grammars

Union of languages is straightforward too!

Big/Little Endian Encodings

Consider the language L defined on the alphabet $\{0, 1, a\}$, as the union of 2 sublanguages L_l and L_b:

- $L_l = \{(na^n)^+ \mid 0 \leq n_{(10)} \leq 3\}$
  - ↪ Here, n is written with little endian binary representation (first digit is the least significant one)
- $L_b = \{(na^n)^+ \mid 0 \leq n_{(10)} \leq 3\}$
  - ↪ Here, n is written with big endian binary representation (first digit is the most significant one)
- For instance, for a sequence of 2 a:
  - Little endian: $01aa$
  - Big Endian: $10aa$

Union With Grammars

Grammar for L_l

- $S_l \rightarrow 0Z \mid 1U$
- $Z \rightarrow 0N_0 \mid 1N_2$
- $U \rightarrow 0N_1 \mid 1N_3$
- $N_0 \rightarrow \epsilon \mid S_l$
- $N_1 \rightarrow aN_0$
- $N_2 \rightarrow aN_1$
- $N_3 \rightarrow aN_2$

Grammar for L_b

- $S_b \rightarrow 0Z \mid 1U$
- $Z \rightarrow 0N_0 \mid 1N_1$
- $U \rightarrow 0N_2 \mid 1N_3$
- $N_0 \rightarrow \epsilon \mid S_b$
- $N_1 \rightarrow aN_0$
- $N_2 \rightarrow aN_1$
- $N_3 \rightarrow aN_2$

Grammar for $L = L_l \cup L_b$

$S \rightarrow S_l \mid S_b$

Union With Grammars

Problem: There are conflicting non-terminal symbols in the sub-grammars! Hence, the grammar generates strings as $01a10a$:

- $S \Rightarrow S_b$
- $S_b \Rightarrow 0Z$
- $0Z \Rightarrow 01N_1$
- $01N_1 \Rightarrow 01aN_0$
- $01aN_0 \Rightarrow 01aS_b$
- $01aS_b \Rightarrow 01a1U$
- $01a1U \Rightarrow 01a10N_1$
- $01a10N_1 \Rightarrow 01a10aN_0$
- $01a10N_0 \Rightarrow 01a10a$

$\hookrightarrow$ This derivation is possible since the merging operation transforms the U rule in: $U \rightarrow 0N_1 \mid 0N_2 \mid 1N_3$

$\hookrightarrow$ Before performing union, the sets of non-terminal symbols of the sub-grammars must be disjoint

Union With Grammars

L is a regular language defined on the alphabet $\Sigma = \{a, b, 0, 1\}$ as:

$L_1 = \{x = a.y \mid y \in \Sigma^* \wedge |y|_0 = 2k + 1 \wedge |y|_1 = 2h, h, k \geq 0\}$

$L_2 = \{x = b.y \mid y \in \Sigma^* \wedge |y|_0 = 2k \wedge |y|_1 = 2h + 1, h, k \geq 0\}$

$L_3 = \{x = (0 \mid 1).y \mid y \in \Sigma^*\}$

Idea: We can easily handle the 3 sub-languages with 3 sub-grammars, with axioms S_1, S_2, and S_3, and then choose among these sub-languages depending on the first character:

Grammar for L

$S \rightarrow aS_1 \mid bS_2 \mid 0S_3 \mid 1S_3$

Union With Grammars

Grammar For L_1

- $S_1 \rightarrow aS_1 \mid bS_1 \mid 0O_e \mid 1E_o$
- $O_e \rightarrow 0S_1 \mid 1O_o \mid aO_e \mid bO_e \mid \epsilon$
- $E_o \rightarrow 0O_o \mid 1S_1 \mid aE_o \mid bE_o$
- $O_o \rightarrow 0E_o \mid 1O_e \mid aO_o \mid bO_o$

Grammar For L_2

- $S_2 \rightarrow aS_2 \mid bS_2 \mid 0O_e^2 \mid 1E_o^2$
- $O_e^2 \rightarrow 0S_2 \mid 1O_o^2 \mid aO_e^2 \mid bO_e^2$
- $E_o^2 \rightarrow 0O_o^2 \mid 1S_2 \mid aE_o^2 \mid bE_o^2 \mid \epsilon$
- $O_o^2 \rightarrow 0E_o^2 \mid 1O_e^2 \mid aO_o^2 \mid bO_o^2$

Grammar for L_3 is easier: $S_3 \rightarrow aS_3 \mid bS_3 \mid 0S_3 \mid 1S_3 \mid \epsilon$

A Left-Linear Version

The grammar for the language L can be written in a more compact form using left-linear productions. With a left-linear grammar, we generate a string from the end to the beginning!

$\hookrightarrow$ Therefore, right linear grammars are often easier to be conceived

Idea for this language: generates the string up to the second character, then allows replacement of the non-terminal symbol based on the parities of 0 and 1:

1. $S \to Sa \mid Sb \mid O_e0 \mid E_o1 \mid 0 \mid 1$. The first character must be 0 or 1
2. $O_e \to O_ea \mid O_eb \mid S0 \mid O_o1 \mid a \mid 0 \mid 1$. The first character cannot be b
3. $E_o \to E_oa \mid E_ob \mid O_o0 \mid S1 \mid b \mid 0 \mid 1$. The first character cannot be a
4. $O_o \to O_oa \mid O_ob \mid E_o0 \mid O_e1 \mid 0 \mid 1$. The first character must be 0 or 1

Simplifying a Grammar

Consider the language $L = \{s = c^m.y \mid y \in L_Y \wedge m > 0\}$

$\hookrightarrow$ $L_Y = \{a^m.x \mid x \in L_X \wedge m > 0\} \cup \epsilon$

$\hookrightarrow$ $L_X = \{b^m.y \mid y \in L_Y \wedge m > 0\} \cup \epsilon$

A Grammar for L

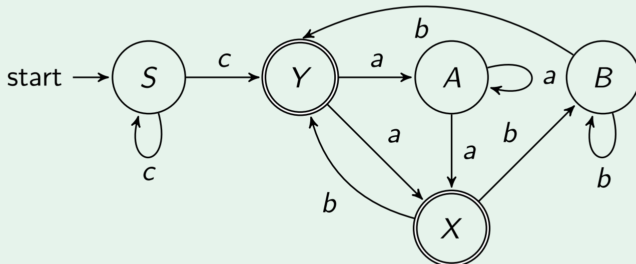
1. $S \rightarrow cS \mid cY$
2. $Y \rightarrow aA \mid aX \mid \epsilon$
3. $X \rightarrow bB \mid bY \mid \epsilon$
4. $A \rightarrow aA \mid aX$
5. $B \rightarrow bB \mid bY$

Simplifying a Grammar: Transforming to FSA

We can generate a FSA from a grammar:

- $\mathbf{Q} = \mathbf{V_n} \cup q_f$
- $\mathbf{I} = \mathbf{V_t}$
- $\delta(q, i) = q' \iff \exists\langle q\rangle \to i\langle q'\rangle \in \mathbf{P} \wedge \delta(q, i) = q_f \iff \exists\langle q\rangle \to i \in \mathbf{P}$
- $\mathbf{F} = \{q \mid \exists\langle q\rangle \to \epsilon \in \mathbf{P}\} \cup q_f$

ND-FSA for L

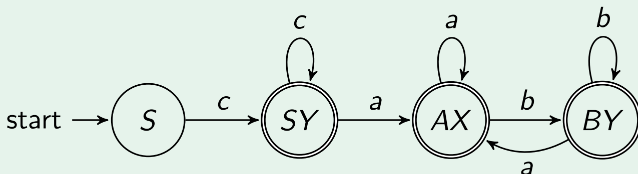


Simplifying a Grammar: Determinization

Generally, it is likely that the automaton derived from a grammar is non-deterministic

We can make it deterministic with the known algorithm:

Deterministic FSA for L



Simplifying a Grammar: FSA Minimization

- Given a FSA, it is always possible to get the FSA recognizing the same language with the minimum number of states
- There is an algorithm, which is based on the concept of indistinguishable states[2]

Indistinguishable States

Given a FSA, 2 states $q, q' \in \mathbf{Q}$ are indistinguishable if:
$q \in \mathbf{F} \iff q' \in \mathbf{F} \land \forall i \in \mathbf{I}(\delta(q,i) = \delta(q',i))$

Idea of the algorithm:

1. Search for indistinguishable states
2. If there are indistinguishable states, merge them in a unique state and go back to 1
3. Otherwise, the minimum automaton has been reached

[2]Simplified definition: general algorithm based on k-distinguishable states

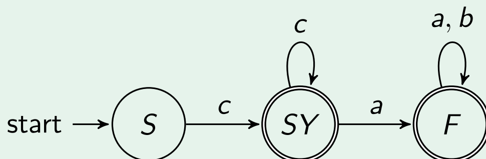
Simplifying a Grammar: FSA Minimization

Are there any indistinguishable states in our FSA?

Yes! AX and BY! Indeed:

1. $AX \in \mathbf{F} \wedge BY \in \mathbf{F}$
2. $\delta(AX, a) = \delta(BY, a) = AX$
3. $\delta(BY, b) = \delta(AX, b) = BY$

Minimum Deterministic FSA for L



Simplifying a Grammar: Getting Back to the Grammar

Once we have our minimum deterministic FSA, we can transform it back to a grammar:

- $\mathbf{V_n} = \mathbf{Q}$
- $\mathbf{V_t} = \mathbf{I}$
- $\exists\langle q\rangle \rightarrow i\langle q'\rangle \in \mathbf{P} \iff \exists\delta(q, i) = q'$
- $\exists\langle q\rangle \rightarrow \epsilon \in \mathbf{P} \iff q \in \mathbf{F}$

Simplified Grammar for L

1. $S \rightarrow cS_y$
2. $S_y \rightarrow cS_y \mid aF \mid \epsilon$
3. $F \rightarrow aF \mid bF \mid \epsilon$

Counting with grammars

Position independent counting

- Target language $L = (a|b)^+, \forall x \in L, |\ x\ |_a = |\ x\ |_b$
- Context-free language, we only need to count one kind of symbols
  - $S \rightarrow aAbG \mid bBaG$
  - $G \rightarrow aAbG \mid bBaG \mid \epsilon$
  - $A \rightarrow aAb \mid bBa \mid \epsilon \mid G$
  - $B \rightarrow aAb \mid bBa \mid \epsilon \mid G$
- or, in a more compact form,
  - $S \rightarrow aGbG|bGaG$
  - $G \rightarrow aGbG \mid bGaG \mid \epsilon$
- The arbitrary choice of the production rule allows to generate every combination

Counting With Grammars

Example of derivations for the string *bbaabaaababb*:

- $S \Rightarrow bBaG$. It starts with a sequence of b
- $bBaG \Rightarrow bbBaaG$. Dealing with the first sequence of b
- $bbBaaG \Rightarrow bbaaG$. The sequence is finished, thus get rid of B
- $bbaaG \Rightarrow bbaabBaG$. Another pair starting with b
- $bbaabBaG \Rightarrow bbaabaG$.The sequence is finished, thus get rid of B
- $bbaabaG \Rightarrow bbaabaaAbG$. This time we have a sequence of a
- $bbaabaaAbG \Rightarrow bbaabaaaAbbG$. Dealing with the sequence of a
- $bbaabaaaAbbG \Rightarrow bbaabaaaGbbG$. There are other sequences between the second a and the matching b, thus we need a new non-terminal G before the b
- $bbaabaaaGbbG \Rightarrow bbaabaaabBaGbbG$. We need to generate a pair ba.
- $bbaabaaabBaGbbG \Rightarrow bbaabaaabaGbbG$. The sequence is finished, thus get rid of B.
- $bbaabaaabaGbbG \Rightarrow bbaabaaababb$. We generate all the necessary A and B, thus we can get rid of G.

The Generative Approach of Grammars

Consider the language $L = \{a^m b^n \mid m \neq n, m, n \geq 0\}$. We want to write a grammar for this language.

A Possible Solution

1. $S \rightarrow aSb \mid aA \mid Bb$ (Generate balanced a, b pairs until an unbalanced character is generated)
2. $A \rightarrow aA \mid aAb \mid \epsilon$ (Generate balanced and unbalanced a)
3. $B \rightarrow Bb \mid aBb \mid \epsilon$ (Generate balanced and unbalanced b)

- Basic idea of the design: non-terminal symbols A and B allows to recognize a string respectively with $m \geq n$ and with $m \leq n$.
- In order to generate an A or B symbol, we generate an unbalanced a or b, thus ensuring that the generated strings have either $m > n$ or $m < n$.

The Generative Approach of Grammars

Grammars are a generative model!

We do not need to allow a grammar to generate both balanced and unbalanced characters, but we can decide to split the generated string in 2 parts: the first where a and b are balanced, the second when we add either a or b.

A Simplified Solution

1. $S \rightarrow aSb \mid aA \mid bB$ (Generate balanced a until an unbalanced character is generated)
2. $A \rightarrow aA \mid \epsilon$ (Add unbalanced a after balanced a)
3. $B \rightarrow bB \mid \epsilon$ (Add unbalanced b before balanced b)

Easier to design the grammar, since we have two simpler separated generation phases

The Generative Approach of Grammars

Consider the language:
$L = \{ab^{n_1}ab^{n_2}\dots ab^{n_k} \mid \forall i, n_i > 0 \wedge k \geq 2 \wedge \exists j, h(1 \leq j < h \leq k \wedge n_j = n_h)\}$
That is, the strings of the form $(ab^+)(ab^+)^+$ where at least two substrings (ab^+) have the same number of b.

A Grammar For L

- $S \rightarrow GaXG$ (This defines the structure of a string)
- $X \rightarrow bXb \mid bGab$ (Generate the substrings with the same number of b)
- $G \rightarrow aH \mid \epsilon$ (Generate a sequence (possibly empty) of substrings ab^+)
- $H \rightarrow bH \mid bG$ (Generate b^+)

The strategy is again forcing the grammar to generate a substring with the require property (2 substrings with the same number of b) somewhere on the string, but without specifying when this substring should be generated

↪ Sooner or later the substring will be generated, and this is enough!

↪ Again, grammars have more "free will" than automata

Proofs on grammars

Mathematical induction on generation

- Goal: prove that $S \rightarrow 1S1 \mid 0S0 \mid 1 \mid 0 \mid \epsilon$ generates all, and only, the palindromes over $\Sigma = \{0, 1\}$
- The theorem we want to prove is a double implication:
  1. $x = w(0 \mid 1 \mid \epsilon)w^R, w \in \Sigma^* \implies \exists S \Rightarrow^* x$
  2. $\exists S \Rightarrow^* x \implies x = w(0 \mid 1 \mid \epsilon)w^R, w \in \Sigma^*$
- We will use thus two different induction steps :
  1. Since the hypothesis is defined on words, induction on word length
  2. Since the hypothesis is defined on the grammar, induction on the number of productions

Proofs on grammars - Part 1

Is a palindrome ⇒ is generated by the grammar

- Base Case : ϵ (length 0) is a palindrome
  - ϵ is generated by the grammar ✓
- Induction Step: the theorem holds for $\mid x \mid = k-1, k \in \mathbb{N}$, prove that it holds for $\mid x \mid = k$
- Split into two cases :
  1. k is odd: $x = u(0 \mid 1)u^R$.
  2. k is even: $x = ww^R$.

Proofs on grammars - Part 1

1. $x = u(0 \mid 1)u^R$. By induction hypothesis, $\exists S \Rightarrow^* uu^R$. We argue that $\exists S \Rightarrow^* uSu^R$. Indeed, the string uu^R has an even number of characters, and since all the productions rewriting S generates 2 characters, we cannot stop generation with the productions $S \rightarrow 1$ or $S \rightarrow 0$, since we would get a string with an odd number of characters. Therefore, $S \rightarrow \epsilon$ must be employed to get uu^R, in turn implying that the grammar generates the string uSu^R. We can get x from this string by applying one of the productions $S \rightarrow 1$ or $S \rightarrow 0$.
2. $x = ww^R = uaau^R, a \in \{0, 1\}$. By induction hypothesis, $\exists S \Rightarrow^* uau^R$. From the previous case, we know also $\exists S \Rightarrow^* uSu^R$, since uau^R is necessarily obtained through applying one of the $S \rightarrow a$ productions. Then, if we apply one of the $S \rightarrow aSa$ productions and $S \rightarrow \epsilon$ one, we can generate x

Proofs on grammars - Part 2

Is generated by the grammar ⇒ is a palindrome

- Base Case : the productions $S \Rightarrow 0 \mid 1 \mid \epsilon$
  - $\epsilon, 0, 1$ are palindromes ✓
- Induction Step: the theorem holds for $S \Rightarrow^{x} w, x < k \in \mathbb{N}$, prove that it holds for $S \Rightarrow^{k} w$
- We need to check that all the grammar productions preserve the palindrome property.
- For $a \in 0, 1$, by inductive hypothesis: $\exists S \Rightarrow^{k-1} x \implies x \in L \implies x = w(0 \mid 1 \mid \epsilon)w^R \implies \exists S \Rightarrow^{k-2} wSw^R \implies \exists S \Rightarrow^{k-1} waSaw^R$
  - $\exists S \Rightarrow^{k-1} waSaw^R \implies \exists S \Rightarrow^{k} waaw^R \in L$ ✓ (using $S \rightarrow \epsilon$).
  - $\exists S \Rightarrow^{k-1} waSaw^R \implies \exists S \Rightarrow^{k} wa(0 \mid 1)aw^R \in L$ ✓(using $S \rightarrow 0$ or $S \rightarrow 1$)
- All the possible productions leading to a valid word at k steps preserve the palindrome property, thus the theorem holds for k

Turing Complete Grammars

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo .felicioni @ polimi . it

March 30, 2021

[1] Mostly based on Nicholas Mainardi's material

Turing Complete Languages and Grammars

We want to design a grammar able to generate
$L = \{a^n b^{2n} c^{\frac{n}{2}} \mid n \geq 1\}$

Hint on CS Grammars

- Basic idea of context-sensitive (CS) grammars is that we have to simulate TM tapes
- The tape stores non-terminal symbols
- We employ other non-terminal symbols to simulate "heads" of the tape
- Symbols in the tape are moved through swap rules

Basic idea for L:

1. write on the tape a pair of symbols BC for each a generated
2. Swaps B and C until on the tape there is B^+C^+
3. Generate 2 b for each B and a c every 2 C

Turing Complete Languages and Grammars

Grammar for $L = \{a^n b^{2n} c^{\frac{n}{2}} \mid n \geq 1\}$

1. $S \rightarrow aXBCE$ (E symbol at the end of the tape is used to generate c)
2. $X \rightarrow aXBC \mid F$ (we write on the tape BC. When the grammar decides to stop writing on the tape, it adds at the beginning of it a F symbol, needed to generate b)
3. $CB \rightarrow BC$ (swap rule. Used to move C symbols after B ones)
4. $CE \rightarrow G$ (Generating c employing E as a head from the end of the tape. The first C found is not translated into a c)
5. $CG \rightarrow Ec$ (The second C found is translated)
6. $FB \rightarrow bbF$ (Generating 2 b employing F as a head from the beginning of the tape)
7. $FE \rightarrow \epsilon$ (Get rid of F and E only if all B and C have been erased, with n being even)
8. $FG \rightarrow \epsilon$ (Get rid of F and G only if all B and C have been erased, with n being odd)

Computing with grammars

Context Matters

- Target language $L = a^{2^i}$
- In this case we will "emulate" a Turing Machine with the grammar behavior
- Basic Idea: At each iteration, you need to double the number of a
- How to do it with a grammar?
- You employ a non-terminal symbol which acts like the head of a TM on a tape
- The tape stores the number of a generated so far, thus we iterate over the tape writing 2 a for each a found

Computing With Grammar

Grammar For a^{2^i}

1. $S \rightarrow BAE$ (place the beginning/end marker plus an A generator)
2. $B \rightarrow BC$ (place a "doubler" mark)
3. $CA \rightarrow AAC$ (every time C swaps places with A it doubles it)
4. $CE \rightarrow E$ (C is discarded at the end)
5. $A \rightarrow a$ (actually generate the a)
6. $B \rightarrow \epsilon$
7. $E \rightarrow \epsilon$

Each time we generate a C, a new iteration, doubling the number of A is started

Computing With Grammar

An Example Of A Derivation

- $S \Rightarrow BAE$
- $BAE \Rightarrow BCAE$, using Rule 2. A new iteration starts ($i = 1$)
- $BCAE \Rightarrow BAACE$, using Rule 3. Doubling the A
- $BAACE \Rightarrow BAAE$, using Rule 4. All the A on the tape have been doubled, we can get rid of C
- $BAAE \Rightarrow BCAAE$, using Rule 2. New iteration ($i = 2$)
- $BCAAE \Rightarrow BAACAE$, using Rule 3. Doubling the A
- $BAACAE \Rightarrow BAAAACE$, using Rule 3. Doubling the A
- $BAAAACE \Rightarrow BAAAAE$, using Rule 4. All the A on the tape have been doubled, we can get rid of C
- $BAAAAE \Rightarrow BaaaaE$, using iteratively Rule 5. We are done with computation, we generate terminal symbols.
- $BaaaaE \Rightarrow aaaa$, using Rule 6 and 7. We get rid of the beginning and end of string markers.

Computing With Grammar

Context-Sensitive Grammars may not be able to generate a terminal string for each sequence of productions!

Example

- $S \Rightarrow BAE$
- $BAE \Rightarrow BaE$, using Rule 5
- $BaE \Rightarrow BCaE$, using Rule2
- $BCaE \Rightarrow Ca$, using Rule 6 and 7

To remove C, we need E, but we have just erased it! The grammar cannot go on in the generation process!

However, this is not an issue! The grammar is correct if it generates all and only the strings of the language, that is:

1. $\forall x(x \in L \implies S \Rightarrow^* x)$
2. $\forall x(S \Rightarrow^* x \implies x \in L)$

Computing With Grammar

Consider now a similar language: $L = \{a^n b^{2^n} \mid n \geq 0\}$

Idea: Each time an A is generated, double the number of B

Grammar For L

1. $S \rightarrow GBE$ (Generate one B, for the case $n = 0$, and place the usual end marker E)
2. $G \rightarrow aGA \mid \epsilon$ (Generate an a altogether with A, used to double the B)
3. $AB \rightarrow BBA$ (Using A to double the number of B)
4. $AE \rightarrow E$ (All the B have been doubled, thus we get rid of A)
5. $B \rightarrow b$ (Generate b)
6. $E \rightarrow \epsilon$ (Erase E)

NB If B and E are erased before all A have been matched to them, the grammar cannot erase all $A \Rightarrow$ No issues during erasure!

The Parrot Language

Consider the parrot language without delimiter c:
$L = \{ww \mid w \in \{a, b\}^*\}$ How can we generate it? Basically, we write w and a copy of each of its character, then we move the copies at the end of the string

- Each time we write A or B, we also write copies
- At the end of the generation of w, we mark the center of the string with a non-terminal symbol C
- We need to move copies after C in the same order. How?
- Each copy is replaced by a or b as soon as it swaps with C
- Order among copies is preserved since the first character to swap with C is the last one of w and the subsequent ones are appended to already copied characters
- We need dummy symbols Y, Z for the copies, since otherwise we can alter w after this have been copied

The Parrot Language

Grammar For $L = \{ww \mid w \in \{a, b\}^*\}$

1. $S \rightarrow AYS \mid BZS \mid C$ (We generate w and the copies of each character with dummy symbols)
2. $YA \rightarrow AY$ (Move Y towards the end of the generated string)
3. $YB \rightarrow BY$ (Move Y towards the end of the generated string)
4. $ZA \rightarrow AZ$ (Move Z towards the end of the generated string)
5. $ZB \rightarrow BZ$ (Move Z towards the end of the generated string)
6. $YC \rightarrow Ca$ (Turn a Y into a since the second part of the string has been reached)
7. $ZC \rightarrow Cb$ (Turn a Z into b since the second part of the string has been reached)
8. $A \rightarrow a$ (Generate a)
9. $B \rightarrow b$ (Generate b)
10. $C \rightarrow \epsilon$ (Erase C)

Which Grammar Do We Need For A Language?

Consider the language $L = \{a^n b^{in} \mid n \geq 1, i \geq 1\}$. Can we generate it with a Context-Free grammar?

We can see it as $L = \{a^n b^n\} \cup \{a^n b^{2n}\} \cup \{a^n b^{3n}\} \ldots$, thus it seems we can do it with a CF grammar. But..

This is the union of an infinite number of languages! CF grammar can perform union only of a finite number, since they need productions for each of them!

We need a Context-Sensitive grammar!

Which Grammar Do We Need For A Language?

How can we generate it with a CS grammar? Write A^nB^n, and then add n B at each iteration

Grammar For $L = \{a^n b^{in} \mid n \geq 1, i \geq 1\}$

1. $S \rightarrow IAGb$ (Place beginning marker(I), used to perform each iteration
2. $G \rightarrow AGb \mid C$ (Generate A^nB^n. We split A from B using a C symbol)
3. $I \rightarrow IJ \mid \epsilon$ (Generate J symbol used to generate n B in each iteration)
4. $JA \rightarrow AJB$ (Generate a B for each A read)
5. $BA \rightarrow AB$ (Move B after the sequence of A)
6. $BC \rightarrow Cb$ (Move B after the sequence of A and make it b)
7. $JC \rightarrow C$ (All A have generated one additional B, and all of them have been moved after the sequence of A, thus we get rid of J)
8. $A \rightarrow a$ (Generate a)
9. $C \rightarrow \epsilon$ (Erase C)

Which Grammar Do We Need For A Language?

Consider the language $L = \{(a^n b^n)^m \mid n \geq 1, m \geq 1\}$. Can we do it with a Context-Free grammar?

We necessarily need a Context-Sensitive grammar, since when b are counted, we cannot count a anymore.

How?

- First, we generate M non terminal symbols, which represent the m sequences of $a^n b^n$ we want to generate
- At each iteration, for each M, an A and a B are generated
- If we perform n iterations, we get a string of the language, which can be turned into a valid one by generating an a from each A and a b from each B

Which Grammar Do We Need For A Language?

Grammar For $L = \{(a^n b^n)^m \mid n \geq 1, m \geq 1\}$

1. $S \rightarrow IABMGE$ (Place beginning marker (I), used to perform each iteration, and the end marker. Generate a pair AB before each M)
2. $G \rightarrow ABMG \mid \epsilon$ (Concatenate M markers preceded by an AB pair)
3. $I \rightarrow IJ \mid \epsilon$ (Generate the symbol J, used to pass through A symbols and add one of them at the end of A^+)
4. $JA \rightarrow AJ$ (Propagate J to the end of A^+)
5. $JB \rightarrow ABK$ (Generate an A at the end of A^+, then turn J into K to pass through B symbols and add one of them at the end of B^+)
6. $KB \rightarrow BK$ (Propagate K to the end of B^+)
7. $KM \rightarrow BMJ$ (Generate a B at the end of B^+, then turn back K to J since there may be a new sequence A^+)
8. $JE \rightarrow E$ (A and B have been generated for all m sequences, so we can erase J)
9. $M \rightarrow \epsilon$ (erase M)
10. $B \rightarrow b$ (Generate b)
11. $A \rightarrow a$ (Generate a)
12. $E \rightarrow \epsilon$ (Erase E)

Which Grammar Do We Need For A Language?

Consider these two languages:

- $L_1 = \{(a, b, c)^* \mid \#a = \#b \vee \#a = \#c\}$
- $L_2 = \{(a, b, c)^* \mid \#b = \#a - \#c \wedge \#a > \#c\}$

Which grammar do we need for L_1?

Grammar for L_1

1. $S \to B \mid C \mid \epsilon$
2. $B \to aBbB \mid bBaB \mid cB \mid \epsilon$
3. $C \to aCcC \mid cCaC \mid bC \mid \epsilon$

A CF grammar! Which automaton?

$\hookrightarrow$ We have to guess, while parsing the string, if we have to count b or c

Non-Deterministic PDA!

Which Grammar Do We Need For A Language?

What about L_2?

- We can see it as $L_2 = \{(a, b, c)^* \mid \#a = \#b + \#c \wedge \#b > 0\}$
- Which automaton is sufficient to recognize this language?

A Deterministic PDA!

- Match each a with either one b or one c using the stack to count them
- Accept only if at least one b has been found and Z_0 is on the stack. What about a grammar?

Grammar For L_2

1. $S \rightarrow aGbG \mid aScS \mid bGaG \mid cSaS$
2. $G \rightarrow aGbG \mid aGcG \mid bGaG \mid cGaG \mid \epsilon$

Is it correct? No: the grammar cannot generate $acbabcaa \in L_2$!

Which Grammar Do We Need For A Language?

Design Grammar for L_2

Conflicting requirements in previous grammar:

- The grammar needs to generate at least a pair containing a b
- At the same time, the grammar cannot avoid generating sequences of a and c without b

Solution: generate one pair with b and then generate all the other pairs with either b or c

1. $S \rightarrow GaGbG \mid GbGaG$: The pair with b is concatenated to other sequences of characters
2. $S \rightarrow GaScG \mid GcSaG$: The pair with b is nested in other pairs
3. $G \rightarrow aGbG \mid aGcG \mid bGaG \mid cGaG \mid \epsilon$

Which Grammar Do We Need For A Language?

Consider now $L_3 = L_1 \cap L_2$

- First of all, strings in L_3 cannot have the same number of a and c, since they would not belong to L_2
- Therefore, $x \in L_3 \implies \#a = \#b$, otherwise $x \notin L_1$
- $x \in L_3 \implies \#a = \#b + \#c \wedge \#a = \#b \implies \#c = 0$, which is ok for L_2, unless $\#a = 0$

$$\Downarrow$$

$$L_3 = \{(a, b)^+ \mid \#a = \#b\}$$

We have already written a CF grammar for this language, as well as a Deterministic PDA