



# AUM

**American University Of The Middle East**

## **Grammar**

## Introduction

As a reminder, a **Finite State Automata (FSA)** for short) can be mathematically modelled as follows:

$$M = \langle Q, \Delta, q_s, Q_a, \Sigma \rangle$$

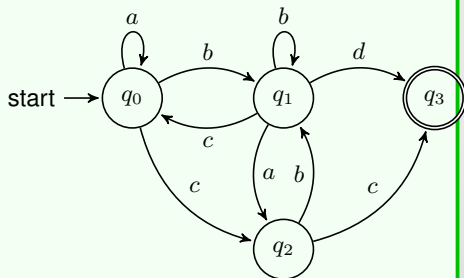
where,

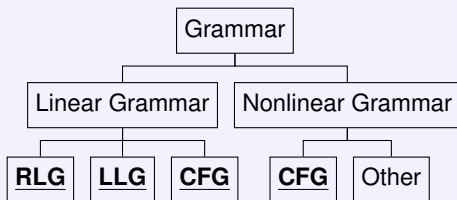
- $Q$  is the set of states in the FSA.
- $\Delta$  is the set of transitions in the FSA.
- $q_s$  is the start state and it has to be one and only one state.
- $Q_a$  is a set of accept states and it could be one or more.
- $\Sigma$  is the alphabet. Remember the alphabet is the set of tokens on the transitions and remember  $\lambda$  is not an alphabet.

$$M = \langle Q, \Delta, q_s, Q_a, \Sigma \rangle$$

- $Q = \{q_0, q_1, q_2, q_3\}$ .
- $q_s = q_0$ .
- $Q_a = \{q_3\}$ .
- $\Sigma = \{a, b, c, d\}$ .
- $\Delta$  :

- 1  $\delta(q_0, a) = q_0$
- 2  $\delta(q_0, b) = q_1$
- 3  $\delta(q_0, c) = q_2$
- 4  $\delta(q_1, a) = q_2$
- 5  $\delta(q_1, b) = q_1$
- 6  $\delta(q_1, c) = q_0$
- 7  $\delta(q_1, d) = q_3$
- 8  $\delta(q_2, b) = q_1$
- 9  $\delta(q_2, c) = q_3$





- **RLG**  $\equiv$  **R**ight **L**inear **G**rammar
- **LLG**  $\equiv$  **L**eft **L**inear **G**rammar
- **CFG**  $\equiv$  **C**ontext **F**ree **G**rammar

The types of grammar that are underlined are the types you are going to be dealing throughout this semester. In these slides, you will be dealing with:

- ↪ Left Linear Grammar
- ↪ Right Linear Grammar
- ↪ Context Free Grammar (Linear case)

Any thing that is called **regular** is called this way because you can translate it to a finite state automata, and you can translate a finite state automata back to it.

## Grammar

$$G = \langle N, \Sigma, R, N_s \rangle$$

where

- ↪  $N$  is a set of **non-terminals** (written in capital letters).
- ↪  $\Sigma$  is a set of **terminals** (written in small letters).
- ↪  $R$  is a set of **rules**.
- ↪  $N_s$  is the starting non-terminal.

Regular grammar and FSA equivalence

- ↪  $N$  to Grammar is  $Q$  to FSA.
- ↪  $\Sigma$  to Grammar is  $\Sigma$  to FSA.
- ↪  $R$  to Grammar is  $\Delta$  to FSA.
- ↪  $N_s$  to Grammar is  $q_s$  to FSA.

## Right Linear Grammar



$$A \rightarrow aB$$

$$A \rightarrow bC$$

$$B \rightarrow cA$$

$$C \rightarrow bD$$

$$C \rightarrow d$$

$$D \rightarrow a$$

$$D \rightarrow \lambda$$

Note the following

- 1 For any rule, the left hand side of the rule has always one and only one Non-terminal.
- 2 It is called linear because in the right hand side of the rule there is one and only one non-terminal.
- 3 It is called right because for the right hand side. If there is a non-terminal then it is in the right hand side of the terminal.

## Left Linear Grammar

$$A \rightarrow Ba$$

$$A \rightarrow Cb$$

$$B \rightarrow Ac$$

$$C \rightarrow Db$$

$$C \rightarrow d$$

$$D \rightarrow a$$

$$D \rightarrow \lambda$$

Note the following

- 1 For any rule, the left hand side of the rule has always one and only one non-terminal.
- 2 It is called linear because in the right hand side of the rule there is one and only one non-terminal.
- 3 It is called left because for the right hand side. If there is a non-terminal then it is in the left hand side of the terminal.

## Linear Context Free Grammar

## Linear Context free grammar

$$A \rightarrow aAb$$

$$A \rightarrow cBd$$

$$B \rightarrow \lambda$$

Note the following

- 1 For any rule, the left hand side of the rule has always one and only one non-terminal.
- 2 It is called linear because in the right hand side of the rule there is one and only one non-terminal.
- 3 It is context free because it is neither left or right.

## Nonlinear Context free grammar

$$A \rightarrow aAbBc$$

$$B \rightarrow CDE$$

$$C \rightarrow bDD$$

$$D \rightarrow aEv$$

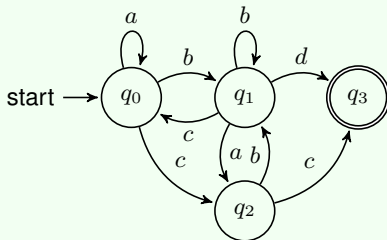
$$D \rightarrow \lambda$$

- 1 For any rule, the left hand side of the rule has always one and only one non-terminal.
- 2 It is called nonlinear because in the right hand side of the rule there is more than one non-terminals.
- 3 It is context free because in the right hand side of the rule there is more than one non-terminal.

**Trick:** To convert from a finite state automata to grammar, it is a good idea to give labels. the easiest way to do it is to use the same state name but in capital letters. in other words,  $q_0$  gets the label  $Q_0$  and  $q_2$  gets the label  $Q_2$ .

**Finite State Automata to Grammar and Vice Vera:** The outcome of the conversion is as follows:

- 1  $Q_0 \rightarrow aQ_0 \parallel bQ_1 \parallel cQ_2$
- 2  $Q_1 \rightarrow cQ_0 \parallel bQ_1 \parallel aQ_2 \parallel dQ_3.$
- 3  $Q_2 \rightarrow bQ_1 \parallel cQ_3.$
- 4  $Q_3 \rightarrow \lambda$  ( because it is an accept).



$Q_0 \rightarrow aQ_0 \parallel bQ_1 \parallel cQ_2$  are three separate rules joined together:  
 $Q_0 \rightarrow aQ_0$  and  $Q_0 \rightarrow bQ_1$  and  $Q_0 \rightarrow cQ_2$  same applied for all the rules above.

Discussion: How is the conversion done from FSA to grammar?!

- 1 Assign a label to every state.
- 2 For every transition create a rule. In other words, if you have  $q_0$  goes to  $q_1$  with alphabet  $a$ , then it is translated this way:  $Q_0 \rightarrow aQ_1$ .
- 3 If there is a self loop then the rule goes to the same non-terminal. In other words, if you have  $q_0$  with a self loop using label  $b$  then it is translated as follows:  $Q_0 \rightarrow bQ_0$
- 4 Accept states are rules with  $\lambda$ . In other words, the FSA in Slide 15 has  $q_3$  as an accept. it translates to adding this rule to the grammar:  $Q_3 \rightarrow \lambda$ .

Note that any string that can be recognized by an FSA can be generated by the equivalent grammar. Recognition means the string is accepted by the FSA.

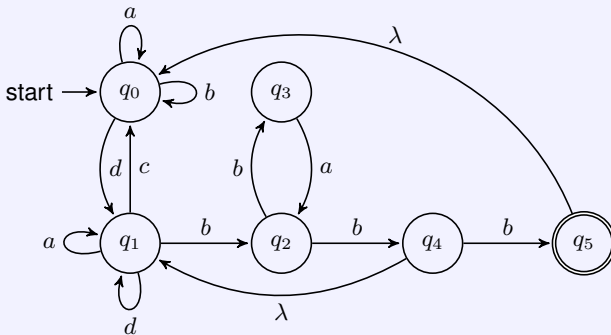


Discussion: How is the conversion done from grammar to FSA?!

- ① For every non-terminal create a state.
- ② For every rule create a transition as follows:
  - If the rule is like this  $A \rightarrow aB$  then create a transition from state  $A$  to state  $B$  with a label  $a$ .
  - If the rule is like this  $A \rightarrow B$  then create a transition from state  $A$  to state  $B$  with a label  $\lambda$ .
  - If the rule is like this  $A \rightarrow aA$  then create a self loop on  $A$  with a label  $a$ .
  - If the rule is like this  $A \rightarrow a$  then create a new state call it  $q_{new}$  and create a transition from  $A$  to  $q_{new}$  with a label  $a$ . Note that in this case  $q_{new}$  has to be an accept state.
  - If the rule is like this  $A \rightarrow \lambda$  then make  $A$  an accept state.

Note that any string that can be generated by certain grammar can be recognized by the equivalent FSA. Recognition means the string is accepted by the FSA.

for the shown FSA find the equivalent grammar



We have the following state  $\mathbb{Q} = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ . Therefore we will be having the following non-terminals  $\mathbb{N} = \{Q_0, Q_1, Q_2, Q_3, Q_4, Q_5\}$ , simply the same name of the states but in capital letters. The starting state is  $q_0$  and that is why the starting non-terminal is  $Q_0$

### Solution

- $Q_0 \rightarrow aQ_0$
- $Q_0 \rightarrow bQ_0$
- $Q_0 \rightarrow dQ_1$
- $Q_1 \rightarrow aQ_1$
- $Q_1 \rightarrow dQ_1$
- $Q_1 \rightarrow cQ_0$
- $Q_1 \rightarrow bQ_2$
- $Q_2 \rightarrow bQ_3$
- $Q_2 \rightarrow bQ_4$
- $Q_3 \rightarrow aQ_2$
- $Q_4 \rightarrow bQ_5$
- $Q_4 \rightarrow Q_1$
- $Q_5 \rightarrow dQ_0$
- $Q_5 \rightarrow \lambda$

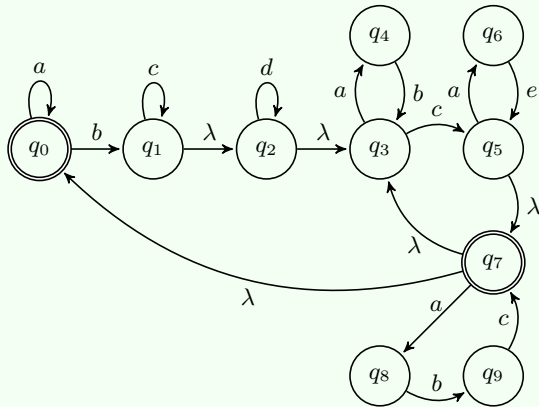
To save space, it can be written this way

- $Q_0 \rightarrow aQ_0 \parallel bQ_0 \parallel dQ_1$
- $Q_1 \rightarrow aQ_1 \parallel dQ_1 \parallel cQ_0 \parallel bQ_2$
- $Q_2 \rightarrow bQ_3 \parallel bQ_4$
- $Q_3 \rightarrow aQ_2$
- $Q_4 \rightarrow bQ_5 \parallel Q_1$
- $Q_5 \rightarrow dQ_0 \parallel \lambda$

For the following grammar find the FSA

- $Q_0 \rightarrow aQ_0 \parallel bQ_1$
- $Q_1 \rightarrow cQ_1 \parallel Q_2$
- $Q_2 \rightarrow dQ_2 \parallel Q_3$
- $Q_3 \rightarrow aQ_4 \parallel cQ_5$
- $Q_4 \rightarrow bQ_3$
- $Q_5 \rightarrow aQ_6 \parallel Q_7$
- $Q_6 \rightarrow eQ_5$
- $Q_7 \rightarrow Q_0 \parallel Q_3 \parallel aQ_8 \parallel \lambda$
- $Q_8 \rightarrow bQ_9$
- $Q_9 \rightarrow cQ_7$

## Solution



String Derivation: Using the given grammar, generate *aaaaaa*

①  $S \rightarrow aS$

②  $S \rightarrow \lambda$

①  $S$  starting non-terminal.

②  $aS$  applying rule number 1

③  $aaS$  applying rule number 1

④  $aaaS$  applying rule number 1

⑤  $aaaaS$  applying rule number 1

⑥  $aaaaaS$  applying rule number 1

⑦  $aaaaaaS$  applying rule number 1

⑧  $aaaaaa$  applying rule number 2

String Derivation: Using the given grammar, generate *aaab* generate string *aaab*  
Given the following grammar:

- ①  $A \rightarrow aA$
- ②  $A \rightarrow B$
- ③  $B \rightarrow bB$
- ④  $B \rightarrow \lambda$

- ① Apply  $A \rightarrow aA$  now you generated *aA*
- ② Apply  $A \rightarrow aA$  now you generated another *a*. total generation *aaA*
- ③ Apply  $A \rightarrow aA$  now you generated another *a*. total generation *aaaA*
- ④ Apply  $A \rightarrow B$  now you can apply B rules. total generation *aaaB*
- ⑤ Apply  $B \rightarrow bB$  now you generated another *b*. total generation *aaabB*
- ⑥ Apply  $B \rightarrow \lambda$  to get rid of *B*. total generation *aaab*

The generation shown in last slide is formulated this way:

$$A \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaab$$

You will see two kinds of arrows in this course:

- 1  $\rightarrow$  and this is used to define a rule like  $A \rightarrow cB$
- 2  $\Rightarrow$  and it is used in derivation.



Having the following left grammar show how you can generate: abb

$$A \rightarrow Ba$$

$$A \rightarrow Cb$$

$$B \rightarrow Ac$$

$$C \rightarrow Db$$

$$C \rightarrow d$$

$$D \rightarrow a$$

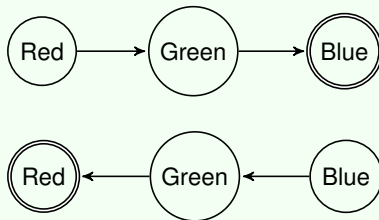
$$D \rightarrow \lambda$$

Derivation goes this way:

$$A \Rightarrow Cb \Rightarrow Db b \Rightarrow abb$$

It is Recursive. it derives the end of the string first then it goes all the way to derive the beginning of the string.

Left Grammar is evil because it is recursive and it a memory hungry monster. That is why Translation engineers do not like it and they tend to convert it to right linear grammar.



We did it in three steps:

- 1 The beginning becomes an end.
- 2 The end becomes a beginning.
- 3 Flip the direction of arcs.

For the following left grammar find the equivalent right grammar

- ①  $A \rightarrow Ba \parallel Ce \parallel D$
- ②  $B \rightarrow Ba \parallel C \parallel a$
- ③  $C \rightarrow Db \parallel b$
- ④  $D \rightarrow Ab \parallel \lambda$

- ① You need to deal with every rule separately.
- ② You need to create a new start state  $S$ .
- ③ You need to specify which is the starting non-terminal in the left grammar and make it an ending non-terminal in the right grammar.
- ④ You need to specify which is the ending non-terminal in the left grammar and make it a starting non-terminal in the right grammar.

Starting non-terminal is non-terminal  $A$ . How did I know? Usually You should be told but if it not specified then take it alphabetically. Since  $A$  is the starting non-terminal, it has to be the end. How will we do that? in the right grammar add the following production rule:  $A \rightarrow \lambda$ . This means there is nothing after  $A$  which makes it the end.

Ending non-terminals The ending non-terminals in the left grammar is every non-terminal that:

- 1 goes to  $\lambda$ . An example since we have the rule  $D \rightarrow \lambda$  then  $D$  is an ending non-terminal in the left grammar.
- 2 goes to a token. An example since we have the rule  $C \rightarrow b$  then  $C$  is an ending non-terminal in the left grammar.  $B$  is also an ending non-terminal since there is a rule  $B \rightarrow a$ .

We need will be solving it with a table based solution. Do not forget to write every single rule in a separate line.

### Solution of problem in slide 27

Left	Right	Comments
$A \rightarrow Ba$	$B \rightarrow aA, A \rightarrow \lambda$	$A \rightarrow \lambda$ as $A$ is an ending non-terminal
$A \rightarrow Ce$	$C \rightarrow eA$	flip the arc
$A \rightarrow D$	$D \rightarrow A$	flip the arc
$B \rightarrow Ba$	$B \rightarrow aB$	flip the arc
$B \rightarrow C$	$C \rightarrow B$	flip the arc
$B \rightarrow a$	$S \rightarrow aB$	make $B$ a start
$C \rightarrow Db$	$D \rightarrow bC$	flip the arc
$C \rightarrow b$	$S \rightarrow bC$	make $C$ a start
$D \rightarrow Ab$	$A \rightarrow bD$	flip the arc
$D \rightarrow \lambda$	$S \rightarrow D$	make $D$ a start

$S$  is an added new non-terminal that works as a starting non-terminal. Every other non-terminal that is supposed to be a start has to be linked to it as seen from the solution above.

Right Linear grammar is a power tool when we need to describe certain statements. Not all statements however can be described by right linear grammar. Some statements need balance. Example of those cases:

- ① Number of **begin** words has to equal the number of **end** words.
- ② Number of **if** words has to equal the number of **then** words which also need to equal the number of **end if** words.
- ③ In a mathematical expression, number of **open brackets** has to equal the number of **close brackets**.
- ④ Number of **for** words has to equal the number of **end for** words.
- ⑤ Number of **while** words has to equal the number of **end while** words.
- ⑥ Number of **do** words has to equal the number of **while** words.
- ⑦ Number of **repeat** words has to equal the number of **until** words.
- ⑧ Number of **Loop** words has to equal the number of **exit loop** words which also need to equal the number of **end Loop** words.

taking the case of **begin** and **end** as an example;

- 1 Use two counters. Count number of **begin** then generate as many **end**.
- 2 There is a better way without using any counters

Lets say there is an organization that wants to make sure that the number of desks is equal to the number of seats without counting. How can this be done? Every time the organization buys one desk , they buy one seat right after. or buy the two of them together. As a result, the number of desks is equal to the number of seats guaranteed without even counting.

What is the difference between:

①  $a^*b^*$

②  $a^nb^n$

$a^*b^*$  means many  $a$  followed by many  $b$ . How many? we do not know and we do not care to know. As you know from last chapter the expression says: number of  $a$  followed by number of  $b$  without specifying the number at all. it could be zero it could be two billions. The expression basically describes the pattern but not the count.

$a^nb^n$  means the number of  $a$  has to equal the number of  $b$  regardless what that  $n$  is.



Find the context free grammar that describes  $a^n b^n$ .

$$S \rightarrow aSb \parallel \lambda$$

Every time  $S$  is going to be replaced by  $aSb$ , we will be generating  $a$  and  $b$  together. That is why without counting it is guaranteed that the number of  $a$  is equal to the number of  $b$ .

Use the above grammar to generate  $a^5 b^5$   $S \Rightarrow aSb \rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaaSbbbbb \Rightarrow aaaaaaSbbbbbb \Rightarrow aaaaaabbbbbbb \equiv a^5 b^5$

All generation steps used  $S \rightarrow aSb$  except for when we are done we used  $S \rightarrow \lambda$ .

Example 2: Find the context free grammar that describes  $a^n c^m d^m b^n$ .

Solution :

$$S \rightarrow aSb \mid A$$

$$A \rightarrow cAd \mid \lambda$$

The problem says that the number of  $a$  is equal to the number of  $b$ . and then after generating equal number of  $a$  and  $b$ , we need to generate number of  $c$  equal to the number of  $d$ . That is why the grammar worries about generating  $a$  and  $b$  first from the production  $S \rightarrow aSb$ . After the required  $n$  is generated, we move to generate  $c$  and  $d$ . We move to the second line by executing the production  $S \rightarrow A$ . After we are on non-terminal  $A$ , we generate  $c$  and  $d$  by using the production  $A \rightarrow cAd$ . Finally we get rid of  $A$  using the production  $A \rightarrow \lambda$ .

Example 3: Find the context free grammar that describes  $a^n c^m d^{m+1} b^n$ .

Solution  $S \rightarrow aSb \parallel A$   
 $A \rightarrow cAd \parallel d$

The problem is similar to the last one except that we need to generate one more  $d$ . So if there is 5  $c$  we need to generate 6  $d$ . This is done by adding one more  $d$  using the production  $A \rightarrow d$  after generating the equal number of  $c$  and  $d$ .

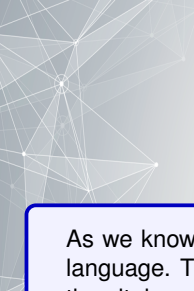
Example 4: Find the context free grammar that describes  $a^{n+2}c^{2m}d^{m+1}b^n$ .

Solution  $S \rightarrow aSb \parallel aaA$

$A \rightarrow ccAd \parallel d$

Now try to explain this yourself. Note that the  $aa$  in production  $S \rightarrow aaA$  is added to the left of  $A$  and that is because it is in the left of  $c^{2m}d^{m+1}$ .

# Grammar Ambiguity



As we known by now, grammar is used to test if certain text belong to certain language. This is done by trying to regenerate the text and if can be generated then it does belong to the language. If the generation failed however, this means that the string does not belong to that particular language expressed by the given grammar.

Having The following grammar generate string: aa

$$A \rightarrow aA \parallel C \parallel aB \parallel cC \parallel a$$

$$B \rightarrow aC \parallel bD \parallel cC \parallel a$$

$$C \rightarrow bD \parallel cC \parallel aD \parallel \lambda$$

$$D \rightarrow aD \parallel a \parallel b \parallel \lambda$$

First parsing path:

$$A \Rightarrow aA \Rightarrow aa$$

Second parsing path:

$$A \Rightarrow aB \Rightarrow aa$$

Third parsing path:

$$A \Rightarrow aB \Rightarrow aaC \Rightarrow aa$$

First parsing path:

Can you find some more?

As seen from the example in the last slide, there are strings that can be generated using many different parsing trees. This makes the grammar an **ambiguous** grammar.

In order to show that a grammar is **ambiguous**, you need to find one string that can be generated by multiple parsing trees. One string is enough to prove the ambiguity of the grammar.

Showing that a grammar is **not ambiguous** is a very difficult task. In order to achieve this, you need to show that each and every string that belongs to the language described by that grammar has one and only one parse tree. One way to do it is using **proof by induction** but it is still a very easy task since the magnitude of the problem is really big.



Given the following grammar, show that the grammar is ambiguous:

$$A \rightarrow B \parallel C \parallel aB \parallel bC$$

$$B \rightarrow bB \parallel D \parallel cC \parallel a$$

$$C \rightarrow aD \parallel bC \parallel \lambda$$

$$D \rightarrow aD \parallel bD \parallel \lambda$$

First try to find a string:  
The string I will be using here is  $ab$

First path :  
 $A \Rightarrow aB \Rightarrow abB \Rightarrow abD \Rightarrow ab$

Second Solution:  
 $A \Rightarrow B \Rightarrow D \Rightarrow aD \Rightarrow abD \Rightarrow ab$

Exercise 1:  
Can you find any other parse tree?

Exercise 2:  
Can you find any other string?

Given the following grammar, show that the grammar is ambiguous

$$A \rightarrow B \parallel C \parallel \lambda$$

$$B \rightarrow aA \parallel aC \parallel bB \parallel D$$

$$C \rightarrow bA \parallel bC \parallel \lambda$$

$$D \rightarrow aA \parallel bA \parallel \lambda$$

First try to find a string The string I will be using here is  $ba$

First path:

$$A \Rightarrow B \Rightarrow bB \Rightarrow baA \Rightarrow ba$$

Second Solution:  $A \Rightarrow C \Rightarrow bA \Rightarrow bB \Rightarrow bD \Rightarrow baA \Rightarrow ba$

Exercise 1:

Can you find any other parse tree?

Exercise 2 Can you find any other string?

# Grammar Equivalence

Two grammars  $G_1$  and  $G_2$  are equivalent if and only of:

$$G_1 \subseteq G_2 \text{ and } G_2 \subseteq G_1$$

In other word, Every string that can be generated by grammar  $G_1$  can also be generated by  $G_2$  and every string that can be generated by grammar  $G_2$  can also be generated by  $G_1$ . Note that this is the condition for any set equivalence.

So in order to prove equivalence of two grammars:  $G_1$  and  $G_2$  You need to show that all string in  $G_1$  are in  $G_2$  and visa verse. which is technically very hard to achieve since the number of strings that can be generated from every grammar is infinite. Again, there is a way to do it through proof by induction, but it is still very hard to achieve.

it is much easier to show that two grammars:  $G_1$  and  $G_2$  are **not** equivalent You need to find one string that can be generated by one of them but not the other. Note that a single string that satisfies this criteria is enough to show that the two grammar are nor equivalent.

### Grammar $G_1$

$$A \rightarrow aB \parallel C \parallel \lambda$$

$$B \rightarrow bC \parallel D$$

$$C \rightarrow bB \parallel aC \parallel \lambda$$

$$D \rightarrow aD \parallel bD \parallel \lambda$$

### Grammar $G_2$

$$A \rightarrow B \parallel C$$

$$B \rightarrow aA \parallel aC \parallel bB \parallel bD$$

$$C \rightarrow bA \parallel aC \parallel b$$

$$D \rightarrow aA \parallel bA \parallel \lambda$$

Show that the two grammars  $G_1$  and  $G_2$  are not equivalent.

How to show it?

Find a string that can be generated by one of them but not the other one. The string I will be using is  $aa$ .

Generation from grammar  $G_1$

$A \Rightarrow C \Rightarrow aC \Rightarrow aaC \Rightarrow aa$

Generation from grammar  $G_2$

$A \Rightarrow B \Rightarrow aA \Rightarrow aB \Rightarrow aaA \Rightarrow aaB \Rightarrow \text{????}$

Lets then try to generate from  $G_2$  through  $A \rightarrow C$

$A \Rightarrow C \Rightarrow aC \Rightarrow aaC \Rightarrow \text{????}B \Rightarrow \text{????}$

Discussion:

It can be generated from  $G_1$  but not from  $G_2$ . Therefore  $G_1$  and  $G_2$  are not equivalent.



### Grammar $G_1$

$$A \rightarrow B \parallel C \parallel \lambda$$

$$B \rightarrow aA \parallel aC \parallel bB \parallel D$$

$$C \rightarrow bA \parallel bC \parallel \lambda$$

$$D \rightarrow aA \parallel bA \parallel \lambda$$

### Grammar $G_2$

$$A \rightarrow B \parallel C$$

$$B \rightarrow aA \parallel aC \parallel bB \parallel bD$$

$$C \rightarrow bA \parallel aC \parallel b$$

$$D \rightarrow aA \parallel bA \parallel \lambda$$

Show that the two grammars  $G_1$  and  $G_2$  are not equivalent.

How to show it?

Find a string that can be generated by one of them but not the other one. The string I will be using is  $\lambda$ .

Generation from grammar  $G_1$   $A \Rightarrow \lambda$  or

$A \Rightarrow B \Rightarrow D \Rightarrow \lambda$

$A \Rightarrow c \Rightarrow \lambda$

(Ambiguous grammar)

Generation from grammar  $G_2$

$A \Rightarrow B \Rightarrow ???$

Lets then try to generate from  $G_2$  through  $A \rightarrow C$

$A \Rightarrow C \Rightarrow ???$

It can be generated from  $G_1$  but not from  $G_2$ . Therefore  $G_1$  and  $G_2$  are not equivalent.



# PARSING AND PARSE TREES

# EXTRACTING GRAMMAR FROM HIGH LEVEL CODE

*Thank  
you!*