# Biçimsel Diller ve Otomata Teorisi

## Sunu XII
## Bağlamdan Bağımsız Söz Dizimi

İZZET FATİH ŞENTÜRK

# *Syntax as a Method for Defining Languages*

- It was necessary to develop a way of writing complicated algebraic expressions in one line of standard typewriter symbols
  - Because of the nature of early computer input devices: keypunches, paper tape, magnetic tape, typewriters, etc
  - The whole expression had to be encoded in a way that did not require a multilevel display

$$\frac{\frac{1}{2}+9}{4+\frac{8}{21}+\frac{5}{3+\frac{1}{2}}}$$

$$((1/2) + 9)/(4 + (8/21) + (5/(3 + (1/2))))$$

We can easily see that the number is a little more than 9 divided by a little more than 5

# *Syntax as a Method for Defining Languages*

- How can a computer scan over this one-line string of typewriter characters and figure out what is going on?
- The conversion from a "high-level" language into a machine-executable language is done by a program called the **compiler**
  - It must do this in a mechanical, algorithmic way. It cannot just look at the expression and understand it
  - Rules must be given by which this string can be processed

# *Syntax as a Method for Defining Languages*

- We want our machine to be able to reject strings of symbols that make no sense as arithmetic expressions such as “((9) +”
  - This input string should not take us to a final state in the machine
- We cannot know that this is a bad input string until we have reached the last letter
  - Change + to a ) and the formula would be valid
- An FA that translated expressions into instructions simultaneously as it scanned left to right like a Mealy machine would already be turning out code before it realized that the whole expression is nonsense

# *Syntax as a Method for Defining Languages*

- Before we try to build a compiling machine, let us recall what is and what is not a <u>valid arithmetic operation</u>

Rule 1 Any number is in the set $AE$.

Rule 2 If $x$ and $y$ are in $AE$, then so are

$$(x) \qquad -(x) \qquad (x + y) \qquad (x - y) \qquad (x * y) \qquad (x/y) \qquad (x{*}{*}y)$$

- First we must design a machine that can figure out how a given input string was built up from basic rules
- Then we should be able to translate this sequence of rules into an assembler language program
  - Because all these rules are pure assembler language instructions except exponentiation

# *Example*

- Consider the input string ((3 + 4) * (6 + 7))
- The machine discovers that this can be produced from the rules in by the sequence

| | |
|---|---|
| 3 is in *AE* | LOAD 3 in register 1 |
| 4 is in *AE* | LOAD 4 in register 2 |
| (3 + 4) is in *AE* | ADD the contents of register 2 into register 1 |
| 6 is in *AE* | LOAD 6 in register 3 |
| 7 is in *AE* | LOAD 7 in register 4 |
| (6 + 7) is in *AE* | ADD the contents of register 3 into register 4 |
| ((3 + 4) * (6 + 7)) is in *AE* | MULTIPLY register 1 by register 4 |

- The difficult part is to figure out how the input string can be produced from the rules

# *Context-Free Grammars*

- Recognizing the structure of a computer language instruction is analogous to recognizing the structure of a sentence in a human language
- Our ability to understand what a sentence means is based on our ability to understand how it could be formed from the rules of grammar
- Determining how a sentence can be formed from the rules of grammar is called *parsing the sentence*

# *Context-Free Grammars*

- Rules that involve the meaning of words we call **semantics** and rules that do not involve meaning of words we call **syntax**
- In English, the meaning of words can be relevant (birds sing vs Wednesday sings) but in arithmetic the meaning of numbers is rarely catastrophic. One number is as good as another. If X = B + 9 is valid, then so are X = B + 473
- In general, the rules of computer language grammar are all syntactic and not semantic, which makes the task of interpretation much easier

# *Context-Free Grammars*

- Some of the rules of English grammar and an example of how to form the sentence: *The itchy bear hugs the jumpy dog* from rules

1. A *sentence* can be a *subject* followed by a *predicate*.
2. A *subject* can be a *noun-phrase*.
3. A *noun-phrase* can be an *adjective* followed by a *noun-phrase*.
4. A *noun-phrase* can be an *article* followed by a *noun-phrase*.
5. A *noun-phrase* can be a *noun*.
6. A *predicate* can be a *verb* followed by a *noun-phrase*.
7. A *noun* can be

   *apple* *bear* *cat* *dog*

8. A *verb* can be

   *eats* *follows* *gets* *hugs*

9. An *adjective* can be

   *itchy* *jumpy*

10. An *article* can be

   *a* *an* *the*

| | |
|---|---|
| *sentence* ⇒ *subject predicate* | Rule 1 |
| ⇒ *noun-phrase predicate* | Rule 2 |
| ⇒ *noun-phrase verb noun-phrase* | Rule 6 |
| ⇒ *article noun-phrase verb noun-phrase* | Rule 4 |
| ⇒ *article adjective noun-phrase verb noun-phrase* | Rule 3 |
| ⇒ *article adjective noun verb noun-phrase* | Rule 5 |
| ⇒ *article adjective noun verb article noun-phrase* | Rule 4 |
| ⇒ *article adjective noun verb article adjective noun-phrase* | Rule 3 |
| ⇒ *article adjective noun verb article adjective noun* | Rule 5 |
| ⇒ *the adjective noun verb article adjective noun* | Rule 10 |
| ⇒ *the itchy noun verb article adjective noun* | Rule 9 |
| ⇒ *the itchy bear verb article adjective noun* | Rule 7 |
| ⇒ *the itchy bear hugs article adjective noun* | Rule 8 |
| ⇒ *the itchy bear hugs the adjective noun* | Rule 10 |
| ⇒ *the itchy bear hugs the jumpy noun* | Rule 9 |
| ⇒ *the itchy bear hugs the jumpy dog* | Rule 7 |

# *Context-Free Grammars*

- A law of grammar is in reality a suggestion for possible substitutions. The arrow indicates that a substitution was made according to the preeceding rules of grammar
- We have started with the initial symbol sentence and then applied the rules for producing sentences listed in the *generative grammar*
- The words that cannot be replaced by anything are called **terminals**. Words that must be replaced by other things we call **nonterminals**
- The job of sentence production is not complete until all the nonterminals have been replaced with terminals

# *Context-Free Grammars*

- We can follow the same model for defining arithmetic expressions
- We have used the word “Start” to begin the process. The only other nonterminal is AE. The terminals are the phrase “ANY-NUMBER” and the symbols + - * / ** ( )

$\underline{\text{Start}} \rightarrow (\underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow (\underline{\text{AE}} + \underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow (\underline{\text{AE}} - \underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow (\underline{\text{AE}} * \underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow \underline{\text{AE}} / \underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow (\underline{\text{AE}} ** \underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow (\underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow -(\underline{\text{AE}})$
$\underline{\text{AE}} \rightarrow \underline{\text{ANY-NUMBER}}$

# *Context-Free Grammars*

- We can also define a set of rules for “ANY-NUMBER”

| | |
|---|---|
| Rule 1 | ANY-NUMBER → FIRST-DIGIT |
| Rule 2 | FIRST-DIGIT → FIRST-DIGIT OTHER-DIGIT |
| Rule 3 | FIRST-DIGIT → 1 2 3 4 5 6 7 8 9 |
| Rule 4 | OTHER-DIGIT → 0 1 2 3 4 5 6 7 8 9 |

- Rules 3 and 4 offer choices of terminals. We put spaces between them to indicate choose one. Soon we will introduce another symbol

# *Context-Free Grammars*

- We can produce the number 1066 as follows

| | |
|---|---|
| ANY-NUMBER ⇒ FIRST-DIGIT | Rule 1 |
| ⇒ FIRST-DIGIT OTHER-DIGIT | Rule 2 |
| ⇒ FIRST-DIGIT OTHER-DIGIT OTHER-DIGIT | Rule 2 |
| ⇒ FIRST-DIGIT OTHER-DIGIT OTHER-DIGIT OTHER-DIGIT | Rule 2 |
| ⇒ 1066 | Rule 3 and 4 |

- The sequence of applications of the rules that produces the finished string of termianls from the starting symbol is called a **derivation** or a **generation** of the word. The grammatical rules are often referred to as **productions**

# *Context-Free Grammars*

- The derivation may or may not be unique
  - By appliying productions to the start symbol in two different ways, we may still produce the same finished product
- We are now ready to define the general concept of which all these examples have been special cases
- This new structure is called a context-free grammar (CFG)
  - Invented by the linguist Noam Chomsky in 1956

# *Definition*

- CFG is a collection of three things

1. An alphabet Σ of letters called terminals
2. A set of symbols called nontermianls (Symbol S for "Start")
3. A finite set of productions of the form

   One Nonterminal -> finite string of termianls and/or Nonterminals

   We require that at least one production has the nonterminal S as its left side

   To not to confuse terminals and nonterminals, we always designate nonterminals by capital letters, wheras terminals are represented by lowercase letters

# *Definition*

- The <u>language</u> *generated* by a **CFG** is the <u>set of all strings</u> of *terminals* that can be <u>produced</u> from the start symbol *S* using the productions as substitutions
- A language generated by a CFG is called a context-free language (**CFL**)

# *Example*

- Let the only terminal be a and the productions be

$$\text{PROD 1} \quad S \rightarrow aS$$
$$\text{PROD 2} \quad S \rightarrow \Lambda$$

- If we apply production 1 six times and then apply production 2, we generate the following. This is a derivation of $a^6$ in this CFG

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow aaS \\ &\Rightarrow aaaS \\ &\Rightarrow aaaaS \\ &\Rightarrow aaaaaS \\ &\Rightarrow aaaaaaS \\ &\Rightarrow aaaaaa\Lambda \\ &= aaaaaa \end{aligned}$$

- If we apply production 2 without production 1, we find that the null string is itself in the language of this CFG. Because the only nonterminal is a, it is clear that no words outside of a* can possibly be generated. The language generated by this CFG is exactly **a***

# *Example*

- Let the only terminal be a and the productions be

$$\begin{array}{ll} \text{PROD 1} & S \rightarrow SS \\ \text{PROD 2} & S \rightarrow a \\ \text{PROD 3} & S \rightarrow \Lambda \end{array}$$

- In this language, we can have the following derivation

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow SSS \\ &\Rightarrow SaS \\ &\Rightarrow SaSS \\ &\Rightarrow \Lambda aSS \\ &\Rightarrow \Lambda aaS \\ &\Rightarrow \Lambda aa\Lambda \\ &= aa \end{aligned}$$

- This language is also a*, but here the string aa can be obtained in infinitely many ways. In the first example, there was a unique way to produce every word in the language. This also shows that the same language can have more than one CFG generating it

# *Example*

- Let the terminals be a and b, the only nonterminal be S, and the productions be

| | |
|---|---|
| PROD 1 | $S \rightarrow aS$ |
| PROD 2 | $S \rightarrow bS$ |
| PROD 3 | $S \rightarrow a$ |
| PROD 4 | $S \rightarrow b$ |

- We can produce the word baab as follows

$$\begin{aligned} S &\Rightarrow bS && \text{(by PROD 2)} \\ &\Rightarrow baS && \text{(by PROD 1)} \\ &\Rightarrow baaS && \text{(by PROD 1)} \\ &\Rightarrow baab && \text{(by PROD 4)} \end{aligned}$$

- The language generated by this CFG is the set of all possible strings of the letters a and b except for the null string

# *Example*

- Let the terminals be a and b, the nonterminals be S, X, and Y, and the productions be

$$
\begin{aligned}
&S \rightarrow X \\
&S \rightarrow Y \\
&X \rightarrow \Lambda \\
&Y \rightarrow aY \\
&Y \rightarrow bY \\
&Y \rightarrow a \\
&Y \rightarrow b
\end{aligned}
$$

- The language generated is (**a** + **b**)*

# *Example*

- Let the terminals be a and b, the only nonterminal be S, and the productions be

$$
\begin{aligned}
S &\to aS \\
S &\to bS \\
S &\to a \\
S &\to b \\
S &\to \Lambda
\end{aligned}
$$

- The word ab can be generated by both derivations:

$$
\begin{aligned}
S &\Rightarrow aS \\
&\Rightarrow abS \\
&\Rightarrow ab\Lambda \\
&= \text{ab}
\end{aligned}
\qquad\qquad
\begin{aligned}
S &\Rightarrow aS \\
&\Rightarrow ab
\end{aligned}
$$

- The language of this CFG is also (**a** + **b**)*****, but the sequence of productions that is used to generate a specific word is not unique

# *Example*

- Let the terminals be a and b, the nonterminals be S and X, and the productions be

$$S \rightarrow XaaX$$
$$X \rightarrow aX$$
$$X \rightarrow bX$$
$$X \rightarrow \Lambda$$

- anything aa anything

or

- (**a** + **b**)***aa**(**a** + **b**)*

which is the language of all words with a double a in them

# *Example*

- Let the terminals be a and b, the nonterminals be S, X, and Y, and the productions be

$$
\begin{aligned}
S &\rightarrow XY \\
X &\rightarrow aX \\
X &\rightarrow bX \\
X &\rightarrow a \\
Y &\rightarrow Ya \\
Y &\rightarrow Yb \\
Y &\rightarrow a
\end{aligned}
$$

- Although it has more nonterminals and more productions, this grammar generates the same language as the last example: (**a** + **b**)***aa**(**a** + **b**)*

# *Example*

- Let the terminals be a and b and the three nonterminals be S, BALANCED, and UNBALANCED. Let the productions be

$$
\begin{aligned}
S &\rightarrow SS \\
S &\rightarrow \text{BALANCED } S \\
S &\rightarrow S \text{ BALANCED} \\
S &\rightarrow \Lambda \\
S &\rightarrow \text{UNBALANCED } S \text{ UNBALANCED} \\
\text{BALANCED} &\rightarrow aa \\
\text{BALANCED} &\rightarrow bb \\
\text{UNBALANCED} &\rightarrow ab \\
\text{UNBALANCED} &\rightarrow ba
\end{aligned}
$$

- The language generated from these productions is the language EVEN-EVEN (even number of a's and even number of b's)

$$
\begin{aligned}
S &\Rightarrow \text{BALANCED } S \\
&\Rightarrow aaS \\
&\Rightarrow aa \text{ UNBALANCED } S \text{ UNBALANCED} \\
&\Rightarrow aa\ ba\ S \text{ UNBALANCED} \\
&\Rightarrow aa\ ba\ S\ ab \\
&\Rightarrow aa\ ba \text{ BALANCED } S\ ab \\
&\Rightarrow aa\ ba\ bb\ S\ ab \\
&\Rightarrow aa\ ba\ bb\ \Lambda\ ab \\
&= aababbab
\end{aligned}
$$

# *Example*

- Let us consider the CFG

$$S \rightarrow aSb$$
$$S \rightarrow \Lambda$$

- The language generated by this CFG is the nonregular language $a^n b^n$

$$\begin{aligned} S &\Rightarrow aSb \Rightarrow aaSbb \\ &\Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \\ &\Rightarrow aaaaaSbbbbb \Rightarrow aaaaaaSbbbbbb \\ &\Rightarrow aaaaaabbbbbb \end{aligned}$$

# Example

- Let us consider the CFG

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \Lambda$$

- All the words generated by this CFG is in the nonregular language PALINDROME. However, it is not true that all the words in the language PALINDROME can be generated by this grammar
- The language generated by this grammar is that of all palindromes with even length and no center letter called EVENPALINDROME

$$\begin{aligned} S &\Rightarrow aSa \\ &\Rightarrow abSba \\ &\Rightarrow abbSbba \\ &\Rightarrow abbaSabba \\ &\Rightarrow abbaabba \end{aligned}$$

# *Example*

- EVENPALINDROME

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \Lambda$$

- ODDPALINDROME

$$S \Rightarrow aSa$$
$$S \Rightarrow bSb$$
$$S \Rightarrow a$$
$$S \Rightarrow b$$

- PALINDROME

$$S \Rightarrow aSa$$
$$S \Rightarrow bSb$$
$$S \Rightarrow a$$
$$S \Rightarrow b$$
$$S \Rightarrow \Lambda$$

# *Example*

- Another nonregular language that can be generated by a CFG is $a^n b a^n$

$$S \Rightarrow aSa$$
$$S \Rightarrow b$$

- But the cousin nonregular language $a^n b a^n b^{n+1}$ cannot be generated by a CFG for reasons we will discuss later

# *Example*

- Let the terminals be a and b, the three nonterminals be S, A, and B. Let the productions be

$$
\begin{aligned}
S &\to aB \\
S &\to bA \\
A &\to a \\
A &\to aS \\
A &\to bAA \\
B &\to b \\
B &\to bS \\
B &\to aBB
\end{aligned}
$$

- The language that this CFG generates is the language EQUAL. Notice that we included **Λ** in this language previously but for now it has been dropped

$$\text{EQUAL} = \{ab \quad ba \quad aabb \quad abab \quad abba \quad baab \quad baba \quad bbaa \quad aaabbb \ldots\}$$

# *Backus Normal Form (BNF)*

- It is common for the same nonterminal to be the left side of more than one production. We now introduce the symbol " | ", a vertical line, to mean disjunction (or)
- For example $\begin{array}{l} S \rightarrow aS \\ S \rightarrow \Lambda \end{array}$ can be writeen simply as $S \rightarrow aS \mid \Lambda$
- The following CFG can also be written more compactly

$$\begin{array}{l} S \rightarrow X \\ S \rightarrow Y \\ X \rightarrow \Lambda \\ Y \rightarrow aY \\ Y \rightarrow bY \\ Y \rightarrow a \\ Y \rightarrow b \end{array} \qquad \begin{array}{l} S \rightarrow X \mid Y \\ X \rightarrow \Lambda \\ Y \rightarrow aY \mid bY \mid a \mid b \end{array}$$