

**Recall: Context-Free Grammar**

Since DFA's, NFA's, and regular expressions are finite, they won't suffice in reading the syntax. We'll need something more powerful, and from that stems **context-free grammar**, which is an approach to interpreting a sequence of tokens and determining if the syntax of a program is correct.

### 13.1 Examples of Context-free Grammar

Context-free Grammar (CFG) involves a set of rules that we can use to manipulate words found in a language.

#### 13.1.1 Example 1

**Typical CS241 Example**

- G: (R1)  $S \rightarrow aSb$  // "aSb" is *concatenation*
- (R2)  $S \rightarrow D$  // 2 rules with S on LHS is *union*
- (R3)  $D \rightarrow cD$  // D on both sides is *recursion*
- (R4)  $D \rightarrow \epsilon$

- the word *accb* is in the language generated by the grammar G, i.e.  $L(G)$ , since we can *derive accb* from G.

- derivation:

$$S \Rightarrow aSb \Rightarrow aDb \Rightarrow acDb \Rightarrow accDb \Rightarrow accb$$

R1    R2    R3    R3    R4

Figure 13.1: We can derive the word "accb" using this particular CFG. This means that "accb" is a valid word in the language. Courtesy of Prof. Lanctot's slides.

#### 13.1.2 Example 2

Consider the language  $\Sigma = \{a, b\}$  whose language consists of words that start with one 'a', followed by an arbitrary amount of 'b's (i.e.,  $\{a, ab, abb, abbb, \dots\}$ ). The rules of the CFG that reads this language are:

- (R1):  $S \rightarrow aB$
- (R2):  $B \rightarrow bB$
- (R3):  $B \rightarrow \epsilon$

Say we want to derive *abb*, we would do the following:

$$S \xrightarrow{(R1)} aB \xrightarrow{(R2)} abB \xrightarrow{(R2)} abbB \xrightarrow{(R3)} abb$$

### 13.1.3 Example 3

Let's create a CFG that access accepts words with balanced parentheses. For example, valid words are

$$\varepsilon, (), (()), ()(), (()), \dots$$

The rules in the grammar are:

- **(R1):**  $S \rightarrow (S)$
- **(R2):**  $S \rightarrow S S$
- **(R3):**  $S \rightarrow \varepsilon$

From these, let's derive some words:

1. Derive  $()$ :

$$S \xrightarrow{(R1)} (S) \xrightarrow{(R1)} ((S)) \xrightarrow{(R3)} (())$$

2. Derive  $((()))$ :

$$S \xrightarrow{(R1)} (S) \xrightarrow{(R2)} (SS) \xrightarrow{(R1)} ((S)S) \xrightarrow{(R3)} ((()S)) \xrightarrow{(R1)} (((()S))) \xrightarrow{(R3)} (((())))$$

### 13.1.4 Example 4

For  $\Sigma = \{a, b\}$ , a CFG that contains an even number of a's:

- **(R1):**  $S \rightarrow bS$
- **(R2):**  $S \rightarrow Sb$
- **(R3):**  $S \rightarrow aSa$  (a's are generated in pairs, which grants that we have an even number)
- **(R4):**  $S \rightarrow \varepsilon$

### 13.1.5 Example 5

Binary Expressions

- In this language the words are binary numbers with no leading 0's (other than 0) and with + or - operators using infix notation (between numbers, not before them)

- |                          |                       |
|--------------------------|-----------------------|
| 1. $E \rightarrow E + E$ | 5. $B \rightarrow D$  |
| 2. $E \rightarrow E - E$ | 6. $D \rightarrow 1$  |
| 3. $E \rightarrow B$     | 7. $D \rightarrow D0$ |
| 4. $B \rightarrow 0$     | 8. $D \rightarrow D1$ |

Here

- E means expression
- B means generate a 0 or D
- D means generate a number with a leading 1

Figure 13.2: Courtesy of Prof. Lanctot's slides.

## 13.2 Parse Trees

Also called a *derivation tree*. It visualizes an entire derivation at once. The tree is structured such that:

- The **internal nodes** are the non-terminals (e.g., E, B, D)
- The **root** of the tree is the start symbol (e.g., E)
- The **children** of a node are given by derivation rules
- The **leaf nodes** are the terminals and show their value (e.g., 1, 0, +1)

### Ambiguous Grammar

Because grammar can be ambiguous, we can have multiple parse trees for the same expression.

The way we'll be processing order in a parse tree is with a **post-order** with a **depth first** traversal.

### Unambiguous Grammar

To make CFG unambiguous, we must change our rules:

Change the first two productions

- |                          |                       |
|--------------------------|-----------------------|
| 1. $E \rightarrow E + E$ | 5. $B \rightarrow D$  |
| 2. $E \rightarrow E - E$ | 6. $D \rightarrow 1$  |
| 3. $E \rightarrow B$     | 7. $D \rightarrow D0$ |
| 4. $B \rightarrow 0$     | 8. $D \rightarrow D1$ |

This change forces the leftmost non-terminal to derive a binary number rather than another expression.

Generates the same words as the previous grammar but the parse tree for each derivation is unique.

Figure 13.3: Courtesy of Prof. Lanctot's slides.