Lecture 7:

- Derivations and $LL(k)$

Announcements:

- HW-1 due Mon

Context-Free Grammars

Derivations: Use grammar rules to generate strings!

- A derivation starts with a single non-terminal (e.g., S)
- Repeatedly replaces one non-terminal until only terminals remain
- Each "step" in the replacement is denoted by $\Rightarrow$

Example *step-by-step* derivation of: $S \to T \mid \varepsilon \quad T \to \mathtt{a}\, S\, \mathtt{b}$

$$S \Rightarrow T \Rightarrow \mathtt{a}\, S\, \mathtt{b} \Rightarrow \mathtt{a}\, T\, \mathtt{b} \Rightarrow \mathtt{aa}\, S\, \mathtt{bb} \Rightarrow \mathtt{aabb}$$

In this case, $S \overset{*}{\Rightarrow} \mathtt{aabb}$ ... $\overset{*}{\Rightarrow}$ means *derives* in zero or more steps

DEF: The language L of a grammar G (with start symbol S) is given as

$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$ ... i.e., all strings that can be derived from S

Context-Free Grammars

Two special types of derivations:

- **Left-most**: replace left-most non-terminal at each step
- **Right-most**: replace right-most non-terminal at each step

Example: (1) $S \to RR$ (2) $S \to \varepsilon$ (3) $R \to \mathtt{a}S\mathtt{b}$

A left-most derivation of "abab" ... $\overset{i}{\Rightarrow}$ means i-th rule applied

$$S \overset{1}{\Rightarrow} \underline{R}\,R \overset{3}{\Rightarrow} \mathtt{a}\,\underline{S}\,\mathtt{b}\,R \overset{2}{\Rightarrow} \mathtt{a}\,\mathtt{b}\,\underline{R} \overset{3}{\Rightarrow} \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\underline{S}\,\mathtt{b} \overset{2}{\Rightarrow} \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\mathtt{b}$$

A right-most derivation of "abab"

$$S \overset{1}{\Rightarrow} R\,\underline{R} \overset{3}{\Rightarrow} R\,\mathtt{a}\,\underline{S}\,\mathtt{b} \overset{2}{\Rightarrow} \underline{R}\,\mathtt{a}\,\mathtt{b} \overset{3}{\Rightarrow} \mathtt{a}\,\underline{S}\,\mathtt{b}\,\mathtt{a}\,\mathtt{b} \overset{2}{\Rightarrow} \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\mathtt{b}$$

Running Example

Simple list of assignment statements

```
<stmt_list> ::= <stmt> | <stmt> ';' <stmt_list>
<stmt> ::= <var> '=' <expr>
<var> ::= 'A' | 'B' | 'C' | ... | 'Z'
<expr> ::= <var> | <expr> '+' <expr> | <expr> '-' <expr>
```

- Note: many possible grammars for this language!

Left-most derivation of `"A = B"`:

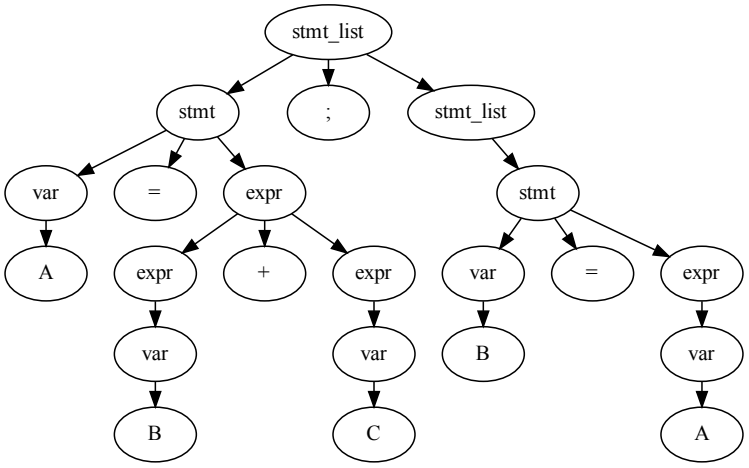
```
<stmt_list>  ⇒  <stmt>
             ⇒  <var> = <expr>
             ⇒  A = <expr>
             ⇒  A = <var>
             ⇒  A = B
```

Parse Trees

Derivations can also be written as "parse trees"

- Each non-terminal is an internal tree node
- Each rhs symbol becomes a child of the lhs node
- The root of the tree is the start symbol

Parse tree of derivation of `"A = B + C; B = A"`



$LL(k)$ Parsing

We will implement an $LL(k)$ parser

- read from left-to-right (1st L), performing a left-most derivation (2nd L)
- parses top down (parse tree built root down)
- at most k look ahead symbols (more later)
- ... some work is usually required to ensure a grammar is $LL(k)$!

Consider this statement rule: ... let <var> be a `VAR` literal

```
<stmt> ::= <var> '=' <expr>
```

Assuming the parser *knows* <stmt> is to be applied ...

(1) calls lexer's `nextToken()` and checks that it is a `VAR` token

(2) calls lexer's `nextToken()` and checks that it is an `ASSIGN` token

(3) and so on until it finishes the <stmt> rule

Produces an error if it finds a token it isn't expecting