Lecture 7:

- Formal Grammars (cont)

Announcements:

- HW-1 out
- Quiz 2 Friday – Lexical analysis, grammars
Using Parentheses: Can use parentheses to simplify rules

\[ S \rightarrow (ab)^* \mid (ba)^* \]

Check In: What is the language of this grammar rule?

Check In: How can the above be rewritten so it doesn’t use parentheses?

\[ S \rightarrow T^* \mid U^* \]
\[ T \rightarrow ab \]
\[ U \rightarrow ba \]

Note: alternation has lower precedence than other “operators”

- The rule: \( S \rightarrow a^* b \mid d^* e \)
- Is the same as: \( S \rightarrow (a^* b) \mid (d^* e) \)

Check In: What is the language of this grammar rule?

\[ S \rightarrow (a \mid b)^* \mid (d \mid e)^* \]

The language consists of the empty string, all combinations of \( a \) and \( b \), and all combinations of \( d \) and \( e \)
Recursion

Either directly when used in same rule, or indirectly ...

Direct Example: \[ S \rightarrow aSb \mid \epsilon \] ... \( S \) occurs (directly) in \( S \) rule
- \( S \) yields the strings \( a^i b^i \) for \( i \geq 0 \)
- note this is not possible to express using * (Kleene star)
- however, * can be implemented using recursion (w/ the empty string ...)

Indirect Example:
\[
\begin{align*}
S & \rightarrow T \mid \epsilon \\
T & \rightarrow aSb
\end{align*}
\]

Derivations: can help decipher language of grammars, especially with recursion
- 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 using the Indirect recursive grammar above:
\[
S \Rightarrow T \Rightarrow aSb \Rightarrow aTb \Rightarrow aaSbb \Rightarrow aabb
\]
Check In: Give a derivation of $abcd$ starting from $S$ using grammar:

\[
\begin{align*}
S &\rightarrow aTUd \\
T &\rightarrow bT | \epsilon \\
U &\rightarrow Uc | c
\end{align*}
\]

\[
S \Rightarrow aTUd \Rightarrow abTUd \Rightarrow abUd \Rightarrow abcd
\]
**MyPL Literals**

We can use grammar rules to define a PL’s literal values

Note that we use BNF below ...

- where ::= used instead of →
- and non-terminals as `<name>`

```plaintext
BOOL_VAL ::= 'true' | 'false'
INT_VAL ::= <pdigit> <digit>* | '0'
DOUBLE_VAL ::= INT_VAL '. ' <digit> <digit>*
STRING_VAL ::= "" <character>* ""
ID ::= <letter>(<letter> | <digit> | '_')*
<letter> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
<pdigit> ::= '1' | ... | '9'
<digit> ::= '0' | <pdigit>
```

... where <character> is any symbol (letter, number, etc.) except ""
**Terminology and Next Steps**

A **regular** language is one that can be defined only using:

- concatenation, alternation, and Kleene star  \( S \rightarrow a \)  
- but no recursion (except for Kleene star)

A **context free** language is one that can be defined using:

- any of the constructs (including recursion)
- but cannot have terminals on the left-hand-side of rules

A **context sensitive** language allows terminals on the left-hand side of rules

- e.g.,  \( aA \rightarrow aB \)  
  substrings  \( aA \)  replaced by  \( abB \)
  - this rule is matched only when a string has an  \( a \)  before  \( A \)
  - the initial  \( a \)  serves as context for when to apply the rule

**PL syntax is defined using context-free grammars**

- but typically not enough to prohibit all invalid programs
- which is a reason for semantic analysis
- we will talk later about additional issues in grammars (e.g., ambiguity)

**Some example syntax rules:** ... use EBNF or variants

- For Java: https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html
- For Python: https://docs.python.org/3/reference/grammar.html
- Summary of C++: https://alx71hub.github.io/hcb/
Summary – Things to Know

1. Basic rules, concatenation, alternation, kleene star

2. How to rewrite a rule to remove alternation

3. How recursion (direct, indirect) generally works with grammar rules

4. How to rewrite Kleene Star using recursion

5. Basic idea of a derivation, how to do basic derivations