Class Announcements

- Remember our course web page:
  
  https://www.cs.rutgers.edu/courses/314/classes/spring_2023_kremer/

- Piazza and canvas sites are up. Please check every other day, at least.

- First homework is due next Tuesday at 11:59pm. Late homework submissions are not accepted.

- TA office hours have been posted. Please check whether major time conflict with popular classes. All TA office hours are held in CoRE 305.
Tokens (Terminal Symbols of CFG, Words of Lang.)

- Smallest “atomic” units of syntax
- Used to build all the other constructs
- Example, Pascal:
  - keywords:  program begin if then ...
  - = * / - < > = <= >= <>
  - ( ) [ ] ; := . , ...
  - number (Example: 3.14 28 ... )
  - identifier (Example: b square addEntry ... )
Review - Regular Expressions

A syntax (notation) to specify regular languages.

<table>
<thead>
<tr>
<th>RE: r</th>
<th>Language L(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{a}</td>
</tr>
<tr>
<td>\epsilon</td>
<td>{\epsilon}</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
</tr>
<tr>
<td>rs</td>
<td>{rs \mid r \in L(r), s \in L(s)}</td>
</tr>
<tr>
<td>r^+</td>
<td>L(r) \cup L(rr) \cup L(rrr) \cup \ldots (any number of r’s concatenated)</td>
</tr>
<tr>
<td>r^*</td>
<td>{\epsilon} \cup L(r) \cup L(rr) \cup L(rrr) \cup \ldots (r^* = r^+</td>
</tr>
<tr>
<td>(s)</td>
<td>L(s)</td>
</tr>
</tbody>
</table>

(all left-assoc. in order of increasing precedence.)

⇒ Note: Inductive definition!
What do we want?

Ideally: The language/compiler designer specifies the tokens using a regular expression, and some automatic tool (scanner generator) produces code that implements the scanner.

How can this be done?

Note: In practice, there are a few more issues that we are not discussing here. For example, how to make sure that a keyword is not recognized as an identifier.
Constructing a DFA from a regular expression

regular expression (RE) \(\rightarrow\) NFA w/\(\epsilon\) moves
   - build NFA for each term
   - connect them with \(\epsilon\) moves

NFA w/\(\epsilon\) moves to NFA
   - coalesce states

NFA \(\rightarrow\) DFA
   - construct the simulation ("subset" construction)
   - minimize DFA (DFA with minimal number of states)

DFA \(\rightarrow\) regular expression
   - construct \(R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^*R_{kj}^{k-1} \cup R_{ij}^{k-1}\)

The entire process as used in compiler generation tools will be covered in 198:415 Compilers.
Converting regular expressions to NFAs

Construction of NFA based on syntactic structure of regular expression. Each intermediate nfa has exactly one final state, no edge entering start state, and no edge leaving final state.

"BASE": Build two-state automaton for atomic regular expression $a$ (single symbol or $\epsilon$) with $a$ as the edge label. One automaton $N(a)$ for each occurrence of $a$.

"INDUCTIVE STEP": Compose automata as follows:

- concatenation: $N(st)$ – given $N(s)$ and $N(t)$

- union: $N(s|t)$ – given $N(s)$ and $N(t)$

- Kleene closure: $N(s^*)$ – given $N(s)$
BNF (Backus-Naur Form): A formal notation for describing syntax—how components can be combined to form a valid program.

- To specify which programs are legal
- To describe the structure of programs (parse tree)
- BNF is a way of writing context free grammars (CFGs)
Context Free Grammars (CFGs)

- A formalism for describing languages
- CFGs are a quadruple \(< T, N, P, S >:\)
  1. A set \(T\) of terminal symbols
  2. A set \(N\) of nonterminal symbols
  3. A set \(P\) production rules
  4. A special start symbol \(S\)
- BNF is a notation for describing CFGs.

A partial example:

\[
\begin{align*}
\text{<if-stmt>} & ::= \textbf{if} \text{ <expr> then } \text{<stmt>} \\
\text{<expr>} & ::= \text{id} <= \text{id} \\
\text{<stmt>} & ::= \text{id} := \textbf{num}
\end{align*}
\]
Elements of BNF Syntax

Terminal Symbol: Symbol-In-Boldface

Non-Terminal Symbol: Symbol-In-Angle-Brackets

Production Rule:

Non-Terminal ::= Sequence of Symbols

or

Non-Terminal ::= Sequence | Sequence | ...

Alternative Symbol: |

Empty String: $\epsilon$
How a BNF Grammar Describes a Language

- A sentence is a sequence of terminal symbols (tokens)
- A language is a set of (acceptable) sentences
- The language $L(G)$ of a BNF grammar $G$ is the set of sentences generated using the grammar:
  
  - Begin with start symbol.
  - Iteratively replace non-terminals with terminals according to rules.

This is a rewrite system!
Simple Grammar ($\mathcal{G}$)

**Terminals** letters, digits, :=

**Nonterminals** <letter> <digit> <identifier> <stmt>

**Productions**

1. <letter> ::= A | B | C | ... | Z
2. <digit> ::= 0 | 1 | 2 | ... | 9
3. <identifier> ::= <letter> |
   <identifier> <letter> |
   <identifier> <letter> |
   <identifier> <digit>
4. <stmt> ::= <identifier> ::= 0

**Start Symbol** <stmt>
Derivation in a Grammar \( (G) \)

Is \( X_2 := 0 \in L(G) \), i.e., can \( X_2 := 0 \) be derived in \( G \)?

In which order to apply the rules?

Typically, there are three options:

- **leftmost** \( (\Rightarrow_L) \)
- **rightmost** \( (\Rightarrow_R) \)
- **any** \( (\Rightarrow) \)

Does it matter?
Derivation in a Grammar ($\mathcal{G}$)

Is $X_2 := 0 \in L(\mathcal{G})$, i.e., can $X_2 := 0$ be derived in $\mathcal{G}$?

<table>
<thead>
<tr>
<th>leftmost derivation</th>
<th>rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stmt&gt;</code></td>
<td>$\Rightarrow_L$</td>
</tr>
<tr>
<td><code>&lt;identifier&gt; := 0</code></td>
<td>$\Rightarrow_L$</td>
</tr>
<tr>
<td><code>&lt;identifier&gt;&lt;digit&gt; := 0</code></td>
<td>$\Rightarrow_L$</td>
</tr>
<tr>
<td><code>&lt;letter&gt;&lt;digit&gt; := 0</code></td>
<td>$\Rightarrow_L$</td>
</tr>
<tr>
<td>$X &lt;digit&gt; := 0$</td>
<td>$\Rightarrow_L$</td>
</tr>
<tr>
<td>$X_2 := 0$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rightmost derivation</th>
<th>rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stmt&gt;</code></td>
<td>$\Rightarrow_R$</td>
</tr>
<tr>
<td><code>&lt;identifier&gt; := 0</code></td>
<td>$\Rightarrow_R$</td>
</tr>
<tr>
<td><code>&lt;identifier&gt;&lt;digit&gt; := 0</code></td>
<td>$\Rightarrow_R$</td>
</tr>
<tr>
<td><code>&lt;identifier&gt; 0 := 0</code></td>
<td>$\Rightarrow_R$</td>
</tr>
<tr>
<td><code>&lt;letter&gt; 0 := 0</code></td>
<td>$\Rightarrow_R$</td>
</tr>
<tr>
<td>$X_2 := 0$</td>
<td></td>
</tr>
</tbody>
</table>
Parsing in a Grammar ($L$)

Can we recognize $x_2 := 0$ as being in $L$?

In other words, can write a program, called a parser for $L(G)$, that given an input string of terminal symbols (tokens) can construct a derivation (leftmost or rightmost) for the input using rules in $G$? If a derivation does not exist, the program should report an error.

**Note:** Different parsing techniques, i.e., the automatic recognition sentences $w \in L(G)$ will be discussed in more detail in 198:415 Compilers.

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).
Parse Trees (in $G$)

A parse tree of $X2 := 0$ in $G$:

Each internal node is a nonterminal; its children are the RHS of a production for that NT.

The parse tree demonstrates that the grammar generates the terminal string on the frontier.

Note: Different parsing techniques, i.e., the automatic recognition sentences $w \in L(G)$ will be discussed in 198:415 Compilers.
Ambiguity, top-down parsing

Things to do:

- Please check our web site every other day for announcements etc.;
- read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)