Observations:

- We have seen in the case of the palindrome generator that SRSs are well suited for generating strings with structure.
- By modifying the standard SRS just slightly we obtain a convenient framework for generating strings with desirable structure – *Grammars*

Definition: [Grammar] A grammar is a triple \((\Gamma, \rightarrow, \gamma)\) such that,

- \(\Gamma = T \cup N\) with \(T \cap N = \emptyset\), where \(T\) is a set of symbols called the *terminals* and \(N\) is a set of symbols called the *non-terminals*,\(^1\)
- \(\rightarrow\) is a set of rules of the form \(u \rightarrow v\) with \(u, v \in \Gamma^*\),
- \(\gamma\) is called the *start symbol* and \(\gamma \in N\).

---

\(^1\)The fact that \(T\) and \(N\) are non-overlapping means that there will never be confusion between terminals and non-terminals.
Grammars

Example: Grammar for arithmetic expressions. We define the grammar $(\Gamma, \rightarrow, \gamma)$ as follows:

- $\Gamma = T \cup N$ with $T = \{a, b, c, +, *, (, )\}$ and $N = \{E\}$,
- $\rightarrow$ is is defined as,

$\begin{align*}
    E & \rightarrow E + E \\
    E & \rightarrow E \ast E \\
    E & \rightarrow (E) \\
    E & \rightarrow a \\
    E & \rightarrow b \\
    E & \rightarrow c
\end{align*}$

$\gamma = E$ (clearly this satisfies $\gamma \in N$).

With grammars, derivations always start with the start symbol. Consider,

$E \Rightarrow E\ast E \Rightarrow (E)\ast E \Rightarrow (E+E)\ast E \Rightarrow (a+E)\ast E \Rightarrow (a+b)\ast E \Rightarrow (a+b)\ast c$.

Here, $(a+b)\ast c$ is a normal form often also called a \textit{terminal} or \textit{derived} \textit{string}. 
Exercise: Identify the rule that was applied at each rewrite step in the above derivation.

Exercise: Derive the string \(((a))\).

Exercise: Derive the string \(a + b \times c\). Is the derivation unique? Why? Why not?
We are now in the position to define exactly what we mean by a language.

**Definition:** [Language] Let $G = (\Gamma, \rightarrow, \gamma)$ be a grammar, then we define the *language of grammar G* as the set of all terminal strings that can be derived from the start symbol $s$ by rewriting using the rules in $\rightarrow$. Formally,

$$L(G) = \{ q \mid \gamma \Rightarrow^* q \land q \in T^* \}.$$ 

**Example:** Let $J = (\Gamma, \rightarrow, \gamma)$ be the grammar of Java, then $L(J)$ is the set of all possible Java programs. The Java language is defined as the set of all possible Java programs.
**Observations:**

- With the concept of a language we can now ask interesting questions. For example, given a grammar $G$ and some sentence $p \in T^*$, does $p$ belong to $L(G)$?
- If we let $J$ be the grammar of Java, then asking whether some string $p \in T^*$ is in $L(J)$ is equivalent to asking whether $p$ is a *syntactically correct program*.
- We can prove language membership by showing that the start symbol is equivalent to the sentence in question,

\[
\begin{align*}
  s & \equiv p \\
  * & \quad * \\
  p & \quad p
\end{align*}
\]
Observations:
- By restricting the shape of the rewrite rules in a grammar we obtain different language classes.
- The most famous set of language classes is the Chomsky Hierarchy.
Table: The Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Rules</th>
<th>Grammar</th>
<th>Language</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \rightarrow \beta$</td>
<td>Type-0</td>
<td>Recursively Enumerable</td>
<td>Turing machine</td>
</tr>
<tr>
<td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Linear-bounded Turing machine</td>
</tr>
<tr>
<td>$A \rightarrow \gamma$</td>
<td>Type-2</td>
<td>Context-free</td>
<td>Pushdown automaton</td>
</tr>
<tr>
<td>$A \rightarrow a$ and $A \rightarrow aB$</td>
<td>Type-3</td>
<td>Regular</td>
<td>Finite state automaton</td>
</tr>
</tbody>
</table>

where $\alpha, \beta, \gamma \in \Gamma^*, A, B \in \mathbb{N}, a \in T$. In Type-1 $\gamma$ is not allowed to be the empty string.
**Observation:** The most convenient language class for programming language specification are the context-free languages – they are decidable – pushdown automata can be efficiently implemented in order to prove language membership.
Example: A simple imperative language. We define grammar $G = (\Gamma, \to, \gamma)$ as follows:

- $\Gamma = T \cup N$ where
  
  $T = \{0, \ldots, 9, a, \ldots, z, \text{true}, \text{false}, \text{skip}, \text{if}, \text{then}, \text{else}, \text{while}, \text{do}, \text{end} +, -, *, =, \leq, !, \&, ||, :=, ;, (, )\}$

  and

  $N = \{A, B, C, D, L, V\}$.

- The rule set $\to$ is defined by,

  - $A \to D \mid V \mid A + A \mid A - A \mid A * A \mid (A)$
  - $B \to \text{true} \mid \text{false} \mid A = A \mid A \leq A \mid !B \mid B & B \mid B || B \mid (B)$
  - $C \to \text{skip} \mid V := A \mid C \mid \text{if } B \text{ then } C \text{ else } C \text{ end} \mid \text{while } B \text{ do } C \text{ end}$
  - $D \to L \mid - L$
  - $L \to 0 L \mid \ldots \mid 9 L \mid 0 \mid \ldots \mid 9$
  - $V \to a V \mid \ldots \mid z V \mid a \mid \ldots z$

- $\gamma = C$.

Observe that this is a context-free grammar!
Here are some elements in $L(G)$,

\begin{verbatim}
x := 1; y := x
v := 1; if v ≤ 0 then v := (−1) ∗ v else skip end
n := 5; f := 1; while 2 ≤ n do f := n ∗ f; n := n − 1 end
\end{verbatim}

**Exercise:** Prove that they belong to $L(G)$. 
Grammars

HW#1 – see website