Topic 1. BNF grammars. Backus-Naur Form (BNF) grammars are a standard formalism for defining languages. All BNF grammars comprise terminals, nonterminals (aka syntactic categories), and production rules, the general form of which is:

\[
\langle \text{nonterminal} \rangle ::= \langle \text{form 1} \rangle | \cdots | \langle \text{form n} \rangle
\]

where each form describes a particular language form— that is, a string of terminals and non-terminals. Any string of terminals and nonterminals is called a sentential form. A term in the language is a string of terminals, constructed according to these rules. Terms are constructed by derivation steps, which relate sentential forms. One form is derived from another by replacing a nonterminal in one with a form appearing on the right-hand-side of a production rule associated with the nonterminal. Usually, one distinguished nonterminal is a start symbol. If \( G \) is a grammar, then the language of \( G \) is the set of terms that can be derived from the start symbol.

example: The language SHEEP. Let \( \{ S \} \) be the set of nonterminals, with \( S \) the start state, let \( \{ a, b \} \) be the set of terminals, and let the grammar definition be:

\[
S ::= b | Sa
\]

Note that this is a recursive definition. Terms in SHEEP include:

\( b, ba, baa, baaa, baaaa, \ldots \)

Since for example, using \( \rightarrow \) to denote a derivation step:

\[
S \rightarrow Sa \rightarrow Saa \rightarrow Saaa \rightarrow baaa
\]

They do not include the following, which are not terms:

\( S, SSa, Saa, \ldots \)

example: The language FROG. Let \( \{ F, G \} \) be the set of nonterminals, \( \{ r, i, b, t \} \) be the set of terminals, and the grammar definition be:

\[
F ::= rF | iG
G ::= bG | bF | t
\]

Note that this is a mutually recursive definition. Note also that each production rule defines a syntactic category. Terms in FROG include:

\( ibit, ribbit, ribiribbit, \ldots \)
**Topic 2. BOOL, a language of boolean expressions.** BNF grammars are used to express PL syntax. Before considering the syntax of Turing complete languages, we will consider simpler algorithmic languages. Here is a simple language of boolean expressions, BOOL:

\[
\begin{align*}
v & ::= \text{True} \mid \text{False} & \quad \text{values} \\
e & ::= v \mid (e \text{ And } e) \mid (e \text{ Or } e) \mid \text{Not } e & \quad \text{expressions}
\end{align*}
\]

Terms in the language of expressions include:

\[
\text{True}, (\text{True And False}), (\text{Not} (\text{True And False}) \text{ Or } \text{True}), \ldots
\]

I may omit parentheses if association is clear from context. Note that \(e\) and \(v\) are nonterminals, though we’ve previously called them metavariables.

**Topic 3. Language semantics.** BNF grammars are sufficient to specify the syntax of languages, but not their meaning. Endowing a language with mathematical or symbolic meaning is the function of a semantics. A semantics can be thought of as a mathematically precise documentation of language behavior. A significant advantage of a formal semantics is platform independence; once the language is implemented via an interpreter or compiler, if the implementation matches the semantic specification, you can be sure that programs can be re-run on any platform without modification. The Java Virtual Machine (JVM) exploits this idea. Their are two major techniques for specifying a semantics:

- **Denotational semantics**— specifies meaning in terms of mathematical entities, e.g. the meaning of BOOL would be specified in terms of boolean algebra

- **Operational semantics**— specifies the meaning of terms in an idealization of computation.

Operational semantics is the more popular technique currently, and we will focus on it.