Review: Formal Grammars

- A formal grammar defines legal strings of a language.
- The grammars we consider have four main elements:
  - A finite set $N$ of nonterminal symbols (grammar variables)
  - A finite set $\Sigma$ of terminal symbols (language symbols)
  - A finite set $P$ of production rules
  - A single start symbol $S \in N$

$$G = (N, \Sigma, P, S)$$

The language of the grammar consists of all strings we can derive from the start symbol, following production (substitution) rules.

### Context-Free Grammars/Languages

- A more complex class of languages requires that we allow a variety of different rules, beyond the three basic types allowed by regular grammars.
- A context-free grammar allows any rule of the form
  $$A \rightarrow \gamma$$
  Where: $A$ is a single nonterminal symbol and $\gamma$ is any combination of nonterminal and terminal symbols (including the same symbol $A$ itself).
- This allows us to define far more complex languages.

### Regular Grammars/Languages

- By allowing various different forms of production rules (or not), we can capture languages of different complexity.
- A regular grammar is one in which every rule in $P$ is of one of three basic forms:
  $$A \rightarrow \epsilon$$
  $$A \rightarrow a$$
  $$A \rightarrow bB$$
  Where: $A \neq B$ are single nonterminal symbols and $a$ and $b$ are any single terminal symbols.
Context-Free Grammars/Languages

- Sometimes the use of a CFG is just for convenience
  - The original integer grammar we saw is context-free, although we don’t have to use a CFG to define it
  - Other times, we can define languages using CFG’s that we can’t define with only regular grammar rules
  - Example: binary strings of \( n \) 1’s followed by \( n \) 0’s \((n \geq 0)\), given by grammar with start state \( S \) and the rules:

\[
S \rightarrow \epsilon \mid 1S0
\]

Recall: while some languages have both regular and context-free grammars, this is not one of them. This language cannot be derived from any regular grammar, and requires a CFG.

Variants of BNF

- The basic BNF form is sufficient for any grammar we are interested in here
  - In fact, it can be simplified even further, since we don’t really need the option bars ( | )
- A rule like:
  \[
  S \rightarrow \epsilon \mid 1S0
  \]
  - Can be replaced by multiple distinct rules:

\[
S \rightarrow \epsilon \\
S \rightarrow 1S0
\]

Extended BNF (EBNF)

- In practice, it is often convenient to extend BNF further with some additional constructs:
  1. **Kleene Star**: applying the star to some symbols (using parentheses if necessary to mark them) means that they can occur zero or more times
     - For example, suppose we have the rule:

\[
S \rightarrow (01)^*
\]
  - This means that we can replace \( S \) with any of:

\[
\epsilon, \ 01, \ 0101, \ 010101, \ 01010101, \ldots
\]

Extended BNF (EBNF)

- In practice, it is often convenient to extend BNF further with some additional constructs:
  2. **Kleene Plus**: means symbols occur one or more times (also with parentheses if necessary)
     - Thus, if we have the rule:

\[
S \rightarrow (01)^+
\]
  - This means that we can replace \( S \) with any of:

\[
01, \ 0101, \ 010101, \ 01010101, \ldots
\]
Extended BNF (EBNF)

In practice, it is often convenient to extend BNF further with some additional constructs:

3. Parenthetical Options: if there are multiple possible replacements, we can group these together as sub-expressions, using parentheses and option-bars

Thus, if we have the rule:

\[ S \rightarrow (a \mid b \mid c) 1 \]

This means that we can replace \( S \) with any of:

\[ a1, b1, c1 \]

EBNF == BNF

It is important to note that these new symbols and constructs do not increase the power of our grammars.

Anything that can be expressed in EBNF can be expressed in regular BNF (and vice versa, obviously)

\[
\begin{align*}
integer & \rightarrow (+ \mid - \mid \epsilon) digit^+
nonterm \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 
\end{align*}
\]

The EBNF grammar we just saw has a BNF equivalent:

\[
\begin{align*}
integer & \rightarrow +digitSeq \mid -digitSeq \mid digitSeq
digitSeq & \rightarrow digit \mid digit digitSeq
digit & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Exercise: Derive -34 in each of the two grammars.

Extended BNF (EBNF)

With these extra resources, we can often right more complex grammars more compactly.

Consider the following grammar for positive, negative, and unsigned integers:

\[
\begin{align*}
integer & \rightarrow (+ \mid - \mid \epsilon) digit^+
digit & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Note: The first rule of this grammar allows replacing \( integer \) with one or more occurrences of \( digit \), preceded by an (optional) sign symbol.

Exercise: A Real-Number Grammar

Suppose we have the rules (with start symbol \( real \)):

\[
\begin{align*}
real & \rightarrow digit^* . digit^+ \mid digit^+ . digit^* 
digit & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Which of the following are derivable in this grammar?

1. 528
2. 0.528
3. .528
4. 528.11
5. 528.0
6. 528.
**Right- and Left-Most Derivations**

- A right-most derivation is one in which we always choose the nonterminal symbol furthest to the right for substitution.

\[
\begin{align*}
\text{integer} & \rightarrow \text{digit} \mid \text{digit integer} \\
\text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

- For example, in the basic integer grammar, we can derive \(123\) as follows:

\[
\begin{align*}
\text{integer} & \Rightarrow \text{digit integer} \\
& \Rightarrow \text{digit digit integer} \\
& \Rightarrow \text{digit digit digit} \\
& \Rightarrow \text{digit 23} \\
& \Rightarrow 123
\end{align*}
\]

**Syntactic Ambiguity**

- A grammar is ambiguous if either:
  1. There exists two distinct right-most derivations
  2. There exists two distinct left-most derivations

- For example, consider the simple grammar:

\[
\begin{align*}
\text{sub} & \rightarrow \text{sub - sub} \mid \text{a} \mid \text{b}
\end{align*}
\]

- There are two distinct right-most derivations of expression: \(a-b-a\)

\[
\begin{align*}
\text{sub} & \Rightarrow \text{sub - sub} \\
& \Rightarrow \text{sub - a} \\
& \Rightarrow \text{sub - sub - a} \\
& \Rightarrow \text{sub - b-a} \\
& \Rightarrow \text{a-b-a}
\end{align*}
\]

**Right- and Left-Most Derivations**

- A left-most derivation does the same, always on the furthest left:

\[
\begin{align*}
\text{integer} & \rightarrow \text{digit} \mid \text{digit integer} \\
\text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

- Thus, we can also derive \(123\) as follows:

\[
\begin{align*}
\text{integer} & \Rightarrow \text{digit integer} \\
& \Rightarrow \text{digit digit integer} \\
& \Rightarrow \text{digit digit digit} \\
& \Rightarrow \text{digit 23} \\
& \Rightarrow 123
\end{align*}
\]

**Syntactic Ambiguity**

- Ambiguous grammars for programming languages are problematic
  - They introduce the possibility that we will not be able to tell, when reading/writing/compiling, how to interpret code properly

- Consider the following ambiguous conditional grammar:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if cond then stmt} \\
& \rightarrow \text{if cond then stmt else stmt}
\end{align*}
\]

- How should we understand code like the following given this grammar?

\[
\begin{align*}
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\end{align*}
\]
Dangling-Else Ambiguity

\[ stmt \rightarrow if \ cond \ then \ stmt \]
\[ stmt \rightarrow if \ cond \ then \ stmt \ else \ stmt \]

Given this grammar, how do we properly derive/understand code like:

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\]

\[
\text{if } C_1 \{ \\
\quad \text{if } C_2 \{ \\
\quad \quad S_1 \} \\
\quad \text{else } \{ \\
\quad \quad S_2 \}
\}
\]

Java Dangling-Else

In Java, the code fragment:

\[
\text{int } x = 3;  \\
\text{if } (x \% 2 == 0) \\
\text{if } (x > 5) \\
\quad \text{System.out.println("Large even number");}  \\
\text{else} \\
\quad \text{System.out.println("Odd number");}
\]

Is parsed as if it had the full set of braces as follows:

\[
\text{int } x = 3;  \\
\text{if } (x \% 2 == 0) \{ \\
\quad \text{if } (x > 5) \{ \\
\quad \quad \text{System.out.println("Large even number");}  \\
\quad \} \\
\text{else} \{ \\
\quad \quad \text{System.out.println("Odd number");}  \\
\}
\]

Eliminating Ambiguities

Such ambiguities have arisen in programming languages since the days of Algol (1960's) and persist today.

Consider the following (legal) Java code; what will it print when run?

\[
\text{int } x = 3;  \\
\text{if } (x \% 2 == 0) \\
\text{if } (x > 5) \\
\quad \text{System.out.println("Large even number");}  \\
\text{else} \\
\quad \text{System.out.println("Odd number");}
\]

Answer: Nothing.

WHY?

The Java programming language, like C and C++ and many programming languages before them, **arbitrarily decrees** that an else clause belongs to the innermost if to which it might possibly belong.

Deriving and Parsing

Consider a grammar of arithmetic expressions:

\[ expr \rightarrow id \mid number \mid -expr \mid (expr) \]
\[ op \rightarrow + \mid - \mid * \mid / \]

Here, we use ‘id’ and ‘number’ as shorthand for all such possible sub-expressions.

That is, it is as if we ‘id’ and ‘number’ are actually nonterminal symbols, and we have the additional rules:

\[ id \rightarrow (a \mid b \mid \cdots \mid z)^+ \]
\[ number \rightarrow (0 \mid 1 \mid \cdots \mid 9)^+ \]

Using shorthand like this allows us to concentrate on the structurally interesting part of our expressions (the use of arithmetic operators), not the trivial parts (the structure of variable identifiers and basic integers).

Deriving and Parsing

\[ expr \rightarrow id \mid number \mid -expr \mid (expr) \]
\[ op \rightarrow + \mid - \mid * \mid / \]

Given this grammar, we can derive, e.g., the expression “x * y – z”:

\[ expr \Rightarrow expr \ op \ expr \]
\[ \Rightarrow expr \ op \ id \]
\[ \Rightarrow expr = id \]
\[ \Rightarrow expr \ op \ expr = id \]
\[ \Rightarrow expr \ op \ id = id \]
\[ \Rightarrow id \ op \ id = id \]
\[ \Rightarrow id \ op id = id \]

Note: the last line compresses multiple steps of further derivation that would be needed to go from each occurrence of ‘id’ to the final actual variable identifiers.

This Week

**Topic:** Formal Syntax

**Read:** Text, chapters 1, 2.1

**Office Hours:** Wing 210

- Wednesday/Friday: 9:00 AM – 10:30 AM
- Tuesday: 3:00 PM – 4:00 PM
- Thursday: 2:00 PM – 3:00 PM