St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 2: Context-Free Languages
Fall 2022

1 Grammars and Context-Free Languages

Recall that, in our discussion on regular languages, we introduced the notion of a regular expression. This expression essentially performed a kind of pattern matching to accept words of a certain form and reject words not of that form.

We can take this rough idea of matching patterns in words and modify it to work not just for symbols within each word, but for the structure and composition of the word itself. We are able to do so using the notion of *grammars*, which essentially provide a set of *rules* that we can follow to produce words that belong to a certain language. (Note that, while we used regular expressions to match a *given* word, we use grammars to produce a *new* word.) If a grammar produces words that belong to a certain language, then we say the grammar *generates* that language.

The idea of using grammars with languages is nothing new; linguists have been using grammars to study natural languages for centuries, dating as far back as the fourth century BCE with the work of the Indian grammarian Pāṇini. Only with the advent of computer science itself has the notion of grammars been applied to formal languages and programming languages, starting with the work of American linguist Noam Chomsky in the 1950s.

While we didn't mention it in the previous lecture, there exists the notion of a *regular grammar*, which corresponds exactly to the class of regular languages. The rules of a regular grammar take on one of the following forms:[1]

1. $A \rightarrow \epsilon$;
2. $A \rightarrow a$ for some $a \in \Sigma$; and
3. $A \rightarrow aB$ for some $a \in \Sigma$.

In each rule, the lowercase letters correspond to alphabet symbols and the uppercase letters correspond to other rules that we can use to produce more symbols of the word. In particular, though, observe how we can associate each rule with the action a finite automaton takes when reading an input word: Rule 1 corresponds to reading no symbol (e.g., while following an epsilon transition), Rule 2 corresponds to reading the last symbol of the word, and Rule 3 corresponds to reading some symbol of the word and transitioning to another state to read the next symbol.

However, this lecture isn't about regular grammars! We already have a number of ways to represent regular languages, and we know also that there exist some non-regular languages. Thus, instead of dwelling on the regular languages, here we will take our first look at a larger class: the class of *context-free languages*.

1.1 Context-Free Grammars

If you look at the specification manual for any programming language, you will likely find tucked away somewhere in the documentation a grammar for that language. This grammar, which could number into the tens of pages, describes precisely what the structure of a program written in that language should look like. Indeed, this grammar is exactly what the language compiler uses to check for syntax errors in your program!

[1] Strictly speaking, these rules define the class of *right-linear* regular grammars. A similar set of rules exists for *left-linear* regular grammars, but both types of grammars generate the regular languages.

As an example, let's consider one small excerpt from the grammar given in the third edition of the *Java Language Specification*:

```
Statement:
    Block
    assert Expression [ : Expression] ;
    if ParExpression Statement [else Statement]
    for ( ForControl ) Statement
    while ParExpression Statement
    do Statement while ParExpression ;
    try Block ( Catches | [Catches] finally Block )
    switch ParExpression { SwitchBlockStatementGroups }
    synchronized ParExpression Block
    return [Expression] ;
    throw Expression ;
    break [Identifier]
    continue [Identifier]
    ;
    StatementExpression ;
    Identifier   :   Statement
```

This part of the Java grammar checks statement blocks such as assignments, if-else blocks, for loops, and so on. All of the words written with an `Uppercase` letter or written in `CamelCase` correspond to rules, and all of the words written in `lowercase` correspond to language keywords. For example, the `if` rule on the fourth line checks that every if-else block in a program conforms to the syntax that the compiler expects: the block begins with the keyword `if` together with some parenthesized expression, followed by some statement or sequence of instructions, and ending with an optional `else` block.

This Java grammar is an example of a *context-free grammar*. Like the regular grammars we saw earlier, a context-free grammar consists of a set of rules that we can use to generate valid programs in Java. Unlike regular grammars, however, these rules take on a much more general form: for example, we can replace the `Statement` rule with any combination of keywords and other rules, as specified by that part of the grammar.

Before we look at some other examples, let's formalize the notion of a context-free grammar.

Definition 1 (Context-free grammar)**.** A context-free grammar is a tuple (V, Σ, R, S), where

- V is a finite set of elements called *nonterminal symbols*;
- Σ is a finite set of elements called *terminal symbols*, where $\Sigma \cap V = \emptyset$;
- R is a finite set of *rules*, where each rule consists of a nonterminal on the left-hand side and a combination of nonterminals and terminals on the right-hand side; and
- $S \in V$ is the *start nonterminal*.

In a context-free grammar, the set of nonterminal symbols V correspond to parts of a word that we have yet to "fill in" with terminal symbols from Σ. The set of rules R tell us how we can perform this "filling in". If we have a rule of the form $A \to \alpha$, then we can replace any instance of A in our word with the symbol α. The start nonterminal S is self-explanatory; it is the first thing in our word that we "fill in".

Returning to our Java grammar example, we can see that (for example) some of the nonterminals in the grammar include `Statement`, `Block`, `Identifier`, and `ParExpression`, while some of the terminals include `if`, `while`, `for`, and `;` (semicolon).

Importantly, we have in our definition of a context-free grammar that $\Sigma \cap V = \emptyset$; that is, the set of terminals and the set of nonterminals must be disjoint. This is to prevent the grammar from confusing terminals and nonterminals. (Incidentally, this is why the Java language designers used uppercase letters in their nonterminals and lowercase letters in their terminals.)

The sequence of rule applications we follow beginning with the start nonterminal S and ending with a completed word containing symbols from Σ is called a *derivation*. Each word of the form $(V \cup \Sigma)^*$ produced during a derivation is sometimes referred to as a *sentential form*.

For any nonterminal A and terminals u, w, and v, if we have a rule $A \to w$ in our grammar and some step of our derivation takes us from uAv to uwv, then we say that uAv *yields* uwv and we write $uAv \Rightarrow uwv$. We can represent a sequence of "yields" relations using similar notation; given words x and y, if $x = y$ or if there exists a sequence $x_1, x_2, \ldots, x_k$ where $k \geq 0$ such that

$$x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_k \Rightarrow y,$$

then we write $x \Rightarrow^* y$. (This is very similar to the Kleene star notation, where the star indicates zero or more "yields" relations taking us from x to y.)

Example 2. Consider the context-free grammar where $V = \{S, A\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and R contains two rules:

$$S \to \mathtt{a}A\mathtt{b}$$
$$A \to \mathtt{a}A\mathtt{b} \mid \epsilon$$

Using this context-free grammar, we can generate words like

$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\epsilon\,\mathtt{b} = \mathtt{ab},$$
$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\mathtt{a}A\mathtt{b}\,\mathtt{b} \Rightarrow \mathtt{aa}\,\epsilon\,\mathtt{bb} = \mathtt{aabb}, \text{ and}$$
$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\mathtt{a}A\mathtt{b}\,\mathtt{b} \Rightarrow \mathtt{aa}\,\mathtt{a}A\mathtt{b}\,\mathtt{bb} \Rightarrow \mathtt{aaa}\,\epsilon\,\mathtt{bbb} = \mathtt{aaabbb},$$

and so on. For each step, the highlighted symbols indicate which symbols were added at that step. As we can see, this context-free grammar generates all words over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ where the number of as is equal to the number of bs, there is at least one a and one b, and all as come before bs in the word.

Observe that our rule A in Example 2 included a vertical bar. This is simply a shorthand for writing multiple rules where each rule contains A on the left-hand side. Writing $A \to \mathtt{a}A\mathtt{b} \mid \epsilon$ is therefore equivalent to writing

$$A \to \mathtt{a}A\mathtt{b}$$
$$A \to \epsilon$$

There are very few limitations we must abide by when we write rules for a context-free grammar. All we need to ensure is that the left-hand side of each rule consists of exactly one nonterminal by itself. The right-hand side of each rule can contain any combination of terminals and nonterminals, including the empty word ϵ.

Example 3. Consider the context-free grammar where $V = \{S\}$, $\Sigma = \{\mathtt{(}, \mathtt{)}\}$, and R contains one rule:

$$S \to \mathtt{(}S\mathtt{)} \mid SS \mid \epsilon$$

This rule allows us to surround an occurrence of S with parentheses, to "duplicate" an occurrence of S, or to replace some occurrence of S with ϵ, effectively removing that occurrence of S from the derivation.

Using this context-free grammar, we can generate a word like

$$S \Rightarrow SS \Rightarrow \mathtt{(}S\mathtt{)}S \Rightarrow \mathtt{(}\,\mathtt{(}S\mathtt{)}\,\mathtt{)}S \Rightarrow \mathtt{((}\,\epsilon\,\mathtt{))}S \Rightarrow \mathtt{(())}S \Rightarrow \mathtt{(())}\,\mathtt{(}S\mathtt{)} \Rightarrow \mathtt{(())(}\,\epsilon\,\mathtt{)} = \mathtt{(())()}.$$

Again, the highlighted symbols indicate which symbols were added at a given step. This context-free grammar generates all words over the alphabet $\Sigma = \{\mathtt{(}, \mathtt{)}\}$ where each word contains *balanced parentheses*: every opening parenthesis is matched by a closing parenthesis, and each pair of parentheses is correctly nested.

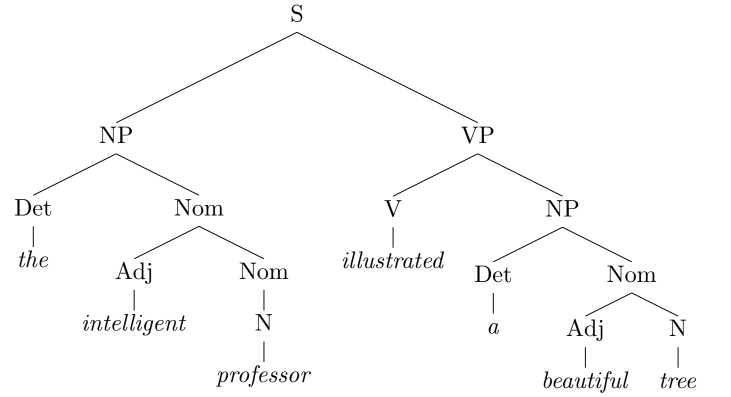
We can define the *language of a grammar G* over an alphabet Σ by $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. Thus, in Example 2, the language of the grammar is $L_{\texttt{a=b}} = \{\texttt{a}^n\texttt{b}^n \mid n \geq 1\}$. Likewise, in Example 3, the language of the grammar is $L_{\texttt{()}} = \{w \in \{\texttt{(},\texttt{)}\}^* \mid$ all prefixes of w contain no more `)`s than `(`s, and $|w|_{\texttt{(}} = |w|_{\texttt{)}}\}$.[2]

Finally, in case you're wondering why we refer to these grammars as being *context-free*: the name stems from the fact that, given a rule of the form $A \to \alpha$, we can replace any occurrence of A with α without looking at the *context* around A; that is, without looking at the symbols to the left and right of A. Thus, the grammar is free of context when we apply a given rule.

1.2 Ambiguous Context-Free Grammars

If we are given a derivation of a word for some context-free grammar, we need not always represent it in a linear fashion like we did in the previous examples. We could alternatively represent it visually as a tree structure, where the root of the tree corresponds to the starting nonterminal S and each branch of the tree adds a new nonterminal or terminal symbol. We refer to such trees as *parse trees*.

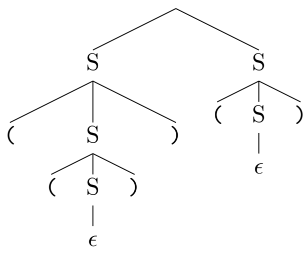
Parse trees are nothing new to linguists; the idea is used all the time to break down sentences or phrases into their constituent components, like nouns, verbs, and so on. In doing so, linguists are able to study the structures of sentences in different languages. For example, consider the following parse tree for an English sentence:



Here, the sentence (S) is broken down into a noun phrase (NP) and a verb phrase (VP); the noun phrase is broken down further into a determiner (Det) and a nominal (Nom); and so on.

Of course, just like we can use grammars with formal languages, we can use parse trees to represent the derivation of any word in the language of a grammar. In our parse trees, the root of the tree is the starting nonterminal S, the leaves of the tree contain terminal symbols from Σ (or ϵ), and all other vertices of the tree contain nonterminal symbols from V. If a parse tree contains an internal (non-leaf) vertex A, and all the children of the vertex A are labelled $a_1, a_2, \ldots, a_n$, then the underlying grammar's rule set contains a rule of the form $A \to a_1a_2 \ldots a_n$.

Example 4. Recall our derivation of the word `(())()` from the language of words with balanced parentheses in Example 3. We can represent the derivation of that word by the following parse tree:



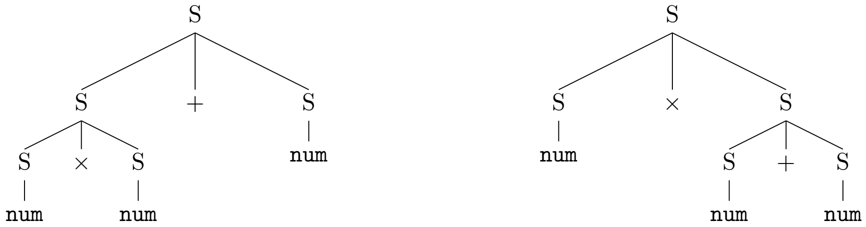
[2] The language of balanced parentheses is also known as the *Dyck language*.

For most grammars we deal with, there exists exactly one way to generate any given word in the language of the grammar. However, this is not always the case. There are some grammars wherein the same word can be generated in more than one way. Perhaps one of the most well-known examples is the grammar generating the language of arithmetic expressions. If you recall grade school mathematics, you'll remember that there is an order of operations that specify the order in which we should apply arithmetic operations in a given expression.[3] We first evaluate expressions in parentheses, then exponents, then multiplications and divisions, and finally additions and subtractions.

Let's consider a simplified set of operations, where we only use parentheses, addition, and multiplication. The grammar generating the language of arithmetic expressions using these three operators together with the standard set of numbers is as follows:

$$\begin{aligned} S &\to \texttt{(}S\texttt{)} \\ S &\to S + S \\ S &\to S \times S \\ S &\to \texttt{num} \end{aligned}$$

If we consider the expression $\texttt{num} \times \texttt{num} + \texttt{num}$, we discover that there exists more than one way to generate this expression, depending on whether we apply the + rule or the × rule first. This is evidenced by the fact that there exist two parse trees for the same expression:

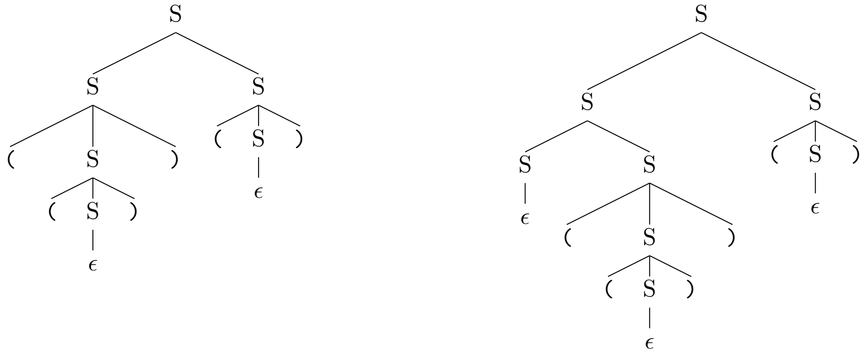


We also assume that, in both of these parse trees, we apply rules to each nonterminal from left to right; that is, at some level of the parse tree where there exists two nonterminals, we apply a rule to the first (left) nonterminal before the second (right) nonterminal. This process is known as a *leftmost derivation*.

If there exists some word in the language of a grammar for which there is more than one leftmost derivation of that word, then we say that the word is derived *ambiguously*. This notion leads us to our main definition.

Definition 5 (Ambiguous context-free grammar)**.** A context-free grammar G is ambiguous if there exists some word $w \in L(G)$ that can be derived ambiguously.

Example 6. The grammar from Example 3 generating our language of words with balanced parentheses is ambiguous. Consider again the word `(())()`. There exist two different parse trees corresponding to leftmost derivations of this word; the left parse tree was given in Example 4 and the right parse tree is new:

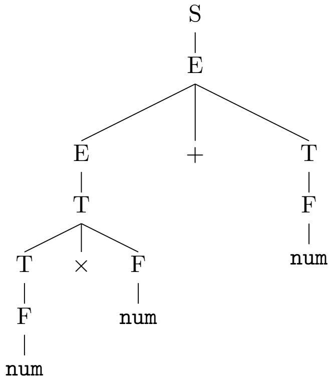


[3] Sometimes referred to using the mnemonics BEDMAS, BIDMAS, BODMAS, or PEMDAS, depending on where you went to school.

In some cases, if we have an ambiguous context-free grammar, we can create an equivalent context-free grammar with reduced or no ambiguity. For example, recalling our three-operation arithmetic grammar from earlier, we can construct an unambiguous grammar generating the same language of arithmetic expressions simply by adding a few "more structured" rules:

$$\begin{aligned} S &\to E \\ E &\to E + T \mid T \\ T &\to T \times F \mid F \\ F &\to \texttt{(}E\texttt{)} \mid \texttt{num} \end{aligned}$$

With this grammar, we're able to ensure that the addition rule is applied before the multiplication rule. This allows us to draw a single, unambiguous parse tree for the expression $\texttt{num} \times \texttt{num} + \texttt{num}$:



However, this process of reducing or removing ambiguity cannot always be done. Some context-free grammars are *inherently ambiguous*, meaning that any grammar generating the specified language has some unavoidable ambiguous component to it, and this ambiguity cannot be removed.

Example 7. Consider the language $L_{\text{twoequal}} = \{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. This language contains all words that have either the same number of as and bs or the same number of bs and cs.

We can generate this language using the following grammar:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to S_1\texttt{c} \mid A \\ A &\to \texttt{a}A\texttt{b} \mid \epsilon \\ S_2 &\to \texttt{a}S_2 \mid B \\ B &\to \texttt{b}B\texttt{c} \mid \epsilon \end{aligned}$$

The rules S_1 and A generate words of the form $\texttt{a}^n\texttt{b}^n\texttt{c}^m$ and the rules S_2 and B generate words of the form $\texttt{a}^m\texttt{b}^n\texttt{c}^n$, each where $m, n \geq 0$.

Now, consider words of the form $\texttt{a}^n\texttt{b}^n\texttt{c}^n$, where $n \geq 0$. All words of this form belong to the language L_{twoequal}, but each such word has two distinct derivations in this grammar: it can be generated either by the rules S_1 and A, or by the rules S_2 and B.

While the formal proof that this grammar is inherently ambiguous is quite long, we can intuitively see that (for instance) any grammar generating this language must have rules similar to S_1 and A to produce balanced pairs of as and bs followed by some number of cs. We can make a similar argument for the rules S_2 and B. Thus, any grammar for this language has inherent ambiguity.