ECS 120 Lesson 10 – Context-Free Grammars

Oliver Kreylos

Friday, April 20th, 2001

In the beginning of the course, we encountered the *Chomsky Hierarchy*, a framework to classify languages into four different categories, CH-0 to CH-3. Until now, we have only had a closer look at the most restrictive level of the Chomsky Hierarchy: CH-3, the class of regular languages. In the last lesson, we found out that there are languages which are not in CH-3, for example the language $\left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ or the language $\mathcal{R}(\Sigma)$ of regular expressions over the alphabet Σ. Today we will look at CH-2, the class of context-free languages (CFL), which includes all regular languages, and also the two nonregular languages mentioned above.

Context-free languages are very important in practical applications; almost every programming language has an underlying context-free structure, and a *parser*, a program to analyze the context-free structure of an input string, is a major component of any compiler for these programming languages. The parser's job is to break up a source program into its syntactical parts, e. g., modules, functions, loops, blocks and expressions, and to invoke the *code generator*, the program that converts syntactical structures into machine code, on these parts independently.

Context-free languages are specified either by *context-free grammars (CFG)* or by *pushdown automata*, a generalization of finite state machines. We will start by having a closer look at context-free grammars.

1 Definition of Context-Free Grammars

We already defined a grammar G as a four-tuple $G = (V, \Sigma, R, S)$, where V is a finite set of variables, Σ is an alphabet with $V \cap \Sigma = \emptyset$, R is a finite set of rules and $S \in V$ is the start symbol. For context-free grammars, all rules

in R are of the form $A \to w$, where $A \in V$ is a variable and $w \in (A \cup \Sigma)^*$ is a string of variables and terminals. If there are several rules with the same left-hand side, e. g., $A \to w_1, A \to w_2, \ldots, A \to w_n$, those are often combined into a single rule $A \to w_1 \mid w_2 \mid \cdots \mid w_n$. The vertical bar is treated like an "or."

If $u, v, w \in (A \cup \Sigma)^*$ are strings of variables and terminals, and $A \to w \in R$ is a rule of G, then $uAv \Rightarrow uwv$ (uAv *yields* uwv). If $u = v$, or there exists an $n \geq 0$ and a sequence $(u_1, u_2, \ldots, u_n) \in \big((A \cup \Sigma)^*\big)^n$ of strings of variables and terminals such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n \Rightarrow v$, then $u \overset{*}{\Rightarrow} v$. The language of grammar G is $L(G) := \left\{ w \in \Sigma^* \;\middle|\; S \overset{*}{\Rightarrow} w \right\}$.

2 Example: The Language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$

Consider the grammar $G = \big(\{S\}, \{\mathtt{0}, \mathtt{1}\}, R, S\big)$, where $R = \{S \to \mathtt{0}S\mathtt{1} \mid \epsilon\}$. We will now prove that this grammar exactly generates the nonregular language L.

2.1 Direction 1: $L(G) \subset L$

The only way to generate a string of only terminals from the start symbol of G is to apply the rule $S \to \mathtt{0}S\mathtt{1}$ any number of times, and then the rule $S \to \epsilon$ exactly once. We will prove by induction that applying the first rule any number of times will yield a string of the type $\mathtt{0}^n S \mathtt{1}^n$.

Claim Applying the rule $S \to \mathtt{0}S\mathtt{1}$ to the start symbol S $n \geq 0$ times yields the string $\mathtt{0}^n S \mathtt{1}^n$.

Induction Basis The start symbol is S. Applying a rule zero times means applying the rule not at all; this leaves the start symbol unchanged. $S = \epsilon S \epsilon = \mathtt{0}^0 S \mathtt{1}^0$.

Induction Hypothesis The claim is true for some $n \geq 0$.

Induction Step Let w be the string generated by applying the rule $S \to \mathtt{0}S\mathtt{1}$ to the start symbol S n times. By induction hypothesis, $w = \mathtt{0}^n S \mathtt{1}^n$. Since w contains exactly one occurrence of any variable (namely S), the only way to apply the rule one more time is to replace that occurrence of S by the string $\mathtt{0}^0 S \mathtt{1}^0$. This will yield the string $\mathtt{0}^n \mathtt{0} S \mathtt{1}\mathtt{1}^n = \mathtt{0}^{n+1} S \mathtt{1}^{n+1}$.

From this follows, that any number of applications of the first rule will only create strings still containing the variable S exactly once. The only way to create a word in $L(G)$ is, therefore, applying the second rule. After the second rule has been applied once, the derived string does not contain any more variables and must be a word in the grammar's language. Applying the second rule to the string $\mathtt{0}^n S \mathtt{1}^n$ yields the word $\mathtt{0}^n\mathtt{1}^n \in L$. Therefore, $L(G) \subset L$.

2.2 Direction 2: $L \subset L(G)$

From the above induction proof, we have seen that applying the rule $S \to \mathtt{0}S\mathtt{1}$ to the start symbol n times, and then applying the rule $S \to \epsilon$ once, yields the word $\mathtt{0}^n\mathtt{1}^n$. Since all the words in L are of this form, each of them can be generated by grammar G, which means $L \subset L(G)$. Together with the other direction $L(G) \subset L$, this is equivalent to $L = L(G)$.

3 The Language of Regular Expressions over an Alphabet Σ, $\mathcal{R}(\Sigma)$

The grammar that generates $\mathcal{R}(\Sigma)$, the language of all (fully and correctly parenthesized) regular expressions over the alphabet Σ, is given by $G_1 = (V, T, R, E)$, where

- $V = \{E\}$ is the set of variables,
- $T = \Sigma \cup \{\cup, \circ, ^*, (,), \emptyset, \epsilon\}$ is the set of terminals,
- R consists of the rules

$$\begin{aligned} E &\to (E \cup E) \mid (E \circ E) \mid (E^*) \mid \emptyset \mid \epsilon \\ E &\to a \quad \text{for every } a \in \Sigma, \text{ and} \end{aligned}$$

- $E \in V$ is the start symbol.

The structure of this grammar very closely resembles the recursive definition for regular expressions we have seen earlier. This is the main reason why context-free grammars are a very powerful tool to specify recursively defined

languages, and to relate their semantics to their syntactical structure. Writing a compiler for a programming language used to be a major undertaking and an art form; now that the theory of context-free grammars has become known to the compiler community, the situation has much improved. One of the most powerful tools for compiler architects is the yacc utility ("Yet Another Compiler Compiler"), a program that takes the description of a context-free grammar and converts it into a `C` program that parses strings from the grammar's language and executes user-supplied program code creating its semantics. Using yacc, a compiler for a simple programming language can literally be written in a matter of hours.

4 Parse Trees

Parse trees are a way to visualize the derivation of a word in the language of a grammar G. A parse tree does not only show the word resulting from a derivation, but it also shows all performed substitution steps and the order in which they were done. Formally, parse trees are defined recursively:

Base Case If $a \in \Sigma \cup \{\epsilon\}$ is a character or the empty word, then the single *leaf node* labeled a is a *partial parse tree* with root a.

Inductive Case If $A \in V$ is a variable, $A \to w \in R$ is a rule with right-hand side $w = w_1 w_2 \dots w_n \in (\Sigma \cup V)^n$, and $t_1, t_2, \dots, t_n$ are n partial parse trees with roots $w_1, w_2, \dots, w_n$, respectively, then the graph obtained by connecting a new *interior node* labeled A to the root of every tree t_i is a partial parse tree with root A. In the layout of the graph, the trees t_i are ordered from left to right: For all $1 \leq i < j \leq n$, all nodes of tree t_i are on the left of all nodes of tree t_j. A partial parse tree is a *parse tree* if and only if its root node is labelled with the start symbol S.

The leaves of a parse tree, read from left to right, spell out the word $w \in L(G)$ that is generated by the parse tree. The graph is usually laid out such that all leaves are aligned on a single line to make the generated word easily readable, see Figure 1.

Besides visualizing the derivation of a word, parse trees also have another, more important function: In all applications, a grammar does not only define the *syntax*, i. e., the structure for all words of a language, but also the

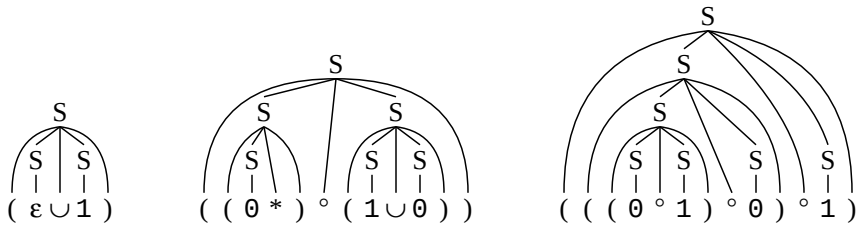

Figure 1: Example parse trees for words in the language $\mathcal{R}(\{0, 1\})$, as defined by grammar G_1.

semantics, i. e., the meaning of all words of the language. For example, the grammar of regular expressions does not only say that, if R_1 and R_2 are regular expressions, so is $(R_1 \cup R_2)$, but it also define the meaning of that word: The language defined by $(R_1 \cup R_2)$ is the union of the languages defined by R_1 and R_2. Every substitution has a certain *action* associated with it, that is executed in reverse order of derivation when the meaning of a word is computed. Using parse trees, the actions are associated with interior nodes of the parse tree. In the context of programming language compilers, the parser's job is to build the parse tree, and the code generator's job is to traverse the parse tree bottom-up and emit code fragments for the derivation symbolized by each node. The yacc utility mentioned earlier creates code that combines the building and executing of the parse tree into one block; the full parse tree is never actually created. Another parser generator, Antlr, explicitly creates the parse tree and passes it to a user-supplied program to execute it.

5 $\mathcal{R}(\Sigma)$ Revisited

The above grammar G_1 has one serious flaw – it only describes fully parenthesized regular expressions. We have already seen that those are almost unreadable, and hence almost unusable. To fix this flaw, we introduced precedence rules that make many of the parentheses superfluous. Here is another grammar that generates regular expressions with arbitrary parenthezation (but still correct nesting of parentheses): $G_2 = (V, T, R, E)$, where

- $V = \{E\}$ is the set of variables,
- $T = \Sigma \cup \{\cup, ^*, (,), \emptyset, \epsilon\}$ is the set of terminals,

- R consists of the rules

$$\begin{aligned} E &\rightarrow E \cup E \mid EE \mid E^* \mid (E) \mid \emptyset \mid \epsilon \\ E &\rightarrow a \quad \text{for every } a \in \Sigma, \text{ and} \end{aligned}$$

- $E \in V$ is the start symbol.

This grammar can now generate regular expressions like `abaa` or $\mathtt{a} \cup \mathtt{ba}$ or (((((a))))), if one were so inclined.

6 Ambiguity

The grammar G_2 still has one problem that must be fixed before it can be used. Consider, for example, the word $\mathtt{a} \cup \mathtt{ba}$. It has two valid parse trees in G_2 which have different structures, see Figure 2. These two different trees correspond to the two different orders in which the grammar can apply the substitution rules $E \rightarrow E \cup E$ and $E \rightarrow EE$: The left parse tree in Fig. 2 corresponds to applying the rule $E \rightarrow EE$ first, whereas the right parse tree corresponds to applying $E \rightarrow E \cup E$ first. Parsing the word in the first way will yield the regular expression $\big((\mathtt{a} \cup \mathtt{b})\mathtt{a}\big)$, whereas the second way will yield the regular expression $\big(\mathtt{a} \cup (\mathtt{ba})\big)$. We know that these two expressions have different meanings, i. e., they generate different languages. A grammar that has different parse trees for one input word is called *ambiguous*. To be able to use a grammar, it has to be unambiguous – a grammar for `C` programs would be useless if there were two different interpretations for the meaning of a syntactically correct program.

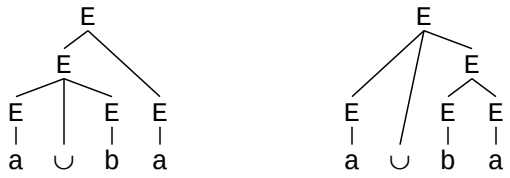

Figure 2: Two different parse trees for the word $\mathtt{a} \cup \mathtt{ba}$ in the ambiguous grammar G_2. The left parse tree interprets the expression as $\big((\mathtt{a} \cup \mathtt{b})\mathtt{a}\big)$; the second one interprets it as $\big(\mathtt{a} \cup (\mathtt{ba})\big)$.

To formalize our understanding of ambiguity, we have to look at the order of derivations instead of informally arguing about the structure of parse trees.

An important detail is, that sometimes derivations can be applied in different orders, but the resulting parse trees will still have the same structure, and the grammar will not be considered ambiguous. Consider a derivation that, at some point, creates a string $s = uAvBw$, where $A, B \in V$ are variables and $u, v, w \in (\Sigma \cup V)^*$ are strings of variables and characters. If the grammar contains rules $A \to a$ and $B \to b$, then these two rules can be applied in any order without changing the structure of the resulting parse tree. The reason is, that the two nodes corresponding with the occurrences of A and B in s are in different branches of the parse tree. Changing the order of two substitutions only changes the structure if one of the two nodes is a part of the other node's subtree, as in Figure 2.

6.1 Leftmost Derivations

By this observation, we can now formalize the notion of ambiguity by imposing a specific order on independent substitutions. Let $w = w_1 w_2 \dots w_n \in (\Sigma \cup V)^n$ be a string of characters and variables, and let $1 \leq i \leq n$ be a number such that $w_i \in V$ and for all $j < i : w_j \in \Sigma$. In other words, w_i is a variable, and all symbols to the left of w_i are characters. If $w_i \to v \in R$ is a rule, the derivation $w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n \Rightarrow w_1 \dots w_{i-1} v w_{i+1} \dots w_n$ is called a *leftmost derivation*. A derivation $u \overset{*}{\Rightarrow} v$ is called a leftmost derivation, if all the derivation steps are leftmost derivations. With this definition, a grammar is ambiguous if and only if there is a word that has two different leftmost derivations from the start symbol.

For example, here are the two leftmost derivations for the string $\mathtt{a} \cup \mathtt{ba}$ in grammar G_2: $E \Rightarrow EE \Rightarrow E \cup EE \Rightarrow \mathtt{a} \cup EE \Rightarrow \mathtt{a} \cup \mathtt{b}E \Rightarrow \mathtt{a} \cup \mathtt{ba}$ and $E \Rightarrow E \cup E \Rightarrow \mathtt{a} \cup E \Rightarrow \mathtt{a} \cup EE \Rightarrow \mathtt{a} \cup \mathtt{b}E \Rightarrow \mathtt{a} \cup \mathtt{ba}$.

6.2 Inherently Ambiguous Languages

In practice, many languages that are described by ambiguous grammars can also be described by unambiguous grammars. We will see how to create an unambiguous grammar for the language of regular expressions in the next section. Some languages, however, can not be described by an unambiguous grammar. These languages are called *inherently ambiguous*, and they have little use in computer science, where the meaning of words of a language should be unique. Incidentally, most natural languages have the nasty property of being inherently ambiguous – one reason why natural-language

processing programs are not yet very sophisticated. English, for example, is inherently ambiguous: The sentence "The boy touches the girl with the flower" has two different meanings; which meaning is intended cannot be derived from the sentence alone, but only by taking additional knowledge from context into account.

7 $\mathcal{R}(\Sigma)$ Revisited – Again

The third grammar generating $\mathcal{R}(\Sigma)$ will be unambiguous. It will unambiguously specify regular expressions using the precedence rule discussed earlier, and it will still be able to generate expressions having any number of superfluous parentheses. It is given by $G_3 = (V, T, R, U)$, where

- $V = \{U, C, S, B\}$ is the set of variables,
- $T = \Sigma \cup \{\cup, ^*, (,), \emptyset, \epsilon\}$ is the set of terminals,
- R consists of the rules

$$\begin{array}{lcl} U & \rightarrow & U \cup C \mid C \\ C & \rightarrow & SC \mid S \\ S & \rightarrow & S^* \mid (U) \mid B \\ B & \rightarrow & \emptyset \mid \epsilon \\ B & \rightarrow & a \quad \text{for every } a \in \Sigma, \text{ and} \end{array}$$

- $U \in V$ is the start symbol.

Incidentally, these rules not only enforce the precedence rules for regular operators, but they also enforce an *associativity* for the union and concatenation operators. The rule $U \rightarrow U \cup C$ is *left-associative*, i. e., an input word like $\mathtt{a} \cup \mathtt{b} \cup \mathtt{c}$ is implicitly parenthesized as $((\mathtt{a} \cup \mathtt{b}) \cup \mathtt{c})$, whereas the rule $C \rightarrow SC$ is *right-associative*, i. e., an input word like $\mathtt{abc}$ is implicitly parenthesized as $(\mathtt{a}(\mathtt{bc}))$. Since both these operators are associative, the order of binding does not matter[1]; for the arithmetic subtraction operator, however, $((10 - 4) - 3)$ is different from $(10 - (4 - 3))$.

[1]The associativities were specified differently only to show that it can be done; a motivation that should normally be frowned upon.

It is obvious that this grammar is not as closely related to the definition of regular expressions as the other two, and in fact designing an unambiguous grammar for a given language is not easy. Luckily, there is an algorithm to convert a grammar like the second one into an unambiguous one by assigning explicit precedence levels to all "operators" in it (in the grammars above, operators are the symbols $\cup$, the "invisible" concatenation operator $\circ$ and the Kleene Star *). The yacc utility implements this algorithm; the user can specify precedence levels (and associativity direction) for all operators, and yacc will clean up the specified grammar automatically.