Takeaway Notes: Context-Free Grammars

Contents

1 Introduction

With these notes, we finally cross over to a new language realm: *context-free languages*. Context-free languages represent the next tier past regular languages, and are arguably the most-applied language category to date.

2 Key Concepts

2.1 Context-Free Grammars (CFGs)

Our journey through regular languages started with machines (deterministic and non-deterministic), moving on to regular expressions, then ending with right-linear grammars (RLGs). For context-free languages, we now travel in reverse: we start with context-free grammars (CFGs).

As grammars, CFGs also have the same "production"-oriented perspective on sets of strings that we saw with RLGs. And in fact their formal definition is virtually identical to RLGs, differing only in the form that its production rules may take.

Thus, a context-free grammar G is also a tuple $G = (V, T, P, S)$ where:

- V is the non-terminal or variable alphabet of the grammar (just like before)
- T, swapping in for Σ, is more mnemonically the terminal ($t$erminal, get it?) alphabet of the grammar
- $V \cap T = \emptyset$
- $S \in V$ is the *start symbol*
- P is the set of *productions* or *rules*
- A production has the form $X \to \alpha$, where $X \in V$ and α is a string consisting of symbols from $V \cup T$. Therein lies the only difference with RLGs: CFGs can have more than one variable symbol in their productions, and those variable symbols can appear *anywhere* within α.

Just as with RLGs, one can produce or generate a string within a language by starting at a rule with S on the left, then substituting S with the expression on the right side of the rule. Every non-terminal in the new string can then be substituted with any rule that has the non-terminal on the left. Keep going until there are no more non-terminals in the string. Voila, you should now have a string that belongs to the grammar's corresponding context-free language.

2.1.1 Grammar Shorthand

Following the definition of production set P strictly can lead to a bunch of rules with the same variable on the left, such as:

$$S \to aS$$

$$S \to Sb$$

$$S \to ba$$

As a form of shorthand, such rules can be written as one, with the pipe or vertical bar ("—") separating the various right sides:

$$S \to aS \mid Sb \mid ba$$

Note how this represents the exact same set of productions; they just take advantage of the common left side to produce a more compact representation.

2.1.2 Grammars for Some Familiar Non-Regular Languages

The idea of a context-free grammar instantly makes certain languages that we couldn't touch in regular-land suddenly accessible or definable. For compactness we use the vertical bar shorthand:

$$S \to aSb \mid \epsilon \tag{1}$$

(1) is a language we have seen before: $\{w \mid w \text{ is of the form } a^n b^n\}$.

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon \tag{2}$$

And (2) is $\{w \mid w \text{ is a palindrome}\}$.

2.1.3 Useful Grammars for Thinking About Later Concepts

This section has two more grammars that will be useful for thinking about later concepts, and can also be viewed as being somewhat "practical" (you'll see once you get what they are). For these, our non-terminal alphabet expands quite a bit: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *\}$ for (3) and $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +, *, (,)\}$ for (4).

$$S \to S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{3}$$

$$S \to (S + S) \mid (S * S) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{4}$$

Do you recognize them? Build a few strings from them and see what you get.

2.2 Derivations

A *derivation* is a specific sequence of productions that result in a particular string. For example, for grammar (3), the string $3 + 5 * 2$ can be *derived* as follows:

$$S \to S + S \to 3 + S \to 3 + S * S \to 3 + 5 * S \to 3 + 5 * 2$$

It can also be derived in this way:

$$S \to S + S \to S + S * S \to S + S * 2 \to S + 5 * 2 \to 3 + 5 * 2$$

The difference between the two approaches is that in the first derivation, we apply a production on the leftmost non-terminal first. In the second one, we apply a production on the rightmost non-terminal first. Those approaches are called *left*- and *right*-derivations, respectively.

Going left- vs. right- isn't the only way to derive the same string in a different way. Depending on the grammar, derivations may also choose different *rules* to reach the same string, whether the leftmost or rightmost non-terminal is changed first. Here are two more derivations of $3 + 5 * 2$:

$$S \to S * S \to S + S * S \to 3 + S * S \to 3 + 5 * S \to 3 + 5 * 2$$

And:

$$S \to S * S \to S * 2 \to S + S * 2 \to S + 5 * 2 \to 3 + 5 * 2$$

We have thus specified four derivations for the same string $3 + 5 * 2$ using grammar (3). Two of them are left-derivations and two of them are right-derivations. Two of them use $S \to S + S$ as the first applied rule and two of them use $S \to S * S$ first.

2.3 Parse Trees

A *parse tree* is a way to visualize a derivation: instead of a sequence as seen in the previous section, each application of a production rule creates a subtree with the left-side non-terminal of the applied rule as the parent node and the right-side symbols (terminal or not) as child nodes, written from left to right in order of concatenation.

Parse trees take us full-circle back to the string whose derivation is represented by having that string be its *yield*—i.e., the concatenation of the tree's leaf nodes, going from left to right.

2.4 Ambiguity

A grammar is said to be *ambiguous* when more than one parse tree exists for the same string. Grammar (3) is ambiguous because the string $3 + 5 * 2$ has more than one parse tree: the one where $S \to S + S$ is the first applied rule, and the one where $S \to S * S$ is the first applied rule.

Compare this to grammar (4). In that grammar, $3 + 5 * 2$ is not part of the grammar's language. Instead, the string must be either $(3 + 5) * 2$ or $3 + (5 * 2)$. Every derivation of either string produces the same parse tree. If this can be proven to be true of *all* strings in grammar (4)'s language, then we can say that grammar (4) is not ambiguous.

3 Important Skills

For this portion of the course, make sure you can do the following:

- Derive strings from a given grammar.
- Given a string, determine whether it can be derived from a given grammar.
- Given a derivation, draw the parse tree for that derivation.
- Given a parse tree, write out a derivation corresponding to that parse tree.
- Determine whether derivations result in the same or different parse tree.

3.1 Proofs About Grammars and Languages

One major skill that will likely not stick in the long run, but should at least leave some kind of lasting impression, is the task of proving whether a language L and a grammar G are equivalent: that is, whether every string in L can be generated by G, and whether every string generated by G belongs to L.

The good news is that general strategy for such proofs is pretty uniform; that is what we'll cover here. The details beyond that are of course dependent on the specific language and grammar whose equivalence is being proven (or disproven).

Observe that the equivalency of L and G is an *if and only if* proposition. In one direction, we need to show that if a string is in L, then it can be generated by G. In the other direction, we then need to show that if a string can be generated by G, then it is in L.

So right there our proof will have two parts: prove the "if," then prove the "only if." Each part, in turn, has the same strategy: proof by induction.

The key to any induction proof is to associate cases with some kind of size, indicated by $n \in \mathbb{N}$. We then prove our statement for the *base case* of some minimum size, typically $n = 0$ or $n = 1$. Once the base case is proven, we are allowed to assume that our statement is true for any size n. To conclude the induction, we must then prove that our statement is true for cases of size $n + 1$.

The two parts of a grammar-language equivalency proof are proofs by induction based on different n's: L to G is proof by induction where n is the length of the string, and G to L is proof by induction where n is the number of productions applied.

To show that every string in L can be generated by G, first show that the shortest string(s) in L can all be generated by G. Having done this, you may now assume that all strings of length n or less in L can be generated by G. Finally, finish up by showing that all strings of length $n + 1$ in L can also be generated by G.

To show that every string generated by G belongs to L, first show that all strings generated by applying a single production rule in G (any rule with the start symbol on the left, that is) are in L. Note that *sentential forms*—strings that still have variable or non-terminal symbols in them—do not count here. We are only looking at strings consisting solely of terminal symbols.

Having proven this base case, we may now assume that all terminal strings resulting from n or fewer productions are in L. We finish up completely by proving that applying one more production ($n + 1$ case) results in a string that is also in L.

4 Real-World Notes

Of all topics in the theory of computation, context-free grammars and languages perhaps have the most tangible real-world impact. The fact alone that they form the core of all programming languages to date speaks to their significance and the potential of theoretical concepts when applied well.

4.1 Alternate Notations

The "math-y" notation of context-free grammars did not mesh well with early plain-text input and output subsystems. The production rule arrow alone would be tough to type out even today. As a result, many alternate notations emerged for CFGs. They are all conceptually the same, but use symbols that are easier to generate from a standard keyboard.

Perhaps the best known of these is Backus-Naur Form (BNF). This is easy to look up on the web, but as examples of how BNF makes CFGs more "typeable," the production arrow $\rightarrow$ is replaced by two colons and an equal sign (`::=`) and multi-letter variables are indicated as such with angle brackets (e.g., `<program>`). Other notations exist as well, but what's important is that the core concept of a context-free grammar and language remains the same throughout.

4.2 Grammars for Grammars

The very concept of different notations for grammars ties into the insight that a grammar can be defined for expressing a context-free grammar. That is:

- the notation for a context-free grammar constitutes a language in its own right,
- this language is context-free, and
- thus *a context-free grammar can be used to define any context-free grammar*

This leads to the notion of "compiler compilers" or "parser parsers:" programs that accept a context-free grammar as input and generate a parser as output. This level of automation revolutionized our ability to design and implement all kinds of computer languages, whether for programming, data representation, or other uses.

The program `yacc` ("Yet Another Compiler Compiler") is perhaps the best known of these, but there are many others, including `javacc` and Ohm. These tools make the step of *parsing* a language virtually effortless, allowing language implementers to focus on tougher issues such as semantic analysis, code generation, and optimization.

4.3 Ambiguity in the Real World

The theoretical concept of ambiguity—very precisely defined as the existence of more than one parse tree for the same string in a grammar's language—has very potent real-world implications, influencing ideas like precedence and associativity in expressions as well as whether delimiters like { } are required. The classic notion of "where does the `else` belong" in statements like:

```
if expr1 then if expr2 then
    stmt1
else
    stmt2
```

. . . can be traced back to ambiguity in a language's grammar. Sometimes these issues can be fixed at the grammar level; other times these are addressed as heuristics outside of parsing. In all cases, having an understanding of ambiguity at the theoretical level helps to inform these issues in practice.

5 Big Picture Points

You likely won't retain all of the skills in the list above for the long term, unless of course you end up writing parsers, compilers, or doing theoretical computer science as part of your career. Even without them, do remember or appreciate the following:

- Context-free languages are the next language category "up" from regular languages.
- Context-free languages "emerge" from regular languages by allowing a grammar to have multiple non-terminals on the right side, at any location.
- A context-free grammar becomes *context-sensitive* when the *left* side of its production rules is allowed to be any string of terminals or non-terminals.
- Today's programming languages are context-free.