Beyond Regular Languages

1 Context-free Languages

In the previous weeks, we have seen two different yet equivalent methods of describing regular languages i.e. automata and regular expressions. We have also seen other examples such as matching brackets, and understood that these are not regular. In this document, we present *context-free grammars*, a more powerful method of describing languages. The set of languages that can be generated using context-free grammars are known as *context-free languages*.

[Note: parts of this handout are adapted from Sipser, which you are encouraged to read.]

For the introductory video, see Context-free Grammars on Canvas.

1.1 Context-free grammars

We begin with an example. The alphabet is

$$\Sigma = \{+, \times, (,), 3, 5, \text{if}, \text{then}, \text{else}, \text{and}, >\}$$

(Perhaps we should call the elements of Σ "tokens" rather than "characters".) Our language L is going to be the set of all integer expressions. For example

$$\text{if } 3 > (3+5) \text{ then } 5 \text{ else } 3$$

is in L, but $3 > (3+5)$ is not. Here is a *context-free grammar* describing L.

$$\begin{array}{rcl}
A & ::= & 3 \\
A & ::= & 5 \\
A & ::= & A + A \\
A & ::= & A \times A \\
A & ::= & (A) \\
A & ::= & \text{if } B \text{ then } A \text{ else } A \\
B & ::= & A > A \\
B & ::= & B \text{ and } B \\
B & ::= & (B) \\
\text{Start:} & & A
\end{array}$$

We can write this in a more concise form:

$$\begin{array}{rcl}
\Rightarrow A & ::= & 3 \mid 5 \mid A + A \mid A \times A \mid (A) \mid \text{if } B \text{ then } A \text{ else } A \\
B & ::= & A > A \mid B \text{ and } B \mid (B)
\end{array}$$

which is called BNF (Backus-Naur Form). A and B are called *nonterminals* while the characters in Σ are called *terminals*. Each line with $::=$ is called a *production*, and it tells us how to replace a nonterminal with a string of terminals and nonterminals.

Formally, a context-free grammar is a 4-tuple (V, Σ, R, S), where:

1. V is a finite set called the **variables** or **non-terminals**,
2. Σ is a finite set, disjoint from V, called the **terminals**,
3. R is a finite set of **rules** or **productions**, with each rule consisting of a variable and a string of variables and terminals, and
4. S $\in$ V is the start variable.

Here is a derivation of the above term.

$$\begin{array}{rcl} A & \rightsquigarrow & \text{if } B \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } B \text{ then } 3 \text{ else } A \\ & \rightsquigarrow & \text{if } B \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } A{>}A \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } A > (A) \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } 3 > (A) \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } 3 > (A + A) \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } 3 > (A + 5) \text{ then } 3 \text{ else } 5 \\ & \rightsquigarrow & \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5 \end{array}$$

Note the principles:

- We begin with the Start nonterminal.
- At each step we replace a nonterminal by a string of terminals and nonterminals according to one of the productions in the grammar.
- At the end, we have the desired word in Σ^*.

The grammar is called "context free" because you can apply a production to any nonterminal regardless of the other symbols in the string. The set of strings that can be produced or generated from a grammar is known as the **language of this grammar**, and can be written as $\mathrm{L}(G)$. A language produced by a context-free grammar is known as **Context-free Language**(CFL).

1.2 Leftmost and Rightmost Derivations

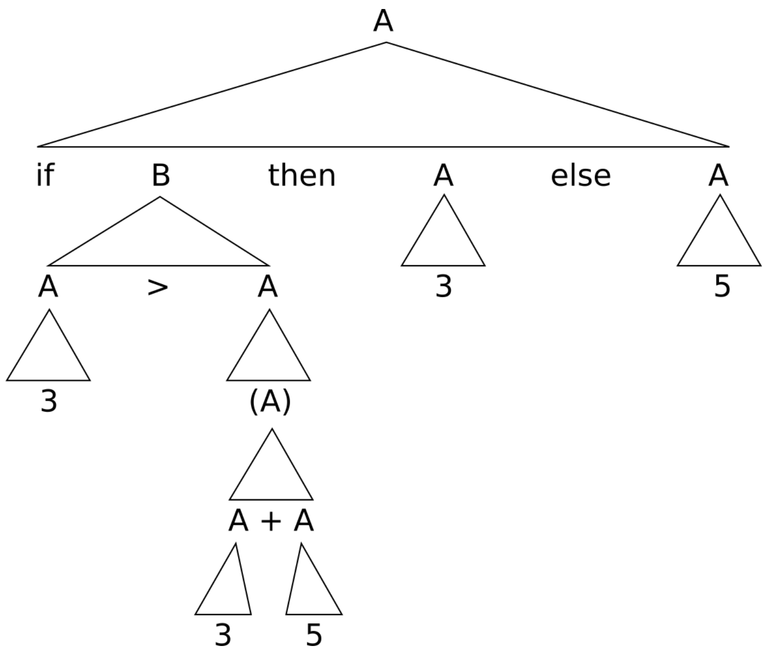
See also the video lecture recording: Leftmost and Rightmost Derivations on Canvas.
The above derivation jumps all over the word. The following performs the same replacements, but it is a *leftmost* derivation, meaning that at each step the nonterminal replaced is the leftmost one.

$$\begin{array}{rcl} A & \rightsquigarrow & \text{if } B \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } A > A \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > A \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (A) \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (A + A) \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (3 + A) \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (3 + 5) \text{ then } A \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } A \\ & \rightsquigarrow & \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5 \end{array}$$

Similarly, we can have a *rightmost* derivation of the above, meaning that at each step the non-terminal replaced is the rightmost one.

$$\begin{array}{rcl} A & \Rightarrow & \text{if } B \text{ then } A \text{ else } \underline{A} \\ & \Rightarrow & \text{if } B \text{ then } \underline{A} \text{ else } 5 \\ & \Rightarrow & \text{if } \underline{B} \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } A > \underline{A} \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } A > (\underline{A}) \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } A > (A + \underline{A}) \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } A > (\underline{A} + 5) \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } \underline{A} > (3 + 5) \text{ then } 3 \text{ else } 5 \\ & \Rightarrow & \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5 \end{array}$$

Each of these derivations can be summarized by the following *derivation tree*.



A derivation tree is also called a *parse tree*. You may see it written with the root at the bottom, or with several edges instead of a triangle.

Note the principles:

- At the root we place the Start nonterminal.
- Each triangle has a nonterminal above and a string of terminals and nonterminals below, following one of the productions in the grammar.
- At each leaf, we have a nonterminal.

The desired word appears by reading the leaves from left to right.

Let's look at a "Natural Language" example. The alphabet is

{ the, a, cat, dog, happy, tired, slept, died, ate, dinner, and, . }

The grammar is

Sentence	⇒ S	::=	C.
Clause	C	::=	NP VP | C and C
Noun phrase	NP	::=	Art N | dinner
Noun	N	::=	Adj N | cat | dog
Adjective	Adj	::=	happy | tired
Verb phrase	VP	::=	VI | VT NP
Intransitive verb	VI	::=	slept | died
Transitive verb	VT	::=	ate
Article	Art	::=	a | the

This grammar accepts "words" such as

the happy tired happy dog died and the cat slept.
the tired tired cat ate dinner.
dinner ate a happy dog.

Try writing derivations and derivation trees for these sentences.

2 The matching problem for a context free language

Given a context free grammar, is the matching problem decidable? In other words, is there some program

```
boolean f (string w) {
  ...
}
```

that, when given a word w over our alphabet, returns True if w is derivable and False otherwise? The answer is Yes; the CYK algorithm (which we shall not learn) is a way of doing this. But, for some grammars, it is not efficient (cubic complexity).

Happily, for certain kinds of grammar, there are efficient ways of solving this problem. When people design a grammar for a programming language, they try to design it to fit one of these special kinds.

A program that constructs a derivation tree for a given word (if possible) is called a *parser*. Tools such as Yacc and Antlr are called *parser generators*; you supply a grammar (which must be of the right kind) and the tool will produce an efficient parser. You'll learn more about parsing when you study Compilers.

3 Designing Context-free Grammars

See also the video lecture recording: Designing Context-free Grammars on Canvas.

The following example (taken from Sipser), showcases that in order to get the grammar for the language $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$, we first construct the grammar:

$$\Rightarrow S_1 ::= \quad 0S_11|\varepsilon$$

for the language $\{0^n1^n | n \geq 0\}$ and the grammar

$$\Rightarrow S_2 ::= \quad 1S_20|\varepsilon$$

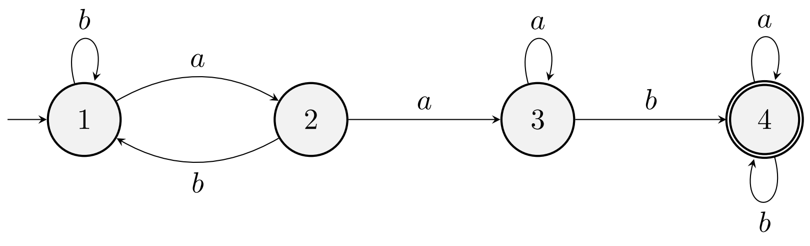
for the language $\{1^n0^n | n \geq 0\}$ and then add the rule $S ::= \quad S_1|S_2$ to give the grammar:

$$\begin{array}{rcl} \Rightarrow S & ::= & S_1|S_2 \\ S_1 & ::= & 0S_11|\varepsilon \\ S_2 & ::= & 1S_20|\varepsilon \\ \text{Start:} & & S \end{array}$$

For regular languages, the task of constructing equivalent CFG is relatively easy. We can construct the DFA for the given language and then convert the DFA into an equivalent CFG as follows:

1. Create a variable R_i for each state q_i of the DFA.
2. Add the rule $R_i ::= \quad aR_j$ to the CFG, if $\delta(q_i, a) = q_j$ is a transition in the DFA.
3. Add the rule $R_i ::= \quad \varepsilon$, if q_i is an accepting state of the DFA.
4. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

You can verify the resultant CFG and the fact that it generates the same language as the given DFA quite easily. Let's consider a DFA that accepts any string that contains the substring "aab".



1. We can make the variables R_1, R_2, R_3 and R_4, corresponding to the states of above DFA.
2. We add the rules using the pattern $R_i ::= \quad aR_j$ to the CFG, for each transition in the DFA.
3. Add the rule $R_4 ::= \quad \varepsilon$, as state 4 is an accepting state of the DFA.
4. Make R_1 the start variable of the grammar.

We get the following grammar after applying the above steps:

$$\begin{aligned} \Rightarrow R_1 &::= & aR_2 \mid bR_1 \\ R_2 &::= & aR_3 \mid bR_1 \\ R_3 &::= & aR_3 \mid bR_4 \\ R_4 &::= & aR_4 \mid bR_4 \mid \varepsilon \end{aligned}$$

We will let you verify the above CFG generates the same language that the DFA recognizes.

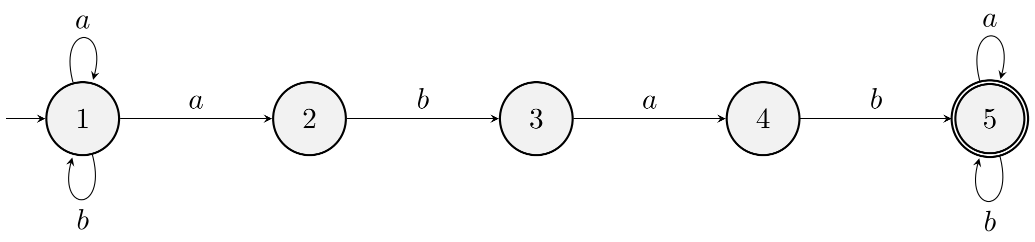
Let us given another grammar for arithmetic expressions. In this example, $G = (V, \Sigma, R, E)$, with $V = \{E, T, F\}$ and Σ is $\{3, 5, +, \times, (,)\}$ is shown below.

$$\begin{aligned} \Rightarrow E &::= & E + T \mid T \\ T &::= & T \times F \mid F \\ F &::= & (E) \mid 3 \mid 5 \end{aligned}$$

In the above example, any time the symbol E appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear *i.e.* the $F ::= (E)$ production.

3.1 Test Your Understanding

1. Let's consider an NFA that accepts any string that contains the substring "abab".



   (a) Convert the above NFA into its equivalent total DFA.
   (b) Convert the resultant DFA in an equivalent CFG.

2. Give a context free grammar for the set of palindromes over the alphabet $\{\mathtt{a}, \mathtt{b}\}$.

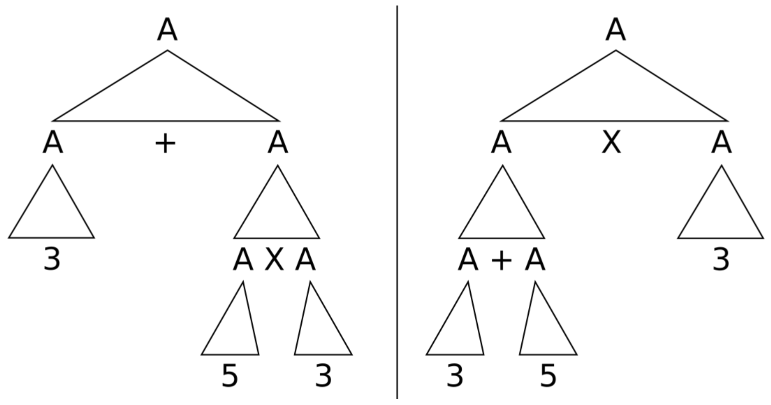
4 Ambiguity

See also the video lecture recording: Ambiguity in CFGs on Canvas.

We have seen that one derivation tree can arise from several different derivations (although only one leftmost derivation), but that's not a serious problem. Much more serious is that a word can have more than derivation tree. For example, using the following grammar:

$$\Rightarrow A \quad ::= \quad A + A \mid A \times A \mid (A) \mid 3 \mid 5$$

the word $3 + 5 \times 3$ has derivation trees



The above grammar does not take into account the order of precedence of operators, that is, it does not ensure that $\times$ operation must be applied before $+$. In contrast, the following grammar generates exactly the same language, but every generated string has a unique derivation tree. Hence, this grammar is unambiguous, whereas the one above is ambiguous. The following grammar takes into account the *precedence* of operators, by moving the higher priority operations lower in the grammar e.g. the $\times$ operation comes later than $+$ operation.

$$\begin{array}{rcl} \Rightarrow A & ::= & A + B \mid B \\ B & ::= & B \times C \mid C \\ C & ::= & (A) \mid 3 \mid 5 \end{array}$$

It is usually desirable to design a grammar to be unambiguous. Therefore it would be useful to have a program that, when we provide a context free grammar, tells us whether that grammar is ambiguous or not. But that is impossible: ambiguity of context free languages is *undecidable*. (We shall not prove this.)

4.1 Test Your Understanding

1. Try deriving the string $3 + 5 \times 3$ in two different ways using leftmost derivation only, using the grammar given above.

2. Show that the following grammar is ambiguous. The alphabet is $\{\mathtt{a}, \mathtt{b}\}$.

$$\begin{array}{rcl} \Rightarrow P & ::= & \varepsilon \mid Q\mathtt{a} \mid \mathtt{a}Q \\ Q & ::= & \mathtt{aa}P \mid \mathtt{b}R \\ R & ::= & Q\mathtt{a} \end{array}$$

5 Chomsky Normal Form (CNF)

See also the video lecture recording: Chomsky Normal Form on Canvas.

In this section we are going to learn how to convert a CFG into a special form known as *Chomsky Normal Form*. It only allows rules of the following kind

$$\begin{array}{rcl} A & ::= & BC \\ A & ::= & a \end{array}$$

where a is any terminal and A, B, and C are any variables - except that B and C may not be the start variable. In addition, there can be a rule $S ::= \ \varepsilon$, where S is the start variable.

Chomsky Normal Form has various uses, but one is especially noteworthy. This is the fact that using a grammar in this form, a derivation of a nonempty word involves $2n - 1$ steps, where n is the word's length. This can be proved using course-of-values induction on n.

And so, given a grammar G and a word w, we can test mechanically whether the word is accepted by G. First we convert G to Chomsky Normal Form (in the manner we shall see below). Then we write out all derivations of length $2n - 1$ and see if any of them work. Highly inefficient, and much worse than the CYK algorithm, but it does the job.

To convert any given CFG into CNF, use the steps outlined below:

1. We begin by introducing a new start symbol (variable) to the grammar.

2. In the second step, we remove all of the ε-rules of the form $A ::= \ \varepsilon$.

3. In the third step, we remove all of the unit productions of the form $A ::= \ B$.

4. We may need to patch-up/fix the grammar to make sure that it still produces the original language.

5. In the end, we will convert the remaining rules into proper form.

Let's convert the following CFG (taken from Sipser) into Chomsky normal form by using the conversion steps outlined above. We will go through the conversion process, step by step, and at each stage show you the new grammar along with the newly added rules shown in bold.

$$\begin{array}{rcl} \Rightarrow S & ::= & ASA \mid aB \\ A & ::= & B \mid S \\ B & ::= & b \mid \varepsilon \end{array}$$

1. Add a new start variable S_0

$$\begin{aligned} \Rightarrow \boldsymbol{S_0} &\;\boldsymbol{::=}\; \boldsymbol{S} \\ S &::= ASA \mid aB \\ A &::= B \mid S \\ B &::= b \mid \varepsilon \end{aligned}$$

2a. Remove ε-rule $B ::= \varepsilon$

$$\begin{aligned} \Rightarrow S_0 &::= S \\ S &::= ASA \mid aB \mid \boldsymbol{a} \\ A &::= B \mid S \mid \boldsymbol{\varepsilon} \\ B &::= b \end{aligned}$$

2b. Remove ε-rule $A ::= \varepsilon$

$$\begin{aligned} \Rightarrow S_0 &::= S \\ S &::= ASA \mid aB \mid a \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\ A &::= B \mid S \\ B &::= b \end{aligned}$$

3a. Remove the unit rule $S ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= S \\ S &::= ASA \mid aB \mid a \mid SA \mid AS \\ A &::= B \mid S \\ B &::= b \end{aligned}$$

3b. Remove the unit rule $S_0 ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ S &::= ASA \mid aB \mid a \mid SA \mid AS \\ A &::= B \mid S \\ B &::= b \end{aligned}$$

3c. Remove the unit rule $A ::= B$

$$\begin{aligned} \Rightarrow S_0 &::= ASA \mid aB \mid a \mid SA \mid AS \\ S &::= ASA \mid aB \mid a \mid SA \mid AS \\ A &::= \boldsymbol{b} \mid S \\ B &::= b \end{aligned}$$

3d. Remove the unit rule $A ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= ASA \mid aB \mid a \mid SA \mid AS \\ S &::= ASA \mid aB \mid a \mid SA \mid AS \\ A &::= b \mid \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ B &::= b \end{aligned}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form.

$$\begin{aligned} \Rightarrow S_0 &::= AC \mid DB \mid a \mid SA \mid AS \\ S &::= AC \mid DB \mid a \mid SA \mid AS \\ A &::= b \mid AC \mid DB \mid a \mid SA \mid AS \\ C &::= SA \\ D &::= a \\ B &::= b \end{aligned}$$

5.1 Test Your Understanding

Convert the following CFG into an equivalent CFG in Chomsky normal form

$$\begin{array}{rcl} \Rightarrow A & ::= & BAB \mid B \mid \varepsilon \\ B & ::= & 00 \mid \varepsilon \end{array}$$

6 Emptiness and Fullness

To test whether a CFG accepts *some* word, we mark each variable that is able to turn into a word. For example, if we see a production

$$A \quad ::= \quad BCaB$$

and we know that B and C can turn into a word, then A can too. Just repeat this until you can't go any further, and see whether the start variable can turn into a word.

Can we test whether a CFG accepts *every* word over the given alphabet? No, this is an undecidable problem. Therefore, it is undecidable whether two CFGs accept the same word.

7 Beyond context free languages

We know that not all languages are context free, because there are uncountably many languages, yet only countably many context free grammars. But are there useful examples of languages that are not context free? Yes. Here's an example.

When you write a program, it's essential that every variable is declared, because a program with an undeclared variable can't run. A compiler will always check that the code being compiled has this property. But the set of words with this property is not context free. (We won't prove this, but the basic problem is that a context free grammar requires the set of nonterminals to be finite.)