Context free languages

More general kind of grammar than regex

A represents integer expressions

B represents Boolean expressions

::= This means can be

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's book: "Introduction to the Theory of Computation".]

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.

A and B are called variables, non-terminals or sorts

Each rule consists of a variable (the LHS) and a string of variables and tokens (the RHS)

If 4 tuple was (X, p, δ, Acc)

- *A finite set X of states.*
- *An initial state*

$p \in X$

- *A transition function*

$\delta : X * \sum -> X$

- *A set of accepting states*

$Acc \subseteq X$

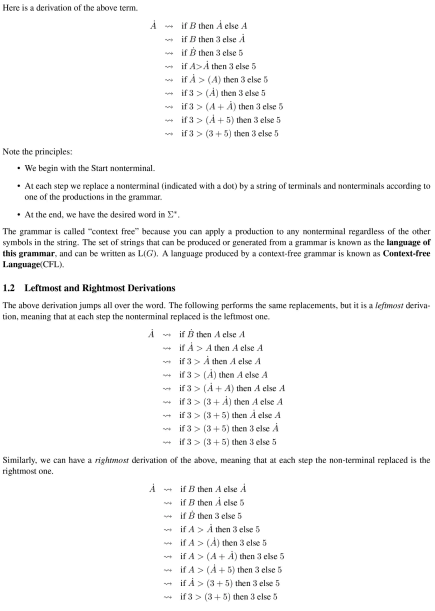
The dot over A means it is expanded/replaced next

> So any variable can be replaced as long as it aligns with a rule in the language

We write $\sum^*$ for the set of all words.

Term:

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



Here is a derivation of the above term.

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

Note the principles:

- We begin with the Start nonterminal.
- At each step we replace a nonterminal (indicated with a dot) 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

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.

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

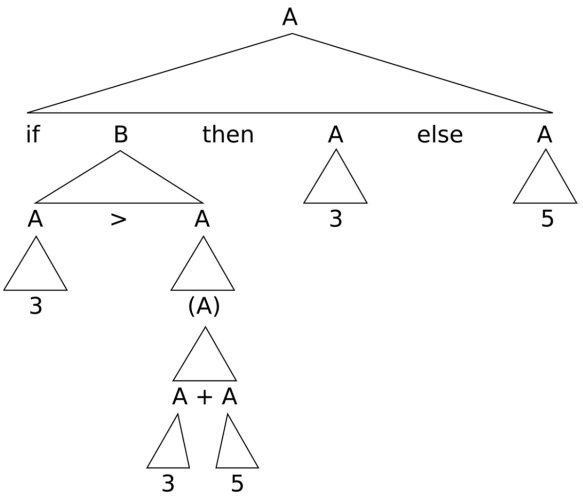
Similarly, we can have a *rightmost* derivation of the above, meaning that at each step the non-terminal replaced is the rightmost one.

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

- *At each stage we have a string of variables and tokens*
- *At the start, we have a start variable*

- *At the end we have a word*
- *At each transition, we replace a variable by a string of variables and tokens*
- *This word is accepted by the grammar*

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 terminal.

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	$\Rightarrow$ 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) {
 ...
}
```


Is it decidable whether a given context-free grammar accepts a given word?

i.e. is there a program that takes as input a context-free grammar and a word and returns true

iff the word is accepted by the grammar

Ans: Yes, this can be done in cubic time for a fixed grammar

Eg.

CYK algorithm

For algorithms in specific forms there are more efficient algorithms

And there are tools called parser generators that take such a grammar and produce parses

i.e. algorithms that convert words into parse trees

Is it decidable whether two context-free grammars are equivalent

Ans: No, this is an undecidable/untractable problem

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

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:

$$\Rightarrow A ::= \quad 0A1|\varepsilon$$

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

$$\Rightarrow B ::= \quad 1B0|\varepsilon$$

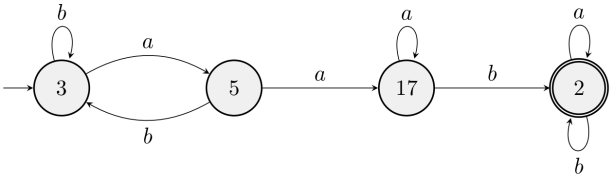
for the language $\{1^n0^n|n \geq 0\}$ and then add the rule $S ::= \quad A|B$ to give the grammar:

$$\begin{aligned} \Rightarrow S &::= \quad A|B \\ A &::= \quad 0A1|\varepsilon \\ B &::= \quad 1B0|\varepsilon \end{aligned}$$

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

1. Create a variable R_q for each state q of the DFA.
2. Add the rule $R_p ::= \quad aR_qj$ to the CFG, if $\delta(p, a) = q$ is a transition in the DFA.
3. Add the rule $R_q ::= \quad \varepsilon$, if q is an accepting state of the DFA.
4. Make R_s the start variable of the grammar, where s 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_2, R_3, R_5 and R_{17}, corresponding to the states of above DFA.
2. We add the rules using the pattern $R_p ::= \quad aR_q$ to the CFG, for each transition in the DFA.
3. Add the rule $R_2 ::= \quad \varepsilon$, as 2 is an accepting state of the DFA.
4. Make R_3 the start variable of the grammar.

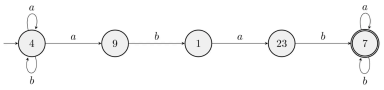
We get the following grammar after applying the above steps:

$$\begin{aligned} R_2 &::= \quad aR_2 \mid bR_2 \mid \varepsilon \\ \Rightarrow R_3 &::= \quad aR_5 \mid bR_3 \\ R_5 &::= \quad aR_{17} \mid bR_3 \\ R_{17} &::= \quad aR_{17} \mid bR_2 \end{aligned}$$

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

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 Induction over a CFG

We often use induction to prove properties about the words in a context-free language. For example, here is a CFG:

$$\begin{array}{rcl} X & ::= & YY \mid abba \\ \Rightarrow Y & ::= & aX \mid XXbbb \mid bYa \end{array}$$

Prove that every word accepted by this CFG has odd length.

To do this, we shall prove a stronger statement: that every X-word has even length, and every Y-word has odd length. We shall induct on the size of a derivation. This means that we shall prove that an X-word or Y-word with derivation d has the required property, assuming that all X-words or Y-words with a smaller derivation have the required property.

Given a derivation d of an X-word x, there are two cases, depending on the root-rule of d.

- $x = y_0y_1$ where y_0 and y_1 are Y-words with derivation smaller than d. By the inductive hypothesis, y_0 and y_1 have odd length. So x has even length.
- $x = abba$, so x has even length.

Given a derivation d of a Y-word y, there are three cases, depending on the root-rule of d.

- $y = ax$, where x is an X-word with derivation smaller than d. By the inductive hypothesis, x has odd length. So y has even length.
- $y = x_0x_1bbb$, where x_0 and x_1 are X-words with derivation smaller than d. By the inductive hypothesis, x_0 and x_1 have even length. So y has odd length.
- $y = by_0a$, where y_0 is a Y-word with derivation smaller than d. By the inductive hypothesis, y_0 has odd length. So y has odd length.

This style of proof is called *structural* induction: we prove a property about a tree (in this case, the derivation d) by using the inductive hypotheses only on the immediate subtrees.

In the induction over a CFG

Base case:

- *X accepts abba*
- *Y accepts aabba*

Next:

- Say X accepts a word w
  - (So w has a derivation root rule that yields X)
- Proof continued in above image

A grammar that allows a word to have more than one derivation tree is called ambiguous

A grammar is ambiguous iff there's a word with more than one leftmost derivations

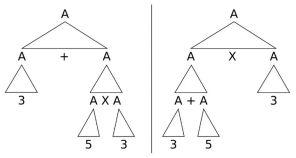
Warning: Not just more than one derivation

5 Ambiguity

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:

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

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 $+$. Here is an equivalent grammar that is unambiguous:

$$\begin{aligned} \Rightarrow X &::= X + Y \mid Y \\ Y &::= Y \times Z \mid Z \\ Z &::= (X) \mid 3 \mid 5 \end{aligned}$$

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 grammars is *undecidable*. (We shall not prove this.)

5.1 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 $\{\mathtt{a}, \mathtt{b}\}$.

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

5.1: Test your understanding:

$$\begin{aligned} &\dot{X} \sim \dot{X} + Y \\ &\sim \dot{Y} + Y \\ &\sim \dot{Z} + Y \\ &\sim 3 + \dot{Y} \\ &\sim 3 + \dot{Y} \times Z \\ &\sim 3 + \dot{Z} \times Z \\ &\sim 3 + 5 \times \dot{Z} \\ &\sim 3 + 5 \times 3 \end{aligned}$$

6 Chomsky Normal Form (CNF)

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

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

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.

💡

Proof by Induction on n (Length of the Word)

Base Case (n=1)

- A single terminal a is derived using the rule A → a.
- This requires **1 step**.
  - Formula check: 2(1) - 1 = 1, which holds.

Inductive Hypothesis

Assume that for a word of length n, the derivation tree has **2n - 1** steps and all derivation subtrees are true.

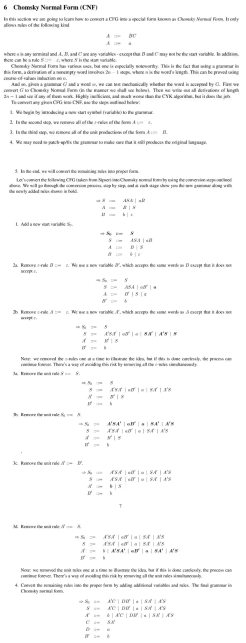
Inductive Step: Prove for n+1

- Since CNF {Chonksy Normal Form} only allows binary splitting via A → BC, any derivation of a word of length n+1 must involve splitting into two subwords.
- Suppose the subwords have lengths k and m such that:
  - k + m = n + 1
- By the induction hypothesis:
  - The number of derivation steps for the subword of length k is 2k - 1.
  - The number of derivation steps for the subword of length m is 2m - 1.
- The additional step comes from applying the rule A → BC, contributing **one extra step**.

Thus, the total number of steps is:

$$(2k-1)+(2m-1)+1=2(k+m)-1=2(n+1)-1=2n-1$$

This matches our formula.



All 2b does is add the steps for accommodating for the epsilon removal from the previous step:

> *You could have AS and SA if one of the As was epsilon or you could have S if both As were epsilon*

All the last step is obey the rules of CNF{Chonksy Normal Form}:

> *Any non-terminal starting rule can either only imply two non-terminals or one terminal*

In Chomsky Normal Form (CNF), a unit rule is a production rule where a non-terminal symbol directly produces another non-terminal symbol. Specifically, a unit rule has the form:

6.1 Test Your Understanding

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

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

Starting CFG:

$A ::= BAB \quad | \quad B \quad | \quad \epsilon$

$B ::= 00 \quad | \quad \epsilon$

Add a new start variable:

$A_0 = A$
$A ::= BAB \quad | \quad B \quad | \quad \epsilon$
$B ::= 00 \quad | \quad \epsilon$

$Remove \quad \epsilon \quad rules$:

$A_0 = A$
$A ::= BAB \quad | \quad B \quad | \quad BB$
$B ::= 00$

Remove unit rules:

$A_0 = BAB \quad | \quad 00 \quad | \quad BB$
$B ::= 00$

Adjust and add to finalise CNF:

$A_0 = BC \quad | \quad 00 \quad | \quad BB$
$C = BA$
$B ::= 00$

7 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 words.

8 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.)