Chapter 1

Languages

I'm a programmar that is not pro-grammar

Cliph

1.1 Introduction

We have now done a ton in regex, but the issue with regular expressions is that they can only do so much. Regular expressions are great when talking about regular languages, but programming languages (and spoken languages) are not regular languages, so we need something more expressive. Let us consider what exactly we need to describe a language.

1.2 Context-Free Grammars

Recall that a language is just a set of strings. We used regular expressions to tell you how to construct the strings in the set. An alternative way is to give a recursive definition of the set. Recall from 250 what a recursive set is (in this case the set of positive multiples of 3):

$$S = \begin{cases} 3 \\ x \in S \Rightarrow x + 3 \in S \end{cases}$$

We can do the same thing to describe sets of strings (although we use a slightly different notation). This is called a grammar. We will be going over context-free grammars (CFGs) as opposed to context-sensitive grammars (CSGs). For example, let us take the CFG we saw for regular expressions:

Any regular expression can be expressed as a string that is in the following set

$$\begin{aligned} S \rightarrow\ & \epsilon \\ & |\sigma \\ & |SS \\ & |S|S \\ & |S^* \\ & |(S) \end{aligned}$$

Any valid regular expression sentence follows the above pattern.

Let's break this down and understand exactly what it means. Here, S is called a Non-terminal because S could be a variety of things. On the other hand symbols like $\epsilon, \sigma, ^*, |, (,)$ are what we call terminals. Grammars also have what is called productions: rules about what non-terminals can be. This example is honestly not the best for these terms, but we will see an example soon, and we will revisit these terms.

The important part right now is that this grammar describes all strings that represent a regular expression. Very much like we can use finite automata to represent a regular expression and show that the regular expression accepts a string, we can derive a string from a grammar using substitution to show that a string is grammatically correct (and hence belongs in the language the grammar describes).

For example, the regular expression /ab∗/ can be described using the following derivation:

$$\begin{aligned} S \rightarrow\ & SS \\ \rightarrow\ & aS \\ \rightarrow\ & aS^* \\ \rightarrow\ & ab^* \end{aligned}$$

Those who have taken a linguistics or hearing and speech sciences class, or even an English class, may know that “grammar" typically refers to the order in which words need to be for the sentence to make sense. That is still true here.

Unlike /ab∗/, the regular expression /∗b/ cannot be derived from the above grammar, so we claim it to be grammatically incorrect and not part of the language.

Now, English is complicated and has a lot of rules, but consider the following simplified grammar for English.

$$\begin{aligned} S \rightarrow\ & NP\, VP \\ NP \rightarrow\ & \text{pronoun} \\ & |\text{proper\_noun} \\ & |\text{det}\ AN \\ AN \rightarrow\ & \text{adj}\ AN \\ & |\text{noun} \\ VP \rightarrow\ & \text{verb} \\ & |\text{verb}\ NP \end{aligned}$$

Let us revisit our terms from earlier to break this down. Non-terminals are symbols that represent other symbols. Conventionally, we give them uppercase letters. Sometimes, the letters mean something; sometimes they are just alphabetical. In this case, they mean something (NP stands for noun phrase, VP for verb phrase, and AN for adjective noun).

- **Terminals**: `pronoun, proper_noun, det, adj, noun, verb` are all terminals. Unlike the previous example where a lowercase letter was a symbol, these all stand for larger sets of things (This can be confusing, so in this course we will typically only be using symbols like in the first example).
- **Non-terminals** S, NP, VP, AN are all non-terminals. We know this because they are all uppercase, and each has a production rule associated with it.
- **Production**: a production rule tells us all the things a non-terminal can be. For example, $S \rightarrow NP\ VP$ is a production rule. It states that any sentence S consists of a noun phrase NP followed by a verb phrase VP.
  $AN- >$ adj $AN|noun$ says that any AN phrase is an adjective followed by another AN phrase, or it is just a noun.

Using this grammar, we can still derive if a sentence is grammatically valid in English. For example, if I had a sentence like "The child ran the race", then I could say that this sentence is grammatically correct and should be in the set of valid English sentences. Before I show the derivation, let us make sure we know our parts of speech:

- "The" is a determiner (det) because it determines the reference of something. Some other examples are "every", "a", "some", and "each".
- "child" and "race" are nouns since they fall under the category of a person, place, or thing (or idea).
- "ran" is a verb since it describes an action.

Knowing all this, let us now show the derivation:

$$\begin{aligned} S \rightarrow\ & NP\, VP \\ \rightarrow\ & \text{The (det)}\ AN\ VP \\ \rightarrow\ & \text{The child (noun)}\ VP \\ \rightarrow\ & \text{The child ran (verb)}\ NP \\ \rightarrow\ & \text{The child ran the}\ AN \\ \rightarrow\ & \text{The child ran the race} \end{aligned}$$

The sentence S is a noun phrase (NP) followed by a verb phrase (VP). In the example above, the first noun phrase is going to use the third definition of a noun phrase: det AN. "The" is the determiner. The following AN then uses the second definition of being just a noun, which in this case is "child". So the noun phrase is "The child". The verb phrase is going to use the second definition of a verb phrase: verb NP. The verb here is "ran", and the noun phrase is going to be "det AN". In this case, the determiner is again "the" and the AN is just a noun: "race". More on this in a bit.

1.3 Designing Grammars

We said that a grammar describes a set of strings, but it is more expressive than regular expressions. This means that any regular expression can be expressed as a CFG but also that CFGs can get around some restrictions that regular expressions have. Let us start with operations that are supported by regular expressions.

1.3.1 Regular Expressions Supported

Let us start with our 3 base cases.

- $\emptyset$: If the language is empty, meaning it is a set containing no strings, then the CFG should reflect that as well. The CFG can still be represented as $\emptyset$, which is the null (empty) set
- ϵ: If the regular expression accepts the empty string, then the CFG can just have a single production.

$$S \longrightarrow \epsilon$$

- σ: If the regular expression is just a single character, we can have our CFG reflect that character in a single production.

$$S \longrightarrow \sigma$$

  I will say though that in larger grammars, we typically describe sentences or statements with words, so if we have a list of words, we can do the same thing. For example, we can have something like

$$S \longrightarrow \text{Cliff}$$

Once we have the base cases (shown above), we can talk about the recursive definitions: concatenation, branching, and kleene closure:

- **Concatenation**: If we wish to concatenate two things together, we can just push them together with either non-terminals or just the string you expect. For example, the corresponding CFG for the regular expressions /ab/ would be

$$S \longrightarrow \text{ab}$$

  Alternatively you could do something like

$$\begin{aligned} S \longrightarrow &\quad AB \\ A \longrightarrow &\quad a \\ B \longrightarrow &\quad b \end{aligned}$$

  This can be helpful if you have branching (next bit), or just a sentence where you want to force one thing to come before (eg. adjective before noun).

- Branching: If we want to branch, we can use the same symbol we used in regex $|$. For example, the grammar for a greeting could be

$$S \longrightarrow \text{Hello}|\text{hi}$$

  You can put each option on a new line if you have the space, but either way is valid.

- Kleene Closure: For allowing repeated values, we can just utilize the recursive property these sets have. For example, the corresponding CFG for the regex /a∗/ is
$$S \rightarrow aS|\epsilon$$
. CFGs also let us have a shortcut for something like /a+/
$$S \rightarrow aS|a$$
We can also use this to repeat whole words:
$$\begin{aligned} S \rightarrow &\quad \text{This is a } T \text{ sentence} \\ T \rightarrow &\quad \text{very } T|\text{long} \end{aligned}$$

1.3.2 Not supported by Regular Expressions

We said that CFGs are more expressive which means we can say more with a CFG than we can with regular expressions. So let us think of some of the restrictions we had with regular expressions. We could not look forward more than one character at a time, and we could never reference what we previously saw. So would couldn't do things like balanced parenthesis. This is all in thanks to the recursive nature of CFGs.

Consider the previous CFG:
$$\begin{aligned} S \rightarrow &\quad \text{This is a } T \text{ sentence} \\ T \rightarrow &\quad \text{very } T|\text{long} \end{aligned}$$

We have a sentence where we have something known ("This is a ") followed by non-terminal (T) which is then followed by some other known string ("sentence"). By allowing for this ability to look at both before and after the non-terminal, we can do things like balance parenthesis, or have relative distinct values.

For something like having a balanced values on either side (parenthesis or palindromes), we can just put the values on either side of the non-termainal.

Balanced parenthesis surrounding "a"	$S \rightarrow (S)|\text{a}$
Palindromes of "a", "b", and "c"	$S \rightarrow \text{a}S\text{a}|\text{b}S\text{b}|\text{c}S\text{c}|\epsilon$

For having relative number of values we are a tad restricted to a few characters that are relative to each other, but supposed we want a string with the some number of "a"s followed by the same number of "b"s. Our notation for this is $a^n b^n$. The following grammar would allow for that:
$$S \rightarrow aSb|\epsilon$$
We can also do distinct relative numbering like $a^n b^{2n}$:
$$S \rightarrow aSbb|\epsilon$$
Or even $a^n b^m, m \geq n$
$$\begin{aligned} S \rightarrow &\quad \text{a}S\text{b}|T \\ T \rightarrow &\quad \text{b}T|\epsilon \end{aligned}$$
Sometimes the order doesn't even matter. If I wanted a string that had the same number of "a"s and "b"s in any order then I could do something like:
$$S \rightarrow S\text{a}S\text{b}|S\text{b}S\text{a}|\epsilon$$
I can also do a string that has an unequal amount of "a"s and "b"s in any order:
$$\begin{aligned} S \rightarrow &\quad A|B \\ A \rightarrow &\quad C\text{a}A|C\text{a}C \\ B \rightarrow &\quad C\text{b}B|C\text{b}C \\ C \rightarrow &\quad \text{a}C\text{b}C|\text{b}C\text{a}C \end{aligned}$$

1.3.3 A basic Grammar

Putting all this together, I could create a rudimentary language that describes basic algebraic expressions.

$$\begin{aligned} A \rightarrow &\quad A + A | A - A | A * A | A / A | N \\ N \rightarrow &\quad number \end{aligned}$$

This grammar is okay because it allows for strings like `"2 + 1 + 0 - 4 * 9 / 3"`. However, this grammar does not allow for things like `"(4-30)*-5`, which of course is allowing order of operations to be expressed. We can just easily modify this expression by adding our parenthesis rule we talked about:

$$\begin{aligned} A \rightarrow &\quad A + A | A - A | A * A | A / A | (A) | N \\ N \rightarrow &\quad number \end{aligned}$$

Now from a compiler/interpreter stand point these this grammar still has some issues, but we will talk about all of this between the next section and the Parsing chapter.

1.4 Modeling Grammars

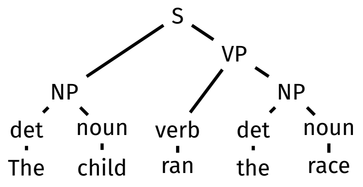
Now that we know what a CFG is and what they represent, we need to discuss how we can model them into something useful (since typically you need to do something with a language and not just know what they are and how they work).

Now one of the leading theories in linguistics and psychology (that I know of) is that we store grammar and the like as a tree in our heads. Regardless if I am up to date or not in the linguistics field, this is what we will be using to model our grammars and sentences in compsci.

Recall that a grammar tells you the structure of a language, so the tree should tell us this as well. We do this by our recursive definition. Consider our basic English Grammar

$$\begin{aligned} S \rightarrow &\quad NP\ VP \\ NP \rightarrow &\quad \text{pronoun} \\ &\quad |\text{proper\_noun} \\ &\quad |\text{det noun} \\ VP \rightarrow &\quad \text{verb} \\ &\quad |\text{verb } NP \end{aligned}$$

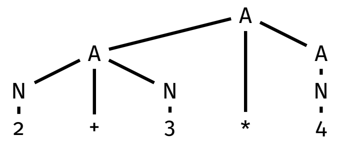
If we take the same sentence we always used: "The child ran the race", we can represent this sentence as a tree thanks to our grammar:



If we consider our basic algebraic expression grammar we can also model sentences with it:

$$\begin{aligned} A \rightarrow &\quad A + A | A - A | A * A | A / A | (A) | N \\ N \rightarrow &\quad number \end{aligned}$$

For example "2 + 3 * 4" can be modeled as:



Technically this tree represents the following derivation:

$$\begin{aligned} A \longrightarrow\ & A * A \\ \longrightarrow\ & A + A * A \\ \longrightarrow\ & N + A * A \\ \longrightarrow\ & 2 + A * A \\ \longrightarrow\ & 2 + N * A \\ \longrightarrow\ & 2 + 3 * A \\ \longrightarrow\ & 2 + 3 * N \\ \longrightarrow\ & 2 + 3 * 4 \end{aligned}$$

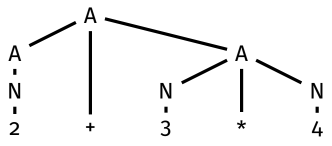
This derivation I got via something we call a "left hand derivation". That is, we will substitute the left most variable for a recursive definition. Consider the right hand derivation for the same tree:

$$\begin{aligned} A \longrightarrow\ & A * A \\ \longrightarrow\ & A * N \\ \longrightarrow\ & A * 4 \\ \longrightarrow\ & A + A * 4 \\ \longrightarrow\ & A + N * 4 \\ \longrightarrow\ & A + 3 * 4 \\ \longrightarrow\ & N + 3 * 4 \\ \longrightarrow\ & 2 + 3 * 4 \end{aligned}$$

Notice that using a left hand or right hand derivation does not change the tree (but if we did more, it would impact the way the tree is build. However, notice we could have used the following left hand derivation instead:

$$\begin{aligned} A \longrightarrow\ & A + A \\ \longrightarrow\ & N + A \\ \longrightarrow\ & 2 + A \\ \longrightarrow\ & 2 + A * A \\ \longrightarrow\ & 2 + N * A \\ \longrightarrow\ & 2 + 3 * A \\ \longrightarrow\ & 2 + 3 * N \\ \longrightarrow\ & 2 + 3 * 4 \end{aligned}$$

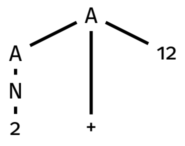
This tree would look like:



Both of these trees and derivations are both valid when substituting the leftmost variable for a definition of A, so we call this grammar ambiguous. A grammar is ambiguous when there are two valid left hand derivations. A grammar can also be ambiguous when where are 2 valid right hand derivation.

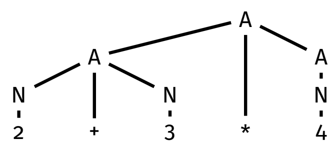
Now that we have some tree model, we need to discuss what we do with these trees. These trees have a proper name as well: parse trees. We will get into parsing in a future chapter, but ultimately should we want to try and obtain meaning from a the tree we need some sort of tree traversal algorithm.

In this case, a post order traversal would probably be helpful here. Consider the second variation to the tree we had. If I wanted to calculate the right-most A, then I would need to figure out what two values my sub children were before I said "3-4". Then I could recursively figure out subtrees until I get to the root. That is I would go from the above tree to the below tree:

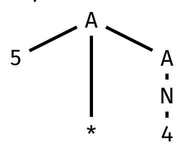


Which would then have the two subtrees be evaluated to 2 and 12 respectively, which we would then multiply to get 24 as the final result.

However, consider the first variation of the tree. If we simplified in this manner we would get the following:



Simplified down to:



And finally to:
20

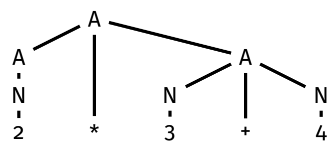
Notice that we get two separate values due to the ambiguity.

There are two ways to help solve ambiguous grammars: we could try and figure out more the grammar and restrict how we go about traversing through the tree, or we can just change the grammar a bit.

For example, much of the ambiguity is because we don't know which path the variable should be when given something. We can fix this by adding separate non-terminals:

$$\begin{aligned} A \longrightarrow &\quad N + A|N - A|N * A|N/A|N|(A) \\ N \longrightarrow &\quad number \end{aligned}$$

Now we know that any expression must start with a number and not an expression. So constructing a tree of "2∗3+4 could only result in the following:

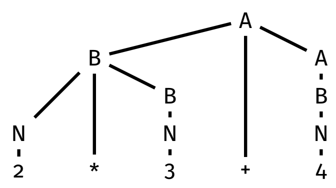


The final issue here is that when we do our traversal, we are still doing addition before multiplication. In order to fix this, we can either mandate that all operations be put in parenthesis or we can change the grammar to have precedence.

The idea of precedence is that we do things of higher precedence closest to the base case as possible. That is notice that when multiplication is closest to the bottom of the tree, we are getting the correct computation. So let us modify our grammar to have precedence. We use the same sort of trick we did for concatenation, we add more non-terminals since we need to figure out non-terminals before terminals. We should get:

$$\begin{aligned} A \longrightarrow &\quad B + A|B - A|B \\ B \longrightarrow &\quad N * B|N/B|N \\ N \longrightarrow &\quad number|(A) \end{aligned}$$

Given this grammar we can only get the following tree:


A
B
A
B
B
N
N
N
2
*
3
+
4