Computer Science

Dr. Y. Liow (March 22, 2024)

Contents

debug: chap.tex

Chapter 14

Context Free Languages

debug: cfg/cfg.tex

14.1 Introduction

Recall the big picture again ... given a problem $\text{PROBLEM}(n)$, instances of it will form a set

$$L_{\text{PROBLEM}} = \{\text{PROBLEM}(n) \mid n \in \mathbb{N}\}$$

Using this formulation, if there's an automata M that accepts L_{PROBLEM}, we can solve the given problem.

We saw in the chapter on regular languages that there are three "devices" that describes the same class of languages: The DFAs, NFAs, and regexes (although they are very different) describe the class of regular languages. (We also saw the GNFA but that's more like a device used in the proof that NFAs can be "converted" to regexes.) We also saw that there are many languages which are not regular. This means that if a problem gives rise to a regular language, then DFAs/NFAs/regexes are not powerful enough to solve this problem.

On the practical side, DFAs/NFAs/regexes can be used to recognize certain substrings (not all of course). This is used in for instance compilers to recognize keywords of a programming language. It's also used to validate user inputs such as in a web form. DFAs and also used to model AI behaviors in, for instance, games. Historically, regexes were used to analyze neurons in our brain.

(Although I mentioned DFAs/NFAs/regexes, there are in fact many other automatas that accept regular languages.)

Now it's time to go on ...

We need more powerful classes of "devices". These devices will accept regular languages ... and more.

We'll look at context free grammars and pushdown automatas. They are used in compilers for instance to build "parse trees" after the source program is broken up into lexemes by regexes. They are also used to analyze human languages and therefore are important in many area of AI such as natural language processing.

14.2 Grammar

Here's an example of a (context-free) grammar (or CFG for short):

$$G : \begin{cases} S & \rightarrow aSb \\ S & \rightarrow \epsilon \end{cases}$$

And here's how you use it. You start with symbol S and choose any arrow and apply the arrow to S to get a string, then you choose a symbol in the string and find an arrow to apply to that symbol, etc. You stop when you see only a's and b's. So here's a string derived from S:

$$S \implies \epsilon$$

We get ϵ.

Here's another example

$$\begin{aligned} S &\implies aSb && \text{choosing first arrow} \\ &\implies a\epsilon b && \text{choosing second arrow} \end{aligned}$$

We get ab.

Here's another example

$$S \implies aSb \implies aaSbb \implies aa\epsilon bb$$

We get $aabb = a^2b^2$.

The set of strings generated by G in the above manner, i.e., when you have no more variables in the replacement process, is denoted $L(G)$. Clearly $L(G) = \{a^nb^n \mid n \geq 0\}$. (Right?) Recall from the chapter on regular languages that this language is not regular.

The arrows are called **production rules** (or rules). There are two types of symbols: S is called a **variable** and symbols such as a and b are called **terminals**. There can be more than one variable. You have to start with a symbol, the **start variable**. In the above example the start variable is S.

You can use any symbol as the start symbol, but S is pretty standard. Note that there must be finitely many product rules.

Note that the production rules simply allows you to replace one variable is a string of variables/terminals.

You can write the above like this

$$G : S \to \epsilon \,|\, aSb$$

The | is read "or".

Example 14.2.1. Let's try something different. Consider the following context free grammar:

$$G : S \to aS \,|\, bS \,|\, \epsilon$$

Clearly $\epsilon \in L(G)$ since

$$S \implies \epsilon$$

using the last rule. Now using the first and last rule we get

$$S \implies aS \implies a\epsilon = a$$

Therefore $a \in L(G)$. Check for yourself that using the second and last rule we get $b \in L(G)$. What about *baab*? Can we derive *baab*?

$$\begin{aligned} S &\implies bS && \text{second rule} \\ &\implies baS && \text{first rule} \\ &\implies baaS && \text{first rule} \\ &\implies baabS && \text{second rule} \\ &\implies baab\epsilon && \text{third rule} \end{aligned}$$

Since $baab \in L(G)$. Your turn: Can you derive ab^2a^3 from this grammar? What is $L(G)$? I hope it's clear that $L(G) = \Sigma^*$ where $\Sigma = \{a, b\}$. □

Example 14.2.2. The grammar:

$$S \to S$$

obviously cannot generate any string. So the language generated by this grammar is $\emptyset$. □

Exercise 14.2.1. Here's a grammar

$$G : S \rightarrow aS \,|\, \epsilon$$

where $\Sigma = \{a, b\}$. Which of the following strings are generated by G? For the strings generated by G, write down a derivation.

1. ϵ
2. a
3. b
4. aa
5. ab
6. ba
7. bb

What is $L(G)$? □

Exercise 14.2.2. Here's another:

$$G : S \rightarrow Sa \,|\, \epsilon$$

What is $L(G)$? □

Exercise 14.2.3. Let G be the following context free grammar:

$$S \rightarrow aS \tag{1}$$
$$S \rightarrow T \tag{2}$$
$$T \rightarrow Tb \tag{3}$$
$$T \rightarrow \epsilon \tag{4}$$

where S is the starting variable. Which of the following strings are generated by G? For each string that is generated by G, write down a derivation for that string citing the rule used for each step.

1. ϵ
2. a
3. b
4. aa
5. ab
6. ba
7. bb
8. $aabb$
9. $abab$
10. $bbaa$

Write down in set notation the language generated by G, i.e., $L(G)$. Can you rewrite the above without variable T?

Exercise 14.2.4.

1. Write down a CFG G_1 such that $L(G_1) = \{\epsilon\}$.
2. Write down a CFG G_2 such that $L(G_2) = \{a\}$.
3. Let x be a string in Σ^* where $\Sigma = \{a, b\}$. Write down a CFG G such that $L(G_3) = \{x\}$. (Of course this would include the previous two problems.)

□

Exercise 14.2.5. Write down a grammar G such that $L(G) = \Sigma^*$ where $\Sigma = \{a, b, c\}$. □

Exercise 14.2.6. Write down a grammar G such that $L(G) = \Sigma^+$ where $\Sigma = \{a, b\}$. □

Exercise 14.2.7. Consider the following CFG:

$$G : \begin{cases} S & \rightarrow Ta^3T \\ T & \rightarrow a \,|\, b \,|\, \epsilon \end{cases}$$

(If the start variable is not specified and there's an S in the grammar, then S is the start variable.)

1. What is the shortest string that can be derived from G? (The length should be 3.)
2. Derive strings of length 4 from G.
3. Derive strings of length 5 from G.
4. OK ... so here's the real question ... what is the language generated by this grammar?
5. Now modify the grammar so that the language is $\{xa^3y \,|\, x, y \in \Sigma^*\}$ where $\Sigma^* = \{a, b\}$.

$\square$

Example 14.2.3. Let's find a CFG G such that $L(G)$ is the set of strings of the form $a^n b^{2n}$ for $n \geq 0$. Here are a few strings in $L(G)$:

$$\epsilon, abb, aabbbb$$

Of course you can try this grammar:

$$S \rightarrow \epsilon \mid abb \mid aabbbb$$

That's well and dandy ... but if you include other stinrgs like $aaabbbbbb$, $aaaabbbbbbbb$, ... you'd get infinitely many rules! But we're only allowed finitely many rules!!! Now think about the design of DFAs (or NFAs). The reason why DFAs/NFAs can generate infinitely many strings despite the fact that there are only finitely many states is that there are loops: some part of the computation repeats. OK ... good ... now look at the strings

$$abb, aabbbb$$

Do you see one of them included in the other, as in:

$$abb, a(abb)bb$$

If you still don't see it, look at this:

$$abb, a(abb)bb, a(a(abb)bb)bb$$

and now this:

$$abb, a(abb)bb, a(a(abb)bb)bb, a(a(a(abb)bb)bb)bb$$

Do you see a rippling out effect? You can see that "something" is spitting out an a on the left and a bb on the right:

$$a\Bigg(a\Big(a(a \ldots \leftarrow ? \rightarrow \ldots bb)bb\Big)bb\Bigg)bb$$

And this is coordinated: You can't have just bb on the right without an accompanying a on the left. See it yet? ...

[WARNING: SPOILERS ON THE NEXT PAGE ...]

Try this:

$$S \to aSbb$$

Of course when you're done with producing the string, you need to stop:

$$S \to \epsilon$$

The CFG is

$$S \to aSbb \mid \epsilon$$

Easy, right? □

The above example illustrates an extremely important difference between DFAs and CFGs. In executing a DFA, you walk from the start state to an accept state and each step (transition) you take, you basically process a character. If you think of that as the producing characters for a string, you see that the process is spitting out characters to the left, one character at a time. In the case of a CFG, a variable can be replaced by a string including terminals and variables. For instance in the case of

$$S \rightarrow aSbb$$

you see that S keeps spitting out characters on the left and on the right:

$$S \implies aSbb \implies aaSbbbb \implies \cdots$$

Of course a variable can be replaced with *many* variables too. So if I have the following rules:

$$\begin{aligned} S &\rightarrow aUbVc \\ U &\rightarrow aUbb \\ V &\rightarrow bbVcccc \end{aligned}$$

I can have the following:

$$S \implies aUbVcc \implies a(aUbb)bV \implies a(aUbb)b(bbVcccc)$$

In this case, I have *two* places that can spit out characters left and right

$$...[... \leftarrow U \rightarrow ...]...[... \leftarrow V \rightarrow ...]$$

And of course the variables can be the same, like so:

$$S \rightarrow aSbSc, \;\; S \rightarrow aSbb, \;\; SbbScccc$$

In the above example

$$S \rightarrow aUbVc, U \rightarrow aUbb, V \rightarrow bbVcccc$$

you see U being replaced by a string containing only variable U and V begin replaced by a string containing only variable V. We can even have replacement between U and V like this:

$$S \rightarrow aUbVc, U \rightarrow aUbb \,|\, V, V \rightarrow bbVcccc \,|\, U$$

Does you head hurt yet?

Example 14.2.4. Let's try to see if it's possible to produce this language

$$L = \{a^m b^n \mid m \leq n\}$$

Well if we have a rule like this

$$S \rightarrow aSb$$

(i.e. split one a on the left and one b on the right) we'll get derivations of the form

$$S \implies \cdots \implies a^m S b^m$$

i.e. same number of a's and b's. Correct? Together with $S \rightarrow \epsilon$, we would have the language $\{a^m b^m \mid m \geq 0\}$. That's the idea of our first grammar and it's no good for us because the number of a's and b's in the same. For this example we need the number of b's to be at least the number of a's. Now what?

Well think about it ... we want to allow the possibility of more b's than a's. That means that you want to allow S to produce b (on the right) but no a's at all. That means that you want this as well:

$$S \rightarrow Sb$$

Note that you do *not* want to throw away the option of

$$S \rightarrow aSb$$

(Right? Think about it.) This means that with the following two rules

$$S \rightarrow aSb \mid Sb$$

you can derive $a^m S b^n$ with $m \leq n$. And to kill the S at the final step, you just need to include $S \rightarrow \epsilon$. Altogether the grammar is

$$S \rightarrow aSb \mid Sb \mid \epsilon$$

□

Exercise 14.2.8. Write down a CFG that generates

$$L = \{a^m b^n \mid m < n\}$$

□

Exercise 14.2.9. Write down a CFG that generates

$$L = \{a^m b^n \mid m \leq 2n\}$$

□

Exercise 14.2.10. Write down a CFG that generates

$$L = \{a^m b^n \mid m + 3 \leq 2n\}$$

□

Exercise 14.2.11. Write down a CFG that generates

$$L = \{a^m b^n \mid 2m > 3n\}$$

□

Now we want to look at languages where the a's and b's are not so "orderly" ...

Example 14.2.5. Write down (if possible) a CFG G such that

$$L(G) = \{w \in \Sigma^* \mid |w|_a = |w|_b|\}$$

where $\Sigma = \{a, b\}$ and $|w|_c$ is the number of c' in w where $c \in \Sigma$.

This language of course includes

$$\{a^n b^n \mid n \leq 0\}$$

But there's more ... it includes

$$\{b^n a^n \mid n \leq 0\}$$

or a mumbo-jumbo such as this word

$$b^{100} abaabbaaabbba^{100}$$

Here's another example (a famous one) where the a's and b's are not so orderly.

Example 14.2.6. Consider the symbols (and). A string over (and) is balanced if it corresponds to a correctly parenthesized algebraic expression with the variables and numbers removed. For instance

$$(w + (xy))(z + w)$$

is a valid algebraic expression. If you remove w, x, y, z, you get the string

$$(())()$$

which is a balanced string. Replacing the (and) by a and b, we get the string

$$aabbab$$

Of course this is not balanced:

$$abbaab$$

since this corresponds to parentheses as follows:

$$()\underline{)}(()$$

where I've underlined the leftmost parenthesis that does not match up nicely. It's clear now that to say that x is a balanced string of the symbol (and) is the same as saying that

1. The number of (in x is the same as the number of).
2. If y is a left substring of x (i.e. a prefix substring of x), then the number of (in y is $\geq$ the number of)'s in y.

Now let me replace (with a and) with b ...

We want to find a CFG that generates

$$L = \{w \mid w \text{ is a balanced expression in } a, b\}$$

Whoa!!! Where do we begin? For

$$\{a^n b^n \mid n \geq 0\}$$

we immediately see strings $a^n b^n$ right away ... but *this* ... we need to write down an algebraic expression and then ... !?!

Don't panic. Write down your own examples. For instance we already have this example:

$$aabbab$$

Well this grammar is similar to the one for $\{a^n b^n \,|\, n \geq n\}$. For one thing

$$(^n)^n$$

is balanced, i.e. $a^n b^n$ is in our language L for $n \geq 0$. In other words the grammar for $\{a^n b^n \mid n \geq 0\}$

$$G_0 : S_0 \to aS_0b \mid \epsilon$$

must be included in the grammar for L. But the problem is that

$$aabbab$$

is in L but not generated by G_0. But you immediately see that the first part *is* generated by G_0:

$$\underline{aabb}ab$$

WAIT A MINUTE!!! The second part is also generated by G_0:

$$aabb\underline{ab}$$

Holy cow! Is this fact crucial or is this just a red herring? Well if we use the above example as a guide, maybe our grammar is

$$\begin{aligned} S &\to S_0S_0 \\ S_0 &\to aS_0b \mid \epsilon \end{aligned}$$

It does generate out example string (check!) but it wouldn't work if we have three groups like this:

$$aabbabab$$

which corresponds in terms of parentheses to this:

$$(())()()$$

which is clearly balanced. Well we need to be able to produce any number of S_0. Good idea ... so we modify our grammar to get this:

$$\begin{aligned} S &\to SS_0 \mid \epsilon \\ S_0 &\to aS_0b \mid \epsilon \end{aligned}$$

The first rule now allows us to derive this

$$S \implies \cdots \implies S_0S_0S_0S$$

(in fact any number of S_0) and we can kill the last S using $S \to \epsilon$. After this we just make the first S_0 produce $aabb$, the second S_0 produce ab, and the third S_0 produce ab. Yeah!

Now let's try a few more examples (because you're not the overly optimistic type right?) such as

$$()()(), ((())), (())(), ...$$

Go ahead and do it. [PAUSE]

Unfortunately the grammar does not work for this:

$$aababb$$

which corresponds in parentheses to the string $(()())$.

If you step back and ask yourself what the grammar

$$S \to SS_0 \mid \epsilon$$
$$S_0 \to aS_0b \mid \epsilon$$

can really produce you realize that it does this: It generates any number of S_0's and each S_0 can generate strings of the form a^nb^n. In other words the grammar can generate string that look like this:

$$aaabbb \cdot aabb \cdot aaaabbbb \cdot ab$$

So *of course* it cannot generate the string

$$aababb$$

which corresponds in parentheses to the string $(()())$.

The problem with the above grammar is that once you get to S_0, you must immediately product a^n and b^n, left and right. Now S produce the S_0. What we can do is to allow S_0 to produce like this:

$$S_0 \to S_0S_0$$

then S_0 can become *two* sources of producion of a^n and b^n:

$$S_0 \implies S_0 S_0 \implies \cdots \implies a \cdots S_0 \cdots b \ \cdot \ a \cdots S_0 \cdots b$$

Verifies that it works.

Now if you think about it carefully ... how do we write mathematical expressions containing parentheses? Of course you have

$$(a+b)$$

and you also have

$$((a+b))$$

But in general, we use parentheses to indicate which operator to use during an evaluation:

$$((a+b)*(c-d))$$

From the last mathematical expression you see that "a parenthesized expression" can contain a parenthesized expression:

$$(\cdots()\cdots)$$

and a "parenthesized expression" is made up of the concatenation of two:

$$(\cdots)(\cdots)$$

which comes from the mathematical expresion $(\cdots)+(\cdots)$ (or with $+$ replaced by $-, \cdot, /$. Right? And there are no others except for the empty string. Now if you think carefully about it, you realize that this is enough:

$$S_0 \to S_0 S_0 \mid a S_0 b \mid \epsilon$$

which the start symbol is S_0. (Recall that $S \to S S_0$ is only ro produce multiple S_0's; since we have $S_0 \to S_0 S_0$, the rule $S \to S S_0$ becomes redundant.)

Again if P is a balanced expression then either

1. P is the empty string
2. P is (P') where P' is a balanced expression
3. P is $P'P''$ where P' and P'' are both balanced expressions.

Or, if you follow the format of defining regex ... you would say this ...

A balanced expression in a and b is a string produced (only) in the following way:

1. ϵ
2. If P is a balanced expression, then aPb is also balanced expression
3. If P and P' are balanced expressions, then PP' is also a balanced expression

With this recursive definition of a balanced expression we could have avoided all the experimentation and errors and arrived at the right grammar immediately. However ... realize this: insights are not easily earned by just pure abstract thought. Also pure abstract thought without experimentation to keep the ideas in check can also be dangerous.

Anyway ... read the above example again *very* carefully.

Exercise 14.2.12. Find a grammar G such that $L(G)$ is the language of palindromes over the symbols a, b. [Hint: Think! Try some examples. See if you can write down a recursive definition of a palindrome.]

Exercise 14.2.13. Here's a grammar with two variables.

$$G : \begin{cases} S \rightarrow E \\ E \rightarrow E + E \mid E * E \mid a \end{cases}$$

where $\{S, E\}$ is set of variables, S is the start variable, and $\{+, *, a\}$ is the set of terminals. Terminals are like Σ in DFAs. Here are some string derivations from this G:

(a) $S \implies E \implies E + E \implies E + a \implies a + a$
(b) $S \implies E \implies E + E \implies a + E \implies a + a$
(b) $S \implies E \implies E + E \implies a + E \implies a + E + E \implies a + a + E \implies a + a + a$

Can you derive $a + a * a$ in two different ways? (i.e., choose different orderings of production rules and/or different order of variables to replace.) What about $a + a + a$?

Exercise 14.2.14. Let G be the grammar

$$G : \begin{cases} S \rightarrow M \mid N \\ M \rightarrow aMb \mid aM \mid a \\ N \rightarrow aNb \mid Nb \mid b \end{cases}$$

What is $L(G)$?

14.3 Formal Definition of Context Free Grammars

Now for some high tech math jargon ...

Definition 14.3.1. A **context-free grammar** G is (V, Σ, P, S) where

- V is a finite set of **variables** (or non-terminals)
- Σ is a finite set of **terminals** (some authors use T instead of Σ for this set)
- P is a finite set of **production rules** of the form

  $$A \to \alpha$$

  where $A \in V$ and α is a string over $V \cup \Sigma$, i.e., $\alpha \in (V \cup \Sigma)^*$
- $S \in V$ is the start symbol

The strings in $(V \cup \Sigma)^*$ are called **sentential forms**.

I will write **CFG** for context-free grammar.

Let $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$. We write $\alpha_1 \implies \alpha_2$ if there is a variable A in α_1 and a production rule $P : A \to \beta$ for A such that when the A in α_1 is replaced by β, we get α_2. In other words, we can write $\alpha_1 = \alpha_1' A \alpha_1''$ where $\alpha_1', \alpha_1'' \in (V \cup \Sigma)^*$ and

$$\alpha_1 = \alpha_1' A \alpha_1'' \implies \alpha_1' \beta \alpha_1'' = \alpha_2$$

If this is the case, we say that α_1 derives α_2.

Definition 14.3.2. Given a CFG G, we define $\implies^*$ we follows:

- $\alpha \implies^* \alpha$
- $\alpha \implies^* \beta$ if β can be derived from α in a finite number of steps.

In other words $\alpha \implies^* \beta$ if either α *is* β or you can derive β from α in a finite number of steps. So you can think of "$\alpha \implies^* \beta$" as a shorthand for "$\alpha = \beta$ or $\alpha \implies \cdots \implies \beta$". You can define this relation more formally like this. It is the smallest relation satifying the following two conditions:

- $\alpha \implies^* \alpha$
- If $\alpha \implies^* \beta_1$ and $\beta_1 \implies \beta$, then $\alpha \implies^* \beta_1$.

Sometimes it's useful to say that a derivation requires a certain number of steps. For instance the following involves 3 derivation steps:

$$\alpha \implies \beta \implies \gamma \implies \delta$$

In this case I will write

$$\alpha \implies^3 \delta$$

Note that there might be many different ways to derive δ from α. I'm just saying that there's *a* derivation of δ from α in 3 steps.

It's not difficult to prove:

Exercise 14.3.1. Prove that $\implies^*$ is transitive, i.e., if $\alpha \implies^* \beta$ and $\beta \implies^* \gamma$, then $\alpha \implies^* \gamma$

Definition 14.3.3. Let G be a CFG with start symbol S. Then $L(G)$ is the set of strings of terminals that can be derifved from S, i.e.,

$$L(G) = \{x \mid x \in \Sigma^*, \ \ S \implies^* x\}$$

A language is **context-free** if is is generated by a CFG.

debug:
cfg/cfl-induction.tex

14.4 Proving the Language is Generated by a CFG

Usually once you've written down the CFG G, if the grammar is simple enough, i.e., there aren't too many production rules and the right-hand side of the production rules are not too complicated, the language accepted by G, i.e., $L(G)$ is usually pretty easy to write down.

However when the CFG is complicated, then you will need to prove the the language generated by G is what it is.

This is usually done using ... mathematical induction!!! (Were you expecting something else???) I'll do a simple example so that you can follow the technique.

Let's consider this grammar:

$$G : S \to aSb \mid \epsilon$$

It should be clear by now that

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

But let's prove it using induction anyway.

First let me label the rules so that they are easier to reference: G is the CFG with the following production rules:

$$S \to aSb \tag{1}$$
$$S \to \epsilon \tag{2}$$

We want to prove

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

Let's give the right-hand side a name (so that it's easier to write): Let

$$L = \{a^n b^n \mid n \geq 0\}$$

I want to prove

$$L(G) = L$$

This means proving

$$L(G) \subseteq L \quad \text{and} \quad L(G) \supseteq L$$

PART 1. First let me prove

$$L(G) \supseteq L$$

Let $x \in L$. Then

$$x = a^n b^n$$

for some $n \geq 0$ by definition of L. I want to prove that

$$a^n b^n \in L(G)$$

To prove this by induction, I define $P(n)$ to be the statement

$$P(n): \ \ a^n b^n \in L(G)$$

Now I will prove $P(n)$ is true for all $n \geq 0$ by induction. First I show that $P(0)$ is true. This means I want to show

$$a^0 b^0 \in L(G)$$

Since $a^0 b^0 = \epsilon$, this means I need to prove

$$\epsilon \in L(G)$$

This means I need to find a derivation (using G) for ϵ. That's easy:

$$S \implies \epsilon$$

by production rule (2). Therefore $P(0)$ is true.

Now I assume $P(n)$ is true; the goal is to prove $P(n+1)$. Since $P(n)$ is true, we have

$$a^n b^n \in L(G)$$

This means

$$S \implies^* a^n b^n \tag{3}$$

Using production rule (1) and the above (3), we have

$$\begin{aligned} S &\implies aSb && \text{by production rule (1)} \\ &\implies^* a(a^n b^n)b && \text{by (3)} \end{aligned}$$

Therefore

$$S \implies^* a(a^n b^n)b = a^{n+1} b^{n+1}$$

This implies that $a^{n+1}b^{n+1} \in L(G)$. Therefore $P(n+1)$ is true.

By the principle of Mathematical induction, $P(n)$ is true for all $n \geq 0$. We have proven that $L \subseteq L(G)$.

Part 2. Now I want to prove

$$L(G) \subseteq L$$

Let $x \in L(G)$. I need to show that $x \in L$. Since $x \in L(G)$, we have

$$S \implies^* x \in L(G)$$

I need to show that $x \in L$. To show $x \in L$ is the same as showing that $x = a^n b^n$ for some n. All in all, I need to show:

$$\text{If } S \implies^* x, \text{ then } x = a^n b^n \text{ for some } n \geq 0$$

The induction in this case is usually on the length of the derivation. In others the induction hypothesis is:

$$P(k) : \text{If } S \implies^k x \in L(G), \text{ then } x = a^n b^n$$

for $k \geq 1$. (Of course this P has nothing to do with the previous induction hypothesis P; to disambiguate, you can use something like P' if you like.)

Make sure you see that the induction is on k and not n!!! The two might be the same or totally different for different problem. One is the length of the derivation and another is a parameter describing the form of the word in our language L.

However instead of proving

$$P(k) : \text{If } S \implies^k x \in L(G), \text{ then } x = a^n b^n$$

by induction, I will prove this instead: if

$$S \implies^k x_1 S x_2$$

then $x_1 = a^n$ and $x_2 = b^n$ for some $n \geq 0$ by induction on k. (I let you think about why I prefer this.) So let's go ahead:

Let $P(k)$ be the following statements:

$$P(k) : \text{If } S \implies^k x_1 S x_2 \text{ then } x_1 = a^n, x_2 = b^n \text{ for some } n \geq 0$$

Note that if $k = 0$, then $x_1 = \epsilon = x_2$ and we're done. Suppose $k > 0$. If $k = 1$, then the first production rule was use and

$$S \implies aSb = x_1 S x_2$$

This implies that $x_1 = a$ and $x_2 = b$. Let $k > 1$. Assume that $P(0), ..., P(k)$ is true. Let

$$S \implies^{k+1} x_1 S x_2$$

Since only the first production rule was used, we have

$$S \implies^k y_1 S y_2 \implies y_1 aSb y_2 = x_1 S x_2$$

Now since

$$S \implies^k y_1 S y_2$$

is a derivation of length k, we have $y_1 = a^n$ and $y_2 = b^n$ for some $n \geq 0$. Therefore $x_1 = y_1 a = a^{n+1}$ and $x_2 = b y_2 = b b^n = b^{n+1}$.

Therefore by the principle of mathematical induction, if

$$S \implies^k x_1 S x_2$$

then $x_1 = a^n$ and $x_2 = b^n$.

Now I'll prove $L(G) \subseteq L$. Let $x \in L(G)$. This means that

$$S \implies^k x$$

for some k. Note that the above sequence of derivations is broken up as follows:

$$S \implies^{k-1} x_1 S x_2 \implies x$$

(The last production rule used in the above is the second production rule. To be absolutely formal, you can produce by induction that is a sequence of derivations does not use the second rule, then the resulting sentential form must contain the variable S. This implies that if you reach a string of terminals, the second rule must be used. I'll leave that fun stuff to you.) Previous, I have shown that $x_1 = a^n$ and $x_2 = b^n$. The above becomes

$$S \implies^{k-1} a^n S b^n \implies x$$

Since the last production rule is $S \rightarrow \epsilon$, the above is also

$$S \implies^{k-1} a^n S b^n \implies a^n \epsilon b^n$$

which means that

$$x = a^n b^n$$

Exercise 14.4.1. Why did I prove

$$\text{"If } S \implies^* x_1 S x_2 \text{ then } x_1 = a^n \text{ and } x_2 = b^n\text{"}$$

instead of

$$\text{"If } S \implies^* x \text{ then } x = a^n b^n\text{"}$$

directly? (Hint: Try to prove the second version by induction yourself and see what happens.) □

debug: closure/closure.tex

14.5 Closure Rules

Exercise 14.5.1. Suppose G' and G'' are two CFGs. How would you define a grammar G such that $L(G) = L(G') \cup L(G'')$? This is your closure law for CFL.

Exercise 14.5.2. Suppose G' and G'' are two CFGs. How would you define a grammar G such that $L(G) = L(G')L(G'')$?

Exercise 14.5.3. Suppose G is a CFG. Construct a grammar G' such that $L(G') = L(G)^*$.

So now you know that union, contenation, and the Kleene star are closed operators of CFLs.

Exercise 14.5.4. Let L be a CFL and L' be regular, prove that $L \cap L'$ is a CFL.

[HINT: We can't prove that a language is not context free yet. However, I will tell you that

$$\{a^n b^n c^n \mid n \geq 0\}$$

is not context free.]

Exercise 14.5.5. Prove that the statement "L is a CFL, L' is a CFL implies $L \cap L'$ is a CFL" is not true. $\square$

Exercise 14.5.6. Prove that the statement "L is a CFL implies $\overline{L}$ is a CFL" is not true. □

Exercise 14.5.7. We have seen some examples where non-regular languages are actually accepted by CFGs. We have also seen that some regular languages are accepted by CFGs. Let's try one regular languages accepted by a regex: Find a CFG that generates the same language as the language accepted by the regular expression $101(01^* \cup 10^*)^+(\epsilon \cup 111)$.

[ANOTHER VERSION]

Proposition 14.5.1. *Here is a summary of the closure properties of CFLs.*

(a) CFLs are closed under union, concatenation, Kleene star **, intersection with regular languages*
(b) CFLs are not closed under intersection complement. However, CFLs are closed under "intersection with regular languages." In other words of L *is a CFL and* L' *is regular, then* $L \cap L'$ *is also a CFL.*

Proof. (a): Let L_i $(i = 1, 2)$ be CFL generated by grammar with starting symbol S_i $(i = 1, 2)$. Create a new grammar by taking the obvious unions (for instance the variable set is the union of the variable set of G_1, G_2, etc.)

1. What do you get when you add the production $S \to S_1 \mid S_2$?
2. What do you get when you add the production $S \to S_1 S_2$?
3. What do you get when you add the production $S \to S_1 S \mid \epsilon$?

Of course we are assuming that the variable sets are disjoint and the S is an unused symbol.

The proof for intersection with regular languages is omitted.

(b): Here's why CFLs are not closed under intersection. Let $L_1 = \{a^i b^i c^j \mid i, j \geq 0\}$ and $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$. Then L_1, L_2 are CFLs (prove it!). The intersection is $\{a^i b^i c^i \mid i \geq 0\}$ which is not a CFL.

The reason why CFLs are not closed under complement is easy. Recall the fact we used in the section on closure properties for regular languages: If L_1, L_2 are sets, then

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$$

Why should this help? □

Example 14.5.1. Let $\Sigma = \{a, b\}$. Prove that the language $L = \{ww \mid w \in \Sigma\}$ is not a CFL.

Proof. Suppose L is a CFL. Since intersection with regular languages is a closed operator for CFLs, when we intersect this with the regular language $L(a^*b^*a^*b^*)$ we get the language

$$\{a^ib^ja^ib^j \mid i, j \geq 0\}$$

However this language is not context-free. Hence L is not a CFL. $\square$

Example 14.5.2. Let $\Sigma = \{a, b\}$. Prove that the language

$$L = \{w \mid w \text{ has the same number of } a\text{'s and } b\text{'s}\}$$

is not context-free.

debug: derivation-tree/derivationtree

14.6 Derivation Trees

Just from playing around with some grammars you see that it's possible to derive a string from a CFG in different ways, i.e., the sequence of rules used can be different and yet produce the same string.

Here's a simple example:

$$\begin{aligned} S &\to TU \\ T &\to aTb \mid \epsilon \\ U &\to bUa \mid \epsilon \end{aligned}$$

For instance the string *abba* can be produced in the following ways:

$$S \implies TU \implies aTbU \implies abU \implies abbUa \implies abba$$

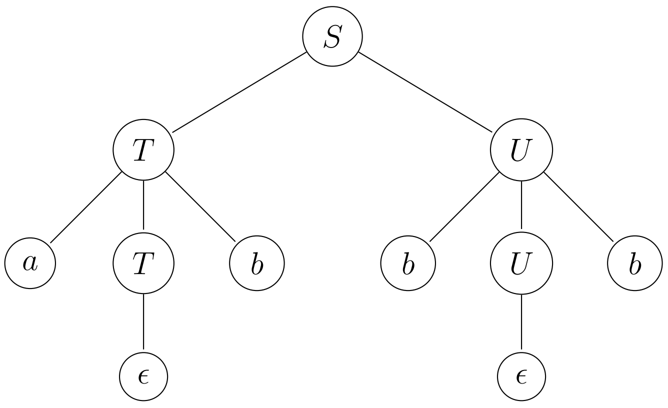
and

$$S \implies TU \implies TbUa \implies Tba \implies aTbba \implies abba$$

One way to illustrate the derivations is to draw a parse tree (or derivation tree). For the derivation

$$S \implies TU \implies aTbU \implies abU \implies abbUa \implies abba$$

the parse tree is



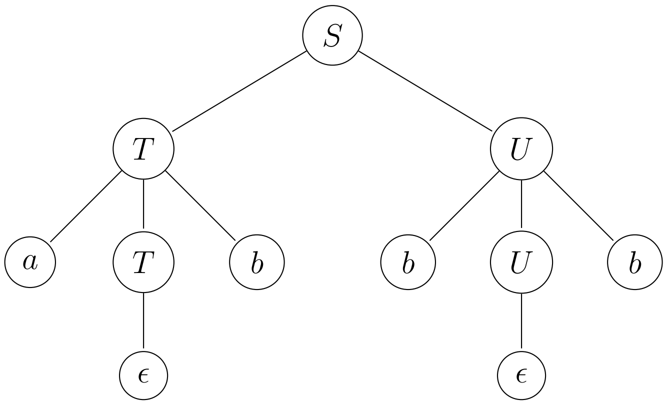
I don't need to explain how to draw the parse tree, right? Anyway the parse tree of the other derivation

$$S \implies TU \implies TbUa \implies Tba \implies aTbba \implies abba$$

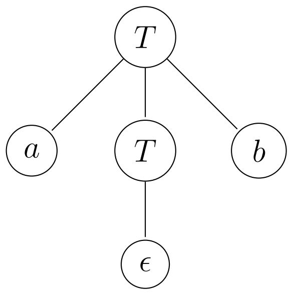
is also the same tree.

Exercise 14.6.1. Write down a derivation for ab^3a^2 for the above CFG. For your derivation, draw the parse tree.

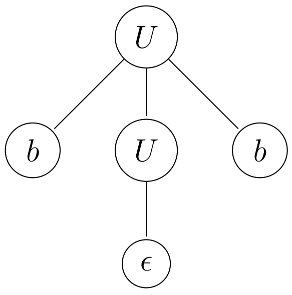
We can think of the two above derivations are more or less the same. The point is that the products following does not really interact or affect the productions following U: They are really in different *subtrees* of the parse trees



where one of the subtree (the left one) is



and the other is



The only difference between the two derivations is only in the choice of which part of the subtree to expand first to get to terminals. Therefore in that sense drawing parse trees for derivation is the right thing to do if we really want to distinguish between derivations.

Of course to fix the way we write derivations (in other words to limit the number of derivations so as to avoid confusion), we can say that we want to always work on subtrees from left to right. That would correspond to always performing derivation on the leftmost variable of a sentential form. We call such a derivation a **leftmost** derivation. For the above derivations, this one:

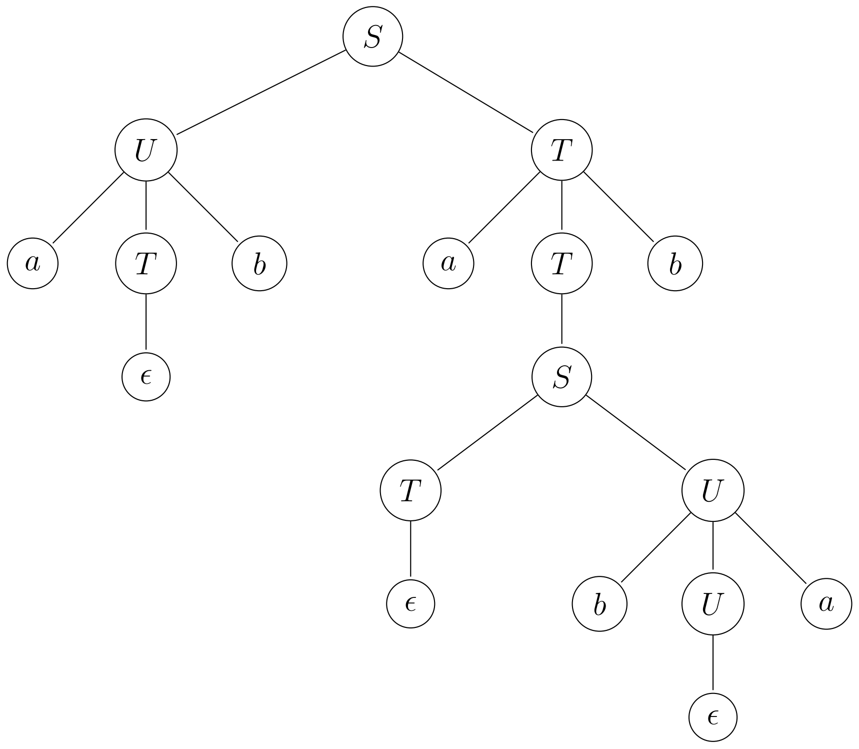
$$S \implies TU \implies ATbU \implies abU \implies abbUa \implies abba$$

is a leftmost derivation. (Right?)

Exercise 14.6.2. Consider the following CFG:

$$\begin{aligned} S &\to TU \mid UT \\ T &\to aTb \mid S \mid \epsilon \\ U &\to bUa \mid \epsilon \end{aligned}$$

Now suppose you're given the following derivation/parse tree:



Write down a left derivation that produces the above tree. How many left derivations can produce this tree? Now do the same for right derivations.

A derivation where the rightmost variable is always chosen to be "processed" is called a **rightmost** derivation (duh).

Some books use the following notation to indicate that the derivation is done using leftmost derivations:

$$S \Longrightarrow_{lm} TU \Longrightarrow_{lm} \cdots$$

(lm = leftmost).

Look at the above grammar. If we fix derivations so that we only talk about *leftmost* derivations, it should be easy to see that for each string generate by the CFG, there is exactly one derivation. (Same for rightmost.) Correct? Think about it and doodle with a couple of examples to see why.

That seems to settle all confusion of multiple derivations for a single string ... or does it???

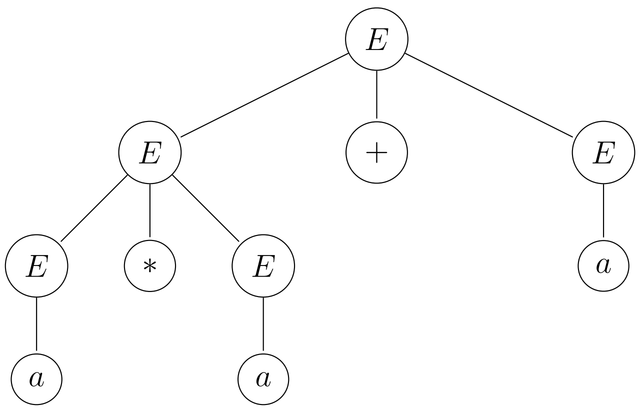
Consider the grammar

$$E \to E + E \mid E * E \mid a$$

Let's derive the string $a * a + a$. Here's one way of doing that:

$$E \implies E + E \implies E * E + E \implies a * E + E \implies a * a + E \implies a * a + a$$

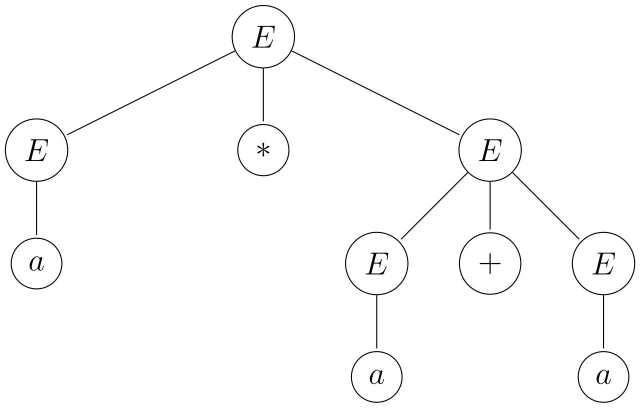
(Exercise: Check the above derivation is leftmost.) Here's the derivation tree for the above derivation:



Here's another derivation:

$$E \implies E * E \implies a * E \implies a * E + E \implies a * a + E \implies a * a + a$$

(Exercise: Check that the above derivation is leftmost.) Let's draw the derivation tree:



Whoa ... the two trees look different. In a sense, the two different sequences of rules applied (even when we are doing a leftmost derivation) are fundamentally different.

So what? Other than being an academic curiosity ...

Recall that regular expressions can be used to match string patterns. CFG can be used to model arithmetic expressions (like the above). The very important point to remember is this: as you walk state to state starting from the start state and ending with the accept state of a DFA, you consume character at each step; the character will ultimately make up the string accepted by the DFA if an accept state is reached. The special thing about CFG is that as you go from derivation to derivation, a variable can be replaced by a string (not just a character). So in the above example we have:

$$E \to E + E$$

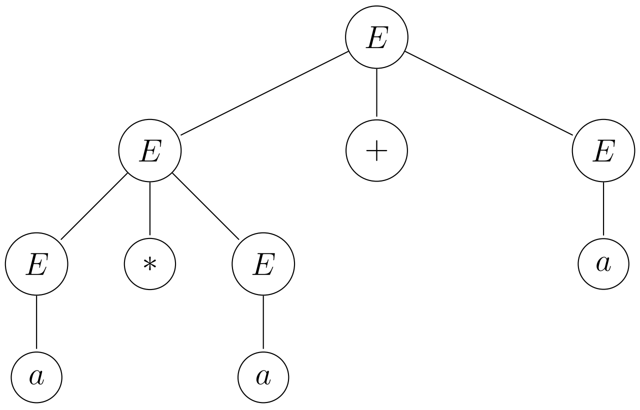
You can (and should!!!) think of string productions in CFG's (in each production step) as expansion in *two* directions - left and right.

$$...V... \implies ...[\leftarrow V \rightarrow]...$$

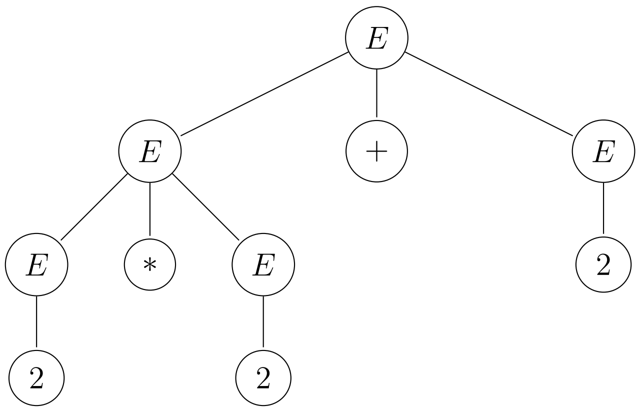
Think for a moment of our first grammar

$$S \implies aSb \implies aaSbb \implies aaaSbbb$$

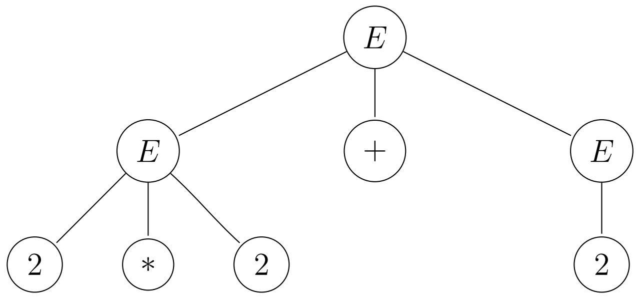
It is because of this that CFG's can be used to model arithmetic expressions. So for instance the first derivation above game us this tree



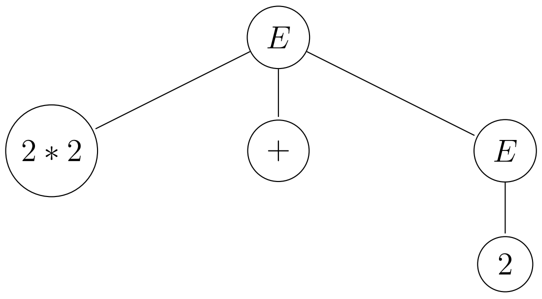
you can ask "What is the value of the expression $a*a+a$ using this tree". Well ... in an expression containing several operators, the operator precedence is important. For the above derivation tree, when viewed as evaluating a mathematical expression (not just as a string anymore), the resulting expression is obtained by going from the bottom symbols up the tree. For instance (just to be concrete) if I replace by a by the number (not string) 2 and $*$ and $+$ by multiplication and addition, then



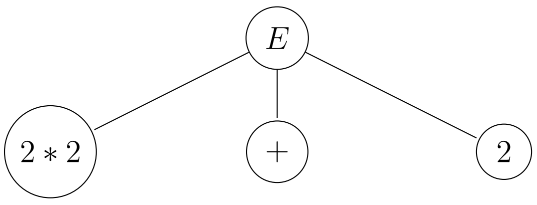
becomes



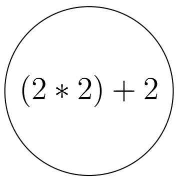
and then



and then



and finally

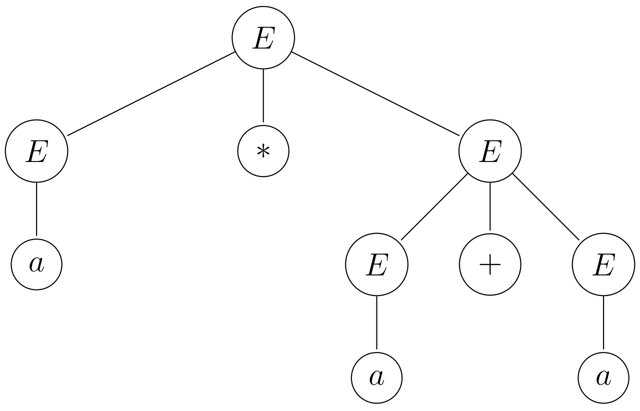


I think you get it.

So ... as a *mathematical* expression, the value modeled by the tree is

$$(a * a) + a$$

On the other hand, for the tree

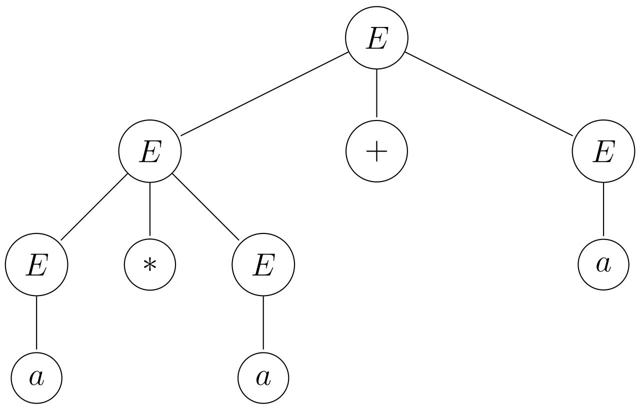


the corresponding mathematical expression is

$$a * (a + a)$$

Remember that $(a * a) + a$ and $a * (a + a)$ is not viewed as a mathematical expression; in the CFG, and during the derivation process, everything is just abstract grammar variables and terminals and of course the two derivations produce the same *string* $a * a + a$.

Obviously from the above trees, following the rule that evaluation of going from bottom symbols up to the top, and we want the resulting expression to be mathematically correct based on our precedence rules, we want to have this tree:



and not the other one. You know now that different grammars can generate the same language. So the really important question is this: can we rewrite grammar so that the "wrong tree" will never appear?

This means that I want to write a CFG that generate mathematical expressions *and* the derivations must produce derivation (parse) trees that can be used to correctly evaluate expressions according to precedence rules.

The idea is very simple. Look at the above parse trees. Because of the precedence rules, $*$ must appear lower in the tree than $+$ in order for $*$ to be evaluated first. Right? Try this:

$$\begin{aligned} &S \to S + S \mid a \mid T \\ &T \to T * T \mid a \end{aligned}$$

Now let's try to derive $a*a+a$ (using left derivations only). We have no choice but to do this first:

$$S \implies S + S$$

(had we chosen $S \implies a$, the derivation would stop and would be incorrect! ... and if we'd chosen $S \to T$, we would not be able to derive the $+$ terminal!)

What next? Again by a process of elimination, the only possible rule is

$$S \implies S + S \implies T + S$$

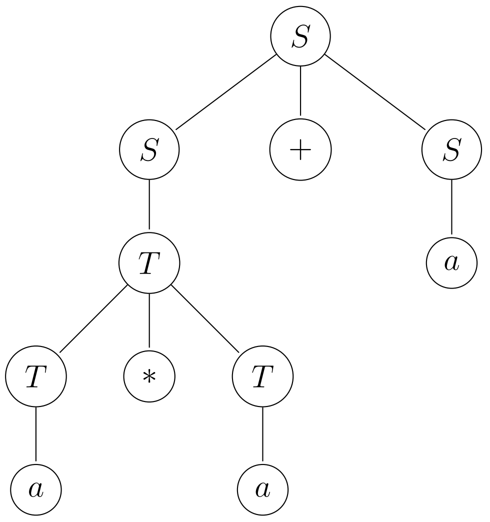
That would move the leftmost variable T to the next "level". We then must use

$$S \implies S + S \implies T + S \implies T * T + S$$

At this point it's clear that the remaining steps must be

$$S \implies S{+}S \implies T{+}S \implies T{*}T{+}S \implies a{*}T{+}S \implies a{*}a{+}S \implies a{*}a{+}a$$

and the derivation tree is



That solves the precedence of certain terminals (or operators if the terminals corresponds to mathematical objects in real-life applications). But there's another issue. Look at the situation where there's only one operator in the mathematical expression:

$$a + a + a$$

In math, this means

$$(a + a) + a$$

In math techno jargon, we say that + is left associative. An operator, say $\oplus$, is said to be right associative if

$$a \oplus b \oplus c = a \oplus (b \oplus c)$$

Note that + for real numbers will result in the same value whether you evaluate it as left or right associative. However it does matter for some operators. For instance $(1 - 2) - 3$ is *not* the same as $1 - (2 - 3)$!!!

By the way, I hope you know that precedence and associativity rules are really rules of convention; we simply say that $a + b + c$ means $(a + b) + c$. There's no deep reason why it has to be other than the fact that we read left to right.

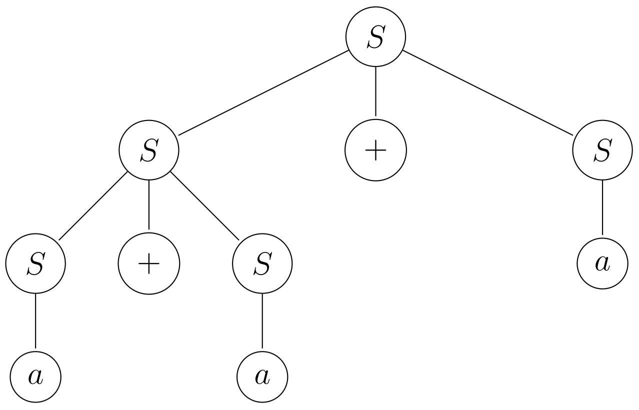
In terms of a string $a + a + a$, we have the following left derivations:

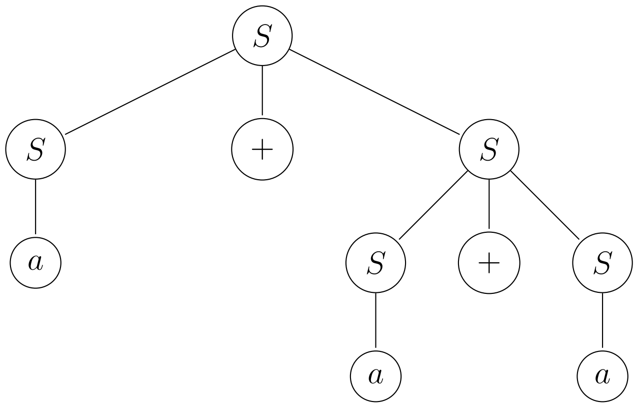
$$S \implies S + S \implies S + S + S \implies a + S + S \implies a + a + S \implies a + a + a$$

and

$$S \implies S + S \implies a + S \implies a + S + S \implies a + a + S \implies a + a + a$$

The following are the derivation trees:





and mathematically, the expressions are (respectively)

$$(a + a) + a \qquad a + (a + a)$$

So we need to modify our CFG to only allow the left parse tree. This is easy: we use left recursion:

$$S \to S + a$$

and our CFG becomes

$$\begin{aligned} S &\rightarrow S + a \mid a \mid T \\ T &\rightarrow T * T \mid a \end{aligned}$$

Mathematically, $*$ is also left associative so ...

$$\begin{aligned} S &\rightarrow S + a \mid a \mid T \\ T &\rightarrow T * a \mid a \end{aligned}$$

Note that not all operators are left associative. For

$$2^{3^4}$$

means

$$2^{(3^4)}$$

if we write $\wedge$ for exponentiation,

$$2 \wedge 3 \wedge 4$$

means

$$2 \wedge (3 \wedge 4)$$

i.e., $\wedge$ is right associative.

Exercise 14.6.3. Add $\wedge$ to our CFG. In terms of precedence

$$\wedge > * > +$$

Alrighty, all the above techniques for modifying CFGs so that precedence and associativity are encoded in the rules should be enough.

Note that

(a) Each derivative tree has a unique leftmost derivation
(b) A string might have two distinct leftmost derivation and hence two distinct derivation trees.

Definition 14.6.1. (a) A CFG G is **ambiguous** if there is a string in $L(G)$ that has two distinct leftmost derivations and hence two distinct derivation trees.

(b) A context free language L is **inherently ambiguous** if for all CFG G with $L(G) = L$, G is ambiguous.

Here aare some bad news ...

Theorem 14.6.1. *(a) There exists inherently ambiguous languages.*
(b) There is no algorithm that can determine whether or not G is ambiguous.
(c) There is no algorithm that can determine, given a CFG G, whether $L(G)$ is inherently ambiguous.

The most famous inherently ambiguous language is $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$. We probably won't have time to prove all the above results.

debug: towardnf/towardnf.tex

14.7 Toward Normal Forms

We want to restrict the form of the productions. The reason is that we want the top-down and bottom-up parsers to terminate. If we have time we'll talk about these but they really belong the to a course such as compiler construction.

Anyway, the goal is to eliminate bad productions. Here are the bad guys:

Definition 14.7.1. (a) An **ϵ-production** is a production of the form $A \to \epsilon$ where A is a variable.
(b) A **unit production** is a production of the form $A \to B$ where A and B are variables.
(c) A variable A is **useless** (nice technical term, isn't it?) if A is not used in the derivation of any string of terminals, i.e., if there is no $x \in T^*$, $\{\alpha, \beta\} \subseteq (V \cup T)^*$ such that

$$S \overset{*}{\Longrightarrow} \alpha A \beta \overset{*}{\Longrightarrow} x$$

Why are they bad?

Example 14.7.1. Consider

$$\begin{aligned} S &\to AB \\ A &\to \epsilon \\ B &\to b \end{aligned}$$

First of all $A \to \epsilon$ (an ϵ–production) is redundant since the above CFG generate the same language as

$$\begin{aligned} S &\to B \\ B &\to b \end{aligned}$$

Correct? The point is that in the above example, the $A \to \epsilon$ is an ϵ-production and can be removed: A is going to disappear during the derivation process anyway:

$$S \implies^* ...A... \implies^* ...\epsilon...$$

Note however that the rule

$$S \to \epsilon$$

cannot be be removed: your grammar would lose it's ability to generate ϵ!!!

Example 14.7.2. Next consider the grammar

$$\begin{aligned} S &\to A \\ A &\to B \\ B &\to Bb \mid \epsilon \end{aligned}$$

The unit products $A \to B$ is redundant. The above grammar generates the same language as

$$S \to B, \;\; B \to Bb \mid \epsilon$$

Example 14.7.3. The last example shows you examples of useless symbols:

$$\begin{aligned} S &\to \epsilon \mid aS \mid A \\ B &\to b \end{aligned}$$

A is useless because it does not derive terminals; B is useless because it cannot be reached from S.

The above are over-simplied examples. But it will help you in understanding the general situation.

14.7.1 ϵ-Productions

First we will get rid of the ϵ-productions. First we need to compute the set of nullable variables:

Definition 14.7.2. A variable A is **nullable** if $A \implies^* \epsilon$. A more recursive definition: A is nullable if either $A \to \epsilon$ or $A \to \alpha$, a product of variables, and each variable in α is nullable. NULLABLES is the set of all nullables of the grammar.

The algorithm to compute the set of nullables is easy:

```
(1) Let NULLABLES = $\{V \mid V \to \epsilon\}$ ($\epsilon$-productions)
(2) Repeat
    (2.1) if $V \notin$ NULLABLES, $V \to \alpha$ is a production, $\alpha$ is a product of
          variables in NULLABLES, then add $V$ to NULLABLES.
(3) Until NULLABLES stops changing
```

Here's the algorithm to get rid of ϵ-productions. Let G be the grammar.

(1) Compute NULLABLES
(2) For each production $A \to X_1 \ldots X_n$ where $X_i \in V \cup T$, create all possible productions $A \to \alpha_1 \cdots \alpha_n$ where

$$\alpha_i = \begin{cases} X_i & \text{if } X_i \notin \text{NULLABLES} \\ X_i \text{ or } \epsilon & \text{if } X_i \in \text{NULLABLES} \end{cases}$$

(3) Eliminate all $V \to \epsilon$ productions.

Let G' be the new grammar. The resulting grammar generates the same language except that if $S \to \epsilon$ was removed, then the new grammar will not generate ϵ. In other words if $S \to \epsilon$ was removed, then

$$L(G') = L(G) - \{\epsilon\}$$

Otherwise

$$L(G') = L(G)$$

What do you do if G does generate ϵ? You perform the above operation to get G' and to G' you add the only exception $S \to \epsilon$. This would be the only ϵ–production. In this case, G' is in Chomsky normal form except for $S \to \epsilon$.

In general when an author says that G does not have ϵ production rules, he/she meant that the grammar does not have ϵ production rules *except possibly for* $S \to \epsilon$. You just have to look at the context very carefully.

Example 14.7.4. Let G be the following context-free grammar:

$$\begin{aligned} S &\to ACA \\ A &\to aAa \mid B \mid C \\ B &\to bB \mid b \\ C &\to cC \mid \epsilon \end{aligned}$$

1. What is the NULLABLES for the grammar
2. Eliminate the ϵ-productions.

Solution. First we compute the NULLABLES.

1. First of all the only production rule directly of the form $X \to \epsilon$ is $C \to \epsilon$. So we have
$$\text{NULLABLES} = \{V \mid V \to \epsilon\} = \{C\}$$
2. Next, we look for new production rules of the form $X \to C$. The only one is $A \to C$. So the NULLABLES at this point is
$$\text{NULLABLES} = \{C, A\}$$
3. Repeating the process again, we look for new production rules of the form $X \to w$ where w is a product of variables in NULLABLES $= \{C, A\}$. The only new rule is $S \to ACA$. Therefore at this point we have
$$\text{NULLABLES} = \{C, A, S\}$$
4. The process stops.

Second, I will perform substitutions on the nullable variables.

1. First I'm going to handle C. By performing the substitution of C by ϵ to get new rules, the grammar
$$\begin{aligned} S &\to ACA \\ A &\to aAa \mid B \mid C \\ B &\to bB \mid b \\ C &\to cC \mid \epsilon \end{aligned}$$

becomes

$$\begin{aligned}
S &\rightarrow ACA|\underline{AA} \\
A &\rightarrow aAa \mid B \mid C \mid \underline{\epsilon} \\
B &\rightarrow bB \mid b \\
C &\rightarrow cC \mid \underline{c} \mid \epsilon
\end{aligned}$$

For readability, I've underlined the right-hand side of the new rules. For instance, the original rule

$$S \rightarrow ACA$$

gives me the new rule

$$S \rightarrow A\epsilon A = AA$$

The reason is that in any derivation, C can be replaced by ϵ because of the rule $C \rightarrow \epsilon$. Right? The whole point of doing this is that later, I want to remove the rule $C \rightarrow \epsilon$ from the grammar. Note that I cannot throw away the rule original $S \rightarrow ACA$; it must be included. Why? Because a derivation that uses $S \rightarrow ACA$ need not follow up with the rule $C \rightarrow \epsilon$. Right? Oh by the way, notice that at this stage, we inherit the ϵ-product rule $A \rightarrow \epsilon$. Think about that: it should be pretty obvious why this is the case.

2. Now I'm going to do the same for A. The grammar becomes:

$$\begin{aligned}
S &\rightarrow ACA \mid \underline{CA} \mid \underline{AC} \mid \underline{C} \mid AA \mid \underline{A} \mid \underline{\epsilon} \\
A &\rightarrow aAa \mid \underline{aa} \mid B \mid C \mid \epsilon \\
B &\rightarrow bB \mid b \\
C &\rightarrow cC \mid c \mid \epsilon
\end{aligned}$$

For instance the rule $S \rightarrow ACA$ gives rise to the following new rules:

$$S \rightarrow \epsilon CA \mid AC\epsilon \mid \epsilon C\epsilon$$

i.e.,

$$S \rightarrow CA \mid AC \mid C$$

3. Finally, I do the same for S. However in this case, since S does not appear on the right-hand side of any rule, there are no new rules.

At this point, the grammar is

$$\begin{aligned}
S &\to ACA \mid CA \mid AC \mid C \mid AA \mid A \mid \epsilon \\
A &\to aAa \mid aa \mid B \mid C \mid \epsilon \\
B &\to bB \mid b \\
C &\to cC \mid c \mid \epsilon
\end{aligned}$$

The last stage is to remove the ϵ–production rules. I get

$$\begin{aligned}
S &\to ACA \mid CA \mid AC \mid C \mid AA \mid A \\
A &\to aAa \mid aa \mid B \mid C \\
B &\to bB \mid b \\
C &\to cC \mid c
\end{aligned}$$

Note that I removed $S \to \epsilon$. The resulting grammar does not have any ϵ-rule but it does generate ϵ (S is in Nullables). This means that the resulting grammar generates the same language except that it's missing the single word ϵ. So I have to make the exception for the rule $S \to \epsilon$. Adding $S \to \epsilon$ back, I get a grammar that does not have any ϵ-production other than the rule $S \to \epsilon$:

$$\begin{aligned}
S &\to ACA \mid CA \mid AC \mid C \mid AA \mid A \mid \epsilon \\
A &\to aAa \mid aa \mid B \mid C \\
B &\to bB \mid b \\
C &\to cC \mid c
\end{aligned}$$

Example 14.7.5. Remove ϵ–production rules from the following grammar:

$$\begin{aligned} S &\to ABC \\ A &\to aA \mid \epsilon \\ B &\to bB \mid \epsilon \\ C &\to cC \mid \epsilon \end{aligned}$$

14.7.2 Unit Productions

The next thing to do is to get rid of unit productions. We will assume that $L(G) \neq \emptyset$. Furthermore we will assume that G has no ϵ-productions. The main idea, like the removal of ϵ-productions, is very simple. Suppose you do have a unit production $A \rightarrow B$. Furthermore suppose $B \rightarrow b$. Then of course A can derive b. So if we include $A \rightarrow b$ as a production, we don't really need the unit production $A \rightarrow B$. Of course you need to look at all possible derivations from A to a terminal string that uses B. For instance there might be another production $B \rightarrow b'$. So here's the algorithm:

(1) For every pair of variables A and B, if $A \implies^* B$:
   (1.1) Add productions $A \rightarrow \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_n$ where $B \rightarrow \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_n$ are all the nonunit productions for B (β_i are sentential forms).
(2) Eliminate all unit productions.

Note that the above algorithm requires verification of $A \implies^* B$. I'll show you how to do that later. First let's prove the following:

Theorem 14.7.1. *Suppose G is the original CFG and G' is the CFG obtained after removal of ϵ-productions and unit productions. Then $L(G) = L(G')$.*

Proof. The fact that $L(G') \subseteq L(G)$ is clear: If $x \in L(G')$ and x is derived without using the new productions, then of course $x \in L(G)$. If x is derived using an new production, say $A \rightarrow b$, then by definition of G', we know that $A \implies^* {}_G B$ (derivation in G) and $B \rightarrow b$ is a production in G. Hence $A \implies^* {}_G b$ (derivation in G). In other words x can be derived using G, i.e., $x \in L(G)$.

Now we prove that $L(G) \subseteq L(G')$. Let $x \in L(G)$. The idea is very simple. Let's look at a leftmost derivation of x which is derived using G and that it involves at least one unit production:

$$S \implies^* wA\alpha \rightarrow wB\alpha \rightarrow wC\alpha \implies^* wD\alpha \implies \ldots \implies x \rightarrow w\gamma\alpha$$

where A, B, C, D are variables, w, w' are all terminal strings, and α, γ are sentential forms (i.e., strings involving variables and terminals); note that we see the unit productions $A \rightarrow B$, $B \rightarrow C$, etc. AHA! This tells us that $A \implies^* D \rightarrow \gamma$ which by our definition of G' implies that $A \rightarrow \gamma$ is in fact in G'!!! In other words the following derivation

$$S \implies^* wA\alpha \implies wD\alpha \rightarrow w\gamma\alpha \implies \ldots \implies x$$

has some unit productions removed and it uses a production *in* G'. Using this idea we can slowly change the original derivation using G to one using G': Just apply the above idea to the first time a unit production appears in the derivation and repeat this process to the next unit product, etc. Since a derivation is finite in length, obviously the process terminates and we have a derivation *in* G. Voila.

□

There's still the problem of verifying whether $A \implies^* B$. The idea is also simple. Suppose G does not have any ϵ-productions. For $A \implies^* B$ to occur, the derivation must involve only unit productions:

$$A \implies X_1 \implies X_2 \implies \cdots\cdots \implies X_n \implies B$$

(Why?). So all you need to do is to make a (directed) graph where the nodes are all the variables of the grammar and $X \to Y$ is an edge in the graph if $X \to Y$ is a production in the grammar. Then $A \implies^* B$ iff there is a directed path in the graph from node A and node B. To check if there is a path from node A to node B, do a depth-first search. That's it!!!

Example 14.7.6. Consider the grammar

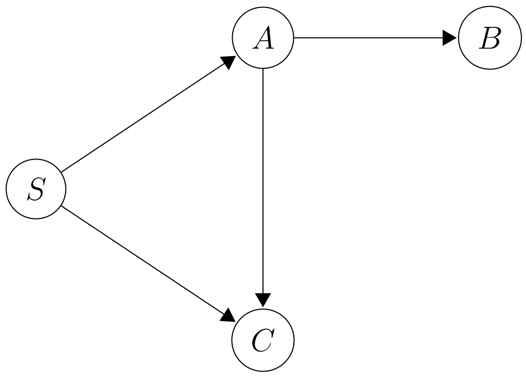
$$\begin{aligned} S &\to ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \epsilon \\ A &\to aAa \mid aa \mid B \mid C \\ B &\to bB \mid b \\ C &\to cC \mid c \end{aligned}$$

Other than $S \to \epsilon$, this grammar does not have any ϵ–productions.

First we need to know when a symbol can derive another. The unit production rules are

$$\begin{aligned} S &\to A \\ S &\to C \\ A &\to B \\ A &\to C \end{aligned}$$

and here's the directed graph of variables taking part in unit productions:



$$\begin{tikzcd} & A \arrow[r] \arrow[dd] & B \\ S \arrow[ur] \arrow[dr] & & \\ & C & \end{tikzcd}$$

The relevant derivations are

$$\begin{aligned} S &\overset{*}{\Longrightarrow} A \\ S &\overset{*}{\Longrightarrow} B \\ S &\overset{*}{\Longrightarrow} C \\ A &\overset{*}{\Longrightarrow} B \\ A &\overset{*}{\Longrightarrow} C \end{aligned}$$

So here are the productions we will add

Unit derivation	Production to add
$S \overset{*}{\Longrightarrow} A$	$S \to aAa \mid aa$
$S \overset{*}{\Longrightarrow} B$	$S \to bB \mid b$
$S \overset{*}{\Longrightarrow} C$	$S \to cC \mid c$
$A \overset{*}{\Longrightarrow} B$	$A \to bB \mid b$
$A \overset{*}{\Longrightarrow} C$	$A \to cC \mid c$

Hence we now have the following productions:

$$
\begin{aligned}
&S \to ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \epsilon \mid aAa \mid aa \mid bB \mid b \mid cC \mid c\\
&A \to aAa \mid aa \mid B \mid C \mid bB \mid b \mid cC \mid c\\
&B \to bB \mid b\\
&C \to cC \mid c
\end{aligned}
$$

And now we remove the unit productions to get

$$
\begin{aligned}
&S \to ACA \mid CA \mid AA \mid AC \mid \epsilon \mid aAa \mid aa \mid bB \mid b \mid cC \mid c\\
&A \to aAa \mid aa \mid bB \mid b \mid cC \mid c\\
&B \to bB \mid b\\
&C \to cC \mid c
\end{aligned}
$$

This grammar does not have ϵ–productions (other than $S \to \epsilon$) or unit productions.

You note that in general, just like the process for ϵ–productions, more productions are added to the grammar.

14.7.3 Useless Symbols

Finally we need to get rid of the useless symbols. Recall that a symbol A is useless if it is not used in any derivation. Formally, A is useless if there does not exist α (sentential form), β (sentential form), x (terminal string) such that

$$S \overset{*}{\Longrightarrow} \alpha A \beta \overset{*}{\Longrightarrow} x$$

The main idea is simple: Symbols which are useful are those which can derive a terminal string and also, beginning with the starting symbol, you can reach a sentential form containing that symbol. Formally, A is useful if

(a) $A \overset{*}{\Longrightarrow} x$ for some terminal string x
(b) $S \overset{*}{\Longrightarrow} \alpha A \beta$ for sentential forms α, β.

We will first focus on (a). Again the algorithm is recursive: We start with symbols A that can derive terminals in one step, then we consider productions $A \to \alpha$ where α is a string of terminals or variables in our set. We continue until the set of symbols does not change. Here's the algorithm:

```
(1) Let TERMINAL be the set of variables $A$ such that there is a production
    $A \to x$ where $x$ is a terminal string
(2) Repeat
    (2.1) For each product $A \to \beta$ where $\beta$ is a string of terminals or symbols
          in TERMINAL, put $A$ into TERMINAL.
(3) Until TERMINAL stops growing
(4) Construct the new grammar with productions that use only TERMINAL$\cup$
    $\Sigma$. The variable set of the new grammar is TERMINAL. The new $\Sigma$ is of
    course the set of terminal symbols that appear in the new grammar.
```

It's clear that at the end of executing this algorithm, Terminal contains variables A such that $A \overset{*}{\Longrightarrow} x$ for some terminal string x.

Now we focus on (b) from the above, i.e., we want to find variables A such that $S \overset{*}{\Longrightarrow} \alpha A \beta$. Such variables are said to be reachable. The idea is again recursive. We continually build a set of variables and set of terminals.

```
(1) Set $V' = \{S\}$ and $T' = \emptyset$
(2) Repeat
    (2.1) If $A \in V'$ and $A \to \alpha_1 \mid \ldots \mid \alpha_n$ in $G$, then add the variables of $\alpha_1$,
          $\ldots$, $\alpha_n$ to $V'$ and add terminals of $\alpha_1, \ldots, \alpha_n$ to $T'$
(3) Until $V'$ and $T'$ stop changing
(4) Let $P'$ be the set of productions using only $V'$ and $T'$.
```

Theorem 14.7.2. *If G is a CFG and $L(G) \neq \emptyset$, then we can find a CFG G'' such that $L(G'') = L(G)$ such that G'' has no useless symbols. In particular, G'' is obtained from G by applying the above algorithms (a) and (b).*

Combining everything above we get this:

Theorem 14.7.3. *Given a CFG G such that $L(G) \neq \emptyset$ we can find a CFG G' such that $L(G') = L(G) - \{\epsilon\}$ and G' has no ϵ-productions, no unit productions, and no useless symbols.*

The new G' is of course derived from G by removing ϵ-productions, unit productions, and useless symbols using the above algorithms.

There are two standard forms of CFG:

1. Chomsky Normal Form: Every production is either $A \rightarrow BC$ or $A \rightarrow a$ where A, B, C are variables and a is a terminal symbol
2. Greibach Normal Form: Every production is of the form $A \rightarrow b\alpha$ where A is a variable, b is a terminal symbol, and α is a string of variables.

debug:
chomsky/chomsky.tex

14.8 Chomsky Normal Form

A grammar is in **Chomsky Normal Form** if every production is of the form

$A \rightarrow BC$	A, B, C are variables
$A \rightarrow a$	A is a variable and a is a terminal

Note that ϵ is not in the language generated by such a language.

For a CFL language that does containing ϵ, you remove ϵ from the language, analyze *that* language. Once you're done, you add production $S' \rightarrow S \mid \epsilon$ to the grammar. This is *almost* a Chomsky Normal Form.

The Chomsky Normal Form is important because it gives rise to efficient algorithms. For instance we will use it in the CYK (Cocke-Younger-Kasami) algorithm which tests if a string can be generated by a grammar; the CYK algorithm also produces the actual derivation for that string.

Theorem 14.8.1. *Every CFL that does not have the ϵ string has a CFG in Chomsky Normal Form.*

Proof. Suppose G is the CFG for a language L without ϵ. We remove the ϵ-productions, the unit productions, and the useless symbols.

Now let's look at the productions $A \rightarrow X_1 \cdots X_n$ ($n \geq 1$) where X_i are either variables or terminals. For the case of $n = 1$, the production $A \rightarrow X_1$ is already of the right form, so we don't have to anything to it. (Why??)

What about the case of $n \geq 2$? What would you replace

$$A \rightarrow X_1 \cdots X_n$$

with so that the resulting grammar generates the same language and the new productions are in Chomsky Normal Form?

Let's look at the case of $A \rightarrow X_1 X_2$. What are the possible cases for X_1 and X_2?

What about the case say $A \rightarrow BcC$ where B, C are variables and c is a terminal. What would you do?

In general, there are two more steps in converting the grammar to a Chomskey Normal Form.

(1) Convert the grammar to one involving productions of the form $A \to a$ (a terminal) and $A \to X_1 \cdots X_n$ where X_i are *variables.*
(2) Convert the productions $A \to X_1 \cdots X_n$ to those involving $A \to A \to YZ$ where Y, Z are variables.

Here's step 1. Let $A \to X_1 \dots X_n$.

(1.1) If X_i is a terminal symbol, create a variable for it, say Y_i. Otherwise Y_i is the same as X_i. Replace $A \to X_1 \cdots X_n$ with $A \to Y_1 \cdots Y_n$.
(1.2) Add productions $Y_i \to X_i$ if X_i is a terminal.

That sounds complicated but it's actually very simple. Let's look at an example. Suppose you look at the production

$$A \to BbCcdD$$

where A, B, C, D are variables and b, c, d are terminals. If you follow the above recipe, you will get the following ...

You create the variables V_b, V_c, and V_d. The above production

$$A \to BbCcdD$$

can be replaced by

$$A \to BV_bCV_cV_dD$$

together with

$$V_b \to b$$
$$V_c \to c$$
$$V_d \to d$$

The first new production

$$A \to BV_bCV_cV_dD$$

is not allowed for Chomsky Normal Forms. The others:

$$V_b \to b$$
$$V_c \to c$$
$$V_d \to d$$

are OK. So we need to fix

$$A \to BV_bCV_cV_dD$$

which is of the form $A \to$ (bunch of variables).

Now for step 2. We only need to consider productions of the form $A \to X_1 \ldots X_n$ where X_i are all variables. Consider the case where $n > 2$. (Why?) How would you replace this production with? [Hint: $X_1 \cdots X_n = X_1(X_2 \cdots X_n)$].

Easy!!!

For instance for

$$A \to UVW$$

where U, V, W are variables just look at the product rule like this:

$$A \to U(VW)$$

You just create a variable Z and replace the above with

$$A \to UZ$$

and

$$Z \to VW$$

Going back an earlier example, this production rule

$$A \to BV_bCV_cV_dD$$

(which is not allowed in Chomsky normal form) can replaced with the following rules:

$$\begin{aligned} A &\to BV_1 \\ V_1 &\to V_bV_2 \\ V_2 &\to CV_3 \\ V_3 &\to V_cV_4 \\ V_4 &\to V_dD \end{aligned}$$

which are allowed in Chomsky normal form.

That's it! Easy, right?

Exercise 14.8.1. Convert the following grammar to Chomsky Normal Form:

$$\begin{aligned}S &\to aA \mid bB \mid b \\ A &\to Baa \mid ba \\ B &\to bAAb \mid ab\end{aligned}$$

debug: greibach/greibach

14.9 Greibach Normal Form

A grammar is in **Greibach Normal Form** if every production is of the form

$$A \to b\alpha$$

where A is a variable, b is a terminal and α is a string of variables. In the case of Chomsky Normal Form, note that ϵ is not in the language generated by such a language. The Greibach Normal Form is used in parsing and proofs of various theorems.

To change a grammar (that does not generate ϵ) to a Greibach Normal Form requires a lot more work.

1. Replace it by a grammar in Chomsky Normal Form (of course without changing language). Rename the variables to $A_1, \ldots, A_n$.
2. Replace grammar by one with with ascending variable property (AVP), i.e., all productions are of the form

   $$A_i \to A_j W \quad j > i$$

   or

   $$A_i \to aW, \quad W \in V^*$$
3. Process $A_n, A_{n-1}, \ldots, A_1$ (in this order) by a short-circuiting trick or by removal of left-recursion (later).

We're already done with Step 1.

Step 2 (AVP) is much harder so we'll delay that. For the time being here's an example for you so that you understand the concept of AVP. Look at the following grammar:

$$\begin{aligned}
A_1 &\to A_3A_1A_1 \mid aA_3 \mid \mathbf{A_1A_2} \\
A_2 &\to A_4A_2A_3 \mid bA_2 \mid \mathbf{A_1A_1A_2} \mid \mathbf{A_2A_1} \\
A_3 &\to A_4 \mid A_4A_3 \mid \mathbf{A_1A_3A_1} \mid \mathbf{A_2} \mid \mathbf{A_3A_3} \\
A_4 &\to aA_4 \mid b \mid \mathbf{A_1A_1} \mid \mathbf{A_2} \mid \mathbf{A_3A_2} \mid \mathbf{A_4A_4}
\end{aligned}$$

The derivation with the right-hand-side in bold are those which violate the AVP. For instance $A_1 \to A_1A_2$ violates the AVP. Right?

Suppose we're done with Step 2. This is how you get from Step 2 to Step 3. I will illustrate this with an example. The idea is very simple. Before we do the example, here's a lemma:

Lemma 14.9.1. *If $A \neq B$ and $A \to \alpha_1 B \alpha_2$, then $B \to \beta_1 \mid \ldots, \mid \beta_n$ are all the productions of with B on the left-hand-side, then you can eliminate $A \to \alpha_1 B \alpha_2$ and replace it with*

$$A \to \alpha_1\beta_1\alpha_2 \mid \alpha_1\beta_2\alpha_2 \mid \ldots \mid \alpha_1\beta_n\alpha_2 \mid$$

without changing the language generated by the original grammar.

We're just short circuiting the derivation. For instance if $A \to aCBd$ and $B \to b \mid cD$ are all the productions with B on the left-hand-side, then we can replace $A \to aCBd$ with

$$A \to aCbd \mid aCcDd$$

OK, now for the example:

Example 14.9.1. Consider the following grammar:

$$\begin{aligned}
A_1 &\to A_3A_1A_1 \mid aA_3 \\
A_2 &\to A_4A_2A_3 \mid bA_2 \\
A_3 &\to A_4 \mid A_4A_3 \\
A_4 &\to aA_4 \mid b
\end{aligned}$$

It satisfies the AVP (right?). Now we need to perform step 3 to make it into a Greibach Normal Form.

First of all productions of A_4 are already in the right form. In fact the variable with the highest index will *always* be OK. (Why? Use AVP!)

Next look at A_3. The trick is to use the productions of A_4. By the above lemma, we can short circuit productions of A_3 and get replace the old A_3 productions as follows: For $A_3 \to A_4$, we get

$$A_3 \to aA_4 \,|\, b \,|\, aA_4A_3$$

and for $A_3 \to A_4A_3$, we get

$$A_3 \to aA_4A_3 \,|\, bA_3$$

So the old production for A_3 is replaced by

$$A_3 \to aA_4 \,|\, b \,|\, aA_4A_3 \,|\, aA_4A_3 \,|\, bA_3$$

As you can see, these are in Greibach Normal Form.

The rest of the story is similar: While processing A_2, use productions of A_3 (the new ones) and A_4. Here are the results. The old production for A_2 should be replaced by

$$A_2 \to aA_4A_2A_3 \,|\, bA_2A_3 \,|\, bA_2$$

and the old production for A_1 is replaced by

$$A_1 \to aA_4A_1A_1 \,|\, bA_1A_1 \,|\, aA_4A_3A_1A_1 \,|\, bA_3A_1A_1 \,|\, aA_3$$

That's it!

OK. So where are we? There are 3 steps to be performs. The first is to change the grammar to Chomsky Normal Form (that itself involves a few steps. For Step 2, change it to have AVP. Finally for Step 3, reverse process it by a short circuiting trick. We're done with Step 1 and Step 3. Now for Step 2 (the AVP).

First there is a simple fact that you should know. Suppose you look at the production

$$A \to A\alpha \,|\, \beta$$

The production $A \to A\alpha$ is called a left recursion. Just look at this derivation and you'd see why:

$$A \implies A\alpha \implies A\alpha\alpha \implies A\alpha\alpha\alpha \overset{*}{\implies} A\alpha^n \implies \beta\alpha^n$$

You can change this to a right recursion by introducing a new variable:

$$\begin{aligned} A &\to \beta \,|\, \beta A' \\ A' &\to \alpha A' \,|\, \alpha \end{aligned}$$

Obviously this also generated the sentential form $\beta\alpha^n$ for $n \geq 0$. More generally, we have the following lemma:

Lemma 14.9.2. *If*

$$\begin{aligned} A &\to A\alpha_1 \,|\, A\alpha_2 \,|\, \ldots \,|\, A\alpha_n \\ A &\to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_m \end{aligned}$$

where the first line gives all the left recursion of A, and the second gives all the other derivation of A, then introducing a new variable A' we can replace the above productions by

$$\begin{aligned} A &\to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_n \\ A &\to \beta_1 A' \,|\, \beta_1 A' \,|\, \ldots \,|\, \beta_m A' \\ A' &\to \alpha_1 A' \,|\, \alpha_2 A' \,|\, \ldots \,|\, \alpha_n A' \\ A' &\to \alpha_1 \,|\, \alpha_2 \,|\, \ldots \,|\, \alpha_n \end{aligned}$$

without changing the language generate.

The proof is easy. Here's the main idea of Step 2. We process $A_1, \ldots, A_n$ (in this order) and while processing A_i, either it's already has the AVP of it looks like $A_i \to A_j\alpha$ and $j < i$ or $j = i$. For the case $j < i$, you can replace the production by another production by short circuiting (lemma 1 again) with a production of A_j. Note that $j < i$ so it has already been processed and since it has AVP, the productions of A_j would look like $A_j \to A_k$ where $k > j$. In other words you can slowly increase the index value of the variable.

If $j = i$, this is then a left recursion. So you use the above lemma. The new variable introduced should be named A_{-i}.

Let's do an example so that you see this in action:

Example 14.9.2. Consider the following grammar:

$$\begin{aligned} A_1 &\to A_2A_2 \\ A_2 &\to A_1A_3 \,|\, a \\ A_3 &\to A_1A_2 \,|\, A_2A_1 \,|\, a \end{aligned}$$

Note that it is in Chomsky Normal Form. Let's process this so that it has the AVP.

The production rule for A_1 is already has the AVP.

For the second line, the production $A_2 \to a$ has the AVP, but the production

$$A_2 \to A_1A_3$$

is does not.

By short circuiting the A_1 in the production with the production $A_1 \to A_2A_2$ (lemma 1), we get

$$A_2 \to A_2A_2A_3$$

(As you can see we are moving the index value up). Now we have a left recursion. We use lemma 2 to change this into a right recursion: Introduce variable A_{-2} to get

$$\begin{aligned} A_2 &\to a \,|\, aA_{-2} \\ A_{-2} &\to A_2A_3 \,|\, A_2A_3A_{-2} \end{aligned}$$

At this point, the variables A_{-2}, A_1, A_2 have the AVP.

Now consider A_3. You should get

$$A_3 \to aA_2A_2 \,|\, aA_{-2}A_2A_2 \,|\, aA_1 \,|\, aA_{-2}A_1 \,|\, a$$

This then creates a grammar with AVP without changing the language.

Example 14.9.3. Let's complete the above example by applying Step 3 to it to get a grammar in Greibach Normal Form. So far we have

$$\begin{aligned}
A_{-2} &\to A_2A_3 \,|\, A_2A_3A_{-2}\\
A_1 &\to A_2A_2\\
A_2 &\to a \,|\, aA_{-2}\\
A_3 &\to aA_2A_2 \,|\, aA_{-2}A_2A_2 \,|\, aA_1 \,|\, aA_{-2}A_1 \,|\, a
\end{aligned}$$

Remember we have to process with the highest index first. The productions for both A_3 and A_2 are already in Greibach Normal Form. Look at A_1:

$$A_1 \to A_2A_2$$

Using the productions for A_2 we short circuit with lemma 1 to get

$$A_1 \to AA_2 \,|\, aA_{-2}A_2$$

Now let's look at the production rules for A_{-2}:

$$A_{-2} \to A_2A_3 \,|\, A_2A_3A_{-2}$$

Let's do this one at a time. For $A_{-2} \to A_2A_3$, we use the productions for A_2 and lemma 1 to get

$$A_{-2} \to aA_3 \,|\, aA_{-2}A_3$$

For $A_{-2} \to A_2A_3A_{-2}$, we again use the production rules for A_2 to get

$$A_{-2} \to aA_3A_{-2} \,|\, aA_{-2}A_3A_{-2}$$

Altogether the grammar is now replaced by the following

$$\begin{aligned}
A_{-2} &\to aA_3 \,|\, aA_2A_3 \,|\, aA_3A_{-2} \,|\, aA_{-2}A_3A_{-2}\\
A_1 &\to aA_2 \,|\, aA_{-2}A_2\\
A_2 &\to a \,|\, aA_{-2}\\
A_3 &\to aA_2^2 \,|\, aA_{-2}A_2^2 \,|\, aA_1 \,|\, aA_{-2}A_1 \,|\, a
\end{aligned}$$

which is clearly in Greibach Normal Form. debug: cyk/cyk

14.10 Cocke–Younger–Kasami Algorithm

The **Cocke–Younger–Kasami algorithm** (**CYK**) is an algorithm that determine if a string is generated by a CFG. And if a string is generated by the CFG, the algorithm can also give all the possible derivations. In other words, it is a string parsing algorithm. The time complexity is $O(n^3)$ where n is the length of the string (details below). Later, faster algorithms were discovered. For instance there is one with time complexity $O(n^{2.8})$. The algorithm uses a very important technique in the design of algorithms – dynamic programming technique – it builds up a solution by combining solutions of subproblems of the given problem. Dynamic programming is an extremely important algorithm design technique an is frequently used even in research. (So it's not just a neat trick used in basic undergrad courses.) One CYK dynamic programming example is definitely not enough. See CISS358 for more details on dynamic programming.

Cocke–Younger–Kasami algorithm
CYK

Of course you can parse a string w without CYK: If w is ϵ, just check if S is a nullable symbol. Assuming the CFG is in Chomsky normal form, then every production will increase the length of the sentential form or replace a variable with a terminal. So you need $|w|$ productions to introduce $|w|$ variables into the derivation process and you need $|w|$ productions to replace the variables with terminals. So there are at most $2|w|$ derivation steps, which is of course finite. So you just check every possible sequence of $2|w|$ productions starting with S and see if you can arrive at w. Is this a good algorithm? (Note sarcasm ...)

The main idea of CYK is very simple. First we assume the grammar G is in Chomsky normal form. Now suppose our string w has length n. Let $w_{i,j}$ denote the substring of w starting at position i with length j. The first position is 1 and the last position is n. The string w is just

Chomsky, Chomsky, Chomsky!

$$w = w_{1,1}w_{2,1}w_{3,1}w_{4,1}\cdots w_{n,1}$$

Example 14.10.1. Let $w = baabaa$. Then $w_{2,3} = aab$.

Exercise 14.10.1.

1. Let $w = baabaa$. What is $w_{2,3}$? What about $w_{5,2}$? What about $w_{2,5}$? What is the relationship between the three?
2. If $w_{i,j}$ is the concatenation of $w_{p,q}$ and $w_{r,s}$, can you tell me something about p, q, r, s in terms of i, j?

First of all we can always determine if each $w_{i,1}$ can be derived: A derives $w_{i,1}$ if and only if $A \to w_{i,1}$ is a production (don't forget that we're assuming the grammar is in Chomsky Normal Form).

Chomsky, Chomsky, Chomsky!

What about the general case? How do you generate $w_{i,j}$? This is a string starting at position i and of length j. What happens when you cut up the string? Suppose you break it up into two strings $w_{i,j} = xy$. Well, x starts at the same index position in w as $w_{i,j}$ right? So it must be of the form

$$x = w_{i,k}$$

Since x is a substring of $w_{i,j}$, it's length must be at most j. Therefore $k \leq j$. But let's assume that we really did cut up the string so that y is not the empty string. So $k < j$. OK. So we have

$$w_{i,j} = w_{i,k}y \text{ where } k < j$$

Now let's think about y.

y starts at position $i + k$, right? And what is the length of y? It's $|w| - |x|$ which is $j - k$. So

$$y = w_{i+k,j-k}$$

Viola.

Altogether, $w_{i,j}$ is the concatenation of $w_{i,k}$ and $w_{i+k,j-k}$ where $k < j$ (i.e., $k = 1, 2, \ldots, j - 1$):

$$w_{i,j} = w_{i,k}w_{i+k,j-k}, \qquad (k = 1, 2, \ldots, j - 1)$$

So what we'll do is to find a variable, say X, deriving $w_{i,k}$:

$$X \overset{*}{\Longrightarrow} w_{i,k}$$

and another, say Y, deriving $w_{i+k,j-k}$

$$X \overset{*}{\Longrightarrow} w_{i,k}$$

for all the given k's. That means that XY derives $w_{i,j}$

$$XY \overset{*}{\Longrightarrow} w_{i,j}$$

If the grammar is in Chomsky Normal Form, we will then look at the production rules to find a rule of the form $Z \to XY$. If this is found, then Z must derive XY which derives $w_{i,j}$:

$$Z \implies XY \overset{*}{\implies} w_{i,j}$$

That's the main idea.

The important thing to understand is that we are breaking down out w into two substrings exactly because the grammar is in Chomsky Normal Form. If the grammar has a production rule of the form $Z \to WXY$, then our string w might be derived through this production rule and in that case we would have to break up w into *three* substrings where W derives the first, X derives the second, and Y derives the third. The more variations there are in the right hand side of the production rules of the grammar, the more cases we would need to analyze to cover all possibilities in the derivation process. Get it?

Chomsky, Chomsky, Chomsky!

Now let's go through an example ...

Example 14.10.2. Consider the grammar:

$$\begin{aligned} S &\to AB \mid AC \\ A &\to BC \mid a \\ B &\to CB \mid b \\ C &\to AA \mid b \end{aligned}$$

Note that this is already in Chomsky Normal Form. Is *baaab* generated by the grammar?

SOLUTION. For this example $w = baaab$. We will systematically tabulate the variables deriving $w_{i,j}$. The variables deriving $w_{i,j}$ will be placed at row j and column i:

j \ i	b 1	a 2	a 3	a 4	a 5	b 6
1						
2						
3						
4						
5						
6						

First Row: We first work out $w_{i,1} = b$:

j \ i	b 1	a 2	a 3	a 4	a 5	b 6
1	?					
2						
3						
4						
5						
6						

b is derived only by $B \to b$ and $C \to b$:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B, C\}$					
2						
3						
4						
5						
6						

If you are interested in not just knowing if *baaaab* is generated by the given grammar, but you actually want to write down the derivation, it's helpful to include some addition information for each variable in the cells. I'll write these extra information in subscript. I'll show you how these extra information will be used later.

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$					
2						
3						
4						
5						
6						

In the above, the B_b means "B will derive b".

a is derived by $A \to a$. This allows us to complete the first row:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2						
3						
4						
5						
6						

In summary, to fill the first row, you need only to look at the productions of the form $\langle variable \rangle \to \langle terminal \rangle$. (Don't memorize this. You should understand the goal and the math involved.)

Second Row: Now let's look at the $j = 2, i = 1$, i.e. $w_{1,2}$:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	?					
3						
4						
5						
6						

Now

$$w_{1,2} = w_{1,1}w_{2,1}$$

(There's only one way to cut up $w_{1,2}$ into two proper substrings.) But we already know that the variable deriving $w_{1,1}$ is at the $(1,1)$ entry in the table: it's either B or C. We also know that the variable deriving $w_{2,1}$ is at the $(1,2)$ entry; this is A. Therefore we want to find a variable V such that

$$V \to BA$$

or

$$V \to CA$$

is a production rule. (Why do we consider only this scenario? Let's hear it ... *Chomsky, Chomsky, Chomsky, ...* BA and CA does not appear on the right-hand side of the productions in the grammar. READ THIS PARAGRAPH AGAIN! So there are no such Vs. We will indicate this by putting $\emptyset$ at $(2, 1)$.

The cells we looked at in order to fill the entry at $(2, 1)$ are shaded.

$j \backslash i$	b 1	a 2	a 3	a 4	a 5	b 6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$					
3						
4						
5						
6						

Using the same idea, we can fill in the entry at $(2, 2)$:

$j \backslash i$	b 1	a 2	a 3	a 4	a 5	b 6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$				
3						
4						
5						
6						

Here, C derives AA. The subscript data of C, i.e. $(1, 2)$ and $(1, 3)$, tells us that C derives XY where variable X is in cell $(1, 2)$ and Y is in cell $(1, 3)$.

Here's the table with row 2 completed:

i	`b`	`a`	`a`	`a`	`a`	`b`
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3						
4						
5						
6						

Again, recall the aim: We're trying to fill the (j, i) entry of the table with a variable that can derive $w_{i,j}$.

<u>Third Row</u>: Now we look at $j = 3, i = 1$:

i	`b`	`a`	`a`	`a`	`a`	`b`
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	?					
4						
5						
6						

We want to to find a variable V that derives $w_{1,3}$. Recall from the above that we want to make use of previous results. The following are the possible ways of breaking up $w_{1,3}$:

$$w_{1,1}w_{2,2} \qquad \text{or} \qquad w_{1,2}w_{3,1}$$

In other words if $w_{1,3}$ is xyz then the two ways to break up $w_{1,3}$ into two proper substrings are:

$$x \cdot yz \qquad \text{or} \qquad xy \cdot z$$

We already know that the variable deriving $w_{1,1}$ is in the entry at $(1, 1)$ of the table: it's B or C. For $w_{2,2}$, just look at the entry at $(2, 2)$; C derives $w_{2,2}$. Therefore BC and CC derives $w_{1,1}w_{2,2}$ which is just $w_{1,3}$. Now we want to find V that derives BC or CC:

$$V \implies XY \text{ where } X \in \{B, C\} \text{ and } Y \in \{C\}$$

and enter that into the $(3, 1)$ entry of the table. Checking the grammar, we see that only A can do this. Therefore we have

i	`b`	`a`	`a`	`a`	`a`	`b`
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$					
4						
5						
6						

We're not done yet because $w_{1,3}$ can also be $w_{1,2}w_{3,1}$.

i	`b`	`a`	`a`	`a`	`a`	`b`
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$					
4						
5						
6						

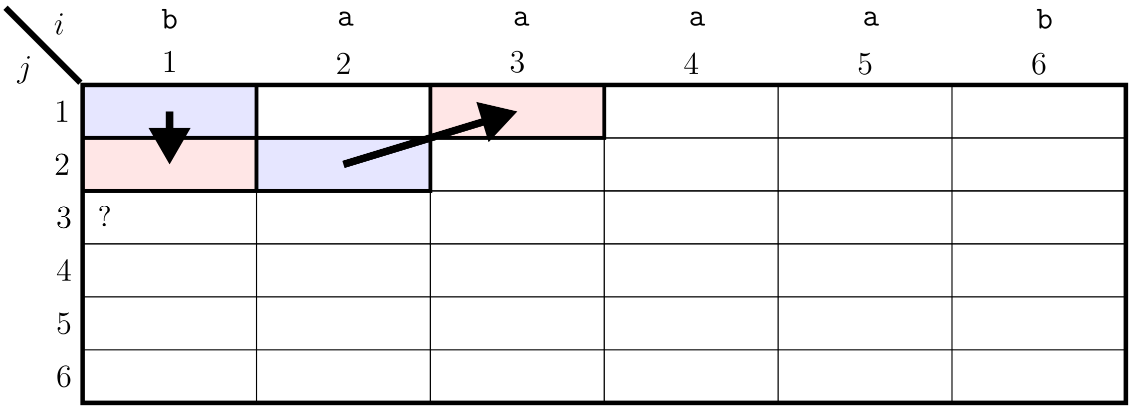
When you check the entry at $(2, 1)$, you see that no variable can derive $w_{1,2}$, i.e., there is no V such that

$$V \implies XY \text{ where } X \in \emptyset, Y \in \{A\}$$

Therefore we have nothing else to add to the entry at $(3, 1)$. Altogether we have:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$					
4						
5						
6						

Notice that while working on the entry at $(3,1)$, we looked at $(1,1),(2,2)$ and also $(2,1),(1,3)$. Do you notice that there's a pattern in the pairs of cells that contribute to the two cases in filling the cell at $(3,1)$?



Now we move on the the entry at $(3,2)$:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	?				
4						
5						
6						

This corresponds to $w_{2,3}$ and

$$w_{2,3} = w_{2,1}w_{3,2} \qquad \text{or} \qquad w_{2,2}w_{4,1}$$

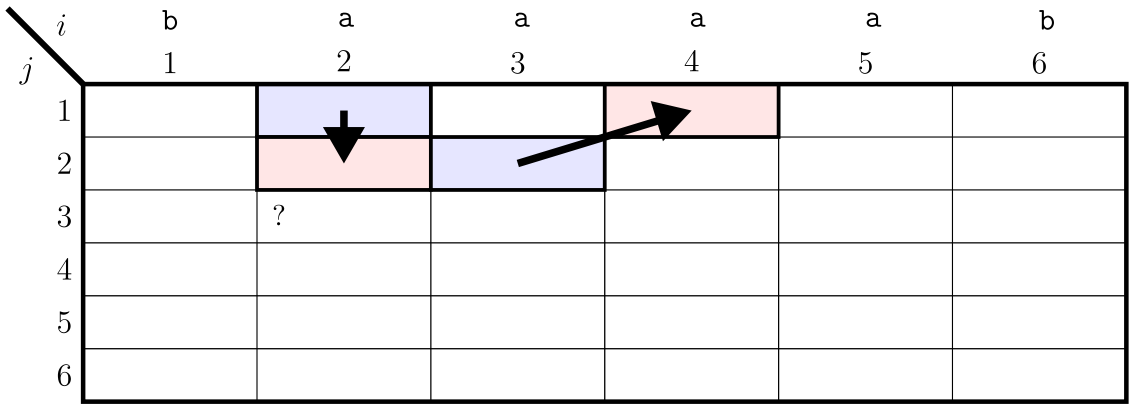
First for $w_{2,3} = w_{2,1}w_{3,2}$, we want to find variables that can derive AC; there's only one: S.

$j \backslash i$	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$				
4						
5						
6						

For $w_{2,3} = w_{2,2}w_{4,1}$, we want to find variables that can derive CA: there isn't any. So there's nothing to add:

$j \backslash i$	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$				
4						
5						
6						

Notice the patterns of the pairs of cells that contribute to the computation at $(3, 2)$:



Now following the same method, complete the second row yourself.

[PAUSE]

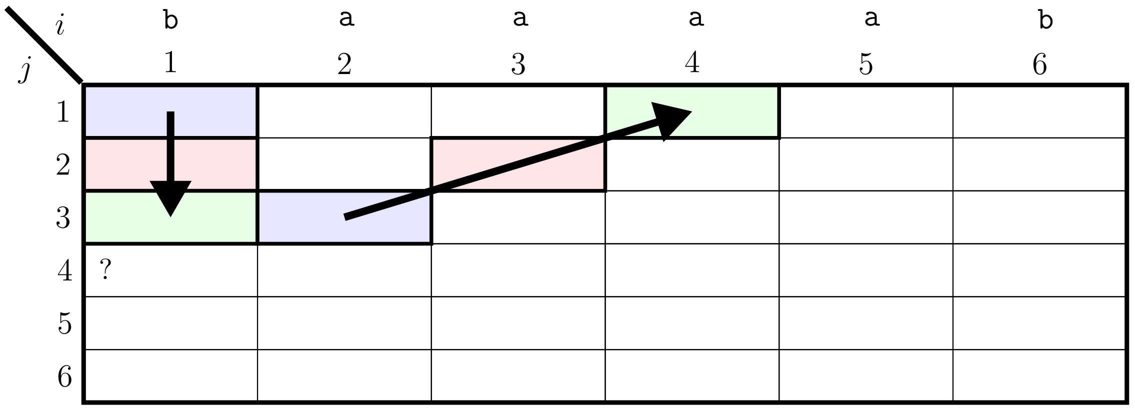
The table should now look like this:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4						
5						
6						

Fourth Row: Now we look at the entry for $(4, 1)$. I think you can see the pattern now. Basically you need to look at the following cases:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	?					
5						
6						

Note the traversal pattern:



For

$j \backslash i$	b 1	a 2	a 3	a 4	a 5	b 6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	?					
5						
6						

we are looking for a variable V such that

$$V \to XY \text{ where } X \in \{B, C\}, Y \in \{S\}$$

There is no such V. For

$j \backslash i$	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	?					
5						
6						

since there are no variables at $(2,1)$, we don't have any V's to add to $(4,1)$. For

$j \backslash i$	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	?					
5						
6						

we want V such that $V \to AA$. There's only one case: $V = C$. Altogether we have

$j \backslash i$	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	$\{C_{(3,1),(1,4)}\}$					
5						
6						

For entry (4,2) you should get:

$j \backslash i$	b 1	a 2	a 3	a 4	a 5	b 6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	$\{C_{(3,1),(1,4)}\}$	$\emptyset$				
5						
6						

For entry (4,3) you should get:

$j \backslash i$	b 1	a 2	a 3	a 4	a 5	b 6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	$\{C_{(3,1),(1,4)}\}$	$\emptyset$	$\{S_{(1,3),(3,4)}\}$			
5						
6						

Fifth Row:

j \ i	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	$\{C_{(3,1),(1,4)}\}$	$\emptyset$	$\{S_{(1,3),(3,4)}\}$			
5	$\{S_{(3,1),(2,4)}\}$	$\{B_{(2,2),(3,4)}\}$				
6						

At $(5,1)$ we use $S \to AC$ and at $(5,2)$ we use $B \to CB$.

Sixth Row. Using the following traversal shown, you should get the following entry for (6,1): ... and we're done!!! Here's the final table:

j \ i	b	a	a	a	a	b
	1	2	3	4	5	6
1	$\{B_b, C_b\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{A_a\}$	$\{B_b, C_b\}$
2	$\emptyset$	$\{C_{(1,2),(1,3)}\}$	$\{C_{(1,3),(1,4)}\}$	$\{C_{(1,4),(1,5)}\}$	$\{S_{(1,5),(1,6)}\}$	
3	$\{A_{(1,1),(2,2)}\}$	$\{S_{(1,2),(2,3)}\}$	$\{S_{(1,3),(2,4)}\}$	$\{B_{(2,4),(1,6)}\}$		
4	$\{C_{(3,1),(1,4)}\}$	$\emptyset$	$\{S_{(1,3),(3,4)}\}$			
5	$\{S_{(3,1),(2,4)}\}$	$\{B_{(2,2),(3,4)}\}$				
6	$\{B_{(1,1),(2,5)}$ $S_{(3,1),(3,4)}\}$					

So what have we achieved? First of all S is in the entry at $(6,1)$. What does this mean? Recall again that this means S can derive $w_{1,6}$, which is the string we started with, i.e. *baaaab*. Secondly, you can write down the productions that derive *baaaab* from S. Can you write them down using the table? [SPOILERS AHEAD ...]

Of course CYK gives you even more. For instance we also know that B is also in the entry at (6,1). This means that B can also derive *baaab*. If you look at the entry at (2,5), you conclude that B can derive *aaab*. In other words besides addressing the question of whether *baaaab* can be generated by the grammar, we have also answered the question for all substrings of *baaaab*. Make sure you see this.

OK ... so you know that S can derive $baaaab$ (i.e. $baaaab \in L(G)$ for the above CFG G). But what is the derivation? If you've understood the mathematical reasoning so far, you should have no problems writing down the derivation.

$$\begin{aligned}
S_{(3,1)(3,4)} &\Longrightarrow A_{(1,1)(2,2)} \cdot B_{(2,4)(1,6)} \\
&\Longrightarrow B_b \cdot C_{(1,2)(1,3)} \cdot B_{(2,4)(1,6)} && \text{(or } C_b \text{ instead of } B_b\text{)} \\
&\Longrightarrow b \cdot C_{(1,2)(1,3)} \cdot B_{(2,4)(1,6)} \\
&\Longrightarrow b \cdot A_a \cdot A_a \cdot B_{(2,4)(1,6)} \\
&\Longrightarrow b \cdot a \cdot A_a \cdot B_{(2,4)(1,6)} \\
&\Longrightarrow b \cdot a \cdot a \cdot B_{(2,4)(1,6)} \\
&\Longrightarrow b \cdot a \cdot a \cdot C_{(1,4)(1,5)} \cdot B_b && \text{(or } C_b \text{ instead of } B_b\text{)} \\
&\Longrightarrow b \cdot a \cdot a \cdot A_a \cdot A_a \cdot B_b \\
&\Longrightarrow b \cdot a \cdot a \cdot a \cdot A_a \cdot B_b \\
&\Longrightarrow b \cdot a \cdot a \cdot a \cdot a \cdot B_b \\
&\Longrightarrow b \cdot a \cdot a \cdot a \cdot a \cdot b
\end{aligned}$$

Note that if you are only interested in whether $baaaab$ is generated by the grammar (and not the actual derivation), then you only need to fill the CYK table in the following way:

i	b	a	a	a	a	b
j	1	2	3	4	5	6
1	$\{B, C\}$	$\{A\}$	$\{A\}$	$\{A\}$	$\{A\}$	$\{B, C\}$
2	$\emptyset$	$\{C\}$	$\{C\}$	$\{C\}$	$\{S\}$	
3	$\{A\}$	$\{S\}$	$\{S\}$	$\{B\}$		
4	$\{C\}$	$\emptyset$	$\{S\}$			
5	$\{S\}$	$\{B\}$				
6	$\{B, S\}$					

The CYK algorithm requires us to fill data into half of an n–by–n array – there are altogether $(1/2)n^2$ cells to fill. Each cell X of the array requires us to go through at most n pairs of cells which are already fill and see if we can find a production rule that can help in filling cell X. (We can speed up the search for a suitable production rule if we find a suitable organization for the collection of production rules. Which one?) Therefore the total runtime is $O(n^3)$. The space complexity is of course $\Theta(n^2)$. (If you want to take into account the grammar, then the runtime is $O(n^3|G|)$ where $|G|$ is the size of the grammar which is the total count of all symbols that appear in all the production rules.)

Exercise 14.10.2. The above table also tells you that B can also derive *baaaab*. Write down a dervation of *baaaab* using variable B.

Exercise 14.10.3. Using the grammar G above, check if the following words are generated by G using the CKY algorithm. Provide the completed CYK tables. For words generated by G, write down all their possible derivations.

1. $aaaa$
2. $ababba$
3. $bbbaaa$
4. $abbaab$

Exercise 14.10.4. Consider the grammar G given by

$$\begin{aligned}
S &\to AB \mid BA \\
A &\to AC \mid a \\
B &\to SA \mid SB \mid b \\
C &\to SS \mid BA \mid a
\end{aligned}$$

Using the grammar G above, check if the following words are generated by G using the CKY algorithm. Provide the completed CYK tables. For words generated by G, write down all their possible derivations.

1. *aa*
2. *bab*
3. *abab*
4. *aabaab*

Exercise 14.10.5. Consider the grammar:

$$\begin{aligned} S &\to AB \mid BA \\ A &\to AC \mid AD \mid a \\ B &\to SA \mid SD \mid a \mid b \\ C &\to DS \mid BA \mid b \\ D &\to AS \mid BB \mid a \end{aligned}$$

Using the grammar G above, check if the following words are generated by G using the CKY algorithm. Provide the completed CYK tables. For words generated by G, write down all their possible derivations.

1. aa
2. bab
3. $abab$
4. $aabaab$

1.

j \ i	`a` 1	`a` 2
1	$\{A_a, B_a, D_a\}$	$\{A_a, B_a, D_a\}$
2	$\{S_{(1,1)(1,2)}, A_{(1,1)(1,2)}, C_{(1,1)(1,2)}, S_{(1,1)(1,2)}, D_{(1,1)(1,2)}\}$	

The derivations are

$$\begin{aligned} S &\implies AA \implies aA \implies aa \\ S &\implies AB \implies aB \implies aa \\ S &\implies BA \implies aA \implies aa \end{aligned}$$

Exercise 14.10.6. Of course with the above information, you can now write a simple program to check if a string is generated by a CFG (you would need the grammar to be in Chomsky Normal Form). By the way, you can also store the production rule in each cell if you want your program to list the production rule for each derivation step. Another thing is that you need to search the list of production rules to fill up the table. You want to think about how you want to represent the list of production rules and how they should be organized in a container to allow fast searches relevant to completing the CYK table. Also, note that in our algorithm above, when we compute the variables V in a cell, we include the information

$$V_{(i,j)(k,l),...}$$

to indicate that V can derive XY where X is in the set of variables in cell (i, j) and Y is in the set of variables in cell (k, l). You might want to include more information about X and Y. For instance you might want to do

$$V_{(i,j,m)(k,l,n),...}$$

to include the fact that the X is the m–th variable at cell (i, j) (if you're using an array for each cell) and Y is the n–th variable at cell (k, l).

Exercise 14.10.7. What if you have a string that's not generated by the grammar? What do you get when you perform the CYK algorithm on such a string?

Exercise 14.10.8. Using the above table, are there strings of length 2 in $L(G)$? What about 3? Or 4? Or 5? Verify by writing down their derivations (if possible).

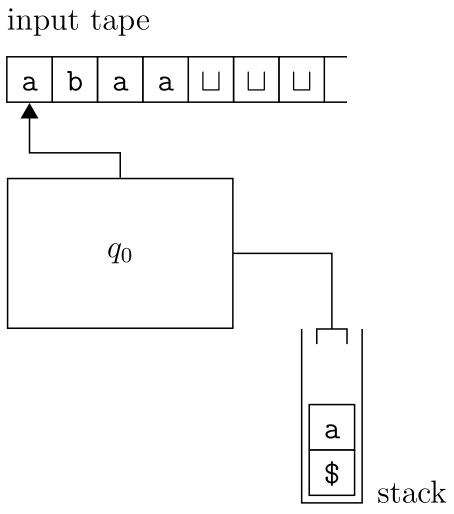
debug: pda/pda

14.11 Push Down Automata

A **push down automata** (PDA) is like an NFA except that it has an infinite memory. The infinite memory device is just a stack. (Recall that in a DFA, the states play the role of memory. Therefore a DFA can only remember a finite number of things since, by definition, a DFA can only have a finite number of states.)

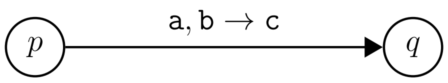
OK, what's a stack? A stack is a container of things (in our case symbols/characters). You can put things into it and you can also take things out of it. But the organization of the stack is such that it works like a stack of plates at a buffet. When you put something into the stack, it goes onto the top. When you take things out of it, you have to take the thing that is at the top of the stack.

Here's the picture of a PDA:



Remember that the stack is infinite – you can put any amount of data into it. Putting something onto the top of the stack is called "pushing" and removing something from the top is called "popping".

The operation of a PDA is described by a state diagram just like an NFA diagram except for some modification. Each edge from state p to q is labelled $(a, b \to c)$:



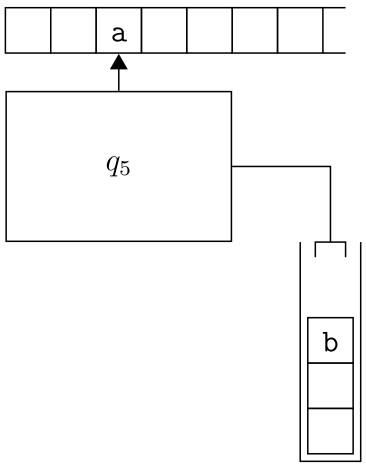
Some books also write $(a, b/c)$ instead of $(a, b \to c)$.

This means:

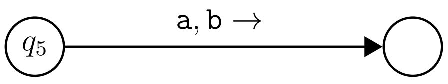
> "If your PDA is at state p, is about to reading character a from the string (the input tape), and the top of stack is b, then your PDA pops off b from the stack and push c onto the stack, move the read head (to the right) past the character it was reading and get ready to read the next character, and finally, the PDA go into state q."

If you're at a state where there is no valid next-state to do to, then the machine crashes and the string is not accepted. Also, because the PDA is like an NFA, there could be several applicable transitions – you might need to spawn copies of your PDAs.

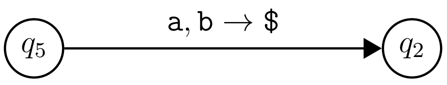
For instance, suppose your PDA is at state q_5, it is about to read `a` (the third character to be processed), and the top of the stack is `b`:



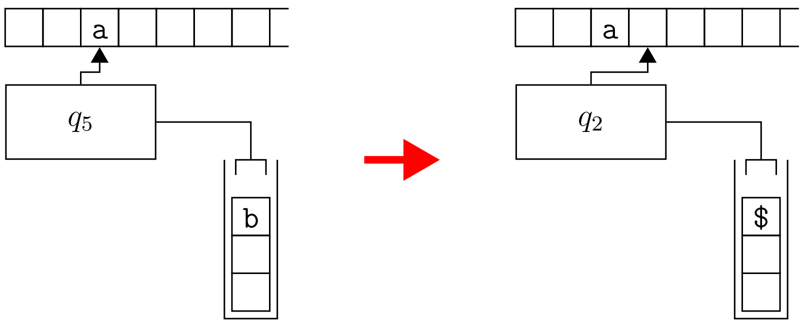
In your PDA state diagram, you look for the state q_5, and then for the relevant transitions (there might be more than one) where you are about to read `a` and the top of stack if `b`:



Say one of the transitions matching the above is



Then your PDA goes through this computational step:



There are a few caveats. First, for the label

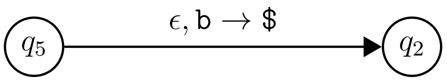
$$(a, b \to c)$$

a or b or c can be ϵ – remember that PDA is like an NFA.

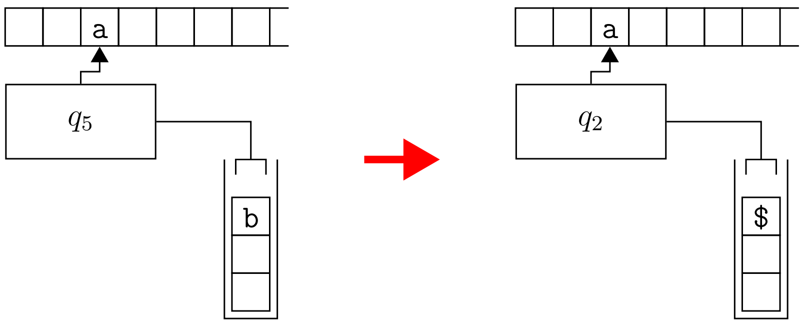
A transition

$$(\epsilon, b \to c)$$

means you're working on the stack without reading any character from the input, i.e., you are replaced b on top of the stack with c but your read head does not move (note: in this case it does not matter what your read head is pointing to). For instance if the applicable transition of your PDA is



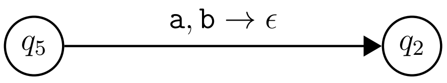
(i.e., the PDA is at state q_5 and the top of the stack is b) then the PDA will go through this computational step:



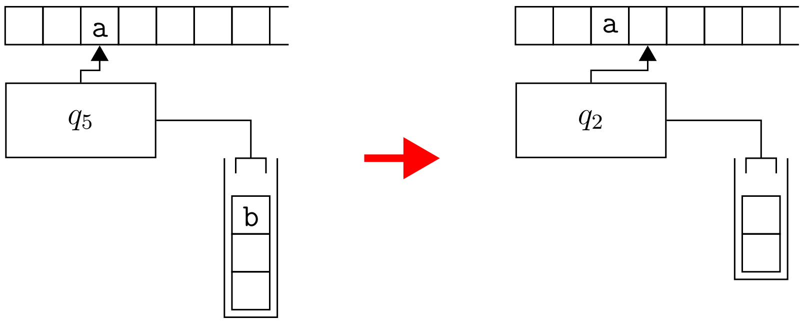
A transition like

$$(a, b \to \epsilon)$$

means that if the character being read is a and the top of the stack is b, then remove b from the stack and put ϵ (i.e., nothing) onto the stack. This means of course that you're simplying popping the stack. For instance if the applicable transition of your PDA is



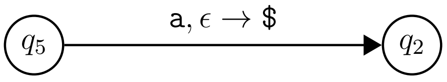
then the PDA will go through this computational step:



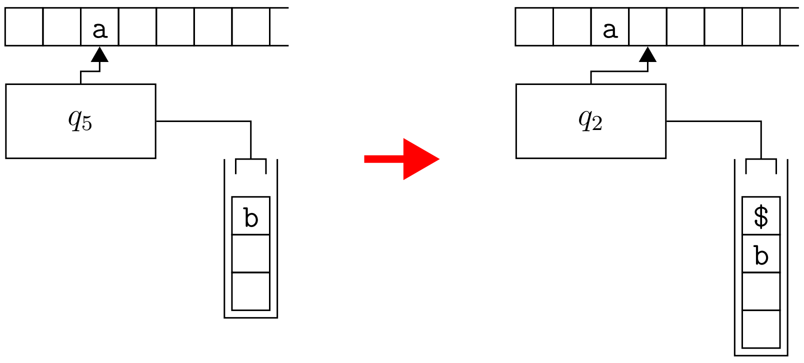
On the other hand

$$(a, \epsilon \to c)$$

means that you're pushing c onto the stack (if the character to be read is `a`). For instance if the applicable transition of your PDA is



(your PDA is at state q_5 and it's reading `a` – it doesn't matter what your top–of–stack is) then the PDA will go through this computational step:



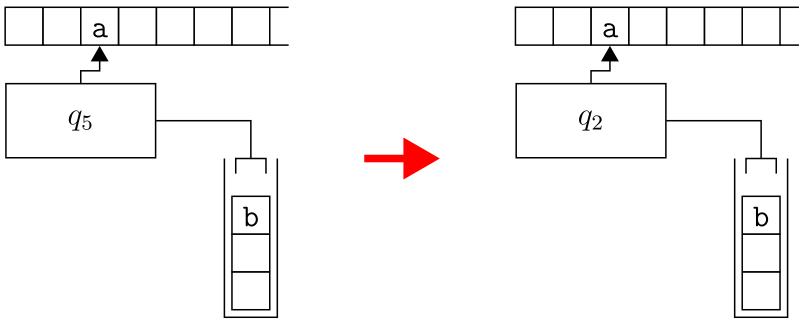
Also, a transition such as

$$(\epsilon, \epsilon \to \epsilon)$$

can be used just to enter a new state without reading any input or modifying the stack. (This does not depend on what you are reading and what's on the top of the stack.) For instance if the applicable transition of your PDA is

$$\begin{tikzcd} q_5 \arrow[r, "{\epsilon, \epsilon \to \epsilon}"] & q_2 \end{tikzcd}$$

(i.e., your PDA is at state q_5 – it doesn't matter what your PDA is about to read and it does not depend on what the top–of–stack is) then the PDA will go through this computational step:

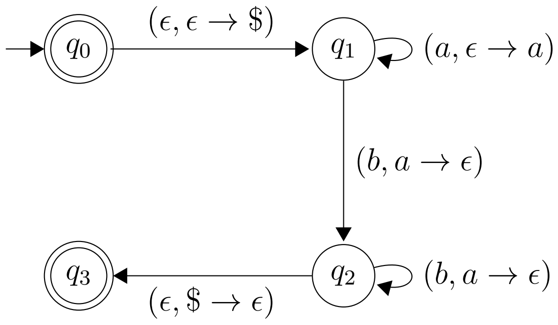


This transition is most similar to the ϵ–transition of an NFA.

And remember that there can be multiple transition from a state labeled in the same way (I said it's non-deterministic right?)

A string is **accepted** if after reading all the characters and going the computational steps starting with a PDA at the initial state, with the read head at the first character, and with an initial empty stack, one your PDAs (remember there can be more than one even though you start with one), you end up in an accepting state.

Try running the following PDA with the string *aabb*:



Clearly the start state of the PDA is q_0 (just like a DFA/NFA). You can also see that there are accept states in a PDA.

Note that the PDA has to read input characters from a set (denoted by Σ just like for DFA/NFA) while the stack can hold characters from another set. The two set of characters (for input and for the stack) can be different. For this example, the input characters are from $\{a, b\}$ while the stack characters are from $\{a, b, \$\}$.

Here's an execution of the PDA. I'm using $\triangleright$ to denote the position of the read head; the character to be read is on the right of $\triangleright$. As you can see, I'm drawing the stack horizontally instead of vertically. Initially we have the following:

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	(top)(bottom)

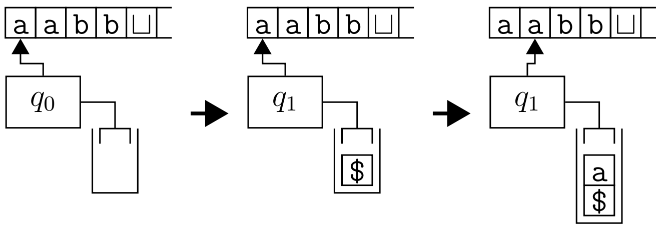
We're at state q_0. The transition (the only one) is $(\epsilon, \epsilon \to \$)$. So we get this:

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	(top)(bottom)
$\triangleright aabb$	q_1	(top) \$ (bottom)

At this point, we're at q_1 and there are two transitions. The only one we can use is $(a, \epsilon \to a)$. This means that we're reading a (the read head must move to the irght by one step) and we're replace ϵ on the top of the stack by a (which means that we're just putting a onto the stack). So the table of computations look like this:

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	(top)(bottom)
$\triangleright aabb$	q_1	(top) \$ (bottom)
$a \triangleright abb$	q_1	(top) a\$ (bottom)

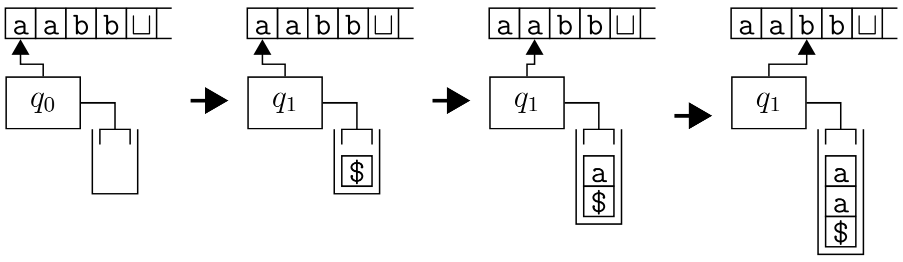
Note very carefully that a is pushed onto the stack!!! In terms of a sequence of pictures of a PDA computing with input $aabb$ up to this point, you have:



Get it? The next step gives us this (I've removed (top) and (bottom) from the stack column to make it more readable):

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	
$\triangleright aabb$	q_1	\$
$a \triangleright abb$	q_1	a\$
$aa \triangleright bb$	q_1	aa\$

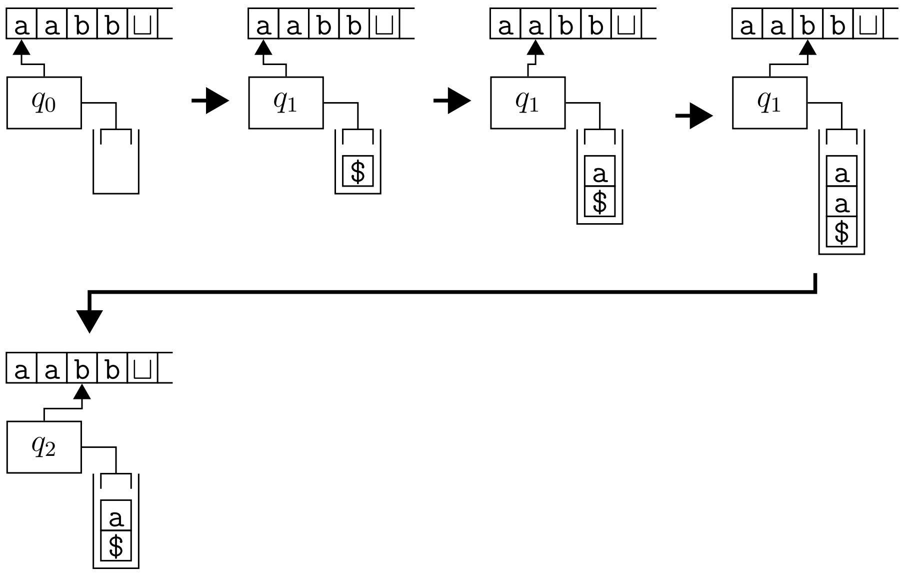
In terms of PDA pictures, we have this:



Note that the next character to be read is b; the top of the stack is a. The only transition we can apply is $(b, a \to \epsilon)$. The action on the stack means to replace the a on top of the stack by ϵ. So we get this:

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	
$\triangleright aabb$	q_1	\$
$a \triangleright abb$	q_1	$a\$$
$aa \triangleright bb$	q_1	$aa\$$
$aab \triangleright b$	q_2	$a\$$

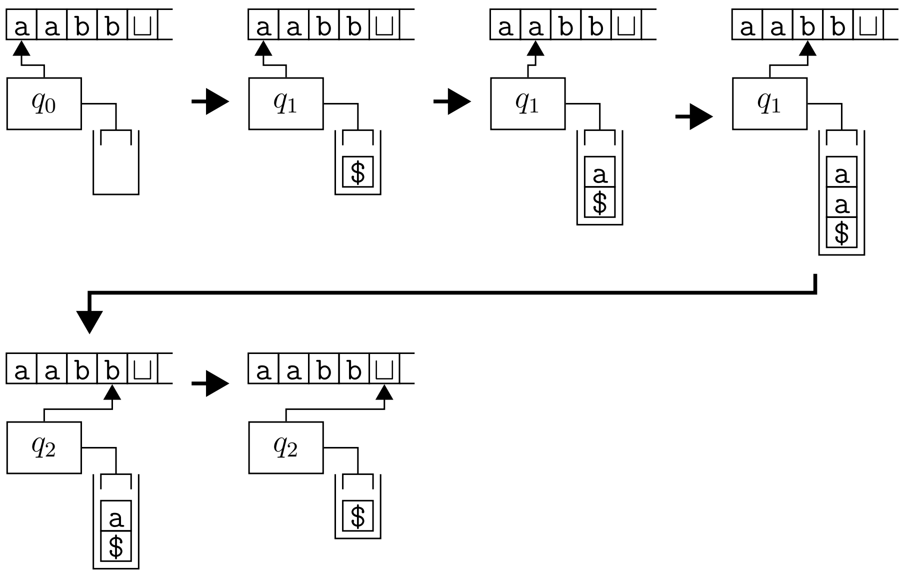
Look at the table carefully! In terms of a picture, we have



The next step gives us this:

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	
$\triangleright aabb$	q_1	\$
$a \triangleright abb$	q_1	$a\$$
$aa \triangleright bb$	q_1	$aa\$$
$aab \triangleright b$	q_2	$a\$$
$aabb\triangleright$	q_2	\$

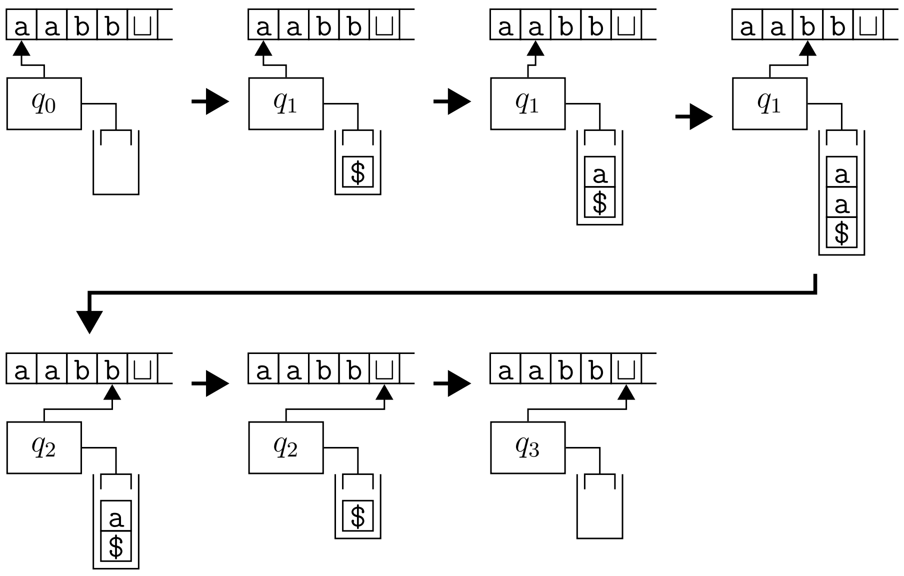
In terms of a picture of PDAs, we have



One more step is applicable ... so we get ...

input tape	state	stack (top on the left)
$\triangleright aabb$	q_0	
$\triangleright aabb$	q_1	\$
$a \triangleright abb$	q_1	$a\$$
$aa \triangleright bb$	q_1	$aa\$$
$aab \triangleright b$	q_2	$a\$$
$aabb\triangleright$	q_2	\$
$aabb\triangleright$	q_3	

or a terms of a picture:



Since we land in an accept state at the end of reading the input, $aabb$ is accepted by the above PDA. If we let P denote the above PDA, we then write $aabb \in L(P)$.

In the table of computational steps, we can also ignore characters already read. So we can display the execution like this:

remaining input	state	stack (top on the left)
$aabb$	q_0	
$aabb$	q_1	$\$$
abb	q_1	$a\$$
bb	q_1	$aa\$$
b	q_2	$a\$$
ϵ	q_2	$\$$
ϵ	q_3	

Drawing pictures or writing a table is fine. But the formal way (which uses less ink) is to do the following ...

A **configuration** or **instantaneous description** is just a tuple representing each step of the above computation. The state is usually placed first, followed by the remaining input, and finally by the stack contents. So here's a configuration:

$$(q_0, aabb, \epsilon)$$

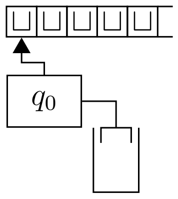
Mathematically the table above is usually written in the following format called a computation or a derivation:

$$\begin{aligned}(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$) \\ &\vdash (q_1, abb, a\$) \\ &\vdash (q_1, bb, aa\$) \\ &\vdash (q_2, b, a\$) \\ &\vdash (q_2, \epsilon, \$) \\ &\vdash (q_3, \epsilon, \epsilon)\end{aligned}$$

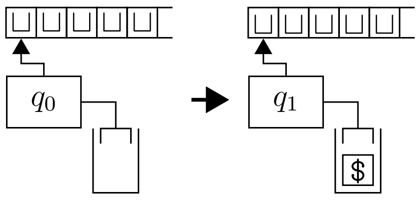
(Refer to the chapter on regular languages; a derivation for the execution of a DFA/NFA is the same except that it does not have the third component, i.e. the stack).

Note that a PDA is nondeterministic.

I've only shown you one that executes to the accept state. In general, there can be multiple executions (and multiple PDAs just like in the case of NFAs.) For instance in the case of ϵ input, there's one PDA with this computation (actually this has no computation at all):



and this one:


$\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$
q_0
$\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$
q_1
\$

The first one reaches an accept state while the second does not. Therefore ϵ in accepted by the PDA.

Exercise 14.11.1. Now's your turn: Execute P with inputs $aaabbb$, $aabbb$, $aaabb$, ba, ϵ. Write down all possible derivations for each string.

You should be able to see after a few experiments that

$$L(P) = \{a^n b^n \mid n \geq 0\}$$

Now step back and try to understand the big picture: Basically the machine (or mathematical device) pushes all the a's on the input onto the stack and then pops one a for each b read on the input; if there's nothing left on the stack the a's must match the b's.

What about the \$ symbol?

You see, the stack we're using does not have the ability to tell you if it's empty or not. So the \$ actually acts as a bottom-of-stack marker. This is important. So remember that if you need to check if you've reached the bottom of stack, the *first* thing you always do is to push a \$ onto the stack without reading any character from the input. In other words if you need to check if the stack is empty (or rather you've reached the bottom-of-stack), the only transition from the start state is always

$$(\epsilon, \epsilon \to \$)$$

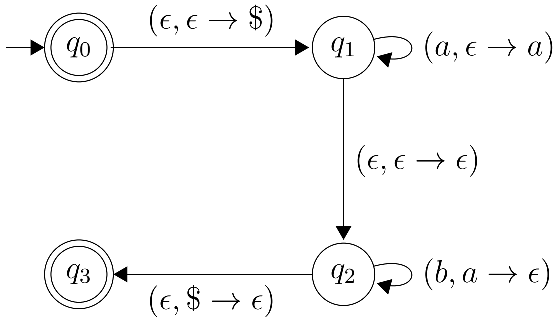
And if you need to remove the bottom-of-stack marker (without reading anything from the input) is

$$(\epsilon, \$ \to \epsilon)$$

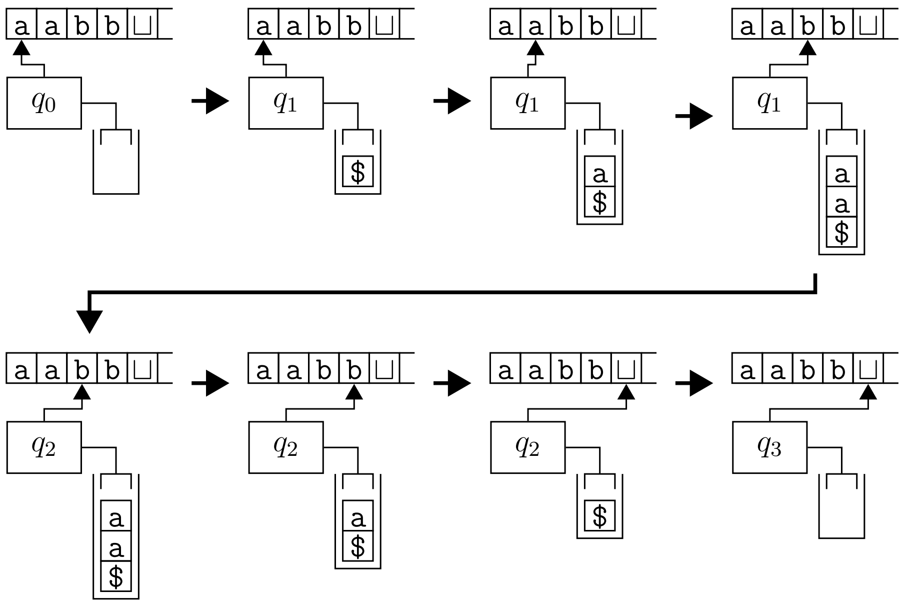
Another important point to note is this: Do you realize that an input character is either a or b, but a character in the stack can be a or b or \$? So in general a PDA has *two* sets of characters: input characters and stack characters. Remember that!!!

You see from the above that the stack is used to "remember" the number of a's on the input. You can (and should!) view the stack as a memory device. The only problem is that it's not like RAM where you have random access to anything you like.

Here's another PDA:


q_0
$(\epsilon, \epsilon \to \$)$
q_1
$(a, \epsilon \to a)$
$(\epsilon, \epsilon \to \epsilon)$
q_2
$(b, a \to \epsilon)$
q_3
$(\epsilon, \$ \to \epsilon)$

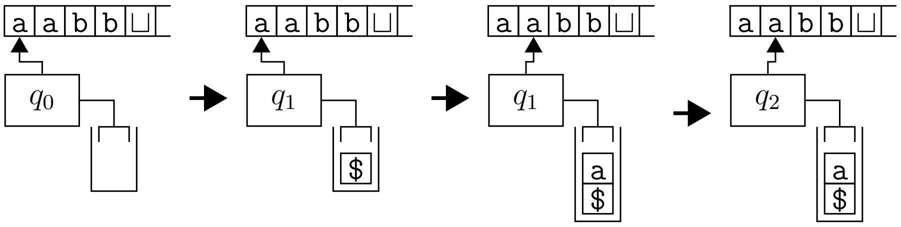
Let's try it with $aabb$. Here's the picture of a computation:



Formally, I would write

$$\begin{aligned}(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$)\\ &\vdash (q_1, abb, a\$)\\ &\vdash (q_1, bb, aa\$)\\ &\vdash (q_2, bb, aa\$)\\ &\vdash (q_2, b, a\$)\\ &\vdash (q_2, \epsilon, \$)\\ &\vdash (q_3, \epsilon, \epsilon)\end{aligned}$$

In this PDA, the transition from q_1 to q_2 represents nondeterministically going from "remembering a's" (using the stack) and "matching b's with a's". Another execution might go like this:



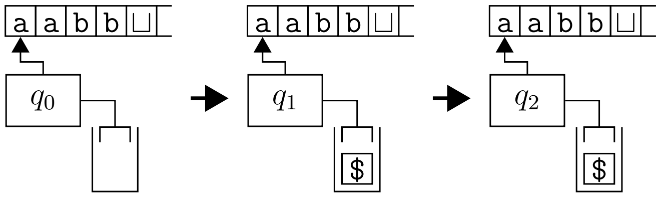
The derivation is

$$\begin{aligned}(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$)\\ &\vdash (q_1, abb, a\$)\\ &\vdash (q_2, abb, a\$)\end{aligned}$$

At this point the derivation stops since there's no applicable transition. In this case, the PDA decides to go to state q_2 after reading one a (and therefore there's only one a in the stack.) An even short derivation that does not accept is

$$\begin{aligned}(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$)\\ &\vdash (q_2, aabb, \$)\end{aligned}$$

Here's the picture:



At this point, this PDA dies because at state q_2, there is no transition for "read a on the input tape and the top of stack is a."

In the above, there are three computations of $aabb$:

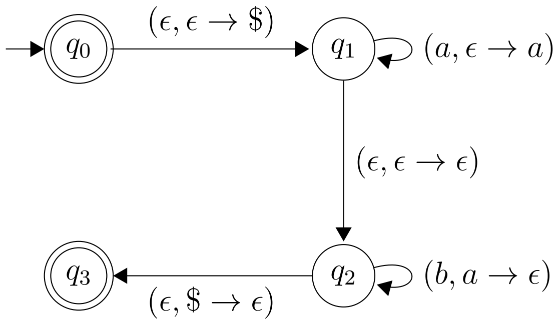
$$\begin{aligned}
(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$) \\
&\vdash (q_1, abb, a\$) \\
&\vdash (q_1, bb, aa\$) \\
&\vdash (q_2, bb, aa\$) \\
&\vdash (q_2, b, a\$) \\
&\vdash (q_2, \epsilon, \$) \\
&\vdash (q_3, \epsilon, \epsilon)
\end{aligned}$$

$$\begin{aligned}
(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$) \\
&\vdash (q_1, abb, a\$) \\
&\vdash (q_2, abb, a\$)
\end{aligned}$$

$$\begin{aligned}
(q_0, aabb, \epsilon) &\vdash (q_1, aabb, \$) \\
&\vdash (q_2, aabb, \$)
\end{aligned}$$

The first computation reads all characters in the input and ends in an accept state. The other two are not able to finish reading all the characters of the input. Therefore $aabb$ is accepted by this PDA.

Exercise 14.11.2. Here's the PDA state diagram again:


q_0
$(\epsilon, \epsilon \to \$)$
q_1
$(a, \epsilon \to a)$
$(\epsilon, \epsilon \to \epsilon)$
q_2
$(b, a \to \epsilon)$
q_3
$(\epsilon, \$ \to \epsilon)$

1. There are two possible executions that will accept the string ϵ. What are the list of states visited by these two executions?
2. Is a accepted by the PDA? Describe all possible executions.
3. Is b accepted by the PDA? Describe all possible executions.
4. Is ab accepted by the PDA? Describe all possible executions.
5. Is ba accepted by the PDA? Describe all possible executions.
6. Is aab accepted by the PDA? Describe all possible executions.
7. Is abb accepted by the PDA? Describe all possible executions.
8. Is $aabb$ accepted by the PDA? Describe all possible executions.
9. Is $bbaa$ accepted by the PDA? Describe all possible executions.
10. Now can you tell me the language accepted by this PDA?
11. What if I make the first state a non–accepting state? What is the resulting language?

Exercise 14.11.3. Design a PDA that accepts $\{a^n b^{2n} \mid n \geq 0\}$.

Exercise 14.11.4. Design a PDA that accepts $\{a^{2n}b^n \mid n \geq 0\}$.

Exercise 14.11.5. Design a PDA that accepts $\{a^m b^n \mid m = 2n \text{ or } 2m = n\}$.

Here's the formal definition of a PDA.

Definition 14.11.1. A **pushdown automata** (PDA) is $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ (phew!) where

- Q is a finite set of states
- Σ is a finite set of input symbols (the input alphabet)
- Γ is a finite set of stack symbols (the stack alphabet)
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma^*)$ is the transition function. Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$. If

  $$\delta(q, a, b) = \{(p_1, c_1), ..., (p_n, c_n)\}$$

  this means that while at state q, reading character a (allowing the case $a = \epsilon$, i.e., ignore the input tape) from the input string, and there is a character b on top of the stack (allowing the case $b = \epsilon$, i.e., ignore the stack), then you can transition to state p_i after popping off b (popping ϵ is the same as not popping), and pushing c_i (allowing the case $c_i = \epsilon$) onto the stack (pushing ϵ is the same as not pushing). In other words from the fact

  $$(p_i, c_i) \in \delta(q, a, b)$$

  you have the following transition in the PDA's state diagram:

  $$q \xrightarrow{a,b \to c_i} p_i$$

Definition 14.11.2. The following definition models the way we compute with PDAs. A **configuration** or **instantaneous description** for a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a tuple of the form

$$(q, x, y)$$

where $q \in Q$, $x \in \Sigma^*$, $y \in \Gamma^*$. In other words a configuration is just an element of $Q \times \Sigma^* \times \Gamma^*$. We define the **derives** relation, denoted $\vdash$, on $Q \times \Sigma^* \times \Gamma$ as follows: Let (q, x, y) and (q', x', y') be configurations in $Q \times \Sigma^* \times \Gamma^*$. Then

$$(q, x, y) \vdash (q', x', y')$$

if there is some $a, b, c \in \Sigma_\epsilon$ such that

(i) $x = ax'$
(ii) $y = bz$
(iii) $y' = cz$
(iv) $(q', c) \in \delta(q, a, b)$

We also define $\vdash^*$ as follows: Let (q, x, y) and (q', x', y') be configuration in $Q \times \Sigma^* \times \Gamma^*$. Then

$$(q, x, y) \vdash^* (q', x', y')$$

if either $(q, x, y) = (q', x', y')$ or there is a sequence of configurations (q_i, x_i, y_i) $(i = 1, \ldots, n)$ such that

$$(q, x, y) = (q_1, x_1, y_1) \vdash \cdots \vdash (q_n, x_n, y_n) = (q', x', y')$$

Definition 14.11.3. Let $x \in \Sigma^*$ and $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. We say that x is **accepted** by P if there is a computation

$$(q_0, x, \epsilon) \vdash^* (q, \epsilon, y)$$

where $q \in F$ and $y \in \Gamma^*$. Define the **language** accepted by P, denoted $L(P)$, to be the set of strings in Σ^* accepted by P:

$$L(P) = \{x \in \Sigma^* \mid P \text{ accepts } x\}$$

The next example is important and uses nondeterminism as a guessing mechanism:

Example 14.11.1. Let's design a PDA that accepts $\{ww^R \mid w \in \Sigma^*\}$ where $\Sigma = \{a, b\}$. The intuition is

1. Push w onto the stack
2. Guess nondeterministically where the middle is
3. Pop w^R while comparing with the rest of the input

Of course we need to know when the stack is empty. So the first thing we do is to push \$ onto the stack. So here's the PDA:

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "{(\epsilon, \epsilon \to \$)}"] & q_1 \arrow[loop right, "{(a, \epsilon \to a), (b, \epsilon \to b)}"] \arrow[d, "{\epsilon, \epsilon \to \epsilon}"] \\
& q_3 & q_2 \arrow[l, "{(\epsilon, \$ \to \epsilon)}"'] \arrow[loop right, "{(a, a \to \epsilon), (b, b \to \epsilon)}"]
\end{tikzcd}$$

Note that at state q_1, we push some input characters onto the stack. At q_2, we pop contents of the stack for every matching character in the remaining input. There's a transition

$$(\epsilon, \epsilon \to \epsilon)$$

from q_1 to q_2. This represents the nondeterministic guessing of the middle of the string. For the input $abba$, one possible derivation is

$$\begin{aligned}
(q_0, abba, \epsilon) &\vdash (q_1, abba, \$) && \text{using } (\epsilon, \epsilon \to \$) \\
&\vdash (q_1, bba, a\$) && \text{using } (a, \epsilon \to a) \\
&\vdash (q_2, bba, a\$) && \text{using } (\epsilon, \epsilon \to \epsilon)
\end{aligned}$$

and at this point, the machine crashes because there is no valid transition to take. So this machine does not accept $abba$. But that's OK since *this* machine will accept *abba*:

$$\begin{aligned}
(q_0, abba, \epsilon) &\vdash (q_1, abba, \$) && \text{using } (\epsilon, \epsilon \rightarrow \$) \\
&\vdash (q_1, bba, a\$) && \text{using } (a, \epsilon \rightarrow a) \\
&\vdash (q_1, ba, ba\$) && \text{using } (a, \epsilon \rightarrow a) \\
&\vdash (q_2, ba, ba\$) && \text{using } (\epsilon, \epsilon \rightarrow \epsilon) \\
&\vdash (q_2, a, a\$) && \text{using } (b, b \rightarrow \epsilon) \\
&\vdash (q_2, \epsilon, \$) && \text{using } (a, a \rightarrow \epsilon) \\
&\vdash (q_3, \epsilon, \epsilon) && \text{using } (\epsilon, \$ \rightarrow \epsilon)
\end{aligned}$$

Exercise 14.11.6. Design a PDA that accepts

$$L = \{a^n b^{2n} \mid n \geq 0\}$$

make sure your PDA has a transition labeled $(\epsilon, \epsilon \to \epsilon)$. Write down a derivation for $a^4 b^2$ that ends in an accept state.

Exercise 14.11.7. Design a PDA that accepts

$$L = \{a^m b^n \mid m < n\}$$

Write down a derivation for $a^2 b^5$ that ends in an accept state.

Exercise 14.11.8. Can you design a PDA that accepts

$$L = \{a^p b^q a^r \mid p < q, r > 3\}$$

Write down a derivation for $a^2 b^5$ that ends in an accept state.

Exercise 14.11.9. Can you design a PDA that accepts

$$L = \{w \mid w \text{ has an equal number of } a\text{'s and } b\text{'s}\}$$

(Obviously this is not the same as $\{a^n b^n \mid n \geq 0\}$.)

Example 14.11.2. Suppose you have two PDAs, say P_1 and P_2. How would you construct a PDA P such that $L(P) = L(P_1) \cup L(P_2)$? Sounds tough ... but ... NONDETERMINISM TO THE RESCUE!!!

You build a new P that includes all the information from P_1 and P_2 but you add a new start state that transitions to the start state of P_1 and P_2 without consuming any characters from the input or modifying the stack; of course the start states of P_1 and P_2 are not not start state of the new P. And that's it! In terms of a diagram, say P_1 has start state q and P_2 has start state r, and s is the start state of P, then the new PDA would look like this:

```
P
+-----------------------------------------+
|                                         |
|                       |                 |
|                       v                 |
|  +------------------- s                 |
|  |(e, e->e)           |(e, e->e)        |
|  |                    |                 |
|  |  P1                |  P2             |
|  |  +-----------+     |  +-----------+  |
|  |  |           |     |  |           |  |
|  +----> q ...   |     +----> r ...   |  |
|     |           |        |           |  |
|     +-----------+        +-----------+  |
|                                         |
+-----------------------------------------+
```

Of course the accept states of P is the set of all the accept states of P_1 and P_2, the transitions are all the transitions from P_1 and P_2 together with the transitions from s to q and r through

$$(\epsilon, \epsilon \rightarrow \epsilon)$$

and the set of stack symbols is the union of the stack symbols of P_1 and P_2. (We do have to assume that the states of P_1 are named differently from the states in P_2.)

Example 14.11.3. Now let's think about concatenation ... Suppose you have two PDAs, say P_1 and P_2. How would you construct a PDA P such that $L(P) = L(P_1)L(P_2)$? You just need to transition from the accept states of P_1 to the start state of P_2 right? Whoa ... hold your horses there ... you need to remember that the state of computation during the computation of P_2 is exactly as it should be for P_2 when P_2 starts executing by itself. This means that you need to ensure that the stack is empty – there's no requirement on P_1 to clear the stack when the string is accepted. So suppose q is the start state of P_1; it will stay as the start state in P. Suppose the start state of P_2 is r. Next suppose the accept states of P_1 (there might be more than one!!!) be f_1, f_2, f_3, etc. Create a new state s for clearing the contents of the stack after P_1 accepts: you must also ensure that P_1 pushes a \$ to the bottom of the stack if it did not do so. Then at s, you continually pop contents off the stack until the last one popped off is \$. Once you're done with that you go to P_2. Of course the accept states of P_1 are not *not* accepts states of P_2. The picture looks like this:

```
P
+----------------------------------------------------------+
| P1                                         P2            |
| +-------------+                            +----------+  |
| |             | (e, e->e)   +--+           |          |  |
| |         f1 ------------   | /            |          |  |
| |             | (e, e->e) \ |v  (e, $->e)  |          |  |
|---> q ... f2 -------------> s --------------> r ...   |  |
| |             | (e, e->e) /                |          |  |
| |         f3 ------------                  |          |  |
| |             |                            |          |  |
| +-------------+                            +----------+  |
+----------------------------------------------------------+
```

Note that there are transitions from s to s labeled $(\epsilon, a \to \epsilon)$ for each $a \in \Gamma_1$ (the stack symbols of P_1) this is just clearing the stack of all symbols in the stack. The transition from s to r will clear the last symbol on the stack, i.e. \$, and immediately go into the execution of P_2. Once again the first thing P_1 must do is to push a \$ onto the stack.

debug: pdavariants/pdavariants

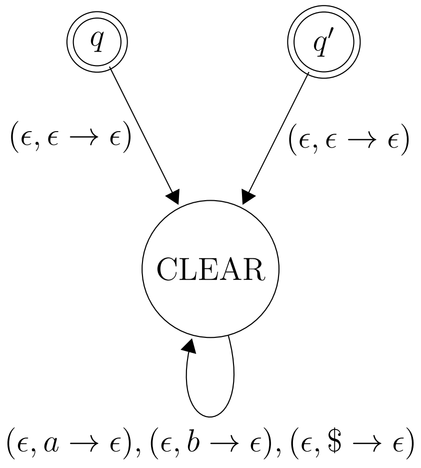
14.12 PDA Variants

There are several variations on our definition of a PDA. It's not too difficult to prove that these definitions are equivalent to our definition. The only exception is the last variant which is a *deterministic* PDA – in the case of DFA, NFA are equal in power to DFA but deterministic PDA is strictly weaker than PDA.

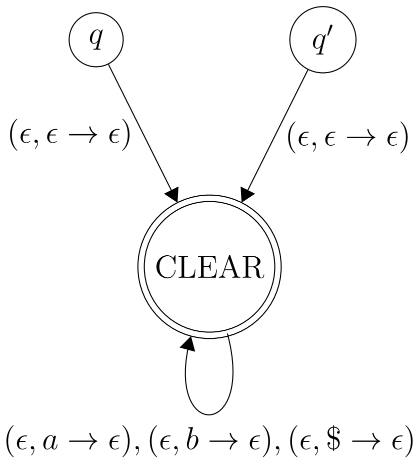
14.12.1 Emptying the stack

Our acceptance condition for a string is based on state information. Note that in our examples, we frequently also end our computation in an accept state and the stack is empty. In some books, acceptance means a computation lands in an accept state *and* the computation *must also end in an empty stack*. A PDA can easily be modified so that when a string is accepted, a computation will land in an accept state with an empty stack. How so?

You add a new state to your PDA, say we call it CLEAR, and you add transition $(\epsilon, \epsilon \to \epsilon)$ from every accept state to CLEAR. This allows the machine to enter CLEAR immediately. Now add transitions from CLEAR to CLEAR that removes everything in the stack. For instance if the set of stack symbols is $\{a, b, \$\}$, then you want to add transitions $(\epsilon, a \to \epsilon)$, $(\epsilon, b \to \epsilon)$, and $(\epsilon, \$ \to \epsilon)$ from CLEAR to CLEAR. Here's the picture where q and q' are accept states from the original PDA:


q
q'
$(\epsilon, \epsilon \to \epsilon)$
$(\epsilon, \epsilon \to \epsilon)$
CLEAR
$(\epsilon, a \to \epsilon), (\epsilon, b \to \epsilon), (\epsilon, \$ \to \epsilon)$

At this point your PDA can accept or accept and clear the stack. To make sure that the stack *is* cleared when a string is accepted, you just make the original accept states non–accepting:


q
q'
$(\epsilon, \epsilon \to \epsilon)$
$(\epsilon, \epsilon \to \epsilon)$
CLEAR
$(\epsilon, a \to \epsilon), (\epsilon, b \to \epsilon), (\epsilon, \$ \to \epsilon)$

That's it!

Now all of this is well and dandy ... but the question is ... *why* would anyone want to empty the stack?

Well frequently you want to build a PDA by assemblying a bunch of PDAs (you have already seen that for instance in DFAs/NFAs). For instance, take a look at the construction of a PDA that accepts the concatenation of the languages accepted by two given PDAs. In that case, you want a second given PDA to use the stack. So it would be good for the first given PDA to play nice by clearly the stack.

14.12.2 Pushing a string

Instead of pushing a character (or ϵ) it's possible to define another type of PDA-like automata where everything is the same as our PDA except that you can push a string onto the stack. In other words the transitions are of the form

$$(a, b \to x)$$

where $a \in \Sigma_\epsilon$, $b \in \Gamma_\epsilon$, and $x \in \Gamma^*$. In *our* definition we require $x \in \Gamma_\epsilon = \Gamma \cup \{\epsilon\}$. See the difference? For instance such a PDA-like thingy can have a transition that looks like this:

$$(a, b \to aab)$$

i.e. the input character read is a, and the top of the stack b is replaced by aab (the b goes into the stack first, followed by a, and then by another a). Let's call such automatas PDA$_2$. It seems that PDA$_2$s are more general and perhaps more powerful than our PDAs.

Well ... actually ... NO!

You can prove that if P' is a PDA$_2$, then there is a PDA, P, such that $L(P) = L(P')$ and vice versa.

Why? Obviously PDA$_2$ includes PDA since the definition of a PDA$_2$ is more general. But if you have a PDA$_2$ with the following transition

$$q \xrightarrow{a, b \to c_1 \cdots c_n} p$$

where $c_i \in \Sigma$, you just modify the PDA$_2$ thingy by removing the above transition and adding the following a corresponding series of transitions:

$$q \xrightarrow{a, b \to c_1} q_1 \xrightarrow{\epsilon, \epsilon \to c_2} q_2 \xrightarrow{\epsilon, \epsilon \to c_3} \cdots \xrightarrow{\epsilon, \epsilon \to c_n} q_n$$

where the q_i are new states. In other words, you simply push one character at a time. Then our PDA would accept the same language as the given PDA$_2$.

14.12.3 Initial stack state

Because you almost always need to put a bottom-of-stack marker in a stack, some definitions of a PDA actually allows one to specfy the initial contents of the stack before executing the PDA. In this case, the PDA is defined by $(Q, \Sigma, \Gamma, \delta, q_0, F, z)$ (it has an extra parameter), where $z \in \Gamma^*$ is the initial contents on the stack. So in this case where you need a bottom-of-stack marker, you could have started off with a PDA of the form $(Q, \Sigma, \Gamma, \delta, q_0, F, \$)$ and not bother to push the \$ onto the stack as a first transition.

All the above are variants of our PDA definition. Remember that they are all equal in power to our usual PDA: The language accepted by one PDA variant is also accepted by another variant.

Exercise 14.12.1. Prove what I just said.

14.12.4 Null stack acceptance

We've defined acceptance as accept a string when the computation ends in an accept state.

There's another definition of acceptance where a string is accepted when the computation ends with an empty stack. In this case, you need not end in an accept state. (In fact for such automatas, there is no concept of accept state.) This is called **acceptance by null stack**. Usually to indicate this type of acceptance, I would write $N(P)$ instead of $L(P)$ where P is a PDA of this type. Again you do not get new languages by using this definition of acceptance.

[PREVIOUS VERSION: In other books, acceptance means the PDA must reach the accept state *and* the stack must be empty. All these definitions of acceptance are actually equivalent to ours in the sense that you can restructure your PDA to conform to another definition of acceptance without any problem (or pain).]

Here's how you convert a regular PDA P to another P' so that the $L(P) = N(P')$. First you simply clear the stack after accepting a string (see above on how to do that.) Is that enough? NO! According to the original definition of acceptance for P, a string is accepted only when there is a computation that lands in an accept state – it's possible that during this computation the stack is empty. So the first thing we need to do is in fact to ensure the P can never have an empty stack. So we pick a special symbol, say #, and push # onto the stack before even running P. Now add to the state that clears the stack the transition to clear # as well. So the picture looks like this where s is the start state of P and, say q and q' are accept states of P, and P' is the new

PDA:

```
P'
+---------------------------------------------------+
|                         P                         |
|                        +-------------------+      |
|             e, e->#    |                   |      |
| -> (START) ------------> (s)      (q) (q') |      |
|                        |           |   |   |      |
|                        +-----------|---|---+      |
|                           (e, e->e)|   |(e, e->e) |
|                                    v   v          |
|                                   (CLEAR)         |
|                                    |   ^          |
|                                    |   |          |
|                                    +---+          |
|                                  (e, a->e)        |
|                                  (e, b->e)        |
|                                  (e, #->e)        |
+---------------------------------------------------+
```

In this example, I'm assuming that the set of stack symbols of P is $\{a, b\}$. In general at state CLEAR you have to clear all the symbols in the stack symbols of P from the stack and the new symbol #. Of course for this new PDA we do not need to make CLEAR an accept state since acceptance is defined in terms of the stack being empty, i.e. we're interested in $N(P')$ and not $L(P')$. In fact you can even keep q and q' as accept states too.

One final thing (and I hope you already realized this): Notice that at START, the stack *is* empty. This means that the empty string is accepted regardless of the PDA (of this type)! So if you want to use acceptance by null stack and the language accepted does not contain ϵ, you can declare that your PDA already has # as the initial stack content.

```
P' (initial stack is #)
+--------------------------------------------------+
|                        P                         |
|                       +-------------------+      |
|                       |                   |      |
| -----------------------> (s)     (q) (q') |      |
|                       |           |   |   |      |
|                       +-----------|---|---+      |
|                          (e, e->e)|   |(e, e->e) |
|                                   v   v          |
|                                  (CLEAR)         |
|                                   |   ^          |
|                                   |   |          |
|                                   +---+          |
|                                 (e, a->e)        |
|                                 (e, b->e)        |
|                                 (e, #->e)        |
+--------------------------------------------------+
```

From now on we will move freely between the different variants of the PDA.

Exercise 14.12.2. Design a PDA accepting $\{a^n b^{2bn} \mid n \geq 0\}$ using a suitable variant. Try to use the smallest number of transitions and ... *be nice* ... by clearing the stack after accepting a string. [Hint: Give the stack an initial state. Push two a's for each a in the input.]

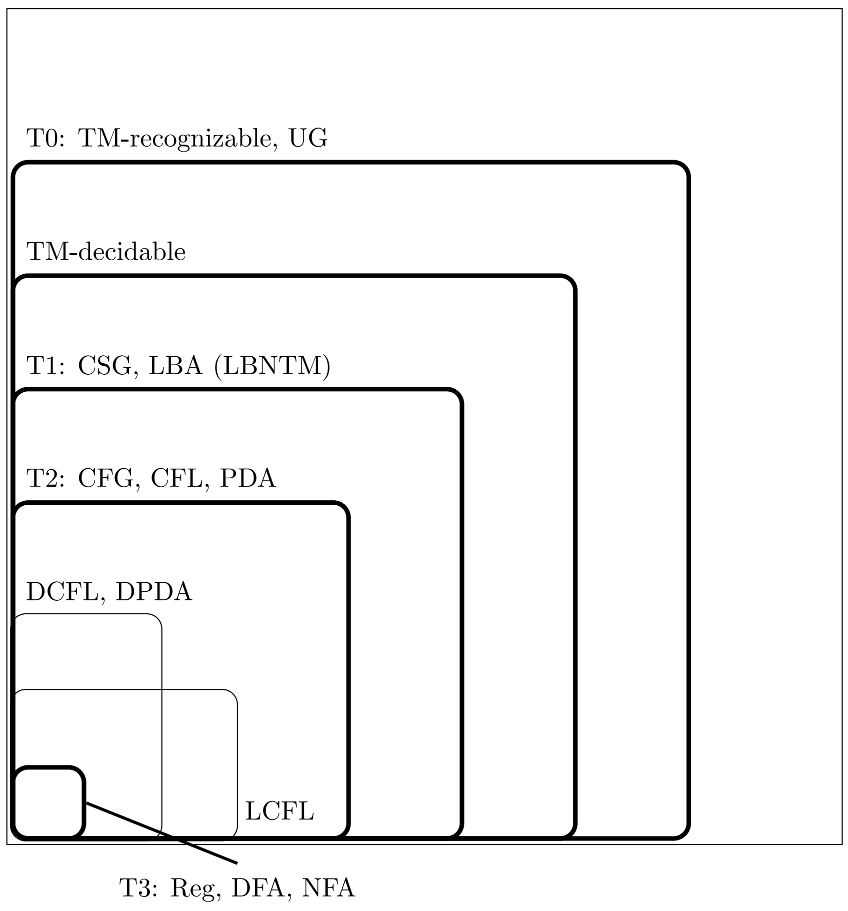
14.12.5 Deterministic PDA (DPDA)

One *final* variant: Note that our PDAs are nondeterministic. You can define a deterministic PDA (DPDA). (You should be able to guess what a DPDA looks like by now, right?)

These are actually *not* equal in power to PDAs.

In other words there are languages accepted by a PDA that is not accepted by any DPDA. Later we will show that PDA languages are the same as CFL and they include regular; DPDA languages are in between. In other words, regular languages are included in DPDA languages and DPDA languages are included in CFLs. Languages accepted by DPDA are said to be DCFL (deterministic CFL).

The reason for DPDA is because of parsing: CFL can be amgiuous and parsing is slow (CYK has a runtime of $O(n^3)$). DCFL are not ambiguous and runtime parsing is $O(n)$. In particular, DCFL are important in compilers for programming languages. We won't talk about these anymore. (See compiler notes.)



debug: cfgpda/cfgpda

14.13 CFG and PDA

I'll call the next theorem our "CFG = PDA" theorem:

Theorem 14.13.1. *L is generated by a CFG iff L is accepted by a PDA.*

Example 14.13.1. Consider the grammar

$$
\begin{aligned}
S &\rightarrow aAB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow a
\end{aligned}
$$

Look at a leftmost derivation:

$$S \rightarrow aAB \rightarrow aaAB \rightarrow aaaB \rightarrow aaabB \rightarrow aaaba$$

You want to be able to "see" a stack and something that tells you where you are in reading the string. Now look at this:

$$\underline{S} \implies a\underline{AB} \implies aa\underline{AB} \implies aaa\underline{B} \implies aaab\underline{B} \implies aaaba$$

Just think of the terms being underlined as terms in a stack. The terminals show what your PDA has read so far. Therefore the PDA's symbols are the variables of the grammar.

Here's the corresponding execution for the PDA we want to build:

remaining input	stack	sentential form in derivation
$aaaba$	S	S
$aaba$	AB	aAB
aba	AB	$aaAB$
ba	B	$aaaB$
a	B	$aaabB$
ϵ	ϵ	$aaaba$

Note in particular that at the end of the computation, the stack is empty. So we use the acceptance rule that a string is accepted if the end of reading all the characters, the stack is empty.

Let's focus on just one derivation:

$$a\underline{AB} \rightarrow aa\underline{AB}$$

What should this correspond to in the computation of the PDA you're trying to build? Your PDA sees an A on top of the stack. What happens after that is that you see a character a and A is back on the stack. In other words this corresponds to the situation where the transition edge of the PDA is labeled $(a, A \rightarrow A)$. Right?

Now what does this step of the computation come from? It comes from the production rule $A \rightarrow aA$, i.e. A is replaced by aA.

Suppose you have a production of the form $A \rightarrow bBA$. How should the corresponding transition be labeled? Do you see that the transition is

$$(b, A \to BA)$$

What is the transition if the rule is $A \to BB$? It's

$$(\epsilon, A \to BB)$$

right?

And what is the transition if the production rule is $A \to aaB$?

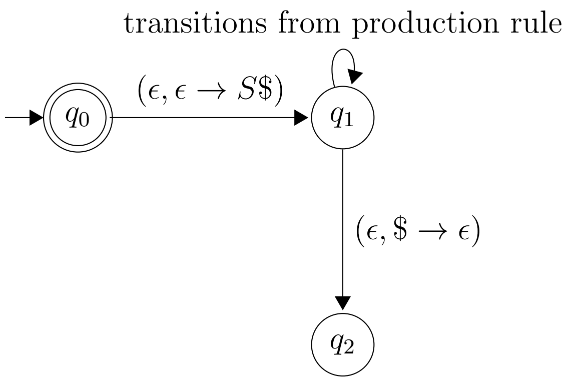
$$(a, A \to aB)$$

For the above example, here are the productions and the corresponding transition labels

production	transition labels
$S \to aAB$	$(a, S \to AB)$
$A \to aA$	$(a, A \to A)$
$A \to a$	$(a, A \to \epsilon)$
$B \to bB$	$(b, B \to B)$
$B \to a$	$(a, B \to \epsilon)$

But wait a minute ... what about the states? The CFG is completely defined in terms of the production rules ... and so far the above is all about transitions ... no mention of states at all!!!

But hang on there. Let's try to understand what we're computing above. We simply continue to compute by looking at the stack. We stop when ... the stack is empty! So we have to use acceptance by null stack. The PDA looks like this:


transitions from production rule
$(\epsilon, \epsilon \to S\$)$
q_0
q_1
$(\epsilon, \$ \to \epsilon)$
q_2

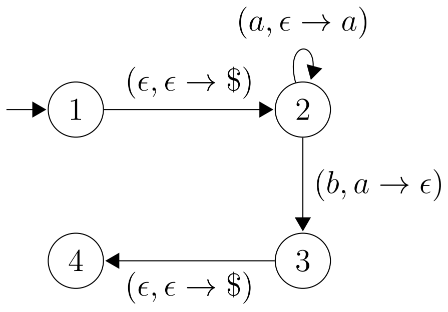
(Note that in the first transition to be executed, I'm pushing a string onto the stack.) Remember that the stack contains a sentential form to be processed.

Exercise 14.13.1. Pick another CFG and convert it to a PDA that accepts the same language that is generated by the CFG. For instance you can try

(a) $\{ww^R \mid w \in \{a, b\}^*\}$,
(b) $\{a^n b^{2n} \mid n \geq 0\}$, etc.

OK, now what about converting a PDA to a CFG? The algorithm is very simple. I'll refer you to our textbook.

Example 14.13.2. Let's look at our PDA that accepts $\{a^n b^n \mid n \geq 0\}$.



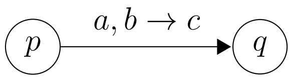
Using the algorithm in the textbook, write down a CFG that accepts the same language as the above PDA.

Let's talk about how to construct a CFG from a PDA. Before doing that we will clean up (simplify) the PDA a little so that the PDA has the following properties:

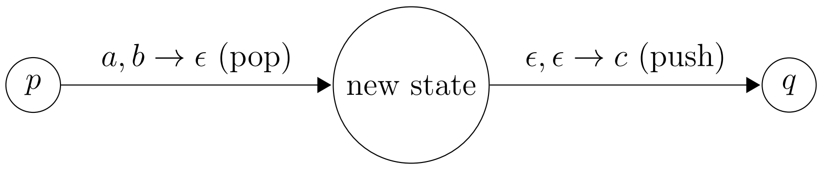
(1) One accept state
(2) Clear stack before accepting
(3) Each transition is push or pop but not both

(1) and (2): EASY! (Right?)

(3) is not too bad:



becomes



Now we assume that the PDA has the above three properties and go on to construct a grammar that accepts the same language.

The main idea is this: As the execution of the PDA go from state p to q, the "work done" is going to be described by a grammar variable V_{pq}, or rather the production rule of V_{pq}. (See the book for details.) Here's the algorithm:

(1) For each state q (in the PDA), we have the production rule

$$V_{qq} \to \epsilon$$

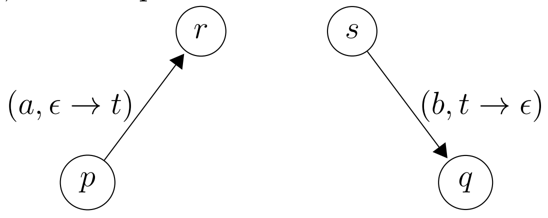
(in the CFG). The intuition here is that if the PDA is at V_q and does not do anything (no transition), then there is no work done.

(2) For states p, q, r (in the PDA), we have the production rule

$$V_{pq} \to V_{pr} V_{rq}$$

(in the CFG). Here, the work done (if any) from p to r is the work done from p to q and q to r.

(3) For states p, q, r, s in this part of the PDA:



we get this production rule in the CFG:

$$V_{pq} \to aV_{rs}b$$

WARNING: the transitions push a t and pops the t ... it's the same stack symbol!!! The intuition here is that the work done from p to q (if any) is a followed by the work done from r to s and b.

Hey ... what about the start symbol? It's

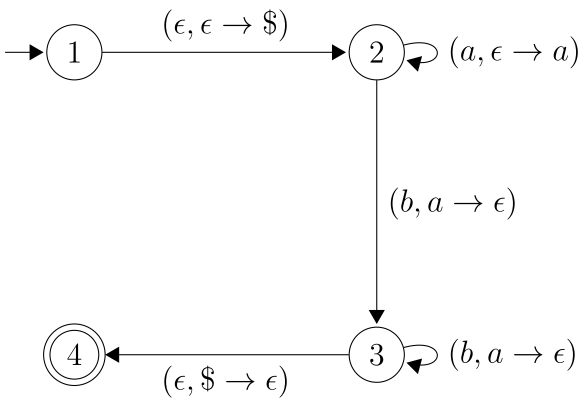
$$V_{q_0, q_{\text{accept}}}$$

where q_0 is the start state and q_{accept} is the unique accept state.

Example 14.13.3. Here's the most famous CFL:

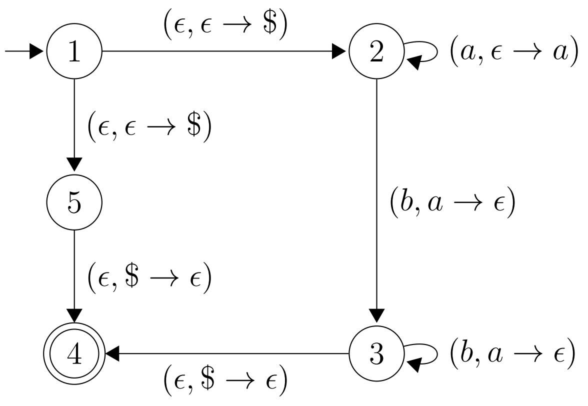
$$L = \{a^n b^n \mid n \geq 0\}$$

Here's a PDA accepting L:



The goal is to construct a CFG using the algorithm.

First we clean up. There are two accepts states; otherwise everything is OK. We make the start state non-accepting and transition to the 4. But each transition must either push or pop so we can't do the above in a single transition. This is what we do:


1
2
3
4
5
$(\epsilon, \epsilon \to \$)$
$(a, \epsilon \to a)$
$(\epsilon, \epsilon \to \$)$
$(b, a \to \epsilon)$
$(\epsilon, \$ \to \epsilon)$
$(b, a \to \epsilon)$
$(\epsilon, \$ \to \epsilon)$

(or create a new transition from state 1 to 3, saving the necessity of state 5.)

Done! Now to build the grammar ...

The variables are

$$V_{11}, V_{12}, \ldots, V_{45}, V_{55}$$

Wow! *25* variables! (Actually that's an overkill ... later you'll see that some are redundant.)

Now for the production rules. First we have these:

$$\begin{aligned} V_{11} &\rightarrow \epsilon \\ &\vdots \\ V_{55} &\rightarrow \epsilon \end{aligned}$$

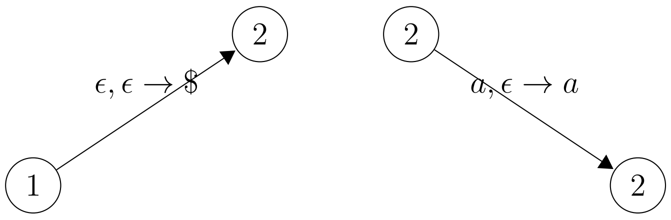
(5 production rules) and these

$$\begin{aligned} V_{11} &\rightarrow V_{11}V_{11} && (p=1, q=1, r=1) \\ V_{12} &\rightarrow V_{11}V_{12} && (p=1, q=1, r=2) \\ V_{13} &\rightarrow V_{11}V_{13} && (p=1, q=1, r=3) \\ &\vdots \end{aligned}$$

(125 production rules!) Now for the third type of production rules.

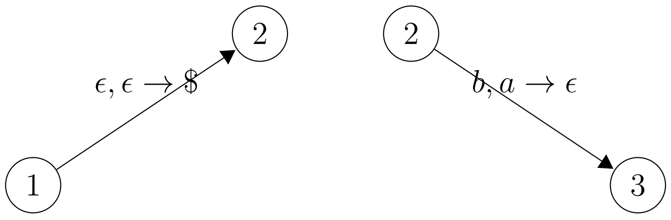
```
                 r       s
p r s q         (1)     (1)
1 1 1 1         ^
               / NO SUCH TRANSITION      No transition from (p) to (r)
             (1)            (1)          NO NEW RULE
              p              q

1 1 1 2       SAME ...                   NO NEW RULE!

etc.
                 r       s
                (2)     (2)
1 2 2 2         ^         \a,e->a
               /e,e->$     v             WHOA! Push $ but pop a!
             (1)            (2)          NO NEW RULE!
              p              q
```

CASE $p = 1, q = 2, r = 2, s = 2$:



WHOA! Push `$` but pop `a`!!!. NO NEW RULE!!!

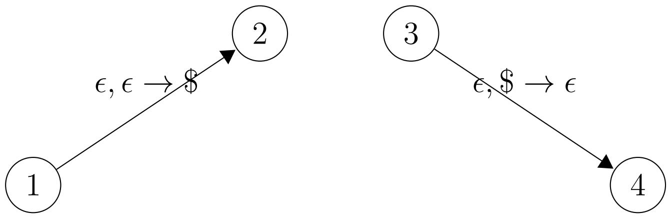
<u>CASE $p = 1, q = 2, r = 2, s = 3$:</u>



WHOA! Push `$` but pop `a`!!!. NO NEW RULE!!!

So far we have not collected any new rules. But ... there *are* some that actually produce rules:

<u>CASE $p = 1, q = 2, r = 3, s = 4$:</u>



AHA! Push \$ but pop \$!!!. Add new rule $V_{14} \to \epsilon V_{23} \epsilon$, i.e., $V_{14} \to V_{23}$.

```
etc.
                   r       s
                  (2)     (3)
2 2 2 3           ^         \b,a->e        AHA! Push a and pop a!!!
                /a,e->a      v             Add new rule V23 -> a V22 b
              (1)           (4)            i.e.  V14 -> V23
               p             q
```

Do you see this:

```
   +--------------------------+
   |        (2)    (2)        |
   |         ^        \       | p,r,s,q = 2,2,2,3
   |       /           v      |
   |     (2)          (3)     |
   +--------------------------+
   |     (2)          (3)     |
   |      ^             \     | p,r,s,q = 1,2,3,4
   |    /                v    |
   |  (1)               (3)   |
   +--------------------------+
```

and then finally $V_{22} \to \epsilon$?

Do you see how to speed up this computation? Look for push and pop of the same character. (Usually easier to make a list of transitions based on push character and another based on pop character – But when the PDA diagram is simple this is not necessary.

OK ... faster now ...

```
                    r       s
p r s q            (5)     (5)
1 5 5 4             ^        \e,$->e         AHA! Push $ and pop $!!!
                  /e,e->$     v              Add new rule V14 -> e V55 e
                (1)           (4)            i.e.  V14 -> V55
                 p             q

etc.
```

Altogether

$$\begin{aligned}
V_{14} &\to V_{23} \\
V_{23} &\to aV_{22}b \\
V_{23} &\to aV_{23}b \qquad \text{(What's the } p, q, r, s \text{ for this one?)} \\
V_{22} &\to \epsilon \\
V_{14} &\to V_{55} \\
V_{55} &\to \epsilon
\end{aligned}$$

where V_{14} is the starting symbol. (I'm only listed the useful rules.) Let's use a simpler notation:

$$\begin{aligned}
S &\to T \\
T &\to aUb \\
T &\to aTb \\
U &\to \epsilon \\
S &\to W \\
W &\to \epsilon
\end{aligned}$$

The last two rules

$$\begin{aligned}
S &\to W \\
W &\to \epsilon
\end{aligned}$$

basically say that S can derive ϵ:

$$\begin{aligned}
&S \to T \\
&T \to aUb \mid aTb \\
&U \to \epsilon \\
&S \to \epsilon
\end{aligned}$$

We get rid of the rule $U \to \epsilon$ to get:

$$\begin{aligned}
&S \to T \\
&T \to ab \mid aTb \\
&S \to \epsilon
\end{aligned}$$

The first two rules can be combined:

$$\begin{aligned}
&S \to ab \mid aSb \\
&S \to \epsilon
\end{aligned}$$

And finally

$$S \to aSb \mid \epsilon$$

Vóila! (You were expecting this CFG right?)

The important thing is to understand the motivation behind this construction. This will then allow you to compute quickly and also understand the proof of this theorem and correctness of the algorithm.

Exercise 14.13.2. Rewrite this part of a PDA so that the transition is replaced by transitions which are push or pop transitions:

$$\begin{tikzcd}
p \arrow[r, "(a, b \to c)"] & q
\end{tikzcd}$$

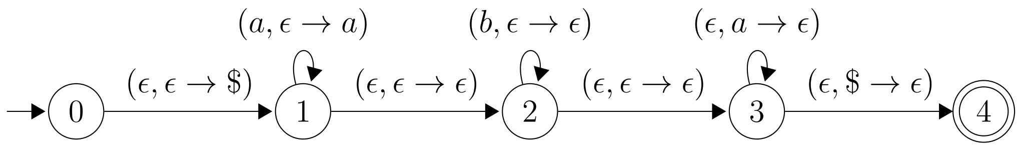
(a, b, c are not ϵ) □

Exercise 14.13.3. Rewrite this part of a PDA so that the transition is replaced by transitions which are push or pop transitions:

$$\begin{tikzcd}
p \arrow[r, "(\epsilon, \epsilon \to \epsilon)"] & q
\end{tikzcd}$$

□

Exercise 14.13.4. Construct a CFG accepting the same language as the following PDA:


$(a, \epsilon \to a)$
$(b, \epsilon \to \epsilon)$
$(\epsilon, a \to \epsilon)$
$(\epsilon, \epsilon \to \$)$
$(\epsilon, \epsilon \to \epsilon)$
$(\epsilon, \epsilon \to \epsilon)$
$(\epsilon, \$ \to \epsilon)$
0
1
2
3
4

□

debug: regularcfl/regularcfl

14.14 Regular ⊆ CFL

In this section, we will look at the big picture: is there a relationship between regular languages and CFLs?

Well ... if you've been reading the notes carefully you should know this by now ...

Theorem 14.14.1. *Every regular language is context free.*

Let L be a regular language. There is therefore a DFA, say M, accepting L. A DFA is simply a PDA that doesn't use its stack!!! In other words you take the state diagram of M, and relabel each transition

$$p \xrightarrow{a} q$$

by

$$p \xrightarrow{a,\epsilon\to\epsilon} q$$

Sweet isn't it? (Of course the same construction works for an NFA too.)

debug: pumpinglemma/pumpinglemma.tex

14.15 Pumping Lemma for CFL

There's corresponding Pumping Lemma for CFL:

Lemma 14.15.1. *If L is a CFL over Σ, then there is some n such that for all $z \in L$, if $|z| \geq n$, then there are $u, v, w, x, y \in \Sigma*$ such that*

1. *$z = uvwxy$*
2. *$|vwx| \le n$*
3. *$|vx| \ge 1$*
4. *for all $i \ge 0$, $uv^iwx^iy \in L$.*

Just like the Pumping Lemma for regular languages, we usually use the contrapositive of the above statement to prove that a language is not a CFL:

Lemma 14.15.2. *Let L is be a language over Σ^*. If for all $n \ge 0$, there is some $z \in L$ such that $|z| \ge n$ and if for any $u, v, w, x, y \in \Sigma^*$ such that*

1. *$z = uvwxy$*
2. *$|vwx| \le n$*
3. *$|vx| \ge 1$*
4. *there is some i_0 such that $uv^{i_0}wx^{i_0}y \notin L$, then L is not a CFL.*

Example 14.15.1. Let $L = \{a^m b^m c^m \mid m \geq 0\}$. Prove that L is not a CFL using the Pumping Lemma for CFL.

Solution. Let $n \geq 0$. Choose $z = a^n b^n c^n$. Note that $z \in L$ and $|z| \geq n$. Now suppose $z = uvwxy$ where

1. $|vwx| \leq n$
2. $|vx| \geq 1$

Note that since $|vwx| \leq n$ and that vwx is a substring of $a^n b^n c^n$, we note that vwx cannot contain more than two symbols in Σ. That means that when we remove v and x from vwx we remove at most two types of symbols. Similar, when we consider uv^0wx^0y, i.e., the string with $uvwxy$ with v and w removed, uv^0wx^0y must have at most two types of characters removed. Note also that since $|vx| \geq 1$, at least one character is indeed removed from $uvwxy$ in order to obtain uv^0wx^0y. This means that uv^0wx^0y cannot be of the form $a^i b^i c^i$ for some i. Hence $uv^0wx^0y \notin L$.

Therefore by the Pumping Lemma for CFL, L is not a CFL. □

Here's a ... DANGEROUS BEND WARNING! ... Note that the part being pumped is *not* a prefix; compare this with the Pumping Lemma for regular languages. Therefore for applications of Pumping Lemma for CFL, the section to be pumped vwx can be anywhere in z. In the above example we chose $z = a^n b^n c^n$. And the strategy was to look at all the possible vwx:

```
      n terms       n terms       n terms
    <--------->   <--------->   <--------->
z = a.........a | b.........b | c.........c


     <------->
           <------->
                   <------->
                          <------->
                                 <------->
```

As you can see we have to consider cases where vwx is not a prefix. And of course we choose our z so as to control possibilities of vwx, i.e. so that it can be made up of at most two type of characters.

Exercise 14.15.1. Prove that $L = \{a^m b^m a^m b^m \mid m \geq 0\}$ is not a CFL.

Exercise 14.15.2. Prove that $L = \{a^m b^n a^m b^n \mid m, n \geq 0\}$ is not a CFL.

Exercise 14.15.3. Prove that $L = \{a^k b^{2k} a^k b^{2k} \mid k \geq 0\}$ is not a CFL.

Exercise 14.15.4. Is $L = \{ab^k a^k b^k \mid k \geq 0\}$ a CFL.

Exercise 14.15.5. Let $L = \{aba^2ba^3b \cdots a^nb \mid n \geq 1\}$. Is L regular? Context free? Neither?

Exercise 14.15.6. Is $L = \{a^{n^2} \mid n \geq 1\}$. context free?

Exercise 14.15.7. Is $L = \{a^p \mid p \text{ prime}\}$ context free? $\square$

Just like in the case of using the pumping lemma for regular languages, sometimes the language is "messy" and you want to tidy it up. Recall that if L_1 is a CFL and L_2 is regular, then $L_1 \cap L_2$ is a also a CFL. So the following example is not too surprising ...

Example 14.15.2. Let $L = \{w \mid w \text{ has equal number of } a\text{'s}, b\text{'s, and } c\text{'s}\}$. Prove that L is not a CFL.

SOLUTION. Assume on the contrary that L is a CFL. Note that $L(a^*b^*c^*)$ is regular. Then $L \cap L(a^*b^*c^*)$ is also CFL. Now note that

$$L \cap L(a^*b^*c^*) = \{a^m b^m c^m \mid m \geq 0\}$$

We have already show that $\{a^m b^m c^m \mid m \geq 0\}$ is not a CFL. Hence L is not a CFL. □

Exercise 14.15.8. Is $\{ww \mid w \in \{a,b\}^*\}$ a CFL?

You've seen this before ...

Exercise 14.15.9. Is $\{a^{n^2} \mid n \geq 0\}$ a CFL?

You want to look at the notes for regular languages for showing that this language is not regular. Analyze the above example very carefully and think about languages like $\{a^{f(n)} \mid n \geq 0\}$ where $f(n)$ is some basic function such as $f(n) = n^2$. How does regularity and context freeness compare for such languages?

Proof. OK. Enough of using PL for CFL. It's time to prove it!

First of all, let's assume L is generated by a grammar G in Chomsky Normal Form. Recall that this means that all productions are either $A \to BC$ or $A \to a$ where A, B, C are variables and a is a terminal.

How can we prove the theorem? Why is there two parts that can be pumped up? (The PL for regular languages have only one).

Suppose we start deriving a string from in L using G. The fact that you can pump the string uv^iwx^iy means that there are infinitely many strings in L. If you recall the proof of the PL for regular languages, you have the same idea and the idea was that you can a loop in the directed graph of transitions so that by going around the circle, you can pump up the string. What about the case of our CFL L? Look at the following:

$$S \overset{*}{\implies} \alpha_1 A \beta_1 \overset{*}{\implies} \alpha_2 A \beta_2 \overset{*}{\implies} uvwxy$$

Suppose $A \overset{*}{\implies} vAx|w$. Let's redo the above:

$$S \overset{*}{\implies} \alpha_1 A \beta_1 \overset{*}{\implies} \alpha_1 (vAx) \beta_1$$

AHA! Now of course we can just derive w from A at this point. But ..., we can instead do this:

$$S \overset{*}{\implies} \alpha_1 A \beta_1 \overset{*}{\implies} \alpha_1 vAx \beta_1 \overset{*}{\implies} \alpha_1 vvAxx \beta_1$$

AHA! We pumped up! Of course from the previous derivation, we could have pumped down instead:

$$S \overset{*}{\implies} \alpha_1 A \beta_1 \overset{*}{\implies} \alpha_1 w \beta_1$$

All we need to say now is that $\alpha_1 \implies^* x$, $\beta_1 \implies^* y$. Everything is done ... except that we still need to prove that there is a production of the form $A \overset{*}{\implies} uAx$.

Now look at a derivation $S \overset{*}{\implies} x$. Supposing that G is already in Chomsky Normal Form, then variable is replaced by two (or by a terminal. That means that if the are n steps in the derivation, then there are at most 2^n leaves (which in this case would be terminals). Now look at $S \overset{*}{\implies} \alpha_1 A \beta_1 \overset{*}{\implies} \alpha_2 A \beta_2 \implies^* z$, specifically look at the derivation tree. There is a path from S to the first A and then to the second A. This is a path of variables. Now why is it that there is a repeat in a variable? That's because there is a path of length at least $k+1$ and k is the number of variables! Pigeonhole Principle again! (Do you see where the $|vwx| \leq n$ and $|vx| \geq 1$ comes from?) □

debug: leftrecursion/leftrecursion.tex

14.16 Left recursion

A production rule is **left recursive** if it is of the form

left recursive

$$A \to Aw$$

where $A \in V$ and $w \in (V \cup T)^*$. We also say the above production rule is a **left recursion**.

left recursion

A **right recursion** is of course a production rule of the form

left recursion

$$A \to wA$$

debug: ll/ll.tex

14.16 LL grammars

Example 14.16.1. Consider the follow CFG:

$$\begin{aligned} S &\to aA \mid B \mid \epsilon \\ A &\to a \mid bA \mid S \\ B &\to bBA \mid aB \mid S \end{aligned}$$

Suppose we want to derive baa.

1. I start off with
$$S \implies ?$$
2. On reading the first symbol of the string $\underline{b}aa$, I see b. The production rule to use cannot be $S \to aA$ or $S \to \epsilon$. It has to be $S \to B$ and then $B \to bBA$. We now have
$$S \implies B \implies \underline{b}BA$$
Note that this is the only way to produce the b on the left from S (see underlined b). There is also $S \implies B \implies S \implies bBA$, but you can see that the $S \implies B \implies S$ is redundant.

3. The leftmost variable of $b\underline{B}A$ is B. I read the next symbol on the input $b\underline{a}a$ and see a. Therefore I need to somehow derive $B \implies^* a....$ The only option is to use $B \to aB$. The derivation is now

$$S \implies B \implies bBA \implies baBA$$

(Hmmm ... do you see a stack in bBA? i.e., $b\underline{BA}$ where B is the top of the stack and A is at the bottom?)
4. Now the leftmost variable of $ba\underline{B}A$ is B and the next input symbol of $b\underline{a}a$ is a If I choose to use $B \to aB$, I will get

$$S \implies B \implies bBA \implies baBA \implies baaBA$$

since the next symbol in the input string is $ba\underline{a}$ (see underlined).
5. At this point, I reach the end of input string. Therefore I need $B \to \epsilon$ which can be achieved by using $B \to S$ followed by $S \to \epsilon$. Altogether, I get

$$\begin{aligned} S &\implies B \implies bBA \implies baBA \implies baaBA \implies baaBA \\ &\implies baaSA \implies baa\epsilon A = baaA \end{aligned}$$

6. In the same way, since I'm done deriving the input baa, I want to derive ϵ from A, which is through $A \implies S \implies \epsilon$. Altogether the derivation is

$$\begin{aligned} S &\implies B \implies bBA \implies baBA \implies baaBA \implies baaBA \\ &\implies baaSA \implies baa\epsilon A = baaA \implies baaS \implies baa\epsilon = baa \end{aligned}$$

An **LL grammar** is a CFG that can be parsed using LL derivations, i.e., derivations which are parsed left-to-right (i.e., the decision on choosing which production rule is based on reading the input string character-by-character left-to-right) and by producing a leftmost derivation (i.e., the leftmost variable is chosen to be replaced). The LL stands for "<u>L</u>eft-to-right" and "<u>L</u>eftmost derivation".

An $LL(k)$ **grammar** is an LL CFG that can be parsed with k lookaheads, i.e., every $\leq k$ symbols read from the input string is sufficient to determine correctly which production rule to use for a leftmost derivation. This means that for any word in generated by the grammar, during the derivation process, when we arrive at

$$S \implies \ldots \implies aabaabaA...$$

where the leftmost variable is A and the next $\leq k$ input symbols are $aabbb$, then there is a unique sequence of production rules that will derive

$$A \implies aabbb...$$

The parser (the algorithm) that determines the correct production rule for such a derivation is called an $LL(k)$ **parser**.

$LL(k)$ parser

Note that only the k symbols read from the input (and the leftmost variable) is used to determine the right production rule to use. For instance this means that if the derivation arrives at a stage where

$$S \implies \ldots \implies aaba\underline{BAaaaaAB}$$

then there are terminal on the underlined sentential form and the parser would also need to look at the sentential form and the input. This can be easily modified simply by add some rules. For instance suppose the $aaaa$ was created by a rule $A \to Aaaaa$, then we just add replace the above rule by $A \to AZZZZ$ and $Z \to a$, effectively changing this rule into Greibach normal form.

I will now focus on the $LL(1)$ parsers, i.e., I will only read one input symbol. Of course the main task is this: when given a variable, say A, and an input symbol, say a, we need to find a sequence of derivations that does the following:

$$A \overset{*}{\implies} a...$$

i.e., A derives a sentential form that begins with a. One possibility is that there's a production rule of the form

$$A \implies aBCDEF$$

But that's not that only case. The a might be derives not by A but by another variable. For instance

$$A \implies BCD \implies a....$$

where $A \to BCD$ is a production rule and $B \to aBEF$ is another production rule. But not only that: it's also possible that the a is actually produced by C or D. That happens if $B \to \epsilon$ is a production rule so that the derivation might be

$$A \implies BCD \implies \epsilon CD \implies \epsilon aaaaGHID....$$

where $C \to aaaaGHI$ is a production rule.

So the base case is to consider the first terminals that A can product by itself.

This involves looking at production rules with A on the left:

$$A \to ?$$

For instance if

$$A \to abbbbBCD$$

then I know "A can produce a" and I want to document that. The set of terminals that A can produce is usually called the **first set** of A. This is usually denoted by $\textbf{First}(A)$.

first set

$\textsc{First}(A)$

Then we have to consider the case where A produces a terminal not directly but through other variables. This means we look at all production rules of the form

$$A \to BCDEF$$

In the following a denotes a terminal.

- If B can derive a... then put a into $\textsc{First}(A)$.
- If B can derive ϵ and C can derive a, then put a into $\textsc{First}(A)$.
- If B, C can derive ϵ and D can derive a, then put a into $\textsc{First}(A)$.
- Etc.
- If B, C, D, E, F can derive ϵ, then A can derive ϵ.

Two things: First, instead of saying "B can derive ϵ", it's also convenient in this case to put ϵ into $\textsc{First}(B)$. (Previously we say that B is nullable and put B is the NULLABLES set.) Second, I need to consider the case when B is a terminal, say

$$A \to bCDEF$$

where b is a terminal. Then I have to put b into $\textsc{First}(A)$. To make less of a fuss so that the the rules are simplified and need not change too much, I might as well define $\textsc{First}(b) = \{b\}$.

With the above, here's a rewrite of the above

- For every terminal a, set $\textsc{First}(a)$ to $\{a\}$.
- For every variable A, set $\textsc{First}(A)$ to $\emptyset$.
- For each production rule of the form $A \to \epsilon$, put ϵ into $\textsc{First}(A)$.
- For each production rule of the form $A \to BCDEF$ where B, C, D, E, F are terminals or variables:
  - Put $\textsc{First}(B) - \{\epsilon\}$ into $\textsc{First}(A)$.
  - If $\epsilon \in \textsc{First}(B)$, then put $\textsc{First}(B) - \{\epsilon\}$ into $\textsc{First}(A)$.
  - If $\epsilon \in \textsc{First}(B)$, $\epsilon \in \textsc{First}(C)$ then put $\textsc{First}(D) - \{\epsilon\}$ into $\textsc{First}(A)$.

- Etc.
- If $\epsilon \in \textsc{First}(B)$, $\epsilon \in \textsc{First}(C)$, ..., $\epsilon \in \textsc{First}(F)$ then put ϵ into $\textsc{First}(A)$.

(Again, note that $\epsilon \in \textsc{First}(A)$ is the same as saying the A is in the sets of nullables.) The above computes $\textsc{First}(A)$ for every variable A.

You can also compute $\textsc{First}(w)$ where w is a string of terminals and variables. (The reason for doing this is because we will be using such FOLLOW sets in later computations.) Suppose $w = w_1 \cdots w_n$ where each w_i is either a terminal or a variable.

1. Set $\textsc{First}(w)$ to $\{\}$.
2. Put all values of $\textsc{First}(w_1) - \{\epsilon\}$ into $\textsc{First}(w)$.
3. If $\epsilon \in \textsc{First}(w_1)$, put all values of $\textsc{First}(w_2) - \{\epsilon\}$ into $\textsc{First}(w)$.
4. If $\epsilon \in \textsc{First}(w_1), \epsilon \in \textsc{First}(w_2)$, put all values of $\textsc{First}(w_3) - \{\epsilon\}$ into $\textsc{First}(w)$.
5. Etc.
6. If $\epsilon \in \textsc{First}(w_1), ..., \epsilon \in \textsc{First}(w_n)$, put ϵ into $\textsc{First}(w)$.

So now with this definition, $\textsc{First}(w)$ is the set of all first terminals that can appear in a derivation starting with w: $w \implies^* aw'$ iff $a \in \textsc{First}(w)$ and $w \implies^* \epsilon$ iff $\epsilon \in \textsc{First}(w)$.

Example 14.16.2. Consider the follow CFG:

$$\begin{aligned} S &\rightarrow aA \mid B \mid \epsilon \\ A &\rightarrow a \mid bA \mid S \\ B &\rightarrow bBA \mid \epsilon \end{aligned}$$

Here are the first sets

x	$\textsc{First}(x)$
a	a
b	ϵ, b
S	ϵ, a, b
A	ϵ, a, b
B	ϵ, b

Note that the above is what you usually see in compiler books. Usually besides the fact that $\textsc{First}(A) = \{a\}$ you also want to know which production rule allows A to derive a sentential form that begins with a. If the rule is $A \to aBC$, then you may want to record $\textsc{First}(A) = \{(a, A \to aBC)\}$ and of course if the first set of A is huge you want to search for a to be fast. In particular the data structure on the right can be an array (or a hashtable). And when all the first set of variables are collected together, it might be arranged as a 2D array with rows labeled by variables and columns labeled by terminals.

For the above example

	ϵ	a	b
S	$S \implies \epsilon$	$S \implies aA$	$S \implies B \implies bBA$
A		$A \implies a$	$A \implies bA$
B	$B \implies \epsilon$		$B \implies bBA$

(Of course the production rules $S \to B$ and $B \to bBA$ can be combined, i.e., I could have run through the grammar to remove unit productions.) If I want to derive aa from the above grammar, using the table I do

$$\begin{aligned} S &\implies aA && \text{using } (S, a) \\ &\implies aa && \text{using } (A, a) \end{aligned}$$

For the derivation to be successful, of course there there cannot be two options in each cell of the above table.

Note that the grammar also generate ba because of the derivation

$$S \implies B \implies bBA \implies b\epsilon a = ba$$

However if I follow the 1–lookahead strategy (i.e., use the table), I would read the b in $\underline{b}a$ and derive

$$S \implies B \implies bBA$$

Reading the next input symbol, I see a, and the leftmost variable now is B. However (B, a) does not give me anything, i.e., B cannot derive a first symbol of a.

Why?

Because it's not the B that derives the a. It's actually the A after the B that derives the a.

More generally, if you have the following derivation

$$ABCDEF \stackrel{*}{\Longrightarrow} a...$$

as part of complete derivation starting with S, it might not be due to A deriving $a...$ (like what we talked about above regarding the First(A)). It might be due to B (or C or D or ...). This means that in the above derivation A produces ϵ:

$$ABCDEF \stackrel{*}{\Longrightarrow} \epsilon BCDEF \stackrel{*}{\Longrightarrow} \epsilon aCDEF$$

This will not be detected by First(A) sets because it really due to the fact that

- $\epsilon \in$ First(A), $a \in$ First(B), or
- $\epsilon \in$ First(A), $\epsilon \in$ First(B), $a \in$ First(C), or
- etc.

Going back to the earlier example when I was trying to derive ba

$$S \Longrightarrow B \Longrightarrow bBA$$

and I cannot move forward to derive a as a first symbol using B, I now need to have a way to derive $B \Longrightarrow \epsilon$ but only is I know $a \in$ First(A) is right after the B. Now if I delay replacing B by ϵ, and replace A by a, the derivation would have been

$$S \Longrightarrow B \Longrightarrow bBA \Longrightarrow bBa$$

I would say that a "follows" B. In terms of the derivation tree, this means that either the a is under the subtree of node A or the a is *not* under the subtree of node A but appears as the first leaf of the subtree immediately to the right.

More generally, if there is some derivation

$$S \stackrel{*}{\Longrightarrow} wAaw'$$

where a is a terminal and A is a variable, then $a \in$ Follow(A). This can happen through applying a production rule where

$$S \stackrel{*}{\Longrightarrow} ...V... \Longrightarrow ...wAaw'...$$

i.e., through the production rule $V \rightarrow wAaw'$ where the "following" terminal of A is introduced during the same derivation step (on the righthand side of the rule). Of course it's possible that A and a are produced from a single production rule but they are not immediately next to each other. For instance the production rule is

$$V \to wAw'aw''$$

and furthermore $w' \implies^* \epsilon$. This means that w' is a string of variables which are nullable. Of course if ϵ–productions have been removed (see earlier section), this w' does does exist.

A second way is where A and a appears in the derivation *not* through the same production rule:

$$S \overset{*}{\implies} ...VV'... \implies ...(wAw')(w''aw''')...$$

where the production rules are $V \to wAw'$ and $V' \to w''Aw'''$; or there might be more productions between wAw' and $w''aw'''$:

$$S \overset{*}{\implies} ...V\underline{...}V'... \implies ...(wAw')\underline{w''''}(w''aw''')...$$

Note that in this case, if a is not derived in wAw', then it's derived in $w''''(w''aw''')...$, i.e., the string after wAw' and for the a be follow A immediately, this means that the w' in wAw' must derive ϵ and the $w''''(w''aw''')...$ must derive a as the *first* symbol. But that means either w' is not there or at least derives ϵ and that a follows V:

$$S \overset{*}{\implies} ...V\underline{...V'...} \implies ...(wAw')\underline{w''''(w''aw''')...}$$

Therefore to construct Follow sets,

1. Set $\text{Follow}(A) = \{\}$ for all variables A.
2. Set $\text{Follow}(S) = \{\$\}$ (the \$ is just used as an end of string marker and we add \$ to the end of the original input; as usual is to fix the issue of ϵ-productions)
3. For each production $V \to wAw'$ where w, w' are strings of terminals and variables and A is a variable, put values of $\text{First}(w') - \{\epsilon\}$ into $\text{Follow}(A)$.
4. For each production $V \to wA$, put values of $\text{Follow}(V)$ into $\text{Follow}(A)$.
5. For each production $V \to wAw'$ where $\epsilon \in \text{First}(w')$, put values of $\text{Follow}(V)$ into $\text{Follow}(A)$.

Index