CONTEXT-FREE GRAMMARS

CFLs are good for describing infinite sets in a finite way.

They are particularly useful for describing the syntax of programming languages.

All regular languages are context free, but not necessarily vice versa.

The following are examples of CFLs that are not regular:

- $\{a^n b^n | n \geq 0\}$
- { palindromes over $\{a, b\}\} =$
  $\{x \in \{a, b\}^* | x =$ **rev**$x\}$
- { balanced strings of parentheses }

Not all languages are CFLs.

$\{a^n b^n a^n | n \geq 0\}$ is not. We will show this with the pumping lemma for CFLs.

Definition 2.1
A context-free grammar is a 4-tuple (V, Σ, R, S) where

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

The following notation is sometimes called BNF (*Backus-Naur form*).

```
< stmt >::=< if-stmt > | < while-stmt > | < begin-stmt >
          | < assg-stmt >
< if-stmt >::=  if  < bool-expr >  then  < stmt >  else  < stmt >
< while-stmt >::=  while  < bool-expr >  do  < stmt >
< begin-stmt >::=  begin  < stmt-list >  end
< stmt-list >::=< stmt > | < stmt >; < stmt-list >
< assg-stmt >::=< var >:=< arith-expr >
< bool-expr >::=< arith-expr >< compare-op >< arith-expr >
< compare-op >::=< | > | ≤ | ≥ | = | ≠
< arith-expr >::=< var > | < const > |
          (< arith-expr >< arith-op >< arith-expr >)
< arith-op >::= +| − | ∗ |/
< const >::= 0|1|2|3|4|5|6|7|8|9
< var >::= a|b|c|...|x|y|z
```

We can consider the above rules as grammar rules for a context-free language. It defines the set of well-formed expressions in some PASCAL-like language.

The string

while $x \leq y$ **do**
begin
$x := (x+1); y := (y-1)$
end
is generated by the non-terminal $< stmt >$.

To show this, we give a sequence of expressions called sentential forms starting from
< stmt >
and ending with

while $x \leq y$ **do**
begin
$x := (x+1); y := (y-1)$
end

< stmt >
⇒< while-stmt >
⇒ **while** < bool-expr > **do** < stmt >
⇒ **while** < arith-expr >< compare-op >< arith-expr > **do** < stmt >
⇒ **while** < var >< compare-op >< arith-expr > **do** < stmt >
⇒ **while** < var > $\leq$ < arith-expr > **do** < stmt >
⇒ **while** < var > $\leq$ < var > **do** < stmt >
⇒ **while** $x \leq$ < var > **do** < stmt >
⇒ **while** $x \leq y$ **do** < stmt >
⇒ **while** $x \leq y$ **do** < begin-stmt >
⇒ ....

$\boxed{\text{EXAMPLE}}$

The nonregular set

{ palindromes over $\{a, b\}$} =
$\{x \in \{a,b\}^* | x =$ **rev**$x\}$ is generated by the grammar
$S \rightarrow aSa|bSb|a|b|\varepsilon$.

$\boxed{\text{EXAMPLE}}$

$\{a^n b^n | n \geq 0\}$ is a CFL.

It is generated by the grammar $G = (V, \Sigma, R, S)$, where
$V = \{S\}$
$\Sigma = \{a, b\}$
$R = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$

$\boxed{\text{Balanced Parentheses}}$

The set PAREN of balanced strings of parentheses is generated by the grammar

$S \to [S]|SS|\varepsilon$.

In this case $V = \{S\}$.
$\Sigma = \{ \ [\ , \ ] \ \}$.

How do we show that PAREN is generated by this grammar?

First we define when a string x of parentheses is balanced.
Let $L(x)$ be the number of left parentheses in x.
Let $R(x)$ be the number of right parentheses in x.

We define a string x of parentheses to be balanced if and only if

1. $L(x) = R(x)$
2. for all prefixes y of x, $L(y) \geq R(y)$

Theorem Let G be the grammar given above. Then $L(G) = \{x \in \{ \ [\ , \ ] \ \}^* | x$ satisfies 1 and 2 $\}$.

Proof: Induction.

The language $PAREN_n$ of all balanced strings of n distinct types is generated by the grammar:

$S \to \ [_1 \ S \ ]_1 \ | \ [_2 \ S \ ]_2 \ | \ ... \ | \ [_n \ S \ ]_n \ | \ SS \ | \ \varepsilon$

$\boxed{\text{AMBIGUITY}}$

See p97-98 in Sipser.

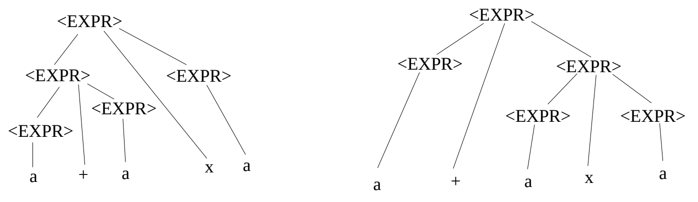
If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously*.

If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

Consider for example the following grammar:
<EXPR>→ <EXPR>+<EXPR> |
    <EXPR>×<EXPR> | (<EXPR>) | a

This grammar generates the string $a + a \times a$ ambiguously.

The following figure shows two different parse trees.



When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees.

A derivation of a string is a *leftmost derivation* if at every step the left most remaining variable is the one replaced.

Definition 2.4
A string w is derived ambiguously in a context-free grammar G if it has two or more different leftmost derivations. A grammar G is *ambiguous* if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.

Some context-free languages however can only be generated by ambiguous grammars. Such languages are called *inherently ambiguous*. The language $\{0^i1^j2^k | i = j \text{ or } j = k\}$ is inherently ambiguous.

$\boxed{\text{CHOMSKY NORMAL FORM}}$

See p99-101 in Sipser.

Definition 2.5
A context-free grammar is in **Chomsky normal form** if every rule is of the form
$A \to BC$
or
$A \to a$
or
$S \to \varepsilon$,
where A, B, C are variables, B and C not the start variable, and a any terminal.

Theorem 2.6
Any context-free language is generated by a context-free grammar in Chomsky normal form.

How do we do this?

- First, add a new start variable S_0 and the rule $S_0 \rightarrow S$

  - This will ensure that the start symbol doesn't occur on the right hand side of a rule.

- Second, remove rules of the form $A \rightarrow \varepsilon$, where A is not the start variable.

  - For each occurence of an A on the right hand side we add a new rule with A deleted.

  - For example the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$ and $R \rightarrow uAvw$ and $R \rightarrow uvw$.

  - If we have the rule $R \rightarrow A$ we add the rule $R \rightarrow \varepsilon$, unless we had previously removed $R \rightarrow \varepsilon$.

  - We repeat these steps until we have no ε rules.

- Third, we handle unit rules.

  - Remove rules of the form $A \rightarrow B$.

  - Whenever a rule $B \rightarrow u$ appears, add $A \rightarrow u$ unless this is a previously removed unit rule (u is a string of variables and/or terminals).

  - Repeat untill no unit rules are left.

- Finally convert all remaining rules into proper form.

  - Replace $A \rightarrow u_1u_2...u_k$ where $k \geq 3$ (and each u_i is a variable or terminal) by $A \rightarrow u_1A_1, A_1 \rightarrow u_2A_2, A_2 \rightarrow u_3A_3, ..., A_{k-2} \rightarrow u_{k-1}u_k$.

  - The A_i's are new variables.

  - If $k \geq 2$ we replace any terminal u_i with a variable U_i and add the rule $U_i \rightarrow u_i$.

Work through Example 2.7 on p100 on your own.

I will do exercise 2.14, p121 in class.

THE COCKE-KASAMI-YOUNGER ALGORITHM

This material is an expansion of the proof of Theorem 7.14 on p240.

Given a CFL A and a string $x \in \Sigma^*$, how can we tell if $x \in A$?

The COCKE-KASAMI-YOUNGER ALGORITHM will do this for us.

Before we use the algorithm, we first convert the grammar to Chomsky normal form. For this algorithm we can allow S on the right-hand side of a rule.

We illustrate the algorithm with the following example.

$S \to AB \mid BA \mid SS \mid AC \mid BD$
$A \to a$
$B \to b$
$C \to SB$
$D \to SA$

We run the algorithm on the input string $x = aabbab$.

Let n be the length of the string, in this case $n = 6$.

Draw $n+1$ vertical lines separating the letters of x and number them 0 to n:

```
| a | a | b | b | a | b |
0   1   2   3   4   5   6
```

For $0 \le i < j \le n$, let x_{ij} denote the substring of x between lines i and j.

In this example $x_{1,4} = abb$ and $x_{2,6} = bbab$. The whole string x is x_{0n}, in our case $x_{0,6}$.

We build a table T with an entry for each pair i, j.

0						
—	1					
—	—	2				
—	—	—	3			
—	—	—	—	4		
—	—	—	—	—	5	
—	—	—	—	—	—	6

The i, jth entry, denoted T_{ij}, refers to the substring x_{ij}.

In T_{ij} we place the nonterminals of G that generate the substring x_{ij} of x. This information is produced inductively, shorter substrings first.

We start with the substrings of length one. These are the substrings of x of the form $x_{i,i+1}$. For each substring $c = x_{i,i+1}$, if there is a production $X \to c$, we write the nonterminal X in the table at location $i, i+1$.

0						
A	1					
—	A	2				
—	—	B	3			
—	—	—	B	4		
—	—	—	—	A	5	
—	—	—	—	—	B	6

Now we proceed to substrings of length 2. This correspond to the diagonal in T immediate below the one just completed.

We break $x_{i,i+2}$ into $x_{i,i+1}$ and $x_{i+1,i+2}$ and check $T_{i,i+1}$ and $T_{i+1,i+2}$.

For each choice $X \in T_{i,i+1}$ and $Y \in T_{i+1,i+2}$, we write Z in $T_{i,i+2}$ if we have a production $Z \to XY$.

In our example, $x_{0,2} = aa$, and we find $A \in T_{0,1}$ and $A \in T_{1,2}$. So we look for a production with AA on the right side. There aren't any, so $T_{0,2}$ is $\emptyset$.

For $T_{1,3}$ we find $A \in T_{1,2}$ and $B \in T_{2,3}$. So we look for a production with AB on the right side and find only $S \to AB$.

We complete the diagonal and end up with:

0						
A	1					
Ø	A	2				
—	S	B	3			
—	—	Ø	B	4		
—	—	—	S	A	5	
—	—	—	—	S	B	6

We now proceed to strings of length 3.

For each such string there are two ways to break it up into two nonnull substrings. For example,

$$x_{0,3} = x_{0,1}x_{1,3} = x_{0,2}x_{2,3}$$

We need to check for both possibilities.

For the first we find $A \in T_{0,1}$ and $S \in T_{1,3}$, so we look for a production with right-hand side AS. There aren't any.

Now we check $T_{0,2}$ and $T_{2,3}$. We find $\emptyset$ in $T_{0,2}$, so there is nothing to check.

We didn't find a nonterminal generating $x_{0,3}$, so we label $T_{0,3}$ with $\emptyset$.

For $x_{1,4} = x_{1,2}x_{2,4} = x_{1,3}x_{3,4}$ we find $A \in T_{1,2}$ and $\emptyset$ in $T_{2,4}$, so there is nothing to check. We find $S \in T_{1,3}$ and $B \in T_{3,4}$, so we look for a production with right-hand side SB and find $C \to SB$. Thus we label $T_{1,4}$ with C.

We continue in this fashion and fill in all entries corresponding to strings of lenght 3, then strings of length 4, etc.

For strings of length 4, there are 3 ways to break them up, all must be checked.

The following is the final result.

0						
A	1					
Ø	A	2				
Ø	S	B	3			
S	C	Ø	B	4		
D	S	Ø	S	A	5	
S	C	Ø	C	S	B	6

We see that $T_{0,6}$ contains S, thus $x_{0,6} = x$ can be derived from S.

Exercise 7.4 p271: Use the Cocke-Kasami-Younger algorithm to do this exercise.

$\boxed{\text{PUMPING LEMMA FOR CFL's}}$

See Sipser p115-119.

Theorem 2.19 Let L be a CFL. There is a number p (the pumping length) such that if $s \in L$ with $|s| \geq p$, s can be divided into 5 pieces $s = uvwxy$ satisfying:

1. $uv^iwx^iy \in L$ for all $i \geq 0$
2. $|vx| > 0$ and
3. $|vwx| \leq p$

Key insights used in the proof

The key insight is that for a grammar in Chomsky normal form, any parse tree for a very long string must have a very long path, and every very long path must have at least two occurences of some variable (nonterminal).

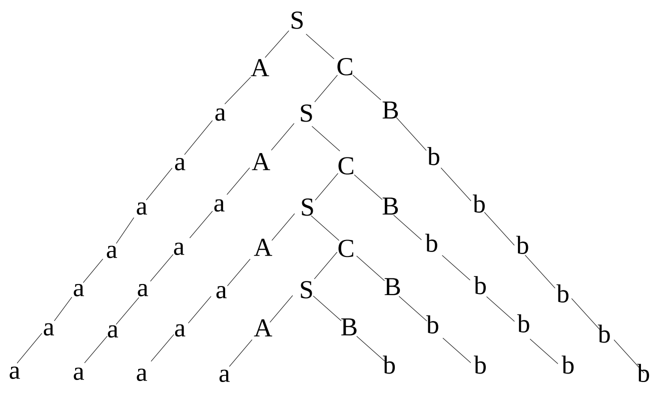
Parse trees of Chomsky grammars for long strings must have long paths, because the number of symbols can at most double when you go down a level. This is because the right hand sides of productions contain at most two symbols.

Take for example the grammar

$$S \to AC \mid AB,\ A \to a,\ B \to b,\ C \to SB$$

and look at a parse tree for a^4b^4.

Duplicate the terminals at each previous level, to keep track of the number of symbols.



The number of symbols at each level is at most twice the number on the level immediately above.

Thus at the most, we can have one symbol at the top (level 0), 2 at level 1, 4 at level 2,..., 2^i at level i.

In order to have at least 2^n symbols at the bottom level, the tree must be of depth at least n, that is, it must have $n + 1$ levels.

Proof: Let G be a grammar for L in Chomsky normal form.

Let $p = 2^{n+1}$, where n is the number of variables (non-terminals) of G. Suppose $s \in L$ and $|s| \geq p$. By the argument on the previous two pages, any parse tree for s must be of depth at least $n + 1$.

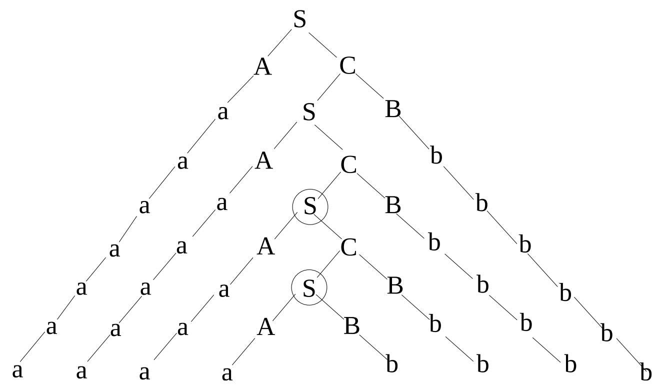
Consider the longest path in the tree. In the example this is the path from S at the root to the leftmost b in the terminal string.

This longest path is of length at least $n + 1$, and thus contains at least $n + 1$ nonterminals.

Some nonterminal must occur twice on this path.

Take the first pair of occurences of a nonterminal reading from bottom to top.

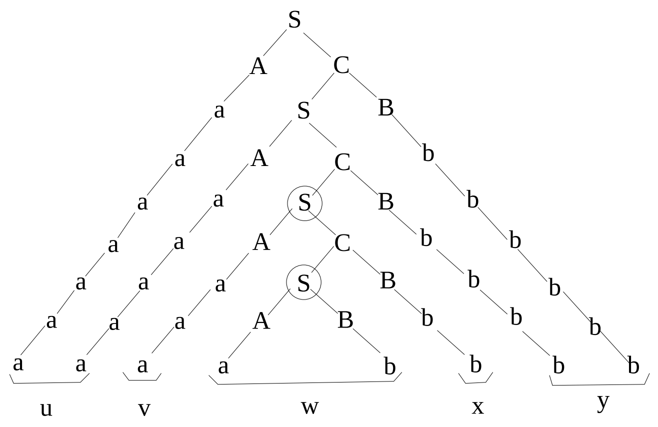
In the example we would take the two circled occurences of S.



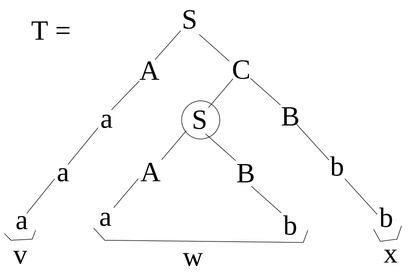
Say X is the nonterminal with two occurences.

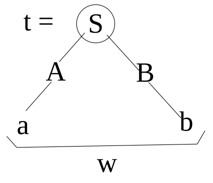
Break s up into $uvwxy$ such that w is the string of terminals generated by lower occurence of X and vwx is the string generated by the upper occurence of X.

In our example, $w = ab$ is the string generated by the lower occurence of S and $vwx = aabb$ is the string generated by the upper occurence of S.

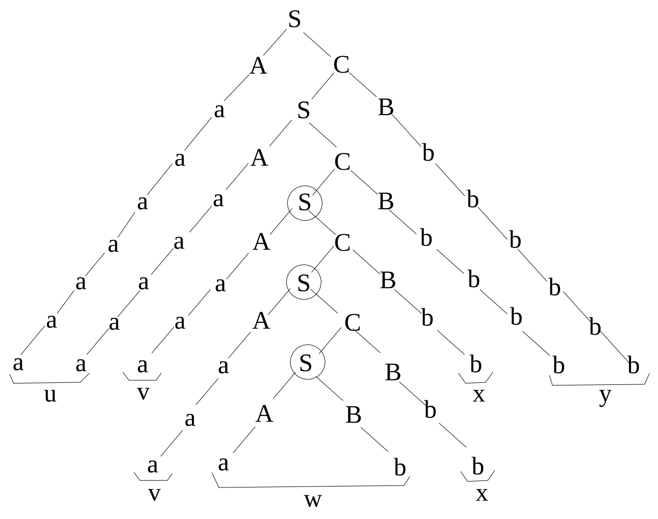


Thus in our example $u = aa, v = a, w = ab, x = b$, and $y = bb$. Let T be the subtree routed at the upper occurence of X and let t be the subtree routed at the lower occurence of X. In our example,



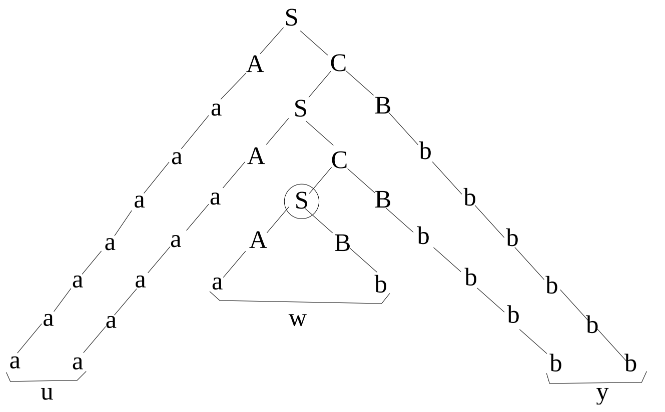


By removing t from the original tree and replacing it with a copy of T, we get a valid parse tree for uv^2wx^2y:



We can repeat this cutting out of t and replacing it with T as many times as we like to get a valid parse tree for uv^iwx^iy for any $i \geq 1$.

We can even cut T out of the original parse tree and replace it with t to get a parse tree for $uv^0wx^0y = uwy$.



Note that $vx \neq \varepsilon$; that is, v and x are not both ε.

We also have $|vwx| \leq 2^{n+1} = p$ since we chose the first repeated occurences of a nonterminal reading from the bottom, and we must have such a repitition within $n + 1$ steps.

$\boxed{\text{PUSHDOWN AUTOMATA}}$

See Sipser p101-106.

Pushdown automata are equivalent in power to context-free grammars.

To show that a language is context-free, we can either give a context-free grammar generating the language or a pushdown automata that recognizes the language.

Definition 2.8
A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is a set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
5. $q_o \in Q$ is the start state
6. $F \subseteq Q$ is the set of accept states

Example 2.9

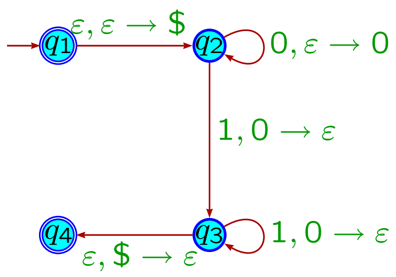
An informal description of the PDA recognizing $\{0^n1^n | n \geq 0\}$:

Read symbols from the input tape. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 of the stack for each 1 read.

If reading the input is finished exactly when the stack becoms empty of 0s, accept the input.

If the stack becomes empty while 1s remain or if 1s are finished while the stack still contains 0s or if 0s appear in the input following 1s, reject.

State diagram for PDA recognizing $\{0^n1^n | n \geq 0\}$



Formal desscription of PDA:

The PDA is given by $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$Q = \{q_1, q_2, q_3, q_4\}$

$\Sigma = \{0, 1\}$

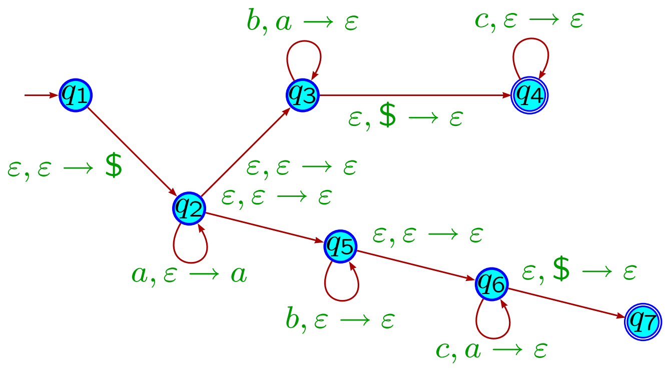
$\Gamma = \{0, \$\}$

$F = \{q_1, q_4\}$

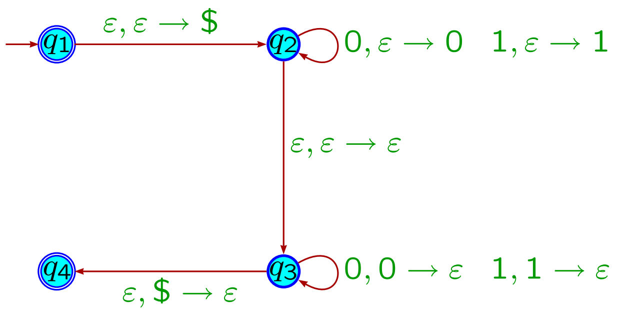
δ is given by the following table where blank entries signifies $\emptyset$.

Input	0			1			ε		
Stack	0	$	ε	0	$	ε	0	$	ε
q1									{(q2,$)}
q2			{(q2,0)}	{(q3, ε)}					
q3				{(q3, ε)}				{(q4, ε)}	
q4									

State diagram for a PDA that recognizes $\{a^ib^jc^k | i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$



State diagram for a PDA that recognizes $\{ww^R | w \in \{0,1\}^*\}$


q_1
$\varepsilon, \varepsilon \to \$$
q_2
$0, \varepsilon \to 0 \quad 1, \varepsilon \to 1$
$\varepsilon, \varepsilon \to \varepsilon$
q_3
$0, 0 \to \varepsilon \quad 1, 1 \to \varepsilon$
q_4
$\varepsilon, \$ \to \varepsilon$

PUSHDOWN AUTOMATA and CONTEXT-FREE GRAMMARS

See Sipser p106-114.

Theorem 2.12 A language is context free if and only if some pushdown automata recognizes it.

Lemma 2.13 If a language is context free, some pushdown automaton recognizes it.

In this lecture we show how to convert a CFG to an equivalent PDA.

Given a CFG G the following is an informal description of an equivalent PDA P.

- Place the marker symbol \$ and the start variable S on the stack.

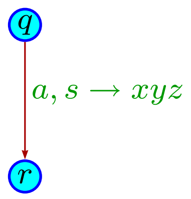
- Repeat the following steps:

- If the top of the stack is the variable A, nondeterministically select one of the rules for A and substitute A by the right-hand side of the rule.

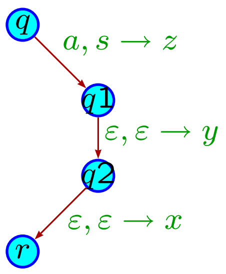
- If the top of the stack is a and the next input symbol is also a, read a from the input and pop a from the stack.

- If the top of the stack is \$, enter the accept state.

We will allow the PDA to push more than one symbol on the stack at a time.

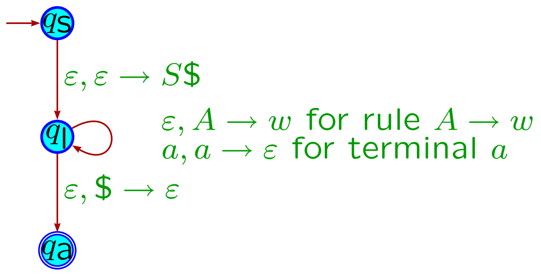


This can for example be done by pushing single symbols as follows:



q
$a, s \longrightarrow z$
$q1$
$\varepsilon, \varepsilon \longrightarrow y$
$q2$
$\varepsilon, \varepsilon \longrightarrow x$
r

The PDA will have states q_{start}, q_{loop}, q_{accept}.



qs
$\varepsilon, \varepsilon \longrightarrow S\$$
ql
$\varepsilon, A \longrightarrow w$ for rule $A \longrightarrow w$
$a, a \longrightarrow \varepsilon$ for terminal a
$\varepsilon, \$ \longrightarrow \varepsilon$
qa

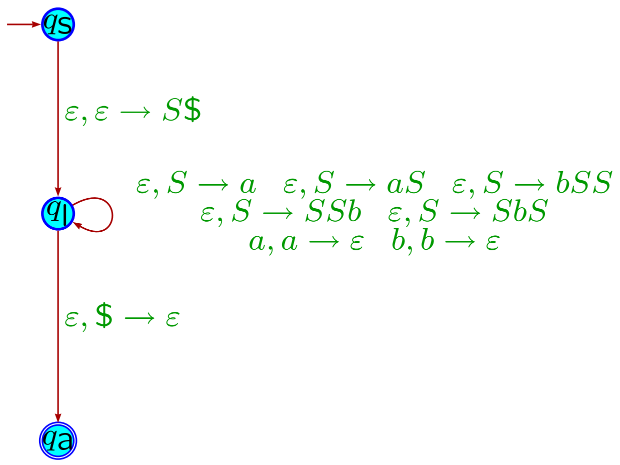
A Top-down PDA for Strings with more a's than b's.

Let $L = \{x \in \{a, b\}^* | n_a(x) > n_b(x)\}$, where $n_a(x)$ denotes the number of a's in x and $n_b(x)$ the number of b's in x.

It can be shown that the following grammar will generate L:

$S \rightarrow a \mid aS \mid bSS \mid SbS \mid SSb$

We now convert this grammar to a PDA. We give the state diagram of the PDA.



qs
$\varepsilon, \varepsilon \longrightarrow S\$$
ql
$\varepsilon, S \longrightarrow a$ $\varepsilon, S \longrightarrow aS$ $\varepsilon, S \longrightarrow bSS$
$\varepsilon, S \longrightarrow SSb$ $\varepsilon, S \longrightarrow SbS$
$a, a \longrightarrow \varepsilon$ $b, b \longrightarrow \varepsilon$
$\varepsilon, \$ \longrightarrow \varepsilon$
qa

Before we continue with the current example, we first introduce more notation.

Saying that (q, x, α) is the current configuration of a PDA means that q is the current state, x is the string of remaining unread input, and α is the current stack content.

We write $(p, x, \alpha) \vdash (q, y, \beta)$ if the PDA can move in one step from the configuration (p, x, α) to the configuration (q, y, β).

We consider the string $x = abbaaa \in L$ and compare the moves made by the PDA in accepting x with a leftmost derivation of x in the grammar.

Each move in which a variable is replaced on the stack by a string corresponds to a step in a leftmost derivation of x.

Observe that at each step the stack contains in addition to \$ the portion of the current string in the derivation that remains after removing the initial string of terminals read thus far.

$(q_s, abbaaa, \varepsilon)$
$\vdash (q_l, abbaaa, S\$)$ (S)
$\vdash (q_l, abbaaa, SbS\$)$ ($\Rightarrow SbS$)
$\vdash (q_l, abbaaa, abS\$)$ ($\Rightarrow abS$)
$\vdash (q_l, bbaaa, bS\$)$
$\vdash (q_l, baaa, S\$)$
$\vdash (q_l, baaa, bSS\$)$ ($\Rightarrow abbSS$)
$\vdash (q_l, aaa, SS\$)$
$\vdash (q_l, aaa, aS\$)$ ($\Rightarrow abbaS$)
$\vdash (q_l, aa, S\$)$
$\vdash (q_l, aa, aS\$)$ ($\Rightarrow abbaaS$)
$\vdash (q_l, a, S\$)$
$\vdash (q_l, a, a\$)$ ($\Rightarrow abbaaa$)
$\vdash (q_l, \varepsilon, \$)$
$\vdash (q_a, \varepsilon, \varepsilon)$

PUSHDOWN AUTOMATA and CONTEXT-FREE GRAMMARS

Lemma 2.15 If a pushdown automaton recognizes some language, then it is context free.

So we start with a PDA and we want to convert it to an equivalent CFG.

First we change the automaton so that it has the following features:

1. It has a single accept state.
2. It empties its stack before accepting.
3. Each transition either pushes or pops a symbol from the stack, but not both.

Say that the PDA P is given by $(Q, \Sigma, \Gamma, \delta, q_0, \{q_a\})$. Here is the equivalent grammar G:

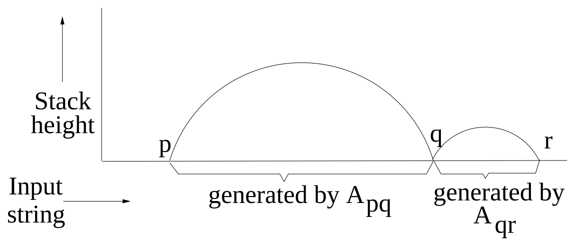
The start variable is A_{q_0,q_a}. Now we describe the rules:

- For each $p, q, r, s \in Q, t \in \Gamma$ and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (q, t) and $\delta(r, b, t)$ contains (s, ε), put the rule $A_{ps} \to aA_{qr}b$ in G.
- For each $p, q, r \in Q$ put the rule $A_{pr} \to A_{pq}A_{qr}$ in G.
- For each $p \in Q$ put the rule $A_{pp} \to \varepsilon$ in G.

For each pair of states p, q, the variable $A_{p,q}$ generates all strings that can take the PDA from state p with empty stack to state q with empty stack.

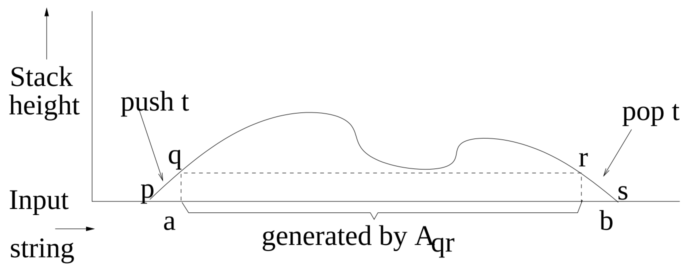
One can prove by induction that the given grammar rules are correct.

In terms of pictures we think of the rules in the following way:



PDA computation corresponding to the rule

$A_{pr} \to A_{pq}A_{qr}$



PDA computation corresponding to the rule

$A_{ps} \to aA_{qr}b$

One can relax the three restriction given on the first page:

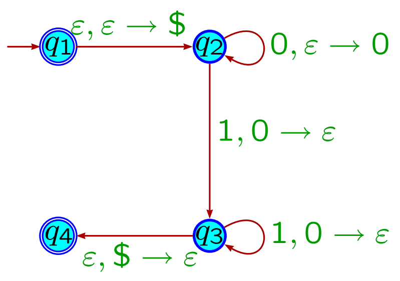
- We can allow more than one accept state. Say for example that you have accept states $q_{a_1}, q_{a_2}, q_{a_3}$. Then we will use S as start variable and add the rule $S \to A_{q_0 a_1} \mid A_{q_0 a_2} \mid A_{q_0 a_3}$, where q_0 is the start state.

- It is not necessary to include a variable A_{pq} if no strings take the PDA from state p with empty stack to state q with empty stack.

- We can allow transitions of the form $\varepsilon, \varepsilon \to \varepsilon$ or $a, \varepsilon \to \varepsilon$ from state p to q. In the first case you add the rule $A_{p,q} \to \varepsilon$ and in the second case $A_{p,q} \to a$.

EXAMPLES

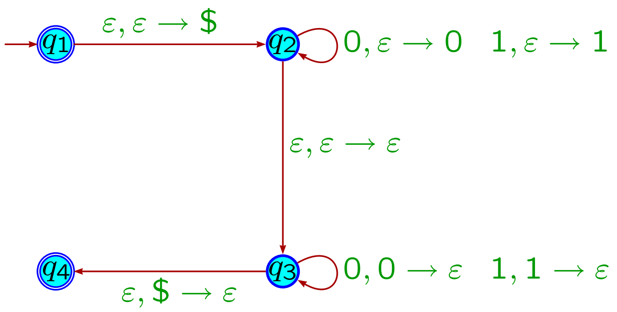
State diagram for PDA recognizing $\{0^n 1^n | n \geq 0\}$



Grammar:

$S \to A_{11} \mid A_{14}$
$A_{ii} \to \varepsilon$ for $i = 1, 2, 3, 4$
$A_{pr} \to A_{pq} A_{qr}$ for $p, q, r \in \{1, 2, 3, 4\}$
$A_{14} \to A_{23}$
$A_{23} \to 0A_{23}1 \mid 0A_{22}1$

State diagram for a PDA that recognizes $\{ww^R | w \in \{0,1\}^*\}$


$\varepsilon, \varepsilon \to \$$
$0, \varepsilon \to 0 \quad 1, \varepsilon \to 1$
$\varepsilon, \varepsilon \to \varepsilon$
$0, 0 \to \varepsilon \quad 1, 1 \to \varepsilon$
$\varepsilon, \$ \to \varepsilon$
$q1$
$q2$
$q3$
$q4$

Grammar:

$S \to A_{11} \mid A_{14}$
$A_{ii} \to \varepsilon$ for $i = 1, 2, 3, 4$
$A_{pr} \to A_{pq}A_{qr}$ for $p, q, r \in \{1, 2, 3, 4\}$
$A_{14} \to A_{23}$
$A_{23} \to 0A_{23}0 \mid 1A_{23}1 \mid \varepsilon$

Exercises from Chapter 2

Exercise 2.2
(a) Use the languages $A = \{a^m b^n c^n | m, n \geq 0\}$ and $B = \{a^n b^n c^m | m, n \geq 0\}$ together with example 2.20 to show that the class of context-free languages is not closed under intersection.

In example 2.20 it is shown with the pumping lemma that the language $C = \{a^n b^n c^n | n \geq 0\}$ is not context-free.

The langauge A is generated by the grammar
$S \to AD$
$A \to aA \mid \varepsilon$
$D \to bDc \mid \varepsilon$
Thus A is context-free. Similarly, B is also context free. But $A \cap B = C$, which is not context-free.

(b) Use part(a) and DeMorgan's law to show that the class of context-free languages is not closed under complementation.

Assume that the class of context-free languages is closed under complementation and let A, B and C be as in part (a).

Then $C = \overline{\overline{C}} = \overline{\overline{A} \cup \overline{B}} = \overline{\overline{A} \cup \overline{B}}$.

But then C will be context-free,

since $\overline{A}$ and $\overline{B}$ are context-free (by the assumption that the complement of a context-free language is context-free),

$\overline{A} \cup \overline{B}$ are also context-free (by problem 2.15 the class of context-free languages is closed under union),

and thus $\overline{\overline{A} \cup \overline{B}}$ is context-free (by the assumption that the complement of a context-free language is context-free).

Exercise 2.17 (b):
We assume that the result in 2.17(a) holds, that is, if C is context-free and R is regular, then $C \cap R$ is context-free.

We assume that $A = \{a^n b^n c^n | n \geq 0\}$ is not context-free and we want to use this to show that $B = \{w | w \in \{a, b, c\}^*$ and w contains equal numbers of a's b's and c's$\}$ is also not context-free. It would of course also be fairly easy to show that B is not context-free by using the pumping lemma.

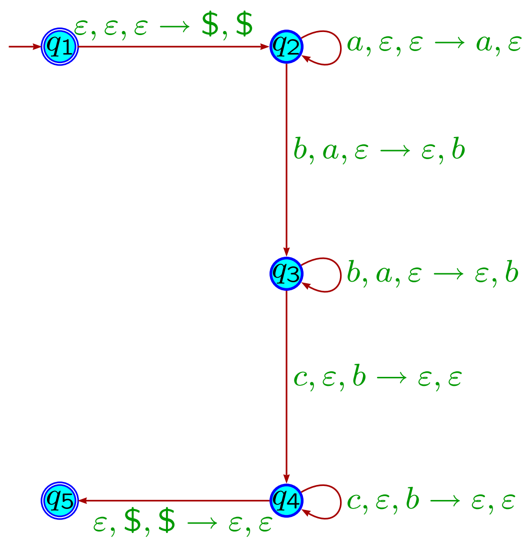
Let R be the regular language described by the regular expression $a^* b^* c^*$.

If B is context-free, then $A = B \cap R$ will also be context-free, but A is not context-free.

My Exercise

On the last two pages we show that if we use two stacks instead of one in a PDA (we will call it a 2-PDA), we can recognize the language $\{a^n b^n c^n | n \geq 0\}$.

In fact, in exercise 3.9 (ok I know this is from the next chapter) one is asked to show that a 2-PDA is more powerful than a PDA, and that a 3-PDA is not more powerful than a 2-PDA. It can be shown that a 2-PDA is just as powerful as a Turing machine.



q1
$\varepsilon, \varepsilon, \varepsilon \rightarrow \$, \$$
q2
$a, \varepsilon, \varepsilon \rightarrow a, \varepsilon$
$b, a, \varepsilon \rightarrow \varepsilon, b$
q3
$b, a, \varepsilon \rightarrow \varepsilon, b$
$c, \varepsilon, b \rightarrow \varepsilon, \varepsilon$
q4
$c, \varepsilon, b \rightarrow \varepsilon, \varepsilon$
q5
$\varepsilon, \$, \$ \rightarrow \varepsilon, \varepsilon$