Administration

- Week 3 midterm: at the end of next week the first midterm will be published. We will have ~2-3 hours (including for technical stuff) once we check it out.
- The final exam will also be online.

Chapter 1 - Recap.

The pumping lemma

Let A be a regular language $\Rightarrow \exists p \in \mathbb{N}\ s.t. \forall s \in A: |s| \geq p \Rightarrow s$ can be written as $s = xyz$ $(x, y, z \in \Sigma^*)$ where:

1) $|y| > 0$
2) $|xy| \leq p$
3) $xy^iz \in A, i \geq 0$

The strings x, z can both be ϵ together.

Consequence:

The language $A = \{0^n1^n | n \geq 0\}$ is not regular. The way to prove it is by finding a string that contradicts the pumping lemma. For instance, choose $s = 0^p1^p$, where p is the pumping length. For that s, $s = xyz$ would mean $y \in \{0\}^+$ necessarily, and then according to the pumping lemma $xy^2z \in A$. But $xyyz$ has more 0's than 1's, thus $xy^2z \notin A$ in contradiction to the pumping lemma. Therefore the pumping lemma cannot apply, then A is not regular.

So we use the pumping lemma to prove that non-regular languages are indeed not regular using proof by contradiction.

Question: is my laptop a DFA?

Answer: yes! The set of states would be the entire domain of memory-configurations (many, many configurations), but it is a finite set. Any stroke of key / button does a transition.

Context-Free Languages

Example:

$E \rightarrow F * F$

$F \rightarrow T \mid (E)$

$T \rightarrow a \mid b$

Where E is the **start symbol** and the rows are **production rules**. The uppercase letters are **variables** and the lowercase are **terminals**. Example of a **derivation**:

$E \rightarrow F * F \rightarrow (E) * T \rightarrow (F * F) * T \rightarrow \cdots \rightarrow (T * T) * a \rightarrow \cdots \rightarrow (b * b) * a$

Where the string derived is constructed of all terminals. We say that this grammar derives this string.

Another example:

$< expr > \rightarrow < expr > + < term > \mid < term >$

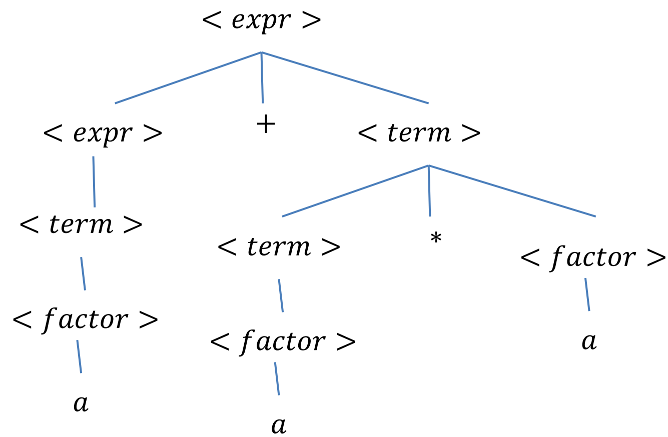
$< term > \rightarrow < term > * < factor > \mid < factor >$

$< factor > \rightarrow (< expr >) \mid a$

With this language we can generate the following expression:

$a + a * a$

In the following way (parse tree):



CFG: Formal definition

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

- V is a finite set of variables
- Σ is a finite set of terminals ($V \cap \Sigma = \emptyset$
- R is a finite set of derivation rules where $R \subset V \times (V \cup \Sigma)^*$
- $S \in V$ is the start state

Let $A \to u$ be a rule, where $A \in V, u \in (V \cup \Sigma)^*$. We say that u **derives** w, denoted $u \overset{*}{\to} w$, if there is a sequence $u \to u_1, u_1 \to u_2, \dots, u_k \to w$.

The $\overset{*}{\to}$ symbolizes the **transitive closure** of the derivation relation

<u>Definition: transitive closure (of a relation)</u>: Given a set S, a relation $R \subset S \times S$, the transitive closure of R is **the** set

$$\bigcap_{\substack{T \subset S \times S \\ R \subset T \\ T \text{ is transitive}}} T$$

The **language of the grammar** $L(G) \coloneqq \left\{ w \in \Sigma^* \mid S \overset{*}{\to} w \right\}$, i.e. there exists a derivation from the start symbol of the grammar G to the words in this language.

<u>Another example</u>:

Let $G_3 = (\{S\}, \{a, b\}, R, S)$ where R are the following rules:

$S \to aSb \mid SS \mid \epsilon$

Example of a production: $S \to aSb \to aSSb \to aaSbaSbb \to aababb$

<u>Interpretation</u>: $a =$"(" and $b =$")" – the opening and closing parenthesis.

Instances like ())(, which is actually $abba \notin L(G_3)$. How do we check that? Using a **pushdown automaton** that accepts $L(G_3)$. Parsing over the input, each a is pushed to the stack, and each b will pop the stack. This way we keep track of the parenthesis.

Another interpretation:

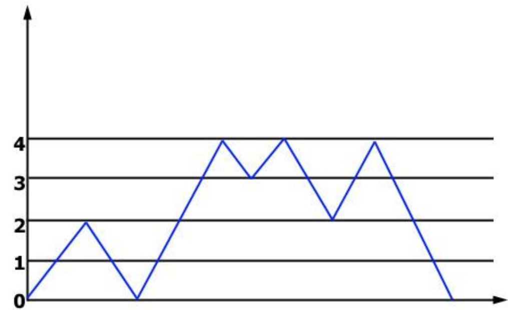
We can simulate a grid, where a will denote going "up" and b – going "down". An input that will be accepted is one that starts at the lowest level of the grid and also finishes there – and we do not go below the "0" line of the grid.

If an input is a line that is always positive, that starts and ends at 0?

In that case, since the function drawn by this line is positive, the first step is "up" and last step is "down", thus we have reduced the question to the line after the first "up" and before the last "down", and look at that smaller line with respect to 1 instead of 0 – but: Somewhere in the middle it may go below 1 to 0 and back to 1, but will still be legal. Therefore this "reduction" is incorrect.

The way to reduce the problem is as follows:

- Break down the line by segments between points on the 0 line
- Then each of these segments are safe to be reduced as suggested above – take the enclosed line without the first and last steps, with respect to 1.
- Repeat recursively.



Another interpretation:

The grammar describes the set of all binary trees, where:

- ϵ is the empty tree
- $SS =$ left and right sub-tree
- $aSb = a$ /\ b
- …?

The point is that we can ask a combinatorial question: how many correctly parenthesized strings of size n are there?

Answer: $C_n = \frac{1}{n+1}\binom{2n}{n}$ - the n'th Catalan number.

And we can use the grammar to get to that number. There are programmatic tools that given a grammar can tell how many words of a given length n does the grammar derives.

Closure under union:

Given 2 grammars G_1, G_2, how do we create the grammar of the union of the languages: Create a new start symbol that derives each of the start symbols of G_1, G_2.

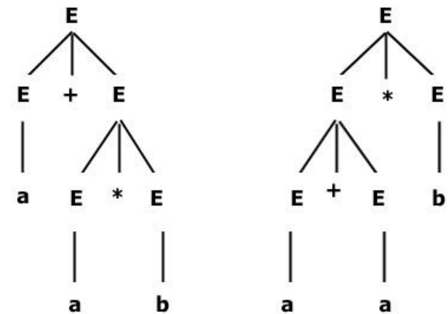
Ambiguity

Example:

$E \rightarrow E + E \mid E * E \mid (E) \mid a \mid b$
Then the string $a + a * b$ can be generated in more than one way:

- $E \rightarrow E + E \rightarrow a + E \rightarrow a + E * E \rightarrow a + a * E \rightarrow a + a * b$
- $E \rightarrow E * E \rightarrow E * b \rightarrow E + E * b \rightarrow a + E * b \rightarrow a + a * b$

In that case we say that the grammar is **ambiguous**.



Chomsky Normal Form (CNF)

Any context free grammar can be converted into CNF, where:

- All rules are of the form $A \rightarrow BC \mid A \rightarrow a, A, B, C \in V, a \in \Sigma, B, C \neq S$
- And an additional rule for the start symbol: $S \rightarrow \epsilon$, and it is the only rule that derives ϵ

Converting to CNF:

1) Removing ϵ derivations:

If $R \rightarrow uAw, A \rightarrow \epsilon$ we can convert it to CNF by eliminating $A \rightarrow \epsilon$ and change R to: $R \rightarrow uAw \mid \epsilon$

If $R \rightarrow uAvAw, A \rightarrow \epsilon$ we convert R to: $R \rightarrow uAvAw \mid uvAw \mid uAvw \mid uvw$ – we simply cover all cases where originally A might have derived ϵ.

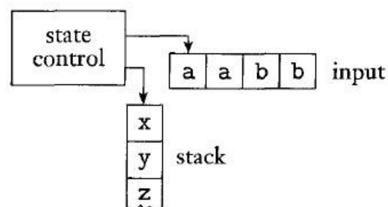
2) Unit rules:

Any rule $A \rightarrow B$ we eliminate by substituting B with anything that B might produce. For instance, if $B \rightarrow u$, we add $A \rightarrow u$. We do this until all unit rules are removed.

If $A \rightarrow u_1 u_2 u_3$ the new convert to: the next part was not clear…

Pushdown Automata (PDA)

Theorem: the languages of context free grammar are the languages of (non-deterministic) PDAs



PDAs that don't use the stack are like DFAs. Using the stack gives them a greater computation power.

Example:

Let $L = \{a^i b^j c^k \mid i = j \ or \ i = k\}$. A PDA for that cannot push down a's, pop an a for every b, since if a's match the c's, after popping the a's we will not be able to check #a's against #c's.

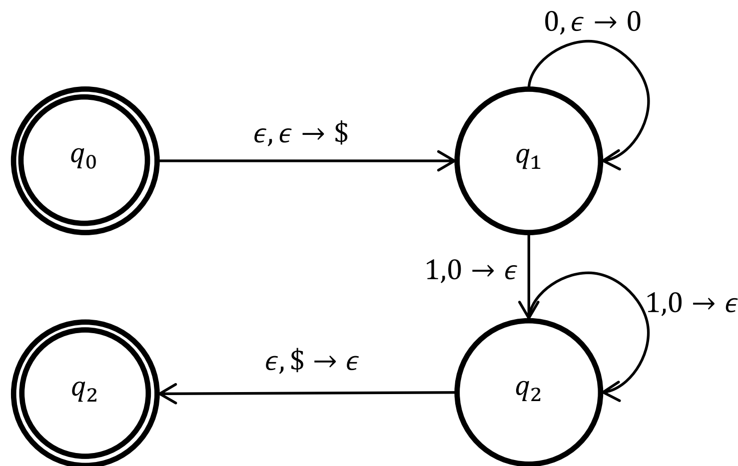
This is where the **non-determinism** of PDAs gets into action. Note that deterministic PDAs are strictly less powerful than non-deterministic PDAs.

Formal definition:

A PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_o, F)$ where:

- Q is a finite set of states
- Σ is the input finite alphabet
- Γ is the stack finite alphabet
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$ where the range of δ is the power set of $Q \times \Gamma_\epsilon$ since it is non-deterministic
- $q_0 \in Q$ is the start state
- $F \subset Q$ is a set of accepting states

A PDA for 0^n1^n:



Where the rules $a, b \to c$ mean that we read a from the input, pop b from the stack and push c to it.

Note that the rule that goes from q_1 to q_2 can be replaced with $\epsilon, \epsilon \to \epsilon$.

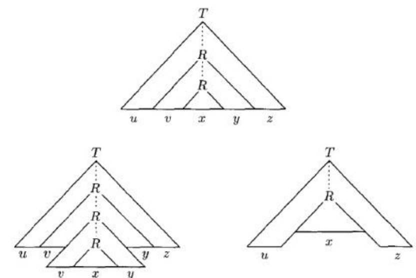
Pumping lemma for CFLs

This is used, like for regular languages, to show that some languages are not context-free.

The lemma:

If $A \in CFL$ then $\exists p \in \mathbb{N}\ s.t. \forall s \in A, |s| \geq p$, then s can be written $s = uvxyz$ where

1) $|vy| > 0$
2) $|vxy| \leq p$
3) $uv^ixy^iz \in A$ for $i \geq 0$



This means that for long-enough words, there will be a repetition of some rule R under itself, such that we can pump it to as many repetitions that we want.