CFGs

CFGs, or nondeterministic pushdown automata, model recursion. Does that add anything to our computational power?

If we have no ability to use a stack, or recursion, we can actually simulate it in a high level way, using variables to model pseudo-stacks. So recursions are not the point here.

But it gives the ability to save current state and push it to the stack, continuing on the rest of the input. The reason we can do that is the fact we have a finite state automaton with finite Q and Σ, thus we can save all pairs in $Q \times \Sigma$ as a configuration saved to the stack. So we can simulate recursion in a finite automaton.

Eventually we add real computing power to finite state automata, giving us the ability to represent CFLs as opposed to just regular languages.

Discussion of Problem Solving #1 (week 2)

1.54)

$F = \{w \mid w = a^i b^j c^k, i, j, k \geq 0, i = 1 \Rightarrow j = k\}$

a.

Following is a proof that F is not regular:

Note that we cannot use the pumping lemma here, so we will use the closure properties of regular expressions.

Consider the languages: $B = \{ab^n c^n \mid n\ is\ an\ integer\}$, $A = \{ab^i c^j \mid i, j\ are\ integers\}$. It is trivial that B is not regular and A is regular. But $B = F \cap A$, therefore F cannot be regular (otherwise B had to be regular).

b.

Following is a proof that F can be pumped:

We will divide it into cases, where i (the number of a's) is 1 and when it's not:

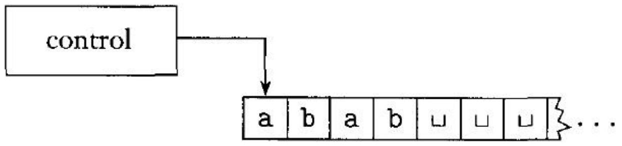
$\underline{i = 1}$: let $s = ab^n c^n$, then we can divide it to $y = a, x = \epsilon, z = b^n c^n$, and then $a^k b^n c^n$ where k is the pumping power is still in F.

$\underline{i \neq 1}$: let $s = a^i b^j c^k$, we choose the partition of s such that y is in b or something like that (and can be pumped as much as we want and stay in the language).

c.

The reason it does not contradict the pumping lemma is because the lemma is not an "iff", i.e. it only states something on regular languages (that if a language is regular, it can be pumped), but not the other way around (if a language can be pumped, it doesn't necessarily mean it is a regular language).

Turing Machines



Formal definition:

A Turing machine is the 7-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where:

- Q is a finite set of states
- Σ is a finite input alphabet
- Γ is a finite tape alphabet, $\Sigma \subset \Gamma$ and " "$\in \Gamma - \Sigma$ (the space character)
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ – the transition function, where L, R indicate moving left or right.
- q_0 is a start state
- q_{accept} is the accept state
- q_{reject} is the reject state, which is different of the accept state.

To define a <u>computation</u> we define first a <u>configuration</u>: any configuration C is $C \in \Gamma^* \times Q \times \Gamma^*$ - the state of the input, where the first string is $u \in \Gamma^*$, the last is $v \in \Gamma^*$ and the state $q \in Q$ is the state of the machine where the head is pointing to the first character in v. A configuration is then written in short: uqv.

Then a computation would be a sequence $C_0, C_1, \dots, C_n$ where:

- $C_0 = q_0 s$ where s is the complete input word
- For any 2 consecutive configurations C_i, C_{i+1}:
  - $C_i = uaqbv \ (a, b \in \Gamma) \Rightarrow C_{i+1} = uacq'v$ where $\delta(q, b) = (q', c, R)$
  - $C_i = uaqbv \Rightarrow uq'acv$ where $\delta(q, b) = (q', c, L)$

A configuration is an <u>accepting configuration</u> if we get to the configuration $uq_{accept}v$, and a <u>rejecting configuration</u> if $uq_{reject}v$.

Definition:

Let M be a Turing machine, then $L(M) = \{w \in \Sigma^* \mid M \ accepts \ w\}$, where <u>accepts</u> means there exists a sequence of configurations starting with $q_0 w$ and ending in an accepting configuration (as defined above). Denote $L \coloneqq L(M)$, then we also say <u>L is recognized by M</u>.

Let $L \subset \Sigma^*$, then <u>M decides L</u> if it accepts L and M rejects all $w \notin L$. Therefore all computations for any input $w \in \Sigma^*$ halt after a finite number of steps (always halts). So if a language is decidable, it is better than if it is only recognizable (the first immediately derives that $\bar{L}$ is decidable).

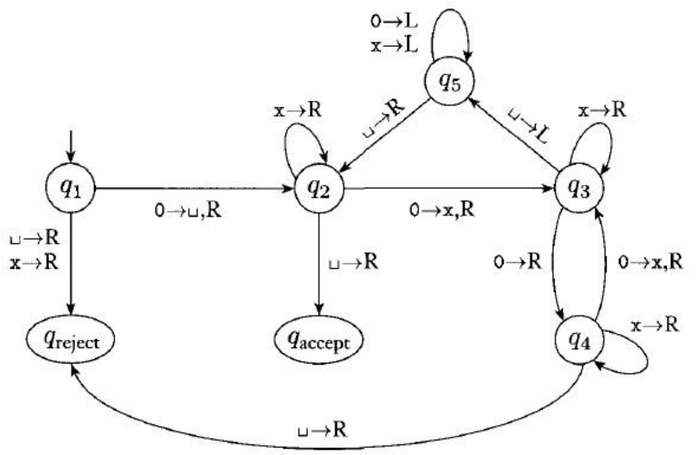
Example: $L = \{0^{2^n} \mid n \geq 0\}$ is Turing-decidable

The idea of the machine is that if we have a sequence of 0's that is a power of 2, we can divide it by 2 again and again until we get 1 (a single 0). That is done by the machine as follows:

- Go from the beginning to the end of the input and mark every other 0, and count the number of 0's not deleted.
- If the number of counted 0's it's odd, reject. If it's 1, accept. Otherwise go back to step 1.

The "deletion" or "mark" is simply a replacement of the character in another one that is in $\Gamma - \Sigma$.

A diagram of the machine:



The " " at the transition $q_1 \rightarrow q_2$ is to mark the beginning of the tape to know later when we get back.

If we read the " " symbol at q_4, we have an odd number of 0's so we go to the rejecting state.

If we read the " " symbol at q_3, we have an even number of 0's so we go back until the " " we placed at the beginning, symbolizing the beginning of the input.

At q_2 we go to the right as long as we see x (deleted), and if all 0's are deleted when getting to the end of the input we accept.

Notes:

When the head is on the left-most character, if the transition function specifies we need to go left, it stays put (but will change the input character according to the transition function).

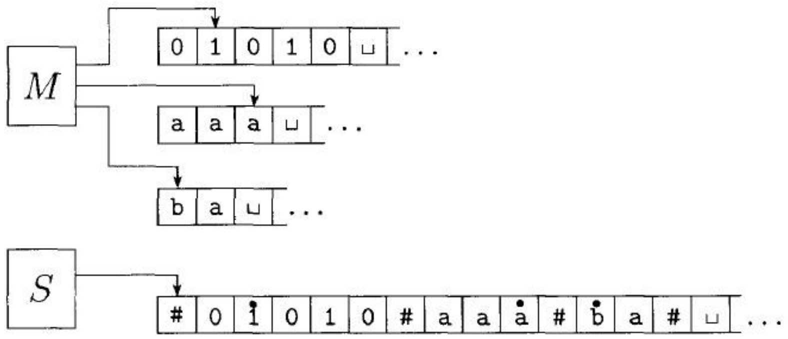
Example: $C = \{a^i b^j c^k \mid i \times j = k, i, j, k \geq 1\}$

This language is Turing-decidable; however drawing the diagram will be complex. The idea is as follows:

- First we check that the input is a sequence of that form (contiguous a's followed by contiguous b's followed by contiguous c's).
- For any a, mark it and then go back and forth between b's (marking) and c's (marking them). If we run out of c's in the middle, reject. If we end up with c's left, reject. If we finish with no c's, accept.

Variants of Turing Machines

Multiple-tape TM:



In M here $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$ where S is a new command "stay" (instead of going either left or right).

They do not offer more computing power, as they can be simulated by a one-tape TM.

Staying in place, for instance, can be simulated by simple go to the right and going back to the left (inflating the number of states by a factor of 2 for the "stay-in-place" version of all states).

How can S simulate M:

- We put all inputs on the single tape separated by a new symbol $\# \in \Gamma$
- The head positions will be simulated by adding a dot above the character, meaning: replacing each a with a new character $\dot{a} \in \Gamma$.
- If we get a an end of one input and need to write another character, we move all characters to the right one spot to the right, and make room for the new character.

From now on we can assume that TMs have the same computational power as multi-tape TMs (which may be useful for future simplifications).

Non-deterministic TMs:

Here $\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$, i.e. any configuration can non-deterministically do some action.

A non-deter' TM N accepts $w \in \Sigma$ iff there exists some path on the non-deterministic path tree that accepts w.

Simulate N with M:

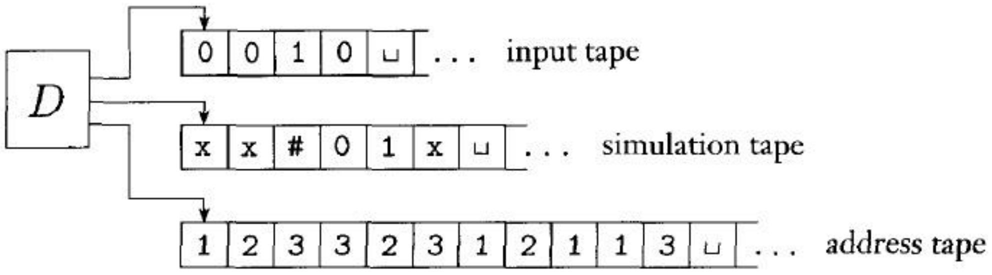
We will use M, a multi-tape TM, to simulate N, and according to the previous proof, N and S (a single-tape deter' machine) have the same computational power.

Let b be the maximal size of $|\delta(q, a)|$, for any $q \in Q, a \in \Gamma$ – the maximal possible branches for some transition. Looking at the non-deter' tree from left to right, we can **enumerate** all possible paths, which is the total number of nodes in that tree.

If $w \in L(N)$ then there exists a path in the computation tree that accepts w. This has to be done in **BFS**, since if we go in **DFS** we might encounter a non-finishing computation in the path we're checking, never getting to an accepting state.

In M:

- First tape: input tape
- Second tape: simulation tape, starts with a copy of the input for one run of N's paths.
- Third tape: path enumeration tape, counting what path in the N tree we're simulating.


D
0 0 1 0 ␣ ... input tape
x x # 0 1 x ␣ ... simulation tape
1 2 3 3 2 3 1 2 1 1 3 ␣ ... address tape

Enumerators:

Turing machines with a connection to a "printer" that prints out each word it recognizes.

The (possibly infinite) set of words printed, is the language of the enumerator.

Theorem:

A language is Turing-recognizable iff there exists an enumerator that enumerates it.

Proof:

Given a TM that recognizes L, we build the enumerator as follows:

- For each string in some enumeration $s_1, s_2, \ldots \in \Sigma^*$:
  - Run $TM(s_i)$ for i steps. If it accepts, print it. If it does not accept, do nothing.

If $w \in L$ it must be recognized by TM after some finite k steps, so at some point it will be printed out. If $w \notin L$, it will never be printed. Therefore this enumerator simulation enumerates L.

Now, if we have an enumerator of L we can build TM that recognizes L as follows:

- Run the enumerator until the printed word equals the input word.
- If it is, accept. Otherwise, continue.

Note that it is ok if for $w \notin L$ it will never stop.

Summary:

All the models above are equivalent.