The main ‘computational tools’ that we have introduced are context free grammars (CFGs) and pushdown automatas (PDAs).

PDAs are DFAs equipped with an additional stack.

Stacks have infinite storage, so in a sense allows us to ‘count’ the number of things we’ve encountered. They actually allow us to do more than that: we can track, in reverse order, the important symbols that we encountered.

Recall the non-regular language that got us started

\[ \{0^n 1^n \mid n \geq 0\} \, . \]

If we have a DFA with a stack, we will put every 0 that we see onto the stack. After we encounter a 1, we only accept 1s, until the stack runs out of them.

Formally, the pushdown automata (PDA) has on extra type of symbols: the stack symbols \( \Gamma \). Its states \( Q \), alphabet \( \Sigma \), starting state \( q_0 \), and accepting states \( F \subseteq Q \) are still the same as before. However its transitions are now

\[ Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow P(Q \times \Gamma \epsilon) \, . \]

That is, the transitions are non-deterministic, and each transition consumes a stack symbol (which is possibly \( \epsilon \)), and adds a symbol to the stack (that’s also possibly \( \epsilon \)).

In the diagram with the states, we use

\[ a, \gamma_1 \rightarrow \gamma_2 \]

to indicate we consume input \( a \), remove \( \gamma_1 \) from the stack, and push \( \gamma_2 \) onto the stack. Note that by adding additional intermediate states, we can allow \( \gamma_1 \) and \( \gamma_2 \) to be arbitrary strings.

Also, note that \( a, X \rightarrow XX \) is different than \( a, \epsilon \rightarrow X \), even though both result in an extra \( X \) on the stack (and consumes an \( a \) from the input). The former requires an \( X \) on the stack to start with to work, while the latter can be invoked as long as the input string has \( a \) as the next character.

1 Additional CFG / PDA Example

Consider the language

\[ \{a^i b^j c^k \mid i, j, k \geq 0 \land (i = j \lor i = k)\} \, . \]
2  CFGs are Recognized by PDAs

We start by showing that any context free language is recognized by some PDA.

The main proof idea is that the non-determinism in the PDA allow us to ‘guess’ the rules to apply when each variable is encountered.

That is, when we encountered a variable $v$, and want to apply

$$v \rightarrow s,$$

we simply go through a sequence of states that pushes $s$ onto the stack.

The only remaining issue is that the stack contains terminal symbols. But these symbols can (and must be) matched with the input greedily.

Finally, we need to start and end with $\$\$ to ensure that we start and end with an empty stack.

Formally, we need to:

1. Augment the PDA transitions to push multiple symbols (corresponding to an entry in $\{V, \Sigma \}^*$ onto the stack: this can be done by having a sequence of states, each transiting to the next consuming no stack/input symbols.

2. Transit from start to an ‘processing’ state by pushing $\$\$ onto the stack.

3. Transit from this processing state to an accepting state by popping $\$\$ off the stack.

4. Self-loops from this processing state to itself consisting of:

   (a) Popping a terminal symbol off the stack, and reading the same symbol from the input.

   (b) Popping a variable off the stack, and pushing a substitution of it onto the stack.

We will carry out this transformation on the example from last time, namely $S \rightarrow 0S1|\epsilon$, which generates the language $\{0^n1^n|n \geq 0\}$.

3  From PDAs to CFGs

I want to discuss how to build a CFG from any PDA.

The main idea is to modify the DFA to regular expression routine, but instead also require the stack to be unchanged.

For each pair of states, $pq$, we build a (CFG) variable,

$$A_{pq}$$

that generates all strings where we can move from $p$ to $q$, starting and ending with the empty stack.

There are several cases, depending on what gets pushed onto the stack in the first / last transition:
1. The stack was empty at some point. Then suppose that state is \( r \), we get

\[
A_{pq} \to A_{pr}A_{rq},
\]

2. We went from \( p \) to \( q \) directly. Then for every \( a \) where there is a transition from \( p \) to \( q \) of the form of:

\[
c : \epsilon \to \epsilon,
\]

we create

\[
A_{pq} \to c.
\]

3. Otherwise there is one symbol pushed, and one symbol popped. For every stack symbol \( \gamma \), state pairs \( p_1 \) and \( q_1 \) with transitions

\[
c_1 : \epsilon \to \gamma
\]

from \( p \) to \( p_1 \) and

\[
c_2 : \gamma \to \epsilon
\]

from \( q_1 \) to \( q \), we create the rule

\[
A_{pq} \to c_1A_{p_1q_1}c_2
\]

To formalize things, we need to modify the PDA so that:

1. It has only one accepting state \( q_{accept} \): this is done similar to GNFA by adding a new accepting state, and \( \epsilon \)-transitions from all current accepting states to it.

2. Each symbol either pushes or pops a symbol from the stack, but don’t do both at the same time: this can be done by having an extra intermediate state with an \( \epsilon \)-transition.

3. It only accepts when the stack is empty: this can be done by pushing an initial $ onto the stack (similar to what we did with balanced parentheses), popping off all symbols ‘for free’ at the end, and only going to the ‘true’ accepting state by popping off this initial $.

Then the start variable is \( A_{q_0q_{accept}} \), and we need to introduce additional rules for the \( p = q \) case:

\[
A_{pp} \to \epsilon.
\]