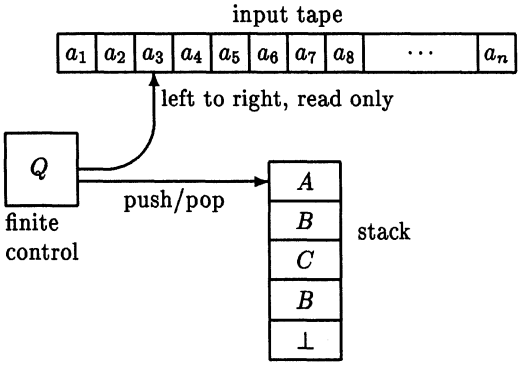
Lecture 23

Pushdown Automata

A *nondeterministic pushdown automaton* (NPDA) is like a nondeterministic finite automaton, except it has a *stack* or *pushdown store* that it can use to record a potentially unbounded amount of information.



The input head is read-only and may only move from left to right. The machine can store information on the stack in a last-in-first-out (LIFO) fashion. In each step, the machine pops the top symbol off the stack; based on this symbol, the input symbol it is currently reading, and its current state, it can push a sequence of symbols onto the stack, move its read head one cell to the right, and enter a new state, according to the transition rules

of the machine. We also allow *ϵ-transitions* in which it can pop and push without reading the next input symbol or moving its read head.

Although it can store an unbounded amount of information on the stack, it may not read down into the stack without popping the top elements off, in which case they are lost. Thus its access to the information on the stack is limited.

Formally, a nondeterministic PDA is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, s, \bot, F),$$

where

- Q is a finite set (the *states*),
- Σ is a finite set (the *input alphabet*),
- Γ is a finite set (the *stack alphabet*),
- $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$, δ finite (the *transition relation*),
- $s \in Q$ (the *start state*),
- $\bot \in \Gamma$ (the *initial stack symbol*), and
- $F \subseteq Q$ (the *final* or *accept states*).

If

$$((p, a, A), (q, B_1 B_2 \cdots B_k)) \in \delta,$$

this means intuitively that whenever the machine is in state p reading input symbol a on the input tape and A on the top of the stack, it can pop A off the stack, push $B_1 B_2 \cdots B_k$ onto the stack (B_k first and B_1 last), move its read head right one cell past the a, and enter state q. If

$$((p, \epsilon, A), (q, B_1 B_2 \cdots B_k)) \in \delta,$$

this means intuitively that whenever the machine is in state p with A on the top of the stack, it can pop A off the stack, push $B_1 B_2 \cdots B_k$ onto the stack (B_k first and B_1 last), leave its read head where it is, and enter state q.

The machine is nondeterministic, so there may be several transitions that are possible.

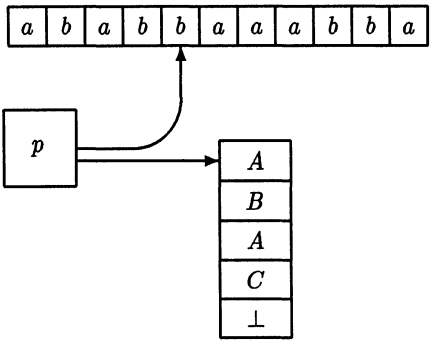
Configurations

A *configuration* of the machine M is an element of $Q \times \Sigma^* \times \Gamma^*$ describing the current state, the portion of the input yet unread (i.e., under and to the

right of the read head), and the current stack contents. A configuration gives complete information about the global state of M at some point during a computation. For example, the configuration

$$(p, baaabba, ABAC\perp)$$

might describe the situation



The portion of the input to the left of the input head need not be represented in the configuration, because it cannot affect the computation from that point on. In general, the set of configurations is infinite.

The *start configuration* on input x is $(s, x, \perp)$. That is, the machine always starts in its start state s with its read head pointing to the leftmost input symbol and the stack containing only the symbol $\perp$.

The *next configuration relation* $\xrightarrow[M]{1}$ describes how the machine can move from one configuration to another in one step. It is defined formally as follows: if

$$((p, a, A), (q, \gamma)) \in \delta,$$

then for any $y \in \Sigma^*$ and $\beta \in \Gamma^*$,

$$(p, ay, A\beta) \xrightarrow[M]{1} (q, y, \gamma\beta); \tag{23.1}$$

and if

$$((p, \epsilon, A), (q, \gamma)) \in \delta,$$

then for any $y \in \Sigma^*$ and $\beta \in \Gamma^*$,

$$(p, y, A\beta) \xrightarrow[M]{1} (q, y, \gamma\beta). \tag{23.2}$$

In (23.1), the ay changed to y, indicating that the input symbol a was eaten; the $A\beta$ changed to $\gamma\beta$, indicating that the A was popped and γ was pushed; and the p changed to q, indicating the change of state. In (23.2),

everything is the same except that the y does not change, indicating that no input symbol was eaten. No two configurations are related by $\xrightarrow[M]{1}$ unless required by (23.1) or (23.2).

Define the relations $\xrightarrow[M]{n}$ and $\xrightarrow[M]{*}$ as follows:

$$C \xrightarrow[M]{0} D \overset{\text{def}}{\iff} C = D,$$

$$C \xrightarrow[M]{n+1} D \overset{\text{def}}{\iff} \exists E\ C \xrightarrow[M]{n} E \text{ and } E \xrightarrow[M]{1} D,$$

$$C \xrightarrow[M]{*} D \overset{\text{def}}{\iff} \exists n \geq 0\ C \xrightarrow[M]{n} D.$$

Then $\xrightarrow[M]{*}$ is the reflexive transitive closure of $\xrightarrow[M]{1}$. In other words, $C \xrightarrow[M]{*} D$ if the configuration D follows from the configuration C in zero or more steps of the next configuration relation $\xrightarrow[M]{1}$.

Acceptance

There are two alternative definitions of acceptance in common use: by *empty stack* and by *final state*. It turns out that it doesn't matter which definition we use, since each kind of machine can simulate the other.

Let's consider acceptance by final state first. Informally, the machine M is said to *accept its input x by final state* if it ever enters a state in F after scanning its entire input, starting in the start configuration on input x. Formally, M *accepts x by final state* if

$$(s, x, \bot) \xrightarrow[M]{*} (q, \epsilon, \gamma)$$

for some $q \in F$ and $\gamma \in \Gamma^*$. In the right-hand configuration, ϵ is the null string, signifying that the entire input has been read, and γ is junk left on the stack.

Informally, M is said to *accept its input x by empty stack* if it ever pops the last element off the stack without pushing anything back on after reading the entire input, starting in the start configuration on input x. Formally, M *accepts x by empty stack* if

$$(s, x, \bot) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$$

for some $q \in Q$. In this definition, the q in the right-hand configuration can be any state whatsoever, and the ϵ in the second and third positions indicate that the entire input has been read and the stack is empty, respectively. Note that F is irrelevant in the definition of acceptance by empty stack.

The two different forms of automata can simulate each other (see Lecture E); thus it doesn't matter which one we work with.

In either definition of acceptance, the entire input string has to be read. Because of ϵ-transitions, it is possible that a PDA can get into an infinite loop without reading the entire input.

Example 23.1 Here is a nondeterministic pushdown automaton that accepts the set of balanced strings of parentheses [ ] by empty stack. It has just one state q. Informally, the machine will scan its input from left to right; whenever it sees a [, it will push the [onto the stack, and whenever it sees a] and the top stack symbol is [, it will pop the [off the stack. (If you matched up the parentheses, you would see that the] it is currently reading is the one matching the [on top of the stack that was pushed earlier.) Formally, let

$$\begin{aligned} Q &= \{q\}, \\ \Sigma &= \{[,]\}, \\ \Gamma &= \{\bot, [\}, \\ \text{start state} &= q, \\ \text{initial stack symbol} &= \bot, \end{aligned}$$

and let δ consist of the following transitions:

(i) $((q, [, \bot), (q, [\bot))$;

(ii) $((q, [, [), (q, [[))$;

(iii) $((q,], [), (q, \epsilon))$;

(iv) $((q, \epsilon, \bot), (q, \epsilon))$.

Informally, transitions (i) and (ii) say that whenever the next input symbol is [, the [is pushed onto the stack on top of the symbol currently there (actually, the symbol currently there is popped but then immediately pushed back on). Transition (iii) says that whenever the next input symbol is] and there is a [on top of the stack, the [is popped and nothing else is pushed. Transition (iv) is taken when the end of the input string is reached in order to dump the $\bot$ off the stack and accept.

Here is a sequence of configurations leading to the acceptance of the balanced string [[[ ]] [ ]] [ ].

	Configuration	*Transition applied*
	$(q,$ [[[ ]] [ ]] [ ], $\perp)$	start configuration
$\rightarrow$	$(q,$ [[[ ]] [ ]] [ ], [$\perp)$	transition (i)
$\rightarrow$	$(q,$ [[ ]] [ ]] [ ], [[$\perp)$	transition (ii)
$\rightarrow$	$(q,$]] [ ]] [ ], [[[$\perp)$	transition (ii)
$\rightarrow$	$(q,$] [ ]] [ ], [[$\perp)$	transition (iii)
$\rightarrow$	$(q,$ [ ]] [ ], [$\perp)$	transition (iii)
$\rightarrow$	$(q,$]] [ ], [[$\perp)$	transition (ii)
$\rightarrow$	$(q,$] [ ], [$\perp)$	transition (iii)
$\rightarrow$	$(q,$ [ ], $\perp)$	transition (iii)
$\rightarrow$	$(q,$], [$\perp)$	transition (i)
$\rightarrow$	$(q,$ ϵ, $\perp)$	transition (iii)
$\rightarrow$	$(q,$ ϵ, $\epsilon)$	transition (iv)

The machine could well have taken transition (iv) prematurely at a couple of places; for example, in its very first step. This would have led to the configuration

$$(q, [\,[\,[\,]\,]\,[\,]\,]\,[\,], \epsilon),$$

and the machine would have been stuck, since no transition is possible from a configuration with an empty stack. Moreover, this is not an accept configuration, since there is a nonnull portion of the input yet unread. However, this is not a problem, since the machine is nondeterministic and the usual rules for nondeterminism apply: the machine is said to accept the input if *some* sequence of transitions leads to an accept configuration. If it does take transition (iv) prematurely, then this was just a bad guess where the end of the input string was.

To prove that this machine is correct, one must argue that for every balanced string x, there is a sequence of transitions leading to an accept configuration from the start configuration on input x; and for every unbalanced string x, *no* sequence of transitions leads to an accept configuration from the start configuration on input x. □

Example 23.2 We showed in Lecture 22 using the pumping lemma that the set

$$\{ww \mid w \in \{a,b\}^*\}$$

is not a CFL (and therefore, as we will show in Lecture 25, not accepted by any NPDA) but that its complement

$$\{a,b\}^* - \{ww \mid w \in \{a,b\}^*\} \tag{23.3}$$

is. The set (23.3) can be accepted by a nondeterministic pushdown automaton as follows. Initially guess whether to check for an odd number of input

symbols or for an even number of the form $xayubv$ or $ubvxay$ with $|x| = |y|$ and $|u| = |v|$. To check for the former condition, we do not need the stack at all—we can just count mod 2 with a finite automaton encoded in the finite control of the PDA. To check for the latter, we scan the input for a nondeterministically chosen length of time, pushing the input symbols onto the stack. We use the stack as an integer counter. At some nondeterministically chosen time, we remember the current input symbol in the finite control—this is the a or b that is guessed to be the symbol in the first half not matching the corresponding symbol in the second half—then continue to scan, popping one symbol off the stack for each input symbol read. When the initial stack symbol $\perp$ is on top of the stack, we start pushing symbols again. At some point we nondeterministically guess where the corresponding symbol in the second half is. If it is the same symbol as the one remembered from the first half, reject. Otherwise we scan the rest of the input, popping the stack as we go. If the stack contains only $\perp$ when the end of the input is reached, we accept by popping the $\perp$, leaving an empty stack. □

We close with a few technical remarks about NPDAs:

1. In *deterministic* PDAs (Supplementary Lecture F), we will need an endmarker on the input so that the machine knows when it is at the end of the input string. In NPDAs, the endmarker is unnecessary, since the machine can guess nondeterministically where the end of the string is. If it guesses wrong and empties its stack before scanning the entire input, then that was just a bad guess.

2. We distinguish the initial stack symbol $\perp$ only because we need it to define the start configuration. Other than that, it is treated like any other stack symbol and can be pushed and popped at any time. In particular, it need not stay on the bottom of the stack after the start configuration; it can be popped in the first move and something else pushed in its place if desired.

3. A transition $((p, a, A), (q, \beta))$ or $((p, \epsilon, A), (q, \beta))$ does not apply unless A is on top of the stack. In particular, *no* transition applies if the stack is empty. In that case the machine is stuck.

4. In acceptance by empty stack, the stack must be empty *in a configuration*, that is, *after* applying a transition. In our intuitive description above, when a transition such as $((p, \epsilon, A), (q, BC))$ is taken with only A on the stack, the stack is momentarily empty between the time A is popped and BC is pushed. This does not count in the definition of acceptance by empty stack. To accept by empty stack, everything must be popped and nothing pushed back on.