Previously:
Context-free grammars, for describing a more powerful class of languages.

Today:
Pushdown automata for recognizing context-free languages.
Take-home Exam 1 due.

Then:
Assignment 4 out later today; due Tuesday after break.

Example of a CFG

\[
\begin{align*}
A & \to aAb \\
A & \to B \\
B & \to \epsilon
\end{align*}
\]

Production rules: substitutions
Non-terminals: variable that can have a substitutions
Terminals: symbols that are part of the alphabet, no substitutions
Start variable: left side of top-most rule

Formal CFG notation

Productions = rules of the form head $\to$ body

head is a variable
body is a string of zero or more variables and/or terminals

Start symbol = variable that represents “the language”

Notation: $G = (V, \Sigma, P, S)$

$V$ = variables
$\Sigma$ = terminals
$P$ = productions
$S$ = start symbol
Pushdown automata

Add a stack to a FA

Typically non-deterministic

An automaton equivalent to CFGs

Schematic of a finite automaton

Finite state control

\[ a \ a \ b \ a \ c \ input \]

Schematic of a pushdown automaton

Finite state control

\[ x \ c \ c \ a \ a \ input \]

Notation

If at state \( p \) with next input symbol \( x \) and top of stack is \( y \):

- go to state \( q \) and replace \( y \) by \( z \) on stack
- \( x = \varepsilon \): Ignore input, don’t read
- \( y = \varepsilon \): Ignore top of stack and push \( z \)
- \( z = \varepsilon \): Pop \( y \)
Example

Notation for transition diagrams:

\[ a, Z \rightarrow X_1 X_2 \ldots X_k \]

Meaning

On input \( a \), with \( Z \) on top of the stack:
- Consume the \( a \)
- Make this state transition
- Replace the \( Z \) on top of the stack by \( X_1 X_2 \ldots X_k \) (with \( X_i \) at the top)

Formal PDA

\[ P = (Q, \Sigma, \Gamma, \delta, q_0, F) \]

- \( Q, \Sigma, q_0, \) and \( F \) have their meanings from FA
- \( \Gamma = \) stack alphabet
- \( \delta = \) transition function
- \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow P(Q \times \Gamma \epsilon) \)

Takes a state, input symbol (or \( \epsilon \)), and a stack symbol and gives you a finite number of choices of:
1. A new state (possibly the same)
2. A string of stack symbols (or \( \epsilon \)) to replace the top stack symbol

PDAs à la Sipser

No intrinsic way to test for an empty stack
- Get the same effect by initially putting a special symbol $ on the stack
- Machine knows stack is empty if it sees $ during computation

No intrinsic way to know when end of input string is reached
- PDA achieves that effect because the accept state takes effect only when the machine is at the end of the input

For \( a^n b^n \):

- \( q_1 = \) starting to see a group of \( a \)s and \( b \)s
- \( q_2 = \) reading \( a \)s and pushing \( a \)s onto the stack
- \( q_3 = \) reading \( b \)s and popping \( a \)s until the \( a \)s are all popped
- \( q_4 = \) no more input and empty stack; accept

NB: Different textbook authors use different conventions and notations
Instantaneous descriptions (IDs)

For a FA, the only thing of interest is its state

For a PDA, we want to know its state and the entire contents of its stack

Represented by an ID \((q, w, \alpha)\), where

\(q = \text{state}\)

\(w = \text{waiting input}\)

\(\alpha = \text{stack [top on left; bottom on right]}\)

Moves of the PDA

If \(\delta(q, a, X)\) contains \((p, \alpha)\), then

\((q, aw, X\beta) \vdash (p, w, \alpha \beta)\)

Extend to \(\vdash^*\) to represent 0 or more moves

Can subscript with the name of the PDA for clarity

Input string \(w\) is accepted if \((q_0, w, \varepsilon) \vdash^* (p, \varepsilon, \varepsilon)\) for any accepting state \(p\)

\(L(P) = \text{set of strings accepted by } P\)

Example

\((q, aabb, \varepsilon) \vdash (q, aabb, \$)\)

\((q, abb, \alpha\$) \vdash (q, b, \alpha\$)\)

\((q, \varepsilon, \$) \vdash (q', \varepsilon, \varepsilon)\)

Acceptance by empty stack

Another one of those technical conveniences:

when we prove that PDAs and CFGs accept the same languages, it helps to assume that the stack is empty whenever acceptance occurs

\(N(P) = \text{set of strings } w \text{ such that}\)

\((q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\) for some state \(q\)

Note: \(q\) need not be in \(F\)

In fact, if we talk about \(N(P)\) only, then we need not even specify a set of accepting states
Example

For our previous example, to accept by empty stack:

1. Add a new transition $\delta(p, \varepsilon, Z_0) = \{(p, \varepsilon)\}$
   That is, when starting to look for a new $a$–$b$ block, the PDA has the option to pop the last symbol off the stack instead.

2. $p$ is no longer an accepting state, in fact, there are no accepting states.

Palindromes

Input: $aaabcbaaa$

Palindromes

Input: $aaabcbaaa$

Palindromes

Input: $aaabcbaaa$

Palindromes

Input: $aaabcbaaa$
Palindromes

Input: aaabcbaaa

Input: aaabcbaaa

Input: aaabcbaaa

Input: aaabcbaaa
PDA exercise

Draw the PDA acceptor for

\[ L = \{ x \in \{a,b\}^* \mid n_a(x) = 2n_b(x) \} \]

Note: The empty string is in \( L \).

The idea is to use the stack to keep count of the number of \( a \)s and/or \( b \)s needed to get a valid string.

If we have a surplus of \( b \)s thus far, we should have corresponding number of \( a \)s (two for every \( b \)) on the stack.

On the other hand, if we have a surplus of \( a \)s, we cannot put \( b \)s on the stack since we can’t split symbols. So instead, put two “negative” \( a \)-symbols, where a negative a will be denoted by capital A.
Another PDA exercise

Draw the PDA acceptor for

\[ L = \{ a^i b^j c^k \mid i = j + k \} \]

Equivalence of acceptance by final state and empty stack

A language is \( L(P_1) \) for some PDA \( P_1 \) if and only if it is \( N(P_2) \) for some PDA \( P_2 \).

Can show with constructive proofs
**Final state ⇒ empty stack**

Given \( P_1 = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \), construct \( P_2 \):

1. Introduce new start state \( p_0 \) and new bottom-of-stack marker \( X_0 \).
2. First move of \( P_2 \): replace \( X_0 \) by \( Z_0X_0 \) and go to state \( q_0 \). The presence of \( X_0 \) prevents \( P_1 \) from "accidentally" emptying its stack and accepting when \( P_1 \) did not accept.
3. Then, \( P_1 \) simulates \( P_2 \), i.e., give \( P_1 \) all the transitions of \( P_2 \).
4. Introduce a new state \( r \) that keeps popping the stack of \( P_2 \) until it is empty.
5. If (the simulated) \( P_1 \) is in an accepting state, give \( P_1 \) the additional choice of going to state \( r \) on \( \varepsilon \) input, and thus emptying its stack without reading any more input.

**Empty stack ⇒ final state**

Given \( P_2 = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \), construct \( P_1 \):

1. Introduce new start state \( p_0 \) and new bottom-of-stack marker \( X_0 \).
2. First move of \( P_1 \): replace \( X_0 \) by \( Z_0X_0 \) and go to state \( q_0 \). Then, \( P_1 \) simulates \( P_2 \), i.e., give \( P_1 \) all the transitions of \( P_2 \).
3. Introduce a new state \( r \) for \( P_1 \), it is the only accepting state.
4. \( P_1 \) simulates \( P_2 \).
5. If (the simulated) \( P_1 \) ever sees \( X_0 \) it knows \( P_2 \) accepts so \( P_1 \) goes to state \( r \) on \( \varepsilon \) input.

**Deterministic PDAs**

The PDAs we are dealing with are almost invariably non-deterministic.

A DPDA never has a choice of move:

\( \delta(q, a, Z) \) has at most one member for any \( q, a, Z \) (including \( a = \varepsilon \)).

If \( \delta(q, \varepsilon, Z) \) is nonempty, then \( \delta(q, a, Z) \) must be empty for all input symbols \( a \).

Why care?

- Parsers are DPDAs
- Thus, the question of what languages a DPDA can accept is really the question of what programming language syntax can be parsed conveniently

**Non-deterministic PDAs**

A non-deterministic PDA allows non-deterministic transitions (e.g., defines multiple possible moves for a given configuration).

Non-deterministic PDAs are strictly stronger than deterministic PDAs.

In this respect, the situation is not similar to the situation of DFAs and NFAs.

Non-deterministic PDAs are equivalent to CFLs.
Real compilers

However, unambiguous, deterministic CFGs are complicated and too restricted

Real parsers cheat by looking ahead one token

This places certain restrictions on the grammar, but not as many

Stay tuned: Learn about LL(1) and LR(1) grammars in CMPU 331

Some ambiguities (e.g., dangling else) are easily handled with one token lookahead

Equivalence of parse trees, leftmost, and rightmost derivations

The following about a grammar $G = (V, \Sigma, P, S)$ and a terminal string $w$ are all equivalent:

1. $S \Rightarrow^* w$ (i.e., $w$ is in $L(G)$).
2. $S \Rightarrow_{lm}^* w$
3. $S \Rightarrow_{rm}^* w$
4. There is a parse tree for $G$ with root $S$ and yield (labels of leaves, from the left) $w$.

Obviously (2) and (3) each imply (1).

Parse tree implies LM/RM derivations

Generalize all statements to talk about an arbitrary variable $A$ in place of $S$.

Except now (1) no longer means $w$ is in $L(G)$.

Induction on the height of the parse tree.

Basis: Height 1: Tree is root $A$ and leaves $w = a_1, a_2, \ldots, a_k$.

$A \stackrel{*}{\Rightarrow} w$ must be a production, so $A \stackrel{*}{\Rightarrow}_{lm} w$ and $A \Rightarrow w$. 
**Induction**: Height > 1: Tree is root $A$ with children = $X_1$, $X_2$, ..., $X_k$.

Those $X_i$s that are variables are roots of shorter trees.

Thus, the IH says that they have LM derivations of their yields.

Construct a LM derivation of $w$ from $A$ by starting with $A \xrightarrow{\text{lm}} X_1X_2...X_k$, then using LM derivations from each $X_i$ that is a variable, in order from the left.

RM derivation analogous.

---

**Example**

Consider derivation $S \Rightarrow AS \Rightarrow AAS \Rightarrow AA \Rightarrow A_1A \Rightarrow A_1A_1 \Rightarrow 0110A_1 \Rightarrow 0110011$

Sub-derivation from $A$ is: $A \Rightarrow A_1 \Rightarrow 011$

Sub-derivation from $S$ is: $S \Rightarrow AS \Rightarrow A \Rightarrow oA_1 \Rightarrow 0011$

Each has a parse tree, put them together with new root $S$.

---

**Derivations to parse trees**

Induction on length of the derivation.

**Basis**: One step. There is an obvious parse tree.

**Induction**: More than one step.

Let the first step be $A \Rightarrow X_1X_2...X_k$.

Subsequent changes can be reordered so that all changes to $X_1$ and the sentential forms that replace it are done first, then those for $X_2$, and so on (i.e., we can rewrite the derivation as a LM derivation).

The derivations from those $X_i$s that are variables are all shorter than the given derivation, so the IH applies.

By the IH, there are parse trees for each of these derivations.

Make the roots of these trees be children of a new root labeled $A$.

---

**Only-if proof (i.e., grammar $\Rightarrow$ PDA)**

Prove by induction on the number of steps in the leftmost derivation $S \Rightarrow^* \alpha$ that for any $x$, $(q, wx, S) \vdash^* (q, x, \beta)$, where

$w\beta = \alpha$

$\beta$ is the suffix of $\alpha$ that begins at the leftmost variable ($\beta = \varepsilon$ if there is no variable).

Also prove the converse, that if $(q, wx, S) \vdash^* (q, x, \beta)$ then $S \Rightarrow w\beta$.

Inductive proofs in book.

As a consequence, if $y$ is a terminal string, then $S \Rightarrow y$ iff $(q, y, S) \vdash^* (q, \varepsilon, \varepsilon)$, i.e., $y$ is in $L(G)$ iff $y$ is in $N(A)$. 