Context-free Languages and Pushdown Automata

Finite Automata vs CFLs

From earlier results:
- every regular language is a CFL
- but there are CFLs that are not regular

Can we extend Finite Automata to equal CFLs?
- i.e., get a machine-like characterization of CFLs?

CFLs

Regular Languages

CF but not Regular

A key feature: recursion

Recursion’s twin: a pushdown stack

Pushdown sufficient? intuitively, yes:

Example
Every CFL is accepted by some PDA

Every regular language is accepted by some PDA (basically, just ignore the stack...)

Above examples show that PDAs are sufficiently powerful to accept some context-free but non-regular languages, too

In fact, they can accept every CFL:

Proof 1: the book’s “top down” parser (next)

Proof 2: “bottom up,” (aka “shift-reduce”) parser (later)
PDAs accept all CFLs

“Bottom-Up” / “Shift-Reduce”

For any CFG $G=(V, \Sigma, R, S)$, build a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where $Q = \{q_0, q_{\text{accept}}\}$, $\Gamma = V \cup \Sigma \cup \{\$\}$, $F = \{q_{\text{accept}}\}$, and $\delta$ is defined by the diagram.

Idea: on input $w$, $M$ nondeterministically picks a rightmost derivation backwards, from $w$ to $S$. Shift input onto stack or “reduce” top few symbols at each step.

Figure 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$
Correctness of shift-reduce construction

CLAIM: For all \( \gamma \in (V \cup \Sigma)^* \) and all \( w \in \Sigma^* \),
\[
\gamma \Rightarrow_R^k w \text{ if and only if } [q, \epsilon, w] \vdash^{k+|w|} [q, \gamma, \epsilon].
\]

COROLLARY: \( L(M) = L(G) \)

Proof:
\[
S \Rightarrow_R^k w \text{ if and only if } [q, \epsilon, w] \vdash^{k+|w|} [q, S, \epsilon]
\]

CLAIM: \( \forall \gamma \in (V \cup \Sigma)^*, \forall w \in \Sigma^*, \gamma \Rightarrow_R^k w \text{ only if } [q, \epsilon, w] \vdash^{k+|w|} [q, \gamma, \epsilon]. \)

Basis \((k = 0)\):
\[
\gamma \Rightarrow_R^0 w \text{ so } \gamma = w, \text{ so } [q, \epsilon, w] \vdash^{|w|} [q, \gamma, \epsilon] \quad \text{(via } |w| \text{ shifts)}
\]

Induction: Assume the claim for some \( k \geq 0 \). Suppose
\[
\gamma \Rightarrow_R^{k+1} w
\]

Let its first step be \( A \rightarrow \beta \). \( \exists \alpha \in (V \cup \Sigma)^*, \exists x, y \in \Sigma^* \) s.t.
\[
\gamma = \alpha Ay \Rightarrow_R \alpha \beta y \Rightarrow_R^k xy = w \text{ so } \alpha A \Rightarrow_R \alpha \beta \Rightarrow_R^k x
\]

By the induction hypothesis, and the definition of "reduce moves"
\[
[q, \epsilon, x] \vdash^{k+|x|} [q, \alpha \beta, \epsilon] \text{ and } [q, \alpha \beta, \epsilon] \vdash [q, \alpha A, \epsilon]
\]

So
\[
[q, \epsilon, xy] \vdash^{k+|xy|} [q, \alpha \beta, y] \vdash^{-1} [q, \alpha A, y] \vdash^{-|w|} [q, \alpha Ay, \epsilon]
\]

Thus
\[
[q, \epsilon, w] \vdash^{k+1+|w|} [q, \gamma, \epsilon]
\]

Proof of this direction is similar, and is left as an exercise.

HINT: Again induction on \( k \); consider the last "reduce" step in the PDA’s computation.
Both top-down & bottom up PDA's above are non-deterministic. With a carefully designed grammar, and by being able to “peek” ahead at the next input symbol, it may be possible to tell deterministically which action to take. The CFG's for which this is possible are called LL(1) (top-down case) or LR(1) (shift-reduce case) grammars, and are important for programming language design.

Every language accepted by a deterministic PDA has an LR (1) grammar, but not all grammars for a given language are LR(1), and for some CFL's no grammar is LR(1).

Q: What L solves this equation?

L \subseteq \{a,b\}^*

L = \{\varepsilon\} \cup \{a\} \cdot L \cdot \{b\}

Answer:

L = \{ a^n b^n | n \geq 0 \}

Compare to:

S \rightarrow \varepsilon \mid a \ S \ b

Q: What L solves this equation?

L, X \subseteq \Sigma^* (X fixed, e.g. “palindromes” or “odd len”)

L = \{\varepsilon\} \cup X \cup L \cdot L

Alt phrasing: the smallest set containing \varepsilon and all of X and is closed under concatenation?

Answer:

L = X^*

Compare to:

S \rightarrow \varepsilon \mid S_{\text{grammar for } X} \mid S \ S
In English? \( L_{22} \) is the set of input strings \( x \) that allow \( M \) to go from state 2 to state 2, starting & ending with empty stack.

An equation?

PDA to CFG, general construction

I. WLOG, assume PDA:
   a) has only one final state
   b) accepts only when stack is empty, and
   c) all transitions either push or pop, never both/neither

   **Goal:** \( A_{pq} \) gives inputs allowing \( M \) to go from state \( p \) to state \( q \), starting & ending with empty stack.

   - read nothing/no transition
   - \( p \) to \( q \) via any intermediate \( r \)

   1. read \( a \) + push \( X \)
   2. go from \( r \) to \( s \) on empty stack, so \( X \) re-exposed
   3. read \( b \) + pop \( X \)

   Grammar start symbol = \( A_{\text{start-state, final-state}} \)

NB: \( G \) can be simplified. E.g., remove \( A_{21}, A_{31} \) & rules using them, since, e.g., \( \exists x \in \Sigma^* \text{ s.t. } \) \( A_{21} \rightarrow^* x \).

This is just fine in the construction, since there is also no \( x \) s.t. \( [2, \varepsilon, x] \vdash^* [1, \varepsilon, \varepsilon] \).

Easier to construct useless rules locally than to sort out such ramifications globally.
I.e., \( A_{pq} \) gives set of inputs that allow \( M \) to go from state \( p \) to state \( q \), starting & ending with empty stack.

\[
L(CG) = L(\Delta M) = \{ x \mid \text{traces of } M \text{ on } x \text{ start & end with empty stack} \}
\]

(and fact that \( M \)'s stack is empty when it enters \( F \))

Summary: \( \text{PDA} \equiv \text{CFG} \)

Pushdown stack conveniently allows simulation of recursion in \( \text{CFG} \)

E.g., \( \{a^n b^n \} \) or \( \{ww^R \} \) or balanced parens, etc.: push some, match later

Nondeterminism sometimes essential

- e.g., "guess middle": there is no "subset constr" for \( \text{NPDA} \)

\( G \subseteq M \): guess deriv., using stack carefully (\( \Rightarrow_L \) or \( \Rightarrow_R \))

- basis for parsers in most compilers, e.g.

\( M \subseteq G \): \( A_{pq} = \{ x \mid \text{go from } p \text{ to } q \text{ on empty stack} \} \)