Pushdown Automata: Introduction

- **Pushdown automaton** (PDA) $M = (K, \Sigma, \Gamma, \Delta, s, A)$ where
  - $K$ is a set of states
  - $\Sigma$ is an input alphabet
  - $\Gamma$ is a set of stack symbols
  - $s \in K$ is the start state
  - $A \subseteq K$ is a set of accepting states, and
  - $\Delta$ is a transition relation:
    $$\Delta \subseteq (K \times (\Sigma \cup \{\epsilon\})) \times (K \times \Gamma^*)$$
- Simply put, a PDA is a FA with a stack
- **Configuration** of a PDA $M \in K \times \Sigma^* \times \Gamma^*$
  - i.e., a triple consisting of a state, remaining input, and contents of the stack
- **Yields-in-one-step** relation:
  - Denoted $\vdash_M$
  - Relates one configuration to another
  - $(q_1, cw, \gamma_1 \gamma) \vdash_M (q_2, w, \gamma_2 \gamma)$ iff $((q_1, c, \gamma_1), (q_2, \gamma_2)) \in \Delta$
- Points of note:
  1. $M$ makes transitions only if $\gamma_1$ matches top of stack
     - This results in pop of $\gamma_1$ and push of $\gamma_2$
     - Top of stack accessible only via a pop
     - Can push and pop more than one symbol at a time
     - Stack provides ability to count and to keep track of structure
  2. If $\gamma_1 = \epsilon$, corresponds to a transition that ignores stack contents
- **Yields** relation:
  - Denoted $\vdash^*_M$
  - Reflexive, transitive closure of $\vdash_M$
- **Computation** of $M$:
  - Finite sequence of configurations $C_0, C_1, ..., C_n, n \geq 0$, where
    - $C_0$ is initial configuration
    - $C_n$ is of the form $(q, \epsilon, \gamma)$ for some $q \in K$ and $\gamma \in \Gamma^*$
    - $C_0 \vdash_M C_1 \vdash_M ... \vdash_M C_n$
Pushdown Automata: Introduction (2)

• Since ∆ is a relation, above definition defines a nondeterministic PDA
  – Accepting computation: One in which \((s, w, \epsilon) \vdash^* (q, \epsilon, \epsilon)\) for some \(q \in A\)
  – Rejecting computation: One in which \((s, w, \epsilon) \vdash^* (q, w', \alpha)\) where \(M\) has no transitions from \(q\) given \(w', \alpha\)

• Language accepted by PDA \(M\) denoted \(L(M)\)

• Label arcs of transition diagram with \(c/\gamma_1/\gamma_2\) to denote \(((q_1, c, \gamma_1), (q_2, \gamma_2))\)
Pushdown Automata: Deterministic PDAs

• **Deterministic PDA**: One in which every configuration allows only a single transition

• Two conditions must hold:
  1. $\Delta_M$ contains no pairs of transitions that compete
  2. If $q \in A_M$, there is no transition $((q, \epsilon, \epsilon), (p, a))$ for any $p, a$

• Can represent operation of a nondeterministic PDA using trees
  – Each node represents a configuration of the machine
  – Path from root to leaf represents a computation
  – Cannot represent all paths in parallel as with NFAs as need a stack for each path

• Nondeterminism is more important for CFLs as opposed to RLs
  – Some CFLs *cannot* be represented by deterministic PDAs

• Two common situations contribute to nondeterministic PDAs
  1. Transitions to be taken only when stack is empty
  2. Transitions to be taken only when input stream is empty

• Nondeterminism can be reduced (not necessarily eliminated) by eliminating the above
  – Both approaches use special markers
    1. Checking for an empty stack
       * Push special marker (#) onto stack at beginning of processing
       * Stack is "empty" when this marker is on top
       * Note that stack must be truly empty in order to accept input
         · Final step in processing must be to pop special marker from stack
    2. Checking for end of input
       * Concatenate special marker ($) to end of input string
Pushdown Automata: Equivalence of CFGs and PDAs -
Converting CFGs to PDAs

• Theorem 12.1
  – Statement: Given a CFG \( G = (V, \Sigma, R, S) \), there exists a PDA \( M \) such that
    \( L(M) = L(G) \)
  – Proof: By construction
    * Two approaches can be taken:
      1. Build a bottom-up parser
      2. Build a top-down parser
    * Strategy is to build a PDA that simulates \( G \) by attempting to derive
      input string \( w \) from \( G \)
    * The two algorithms are opposites in all aspects
  1. Top-down parser
    * Strategy is to generate a leftmost derivation for \( w \)
    * Start by pushing \( S \) onto stack
    * Whenever topmost symbol of stack matches the LHS of some rule,
      replace stack symbol with symbols on RHS of rule (pushing symbols
      from right to left)
    * Whenever symbol on top of stack is terminal, match it against input
      string (read input character, pop stack)
    * PDA \( M \) has exactly 2 states:
      (a) First pushes \( S \) onto stack
      (b) Second performs rest of processing

    * Algorithm: PDA CFGToPDA-TD (CFG \( G \));
      Create \( M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\}) \), where \( \Delta \) contains
      \( ((p, \epsilon, \epsilon), (q, S)) \)
      \( ((q, \epsilon, X), (q, \gamma_1\gamma_2...\gamma_n)) \) for each \( X \rightarrow \gamma_1\gamma_2...\gamma_n \in R_G \)
      \( ((q, \epsilon, X), (q, \epsilon)) \) for each \( X \rightarrow \epsilon \in R_G \)
      \( ((q, c, c), (q, \epsilon)) \) for each \( c \in \Sigma \)
Pushdown Automata: Equivalence of CFGs and PDAs -
Converting CFGs to PDAs (2)

2. Bottom-up parser
   * Strategy is to generate a rightmost derivation for $w$
   * Start by pushing input characters onto stack
   * Whenever topmost sequence of symbols on stack matches the RHS of
     some rule, replace sequence with LHS symbol of rule
   * Accept when input consumed with $S$ on top of stack
   * PDA $M$ has exactly 2 states:
     (a) First performs all processing of input (and appropriate pushes and
         pops)
     (b) Second pops $S$ from stack

```
* Algorithm: PDA CFGToPDA-BU (CFG G);
   Create $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where $\Delta$ contains
       ((($p, \epsilon, S$), ($q, \epsilon$))
       (($p, \epsilon, (\gamma_1\gamma_2...\gamma_n)^R$), ($p, X$)) for each $X \rightarrow \gamma_1\gamma_2...\gamma_n \in R_G$
       (($q, c, \epsilon$), ($p, c$)) for each $c \in \Sigma$

- Theorem important because
  1. Proves that languages that are described by CFGs are equivalent to
     those accepted by PDAs
  2. Provides useful tools for compilers
Pushdown Automata: Equivalence of CFGs and PDAs -
Converting PDAs to CFGs

• Restricted Normal Form (RNF)
  – To convert a PDA to a CFG, it must first be transformed into a particular format
  – PDA $M$ is in RNF iff
    1. Start state does only 1 thing: Push a special marker ($#$) onto the stack
       * No transitions may enter the start state, and only one may leave it
    2. Has a single accepting state
       * All transitions into this state consume no input, and pop $#$
    3. Every transition -except the one from the start state - pops exactly one symbol
Pushdown Automata: Equivalence of CFGs and PDAs -
Converting PDAs to CFGs (2)

- Algorithm:

```plaintext
PDA convertPDAToRNF (PDA M)
{
    M' = M;
    M'.K = M'.K + s';
    M'.Gamma = M'.Gamma + #;
    M'.Delta = M'.Delta + ((s', ε, ε), (s, #));
    M'.s = s';
    M'.K = M'.K + a;
    for(each q in M'.A)
        M'.Delta = M'.Delta + ((q, ε, #), (a, ε));
    M'.A = a;
    for(each t in M'.Delta)
        if(t == ((q, c, a1a2...an), (p, γ))
            M'.K = M'.K + qq1;
            M'.Delta = M'.Delta + ((q, ε, a1), (qq1, ε));
        for (i = 1; i < n - 1; i++)
            M'.K = M'.K + qqi+1;
            M'.Delta = M'.Delta + ((qqi, ε, a1+1), (qqi+1, ε));
        M'.Delta = M'.Delta + ((qqn-1, c, an), (p, γ));
        M'.Delta = M'.Delta - t;
    for(each t in M'.Delta)
        if(t == ((q, c, ε), (p, γ))
            for(each a in M'.Gamma)
                M'.Delta = M'.Delta + ((q, c, a), (p, γa));
        M'.Delta = M'.Delta - t;
    return M';
}
```
Theorem 12.2

- Statement: Given PDA \( M = (K, \Sigma, \Gamma, \Delta, s, A) \), there exists a CFG \( G \) such that \( L(G) = L(M) \)
- Proof: By construction
  * Begin by converting PDA to RNF
  * Primary concerns are generating nonterminal symbols and rules for expanding them
  * Nonterminals will have form \(< q_i \gamma q_j >\)
    - This nonterminal generates only those strings that drive \( M \) from state \( q_i \) to state \( q_j \) while popping all symbols on the stack from the top through the first occurrence of \( \gamma \)
    - For PDA in RNF, \(< s\#a >\) generates all strings that drive \( M \) from the start state to \( a \), popping everything off the stack
    - These are the strings in \( L(M) \)
    - First rule in grammar is thus \( S \rightarrow < s\#a > \)
  * To generate rest of rules, consider three kinds of transitions
    1. Those that push no symbols
      - Consider

\[
\begin{array}{c}
\text{q} \\
\varepsilon \gamma / \varepsilon
\end{array} \rightarrow
\begin{array}{c}
\text{r} \\
\text{w}
\end{array}
\]

and transition \( ((q, c, \gamma), (r, \epsilon)) \in M.\Delta \) where \( c \in \Sigma \cup \{\epsilon\} \)
- After \( M \) reaches \( r \), it could reach \( w \) via any states in \( M \)
- Nonterminal \(< q\gamma w >\) generates strings that drive \( M \) from \( q \) to \( w \) while popping \( \gamma \)
- Some of these strings are those that start with \( c \) and that drive \( M \) from \( r \) to \( w \) without popping anything
- So need rule \(< q\gamma w > \rightarrow c < rew >\)
2. Those that push one symbol
   · Consider
   
   \[
   \begin{array}{c}
   q \xrightarrow{c\gamma\alpha} r \ldots \rightarrow w
   \end{array}
   \]
   
   · \(M\) reads \(c\), pops \(\gamma\), pushes alpha, and transitions to \(r\)
   · Nonterminal \(< q\gamma w >\) generates strings that drive \(M\) from \(q\) to \(w\) while popping \(\gamma\)
   · Some of these strings are those that that drive \(M\) from \(q\) to \(r\) while popping \(\gamma\) and reading \(c\), and then driving \(M\) from \(r\) to \(w\) while popping \(\alpha\)
   · So need rule \(< q\gamma w > \rightarrow c < r\alpha w >\)

3. Those that push more than one symbol
   · Consider
   
   \[
   \begin{array}{c}
   q \xrightarrow{c\gamma\alpha_1\alpha_2\ldots\alpha_n} r \rightarrow r_1 \rightarrow r_2 \ldots \rightarrow r_{n-1} \rightarrow w
   \end{array}
   \]
   
   · Since \(M\) is in RNF, can only pop one symbol per transition
   · For every symbol pushed during one transition, will need an additional transition to pop each
   · Some of the strings that drive \(M\) from \(q\) to \(w\) are those that drive \(M\) from \(q\) to \(r\) while popping \(\gamma\)
   · They then drive \(M\) from \(r\) to \(r_1\) while popping \(\alpha_1\), from \(r_1\) to \(r_2\) while popping \(\alpha_2\), etc.
   · They finally transition from \(r_n\) to \(w\) while popping \(\alpha_n\)
   · So need rule \(< q\gamma w > \rightarrow c < r\alpha_1 r_1 > < r_1 \alpha_2 r_2 > \ldots < r_{n-1} \alpha_n w >\)

4. Need to be able to terminate rule expansion
   · So need rule \(< q\epsilon q > \rightarrow \epsilon\)
Pushdown Automata: Equivalence of CFGs and PDAs -
Converting PDAs to CFGs (5)

Algorithm:

\[
\text{CFG convertPDAToCFG (PDA } M) \{
M' = \text{convertPDAToRNF}(M);
V = M'.S;
R = \{S -> <s # a>\};
\text{for (each } ((q, c, gamma), (r, \epsilon)) \text{ in } M'.\Delta \text{)
  for (each state } w \text{ in } M'.K \text{)
    if (} w \neq M'.s \text{)
      R = R + <q gamma w> \rightarrow c<r \epsilon w>;
\text{for (each } ((q, c, gamma), (r, alpha)) \text{ in } M'.\Delta \text{)
  if (} q \neq M'.s \text{)
    for (each state } w \text{ in } M'.K \text{)
      if (} w \neq M'.s \text{)
        R = R + <q gamma w> \rightarrow c<r alpha w>;
\text{for (each } ((q, c, gamma), (r, alpha_1 alpha_2 \ldots \alpha_n)) \text{ in } M'.\Delta \text{)
  if (} q \neq M'.s \text{) \{ 
    \text{create rules pi of form}
    <q gamma w> \rightarrow 
    c<q alpha_1 r_1><r_1 alpha_2 r_2>\ldots<r_{n-1} alpha_n w>;
    R = R + \{pi\};
  \}
\text{for (each state } q \text{ in } M'.K \text{)
  if (} w \neq M'.s \text{)
    R = R + <q c q> \rightarrow \epsilon; 
G = (M'.V, M'.\Sigma, M'.S, R);
\text{return } G; 
\}
\]
Pushdown Automata: Halting

- Computation $C$ of PDA $M$ halts if at least one of the following holds:
  1. $C$ is an accepting computation
  2. $C$ ends in a configuration from which there is no transition in $\Delta$

- $M$ halts on string $w$ if every computation of $M$ on $w$ halts

- $M$ rejects $w$ if it halts on $w$ and does not accept $w$

- PDAs differ from FAs in the following aspects
  1. Some CFGs cannot be represented by deterministic PDAs
  2. A given PDA may not
     (a) Halt
     (b) Consume all input
  3. There is no algorithm for minimizing PDAs

- Two ways of dealing with these difficulties
  1. Formal solutions that do not restrict the class of the language being considered
     - These approaches - in general - restrict the representation of the language
     - This, in turn, affects the usefulness of resulting parse trees
  2. Practical solutions
     - In general, these apply only to restricted subclasses of a language class
     - Such subclasses are usually large enough to provide useful solutions

- Moral:
  - Cannot guarantee solutions to problems like
    1. Is string $w \in L(M)$?
    2. Can a parse tree be generated for $w$?
    3. Can a parse tree be generated for $w$ in time linear wrt $|w|$?
    4. Is $w \in \neg L(M)$?