16.070

Introduction to Computers & Programming

Theory of computation 3: PDA, CFG

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

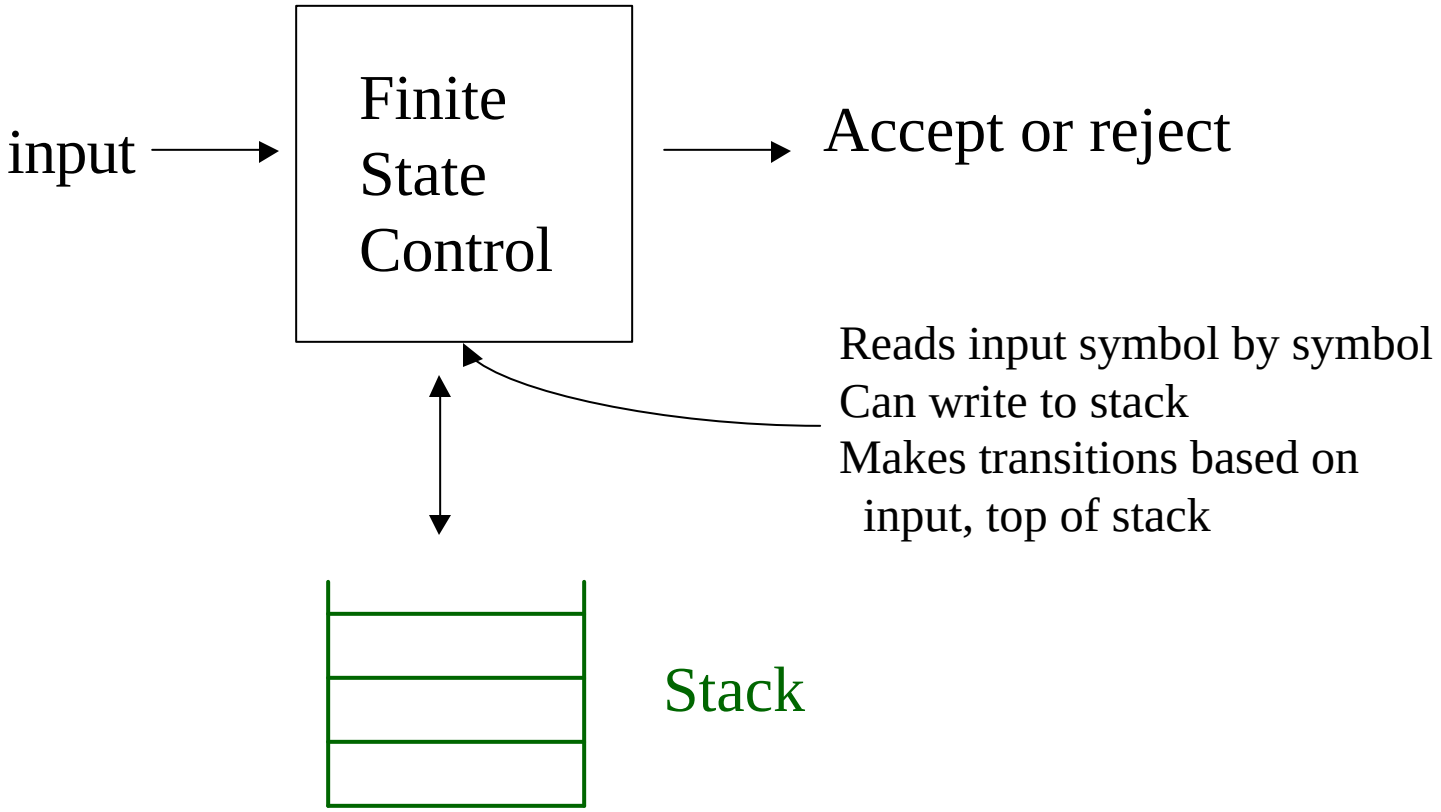
Pushdown Automata (PDA)

- Just as a **DFA** is a way to implement a **regular expression**, a **pushdown automata** is a way to implement a **context free grammar**
  - PDA equivalent in power to a CFG
  - Can choose the representation most useful to our particular problem

- Essentially identical to a regular automata except for the addition of a **stack**
  - Stack is of *infinite size*
  - Stack allows us to recognize some of the non-regular languages

PDA

- Can visualize a PDA with the schematic



Implementing a PDA

- In one transition the PDA may do the following:
  - Consume the **input symbol**. If **e** is the input symbol, then no input is consumed.
  - Go to a **new state**, which may be the same as the previous state.
  - **Replace the symbol** at the **top of the stack** by any string.
    - If this string is **e** then this is a **pop** of the stack
    - The string might be the same as the current stack top (does nothing)
    - Replace with a new string (pop and push)
    - Replace with multiple symbols (multiple pushes)

Informal PDA Example

- Consider the language $L = \{0^n 1^n \mid n \geq 0 \}$.
  - Why is it not regular?

- A PDA is able to recognize this language!
  - Can use its stack to store the number of 0's it has seen.
    - As each 0 is read, push it onto the stack
    - As soon as 1's are read, pop a 0 off the stack
    - If reading the input is finished exactly when the stack is empty, accept the input else reject the input

PDA and Determinism

- The description of the previous PDA was deterministic
- However, **in general the PDA is nondeterministic**.
- This feature is crucial because, unlike finite automata, nondeterminism adds power to the capability that a PDA would have if they were only allowed to be deterministic.
  - i.e. A non-deterministic PDA can represent languages that a deterministic PDA cannot

Informal Non-Deterministic Example

- **$L = \{ ww^R \mid w \text{ is in } (0+1)^* \}$**

- **Informal** PDA description
  - Start in state q_0 that represents the state where we haven't yet seen the reversed part of the string. While in state q_0 we read each input symbol and push them on the stack.
  - At any time, assume we have seen the middle; i.e. "**fork**" off a new branch that assumes we have seen the end of w. We signify this choice by spontaneously going to state q_1. This behaves just like a nondeterministic finite automaton
    - We'll continue in both the forked-branch and the original branch. One of these branches may die, but as long as one of them reaches a final state we accept the input.
  - In state q_1 compare input symbols with the top of the stack. If match, pop the stack and proceed. If no match, then the branch of the automaton dies.
  - If we empty the stack then we have seen ww^R and can proceed to a final accepting state.

Formal Definition of a PDA

- **P = (Q, å, G, d, q_0, Z_0, F)**
  - Q = finite set of states, like the finite automaton
  - ∑ = finite set of input symbols, the alphabet
  - **G** = finite stack alphabet, components we are allowed to push on the stack
  - q_0 = start state
  - **Z_0** = start symbol. Initially, the PDA's stack consists of one instance of this start symbol and nothing else. We can use it to indicate the bottom of the stack.
  - F = Set of final accepting states.

PDA Transition Function

- **d** = transition function, which takes the triple: **d(q, a, X)** where
  - **q** = state in Q
  - **a** = input symbol in Σ
  - **X** = stack symbol in G

- The **output of d** is the finite **set** of pairs **(p, ?)** where **p** is a new state and **?** is a new string of stack symbols that replaces X at the top of the stack.
  - If **?** = e then we pop the stack
  - if **?** = X the stack is unchanged
  - if **?** = YZ then X is replaced by Z and Y is pushed on the stack. Note the new stack top is to the left end.

Formal PDA Example

- Here is a **formal description** of the PDA that recognizes $L = \{0^n1^n \mid n \geq 0\}$.
  - $Q = \{ q_1, q_2, q_3, q_4 \}$
  - $\Sigma = \{0, 1\}$
  - $G = \{0, Z_0\}$
  - $F = \{q1, q4\}$

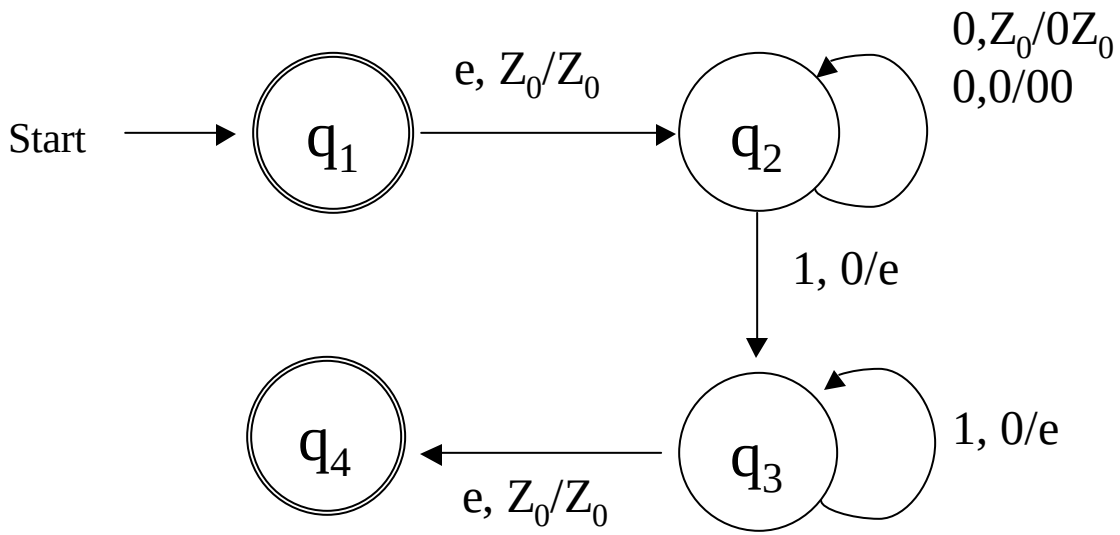
- And **d** is described by the table below

Input:	**0**	**0**	**1**	**1**	**e**	**e**
Stack:	**0**	**Z_0**	**0**	**Z_0**	**0**	**Z_0**
→q_1						$\{(q_2, Z_0)\}$
q_2	$\{(q_2, 00)\}$	$\{(q_2, 0Z_0)\}$	$\{(q_3, e)\}$			
q_3			$\{(q_3, e)\}$			$\{(q_4, Z_0)\}$
→q_4						

Graphical Format

- Uses the format

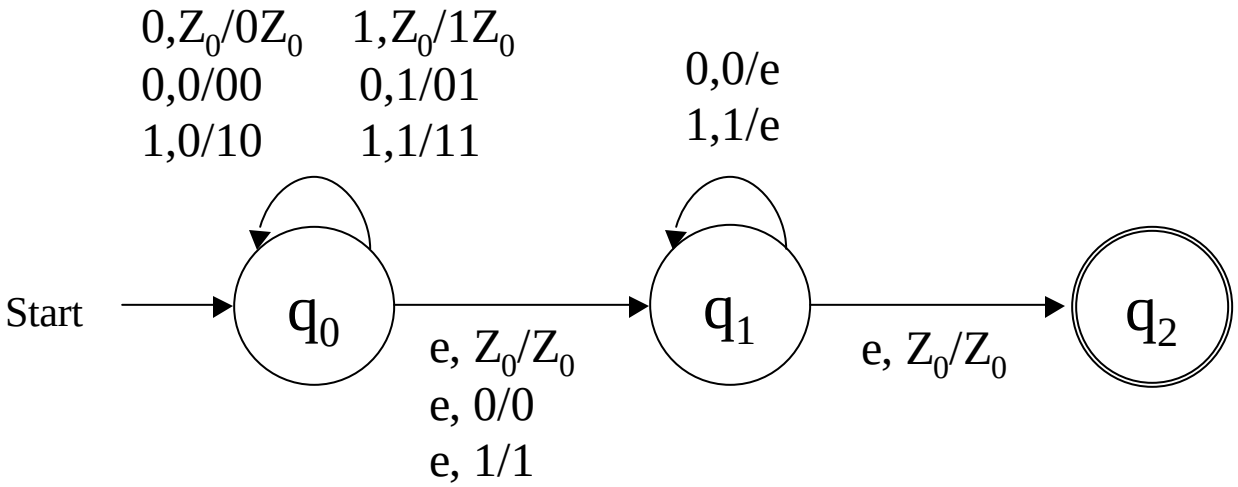
  Input-Symbol, Top-of-Stack / String-to-replace-top-of-stack



Example 2

- Here is the graphical description of the PDA that accepts the language
  - L = { $\mathbf{ww^R}$ **| w is in (0+1)*** }

  - Stays in state q_0 when we are reading w, saving the input symbol. Every time we "guess" that we have reached the end of w and are beginning w^R by going to q_1 on an epsilon-transition.
  - In state q_1 we pop off each 0 or 1 we encounter that matches the input. Any other input will "die" for this branch of the PDA. If we ever reach the bottom of the stack, we go to an accepting state.



Instantaneous Descriptions of a PDA (ID)

- For a **FA**, the only thing of interest about the FA is its **state**.
- For a **PDA**, we want to know its **state and the entire content of its stack**.
  - Often the stack is one of the most useful pieces of information, since it is not bounded in size.
- We can represent the **instantaneous description** (**ID**) of a PDA by the following triple (q,w,?):
  - q is the state
  - w is the remaining input
  - ? is the stack contents
- By convention the **top of the stack** is shown at the **left end of ?** and the **bottom** at the **right** end.

Moves of a PDA

- To describe the **process of taking a transition**, we use the "turnstile" symbol $+$ which is used as:

$$(q, aw, X\beta) + (p, w, \alpha\beta)$$

- In other words, we took a transition such that we went from state q to p, we consumed input symbol a, and we replaced the top of the stack X with some new string α.

- We can extend the move symbol to taking many moves:
  $+^*$ represents **zero or more moves of the PDA**.

Move Example

- Consider the PDA that accepted

  $L = \{ ww^R \mid w \text{ is in } (0+1)^* \}$.

- We can describe the moves of the PDA for the input 0110:
  - $(q_0, 0110, Z_0) + (q_0, 110, 0Z_0) + (q_0, 10, 10Z_0) + (q_1, 10, 10Z_0) + (q_1, 0, 0Z_0) + (q_1, e, Z_0) + (q_2, e, Z_0)$.

- We could have taken other moves rather than the ones above, but they would have resulted in a "dead" branch rather than an accepting state.

Language of a PDA

- The PDA consumes input and **accepts** input when it ends in a **final state**. We can describe this as:

$$\mathbf{L(P) = \{w \mid (q_0, w, Z_0) +^{*} (q, e, a)\} \text{ where } q \in F}$$

- That is, from the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

Alternate Definition for L(PDA)

- It turns out we can also describe a language of a PDA by **ending up with an empty stack with no further input**

$$\mathbf{N(P) = \{w \mid (q_0, w, Z_0) +^* (q, e, e)\}} \text{ where q is any state.}$$

- That is, we arrive at a state such that P can consume the entire input and at the same time empty its stack.
- It turns out that we can show the classes of languages that are L(P) for some PDA P is equivalent to the class of languages that are N(P) for some PDA P.
- This class is also **exactly the context-free languages**.

Equivalence of PDA and CFG

- A context-free grammar and pushdown automata are equivalent in power.

- Theorem: Given a CFG grammar G, then some pushdown automata P recognizes L(G).
  - To prove this, we must show that we can take any CFG and express it as a PDA. Then we must take a PDA and show we can construct an equivalent CFG.
  - We'll show the CFG→PDA process, but only give an overview of the process of going from the PDA→CFG

Context Free Grammars

Context Free Languages (CFL)

- Not all languages are regular
  - There are many classes "**larger**" than that of regular languages
  - One of these classes are called "**Context Free**" languages
- Described by **Context-Free Grammars** (**CFG**)
  - Why named context-free?
  - Property that we can **substitute strings for variables regardless of context**
- CFG's are useful in many applications
  - Describing syntax of programming languages
  - Parsing
  - Structure of documents, e.g.XML
- Analogy of the day:
  - **DFA : Regular Expression** as **Pushdown Automata : CFG**

CFG Example

- Language of palindromes
  - We can show that the language L = { w | w = w^R } **is not regular**.
  - However, we can describe this language by the following **context-free grammar over the alphabet {0,1}**:

$P \rightarrow \varepsilon$
$P \rightarrow 0$
$P \rightarrow 1$
$P \rightarrow 0P0$
$P \rightarrow 1P1$

Inductive definition

More compactly: $P \rightarrow \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$

Formal Definition of a CFG

- There is a finite set of symbols that form the strings, i.e. there is a **finite alphabet**. The alphabet symbols are called **terminals.**
- There is a finite set of **variables**, sometimes called **non-terminals** or syntactic categories. Each variable represents a language (i.e. a set of strings).
  - **Ex**: In the palindrome example, the only variable is P.
- One of the variables is the **start symbol**. Other variables may exist to help define the language.
- There is a finite set of **productions** or production rules that represent the **recursive definition of the language**. Each production is defined:

1. Has a **single variable** that is being defined to the left of the production
2. Has the **production symbol** →
3. Has a string of zero or more terminals or variables, called **the body of the production**. To form strings we can substitute each variable's production in for the body where it appears.

CFG Notation

- A CFG G may then be represented by these four components, denoted G=(V,T,P,S)
  - V is the set of variables
  - T is the set of terminals
  - P is the set of productions
  - S is the start symbol.

Sample CFG

1. E→I // Expression is an identifier
2. E→E+E // Add two expressions
3. E→E*E // Multiply two expressions
4. E→(E) // Add parenthesis
5. I→ L // Identifier is a Letter
6. I→ ID // Identifier + Digit
7. I→ IL // Identifier + Letter
8. D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 // Digits
9. L → a | b | c | … A | B | … Z // Letters

Note **Identifiers** are **regular**; could describe as **(letter)(letter + digit)***

Recursive Inference

- The process of coming up with strings that satisfy individual productions and then concatenating them together according to more general rules is called ***recursive inference***.
- This is a bottom-up process

- For **example**, parsing the identifier "r5"
  - Rule 8 tells us that D → 5
  - Rule 9 tells us that L → r
  - Rule 5 tells us that I→L so I→r
  - Apply recursive inference using rule 6 for I→ID and get
    - I → rD.
    - Use D→5 to get I→r5.
  - Finally, we know from rule 1 that E→I, so **r5 is also an expression.**

Exercise: Show the recursive inference for arriving at *(x+int1)*10* is an expression

Derivation

- Similar to recursive inference, but **top-down** instead of bottom-up
  - Expand start symbol first and work way down in such a way that it matches the input string
- For **example**, given a*(a+b1) we can derive this by:
  - $E \Rightarrow E*E \Rightarrow I*E \Rightarrow L*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(L+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow a*(a+ID) \Rightarrow a*(a+LD) \Rightarrow a*(a+bD) \Rightarrow a*(a+b1)$
- **Note** that at each step of the productions we could have chosen any one of the variables to replace with a more specific rule.

When is this useful? Think for example of your Ada compiler