**Air Force Institute of Technology**

**Graduate School of Engineering and Management**

**Department of Electrical and Computer Engineering**

**CSCE 532 Automata and Formal Languages**

**Winter 2019**

# Day 8

# Pushdown Automata

§2.2 Pushdown Automata

Overview

* Pushdown automata (PDAs) extend NFAs by adding external storage in the form of a single stack with infinite capacity.
* The availability of the stack allows PDAs to recognize some languages that are not regular.
* For example, a PDA can recognize by using its stack to “count” the number of ’s and then compare that to the number of ’s.
* In fact, we will see that the class of languages recognized by PDAs is exactly the same class of languages that can be generated by CFGs, i.e. CFLs.
* Note: PDAs are nondeterministic.
  + A PDA in which every transition is deterministic (in a sense to be made precise later) is a deterministic PDA (DPDA).
  + Whereas DFAs and NFAs each recognize the class of regular languages (so nondeterminism does not add more power), we will see that the same relationship does not hold for PDAs and DPDAs.
  + In particular,
    - DPDAs recognize the class of deterministic context-free languages (DCFLs).
    - All regular languages are DCFLs, but there are DCFLs that are not regular.
    - Also, all DCFLs are CFLs, but there are CFLs that are not DCFLs.

### Notation

If is a set, then .

### Definition

A pushdown automaton (PDA) is a 6-tuple , where , , , and are all finite sets, and

1. is the set of states
2. is the input alphabet,
3. is the stack alphabet
4. is the transition function,
5. is the start state, and
6. is the set of accept states.

A PDA accepts input iff where each and there exist sequences and satisfying

1. and
2. for , we have , where and for some and .
3. .

### Lemma

If a language is context-free, then some pushdown automaton recognizes it.

### Proof

See Sipser (pp. 117-121). In essence, if a language is context-free, then there exists a CFG such that . Given , a PDA can be constructed that simulates derivations in so that .

### Lemma

If a pushdown automaton recognizes some language, then it is context-free.

### Proof

See Sipser (pp. 121-124). This is an example of a situation in which it is easier to prove a stronger result. Given a PDA , a CFG can be constructed that simulates all possible sequences of configurations in . As a special case, it simulates all possible sequences of configurations beginning in the initial state and ending in an accepting configuration, i.e. all sequences in which the input is accepted.

### Theorem

A language is context-free if and only if some pushdown automaton recognizes it.

### Proof

This result follows immediately from the preceding lemmas.

### Corollary

Every regular language is context-free

### Proof

Every regular language is recognized by an NFA. Every NFA is equivalent to a PDA that never uses its stack. Thus, every regular language is recognized by a PDA.

### Example (Sipser Exercise 2.5d)

Give an informal description and state diagram of a PDA for the language .

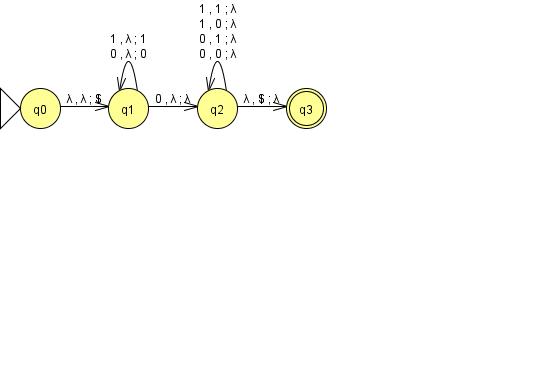
### Solution

#### Informal Description

The PDA with the transition graph shown below pushes input symbols onto the stack until it nondeterministically “chooses” the in the middle of the string. After that, it pops symbols off the stack until it reaches the end of the input. If the stack is empty, then the same number of symbols came before the and after it, so the length of the input was odd and the middle symbol was .

#### State Diagram (a.k.a. Transition Graph)

Note that whereas Sipser’s notation for transition labels uses an arrow to separate the popped stack symbol from the pushed stack symbol, JFLAP’s uses a semicolon.



#### Less Informal Description

Observe the following.

* The transition from to puts a special symbol () at the bottom of the stack.
* Each self-loop on pushes a symbol on the stack.
* The transition from to consumes some in the input without altering the stack.
* Each self-loop on pops a symbol off the stack.
* The transition from to detects the bottom of the stack.

The symbol marks the bottom of the stack. The transition from to “correctly guesses” the “middle ” (i.e. nondeterministically chooses that ), if it exists. If the transition from to is ever taken, then the same number of stack symbols are popped via self-loops on as are pushed via self-loops on . This means there are at least as many input symbols after the “middle ” as before it. If the stack is then empty, the number of input symbols after the “middle ” is exactly the same as the number before. Consequently, accepts.

Note that because we are just counting symbols and not matching them, there is no reason that the stack symbols pushed need to be the same as the input symbols consumed. In fact, anything other than would work, and if we had pushed the same stack symbol for each self-loop on , we would only need half as many self-loop transitions on .

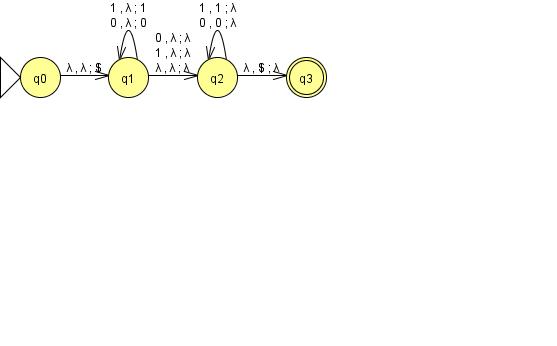
What would a “formal description” look like?

### Practice (Sipser Exercise 2.5e)

Give an informal description and state diagram of a PDA for the language . Note: the given language is similar to but has an essential difference. How is it similar and how is it different?

### Solution

Sipser’s Example 2.18 gives a PDA that recognizes . That PDA has a transition on the empty string that nondeterministically guesses at the middle of the string. The only necessary change is to allow that transition to consume a single input symbol.



### example (Sipser Exercise 2.6a)

Give an informal English description of a PDA recognizing the set of strings over with more ’s than ’s.

### Solution

If we knew that all of the ’s came before all of the ’s, we could simply “count” the ’s and compare to the number of ’s. However, that is not the case. One idea would be to simply “count up” when we see an and “count down” when we see a . However, this fails if any prefix of the string contains more ’s than ’s (e.g. ). Our “counter” needs to be able to represent negative integers as well. We can do that by considering a stack containing ’s to represent a positive integer and a stack containing ’s to represent a negative integer. This leads to the following PDA description.

Push a special symbol (e.g. ) on the stack. Compare each input symbol to the top stack symbol .

* If or , push .
* Otherwise, pop .

After reading the last input symbol, if , accept, and otherwise reject.

### practice (Sipser Exercise 2.6d)

Give an informal English description of a PDA recognizing .

### Solution

1. Push a on the stack.
2. Nondeterministically go to step 4.
3. “Discard” ’s and ’s, and after reading a , go to step 2.
4. Push input symbols on the stack until reaching a . Read it, but don’t push it.
5. Nondeterministically go to step 7.
6. “Discard” ’s and ’s, and after reading a , go to step 5.
7. Compare input symbols to symbols popped off the stack. If they ever disagree, reject. After matching a in the input to one on the stack, accept.

* §2.3 Non-Context-Free Languages

### Practice (Sipser Problem 2.18a)

* Let be a context-free language and be a regular language. Prove that the language is context-free.

### Solution

* See Sipser’s solution on p. 161.

### Practice (Sipser Problem 2.18b)

* Let . Use part (a) to show that is not a CFL.

### Solution

* Assume is context-free. Let . Then is regular, so by part (a), must be context-free. However, we know from Sipser Example 2.36 that is not context-free. Thus, our assumption must be false.