Pushdown Automata

Theory of Computation

CISC 603, Spring 2020, Daqing Yun

Where are we?

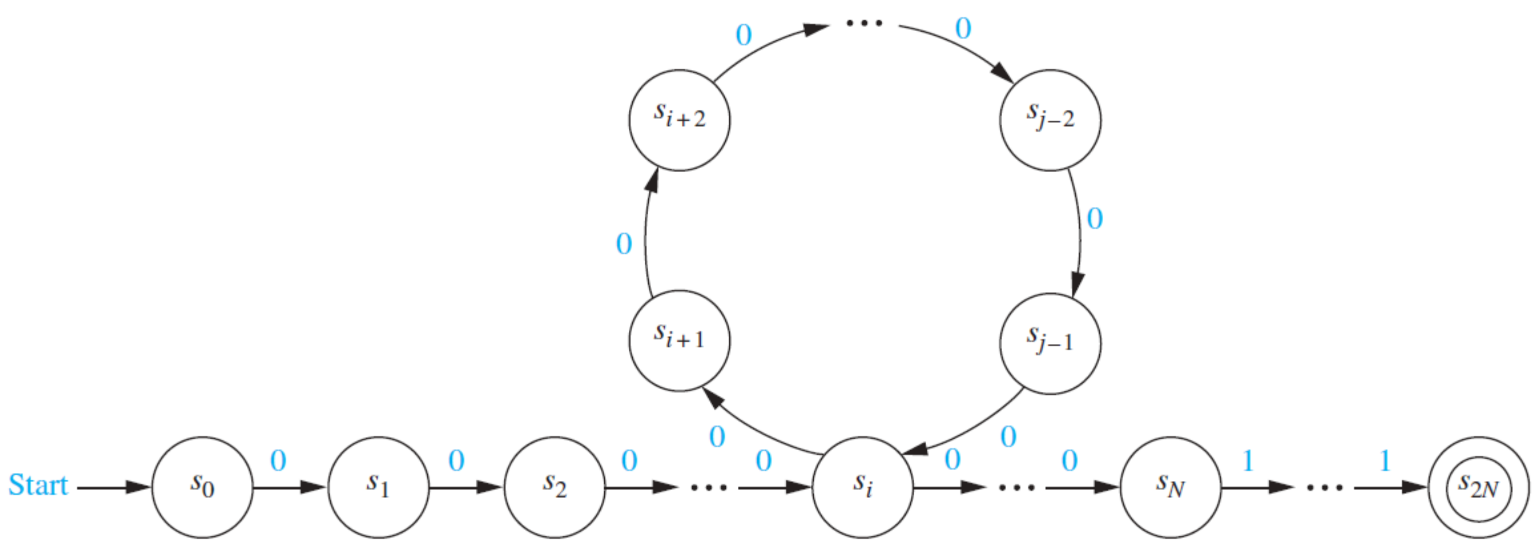
- We have seen how to design a complex system from building block like sets, RegEx, DFA, NFA
- Our machine in its current capability has difficulty recognizing certain sets
- Adding more computational power
  - Build more powerful types of machines like Pushdown Automaton (PDA)
  - They contribute to the building of the "smartest" software computers have, i.e., the *compiler*
  - Helpful for us to understand programming language and get better coding skills

Pushdown Automata

- The Pushdown Automaton (PDA) is an automaton equivalent to CFG in language-defining power
- Only the nondeterministic PDA define all the CFL's
- But the deterministic version models parsers
  - Most programming languages have deterministic PDA's

A Set Not Recognized by an FA (RegEx)

- Show that the set $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$, made up of all strings consisting of a block of 0s followed by a block of an equal number of 1s, is not regular



The path produced by $0^n 1^n$

Intuition

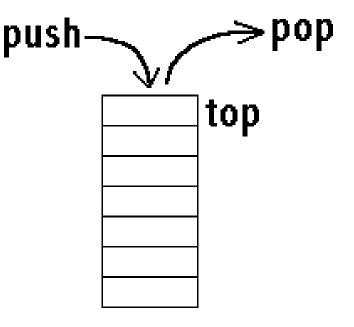
- There are limitations in these machines' capabilities at recognizing certain sets
- If nondeterminism is not enough to make a FA more capable, what else can we do to give it more power?
- The current problem is from the machines' limited storage, let us try to add some extra storage and see what happens

Pushdown Automata

- A language can be generated by a CFG if and only if it can be accepted by a *pushdown automaton*
- A pushdown automaton is similar to a FA but has an auxiliary memory in the form of a stack
- Pushdown automata are, by default, nondeterministic. Unlike FA's, the nondeterminism cannot always be removed

Intuition: PDA

- Think of an ε-NFA with the additional power that it can manipulate a stack
- Its moves are determined by:
  1. The current state (of its "NFA")
  2. The current input symbol (or ε), and
  3. The current symbol on top of its stack



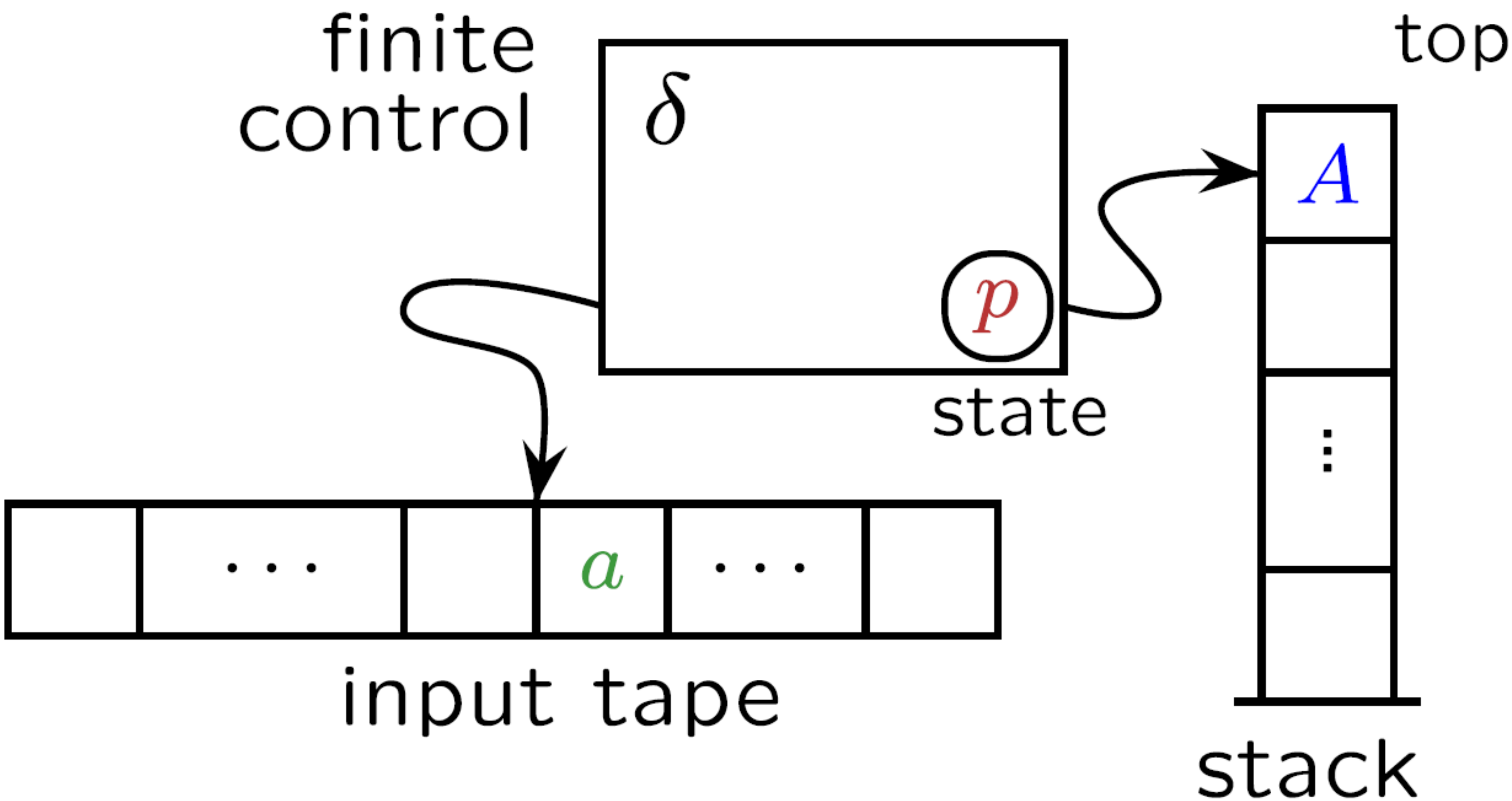
Intuition: PDA (cont'd.)

- Being nondeterministic, the PDA can have a choice of next moves
- In each choice, the PDA can:
  1. Change state, and
  2. Replace the top symbol on the stack by a sequence of zero or more symbols
     - Zero symbols = "pop"
     - Many symbols = sequence of "pushes"

PDA Formalism

- A PDA is described by (a 7-tuple):
  - A finite set of *states* (Q, typically)
  - An *input alphabet* (Σ, typically)
  - A *stack alphabet* (Γ, typically)
  - A *transition function* (δ, typically)
  - A *start state* (q_0, in Q, typically)
  - A *start symbol* (Z_0, in Γ, typically)
  - A set of *final states* ($F \subseteq Q$, typically)

Pushdown Automata



Conventions

- a, b, … are input symbols
  - But sometimes we allow ε as a possible value
- …, X, Y, Z are stack symbols
- …, w, x, y, z are strings of input symbols
- α, β,… are strings of stack symbols

The Transition Function

- Takes three arguments:
  1. A state, in Q
  2. An input, which is either a symbol in Σ or ε
  3. A stack symbol in Γ
- $\delta(q, a, Z)$ is a set of zero or more actions of the form (p, α)
  - p is a state; α is a string of stack symbols

Actions of the PDA

- If $\delta(q, a, Z)$ contains (p, α) among its actions, then one thing the PDA can do in state q, with a at the front of the input, and Z on top of the stack is:
  1. Change the state to p
  2. Remove a from the front of the input (but a may be ε)
  3. Replace Z on the top of the stack by α

Example

- Design a PDA to accept $\{0^n 1^n \mid n \geq 1\}$
- The states:
  - q = start state. We are in state q if we have seen only 0's so far
  - p = we've seen at least one 1 and may now proceed only if the inputs are 1's
  - f = final state; accept

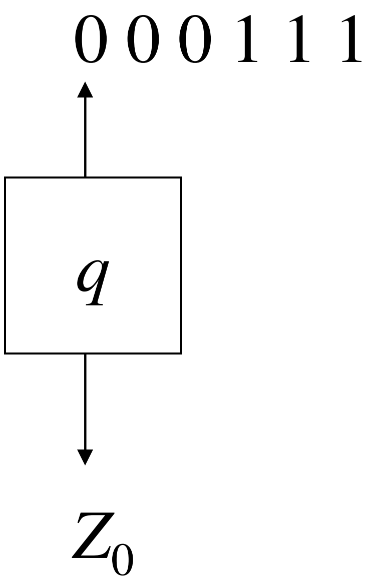
Example (cont'd.)

- The stack symbols:
  - Z_0 = start symbol. Also marks the bottom of the stack, so we know when we have counted the same number of 1's as 0's
  - X = marker, used to count the number of 0's seen on the input

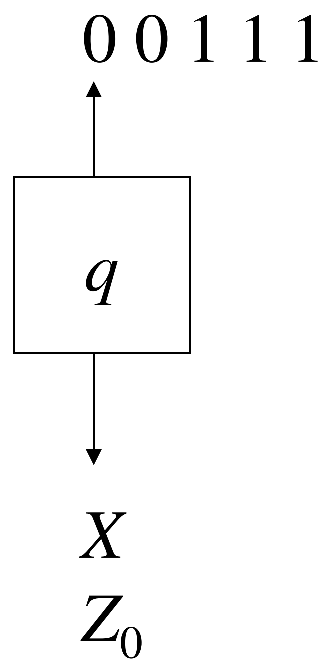
Example (cont'd.)

- The transitions:
  - $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$
  - $\delta(q, 0, X) = \{(q, XX)\}$ – these two rules cause one X to be pushed onto the stack for each 0 read from the input
  - $\delta(q, 1, X) = \{(p, \varepsilon)\}$ – when we see a 1, go to state p and pop one X
  - $\delta(p, 1, X) = \{(p, \varepsilon)\}$ – pop one X per 1
  - $\delta(p, \varepsilon, Z_0) = \{(f, Z_0)\}$ – accept at bottom

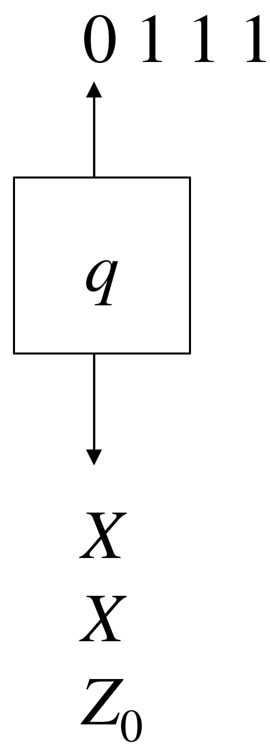
Actions of the Example PDA



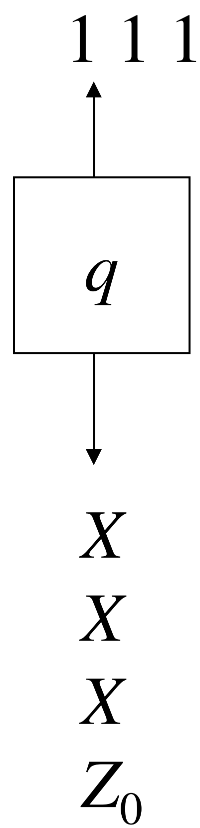
Actions of the Example PDA



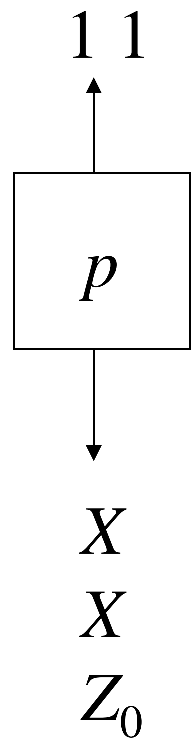
Actions of the Example PDA



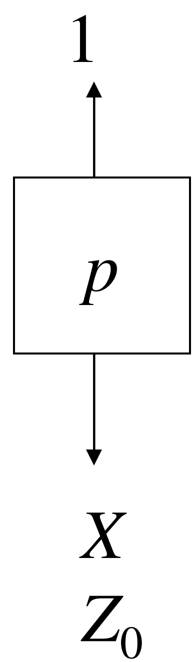
Actions of the Example PDA



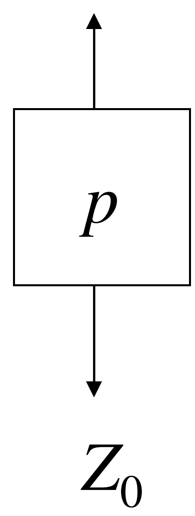
Actions of the Example PDA



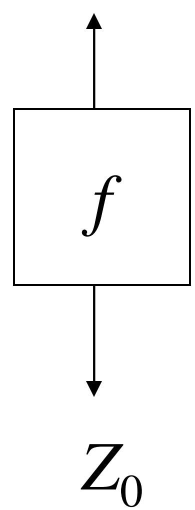
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Instantaneous Descriptions

- We can formalize the pictures just seen with an *instantaneous description* (ID)
- An ID is a triple (q, w, α), where:
  1. q is the current state
  2. w is the remaining input
  3. α is the stack contents, top at the left

The "Goes-To" Relation

- To say that ID I can become ID J in one move of the PDA, we write $I \vdash J$
- Formally, $(q, aw, X\alpha) \vdash (p, w, \beta\alpha)$ for any w and α, if $\delta(q, a, X)$ contains (p, β)
- Extend $\vdash$ to $\vdash^*$, meaning "zero or more moves" by:
  - Basis: $I \vdash^* I$
  - Induction: If $I \vdash^* J$ and $J \vdash^* K$, then $I \vdash^* K$

Example: Goes-To

- Using the previous example PDA, we can describe the sequence of moves by:

  $(q, 000111, Z_0) \vdash (q, 00111, XZ_0) \vdash$

  $(q, 0111, XXZ_0) \vdash (q, 111, XXXZ_0) \vdash$

  $(p, 11, XXZ_0) \vdash (p, 1, XZ_0) \vdash (p, \varepsilon, Z_0) \vdash$

  (f, ε, Z_0)
- Thus, $(q, 000111, Z_0) \vdash^* (f, \varepsilon, Z_0)$
- What would happen on input 0001111?

What would happen on input 0001111?

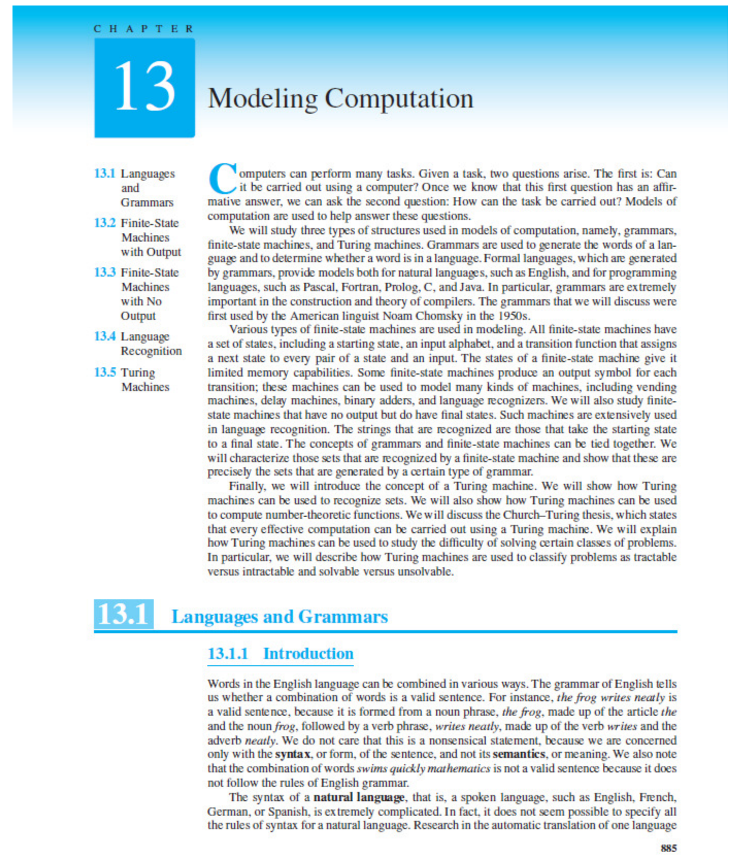
Answer

Legal because a PDA can use ε input even if input remains

- $(q, 0001111, Z_0) \vdash (q, 001111, XZ_0) \vdash (q, 01111, XXZ_0) \vdash (q, 1111, XXXZ_0) \vdash (p, 111, XXZ_0) \vdash (p, 11, XZ_0) \vdash (p, 1, Z_0) \vdash (f, 1, Z_0)$
- Note the last ID has no move
- 0001111 is *not* accepted, because the input is not completely consumed

Reading: Modeling Computation

- Rosen Book, Chapter 13
- See Canvas page



CHAPTER

13 Modeling Computation

Computers can perform many tasks. Given a task, two questions arise. The first is: Can it be carried out using a computer? Once we know that this first question has an affirmative answer, we can ask the second question: How can the task be carried out? Models of computation are used to help answer these questions.

We will study three types of structures used in models of computation, namely, grammars, finite-state machines, and Turing machines. Grammars are used to generate the words of a language and to determine whether a word is in a language. Formal languages, which are generated by grammars, provide models both for natural languages, such as English, and for programming languages, such as Pascal, Fortran, Prolog, C, and Java. In particular, grammars are extremely important in the construction and theory of compilers. The grammars that we will discuss were first used by the American linguist Noam Chomsky in the 1950s.

Various types of finite-state machines are used in modeling. All finite-state machines have a set of states, including a starting state, an input alphabet, and a transition function that assigns a next state to every pair of a state and an input. The states of a finite-state machine give it limited memory capabilities. Some finite-state machines produce an output symbol for each transition; these machines can be used to model many kinds of machines, including vending machines, delay machines, binary adders, and language recognizers. We will also study finite-state machines that have no output but do have final states. Such machines are extensively used in language recognition. The strings that are recognized are those that take the starting state to a final state. The concepts of grammars and finite-state machines can be tied together. We will characterize those sets that are recognized by a finite-state machine and show that these are precisely the sets that are generated by a certain type of grammar.

Finally, we will introduce the concept of a Turing machine. We will show how Turing machines can be used to recognize sets. We will also show how Turing machines can be used to compute number-theoretic functions. We will discuss the Church–Turing thesis, which states that every effective computation can be carried out using a Turing machine. We will explain how Turing machines can be used to study the difficulty of solving certain classes of problems. In particular, we will describe how Turing machines are used to classify problems as tractable versus intractable and solvable versus unsolvable.

13.1 Languages and Grammars

13.1.1 Introduction

Words in the English language can be combined in various ways. The grammar of English tells us whether a combination of words is a valid sentence. For instance, *the frog writes neatly* is a valid sentence, because it is formed from a noun phrase, *the frog*, made up of the article *the* and the noun *frog*, followed by a verb phrase, *writes neatly*, made up of the verb *writes* and the adverb *neatly*. We do not care that this is a nonsensical statement, because we are concerned only with the **syntax**, or form, of the sentence, and not its **semantics**, or meaning. We also note that the combination of words *swims quickly mathematics* is not a valid sentence because it does not follow the rules of English grammar.

The syntax of a **natural language**, that is, a spoken language, such as English, French, German, or Spanish, is extremely complicated. In fact, it does not seem possible to specify all the rules of syntax for a natural language. Research in the automatic translation of one language

885

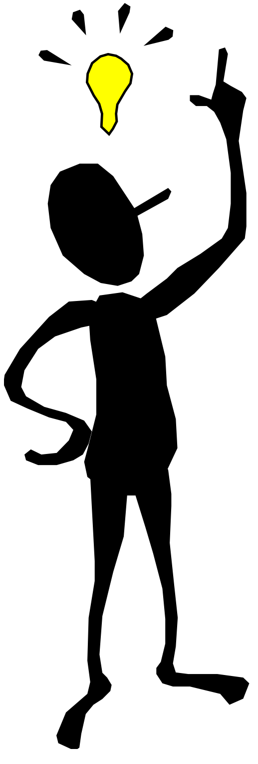
Course Project

A Toy Compiler

- Programming challenge: write a program (using any language of your choice) that can “understand” and “execute” some commands you defined beforehand
- For example, see in-class demo

```
// this is a curve of sin function
origin is (200,200);
rot is 0;
scale is (10,4);
for T from 0 to 2*pi + pi/50 step pi/500 draw(T,-30*sin(T));
```

- Course project deliverable 1 asks you to set up your development environment, and course project deliverable 2 asks you to design the CFG of your “programming language”



Thanks ! ☺

Questions ?