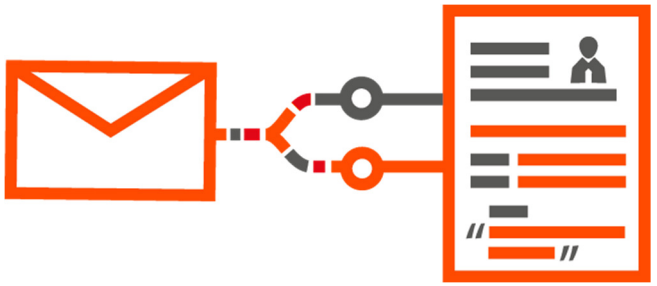
PDA, Tokenization, and Paring

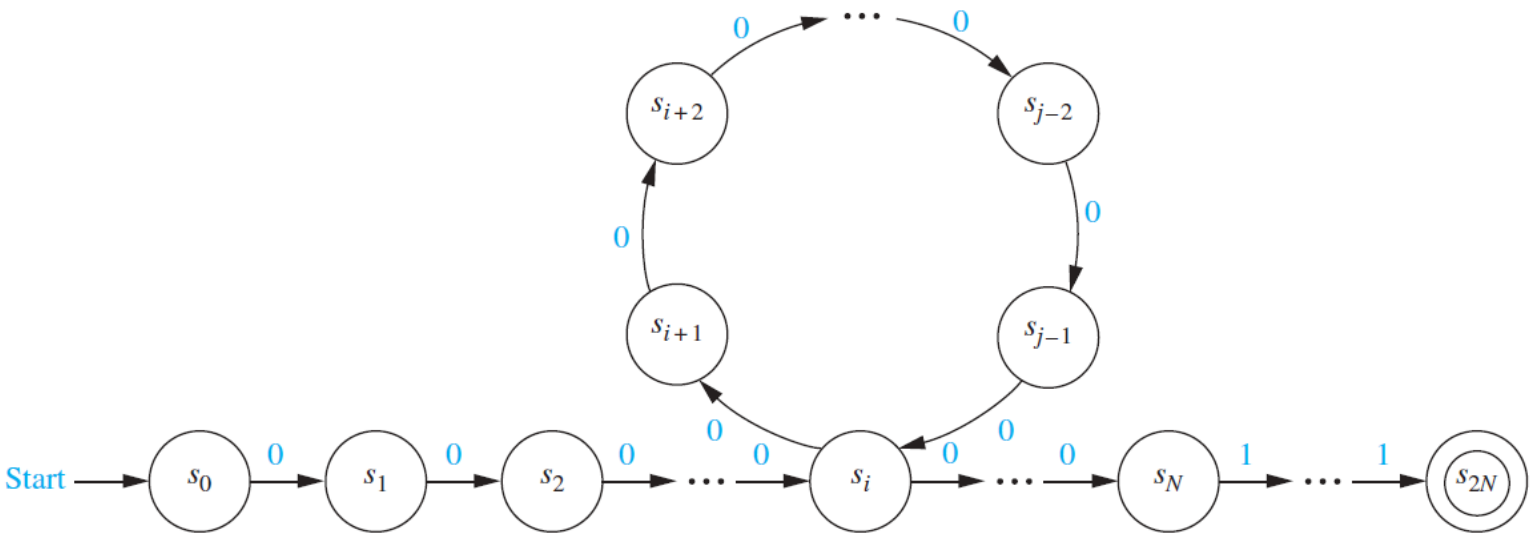


Theory of Computation

CISC 603, Spring 2020, Daqing Yun

Recall – A Set Not Recognized by an FA (RegEx)

- The set $\{0^n1^n \mid n = 0, 1, 2, \ldots\}$, 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^n1^n

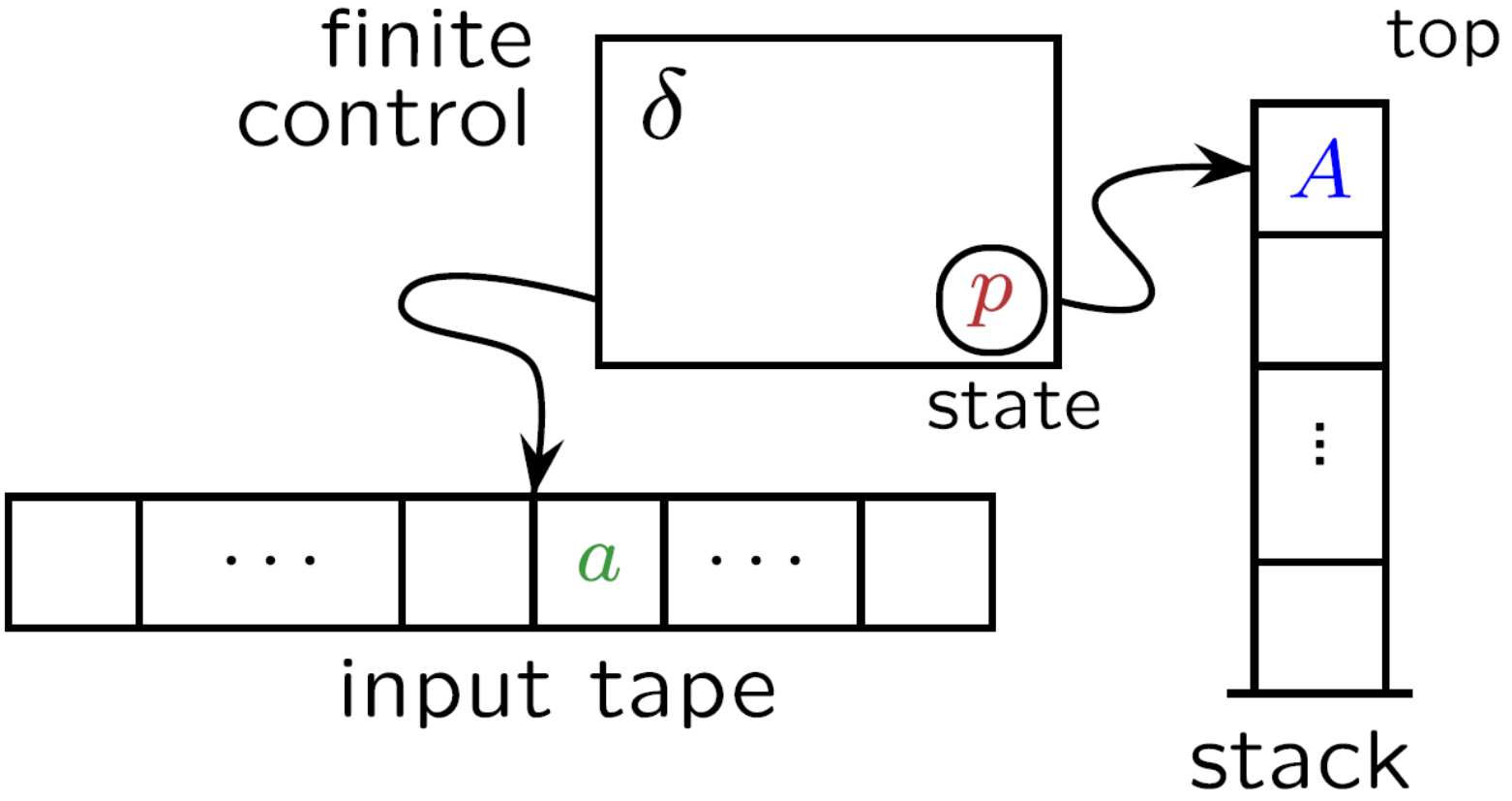
Recall – Where Were We?

- 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 (e.g., $\{0^n1^n \mid n = 0, 1, 2, \ldots\}$)
- 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

Recall – Adding External Storage

- This extra space gives a machine *external* memory in addition to the limited *internal* memory provided by its state
- Having an external memory makes all the difference to a machine's computational power
- Organized as a stack, i.e., a *last-in-first-out* data structure

Recall – Pushdown Automata



How PDAs Recognize "(()(()()))"

1. Given two machine states, 1 and 2, with state 1 being the *accept* state
2. Start the machine in state 1 with an empty stack
3. When in state 1 and an opening bracket is read, push some character – let us use *b* for "bracket" – onto the stack and move into state 2
4. When in state 2 and an opening bracket is read, push the character *b* onto the stack
5. When in state 2 and a closing bracket is read, pop the character *b* off the stack
6. When in state 1 and the stack is empty, move back to state 1

Example – Use (deterministic) PDA to recognize the string “(()(()()))”

State	Accepting?	Stack contents	Input	Action

Example – Use (deterministic) PDA to recognize the string "(( ) (( ) ( )))"

State	Accepting?	Stack contents	Input	Action
1	Yes		(( ) (( ) ( )))	read (, push b, go to state 2
2	No	b	( ) (( ) ( )))	read (, push b
2	No	bb	) (( ) ( )))	read), pop b
2	No	b	(( ) ( )))	read (, push b
2	No	bb	( ) ( )))	read (, push b
2	No	bbb	) ( )))	read), pop b
2	No	bb	( )))	read (, push b
2	No	bbb	)))	read), pop b
2	No	bb	))	read), pop b
2	No	b	)	read), pop b
2	No			go to state 1
1	Yes			—

Determinism

- Whatever state it is currently in, and whichever character it reads, it is always absolutely *certain* which state it will end up in
- This certainty is guaranteed as long as we respect two constraints
  - No contradictions
  - No omissions
- DFA: machines the obey the determinism constraints

Deterministic Pushdown Automaton (DPDA)

- Two important things to know, at each step:
  - What its current state is, and
  - What the current contents of its stack are
- Configuration, or Instantaneous Description
- “Goes-To” Relation
  - Viewed this way, a DPDA just has a current configuration, and the rule tells us how to turn the current configuration into the next configuration each time we read a character

Nondeterministic Pushdown Automaton (NPDA)

- The balanced-brackets machine does need the stack to do its job, it is really just:
  - using the stack as a *counter*, and
  - its rules are only interested in the distinction between "the stack is empty" and "the stack is not empty"
- More sophisticated DPDAs will push more than one kind of symbols to the stack and make use of that information as they perform computations

Nondeterministic Pushdown Automaton (NPDA)

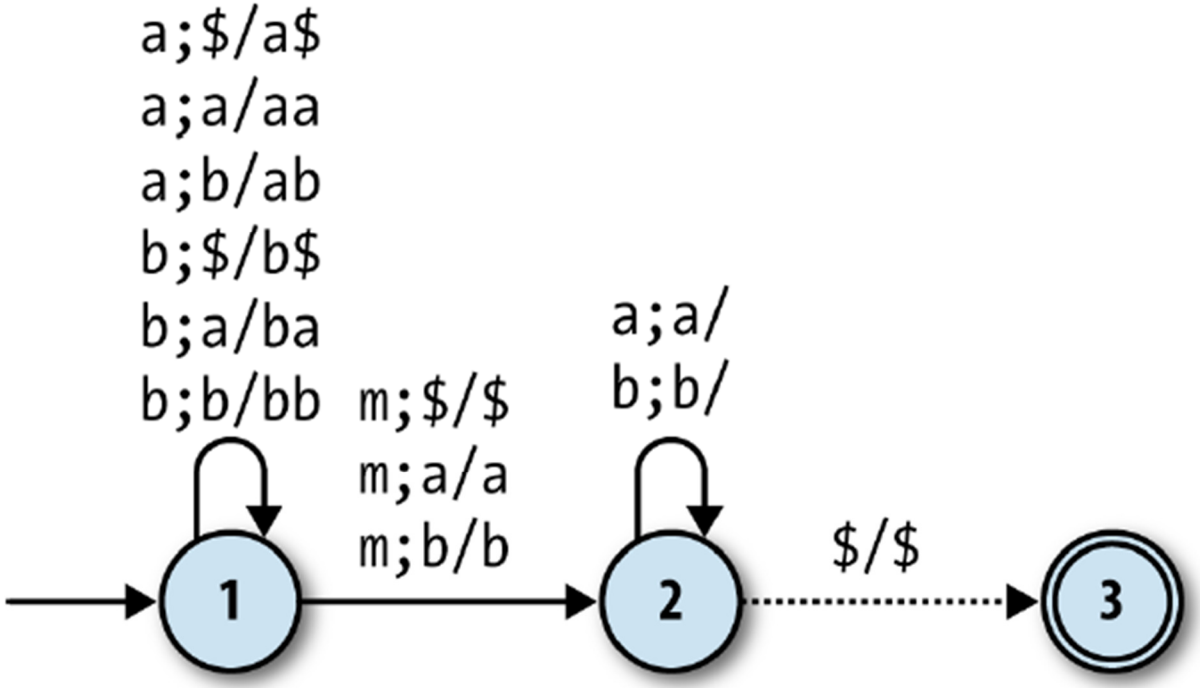
- Can the previous DPDA recognize palindromes like *aa*, *abba*, *babbaabbab*, etc.?
- The machine has to change from state 1 to state 2 as soon as it reaches the *halfway* point of the string, and without a marker, it has no way of knowing when to do that

Nondeterministic Pushdown Automaton (NPDA)

- Solution:
  - Relaxing the determinism constraints
  - Allowing the machine the freedom to make that vital state change at any point
  - Make it possible for it to accept a palindrome by following the right rule at the right time
- Such a pushdown automaton without determinism constraints is called a nondeterministic pushdown automaton (NPDA)

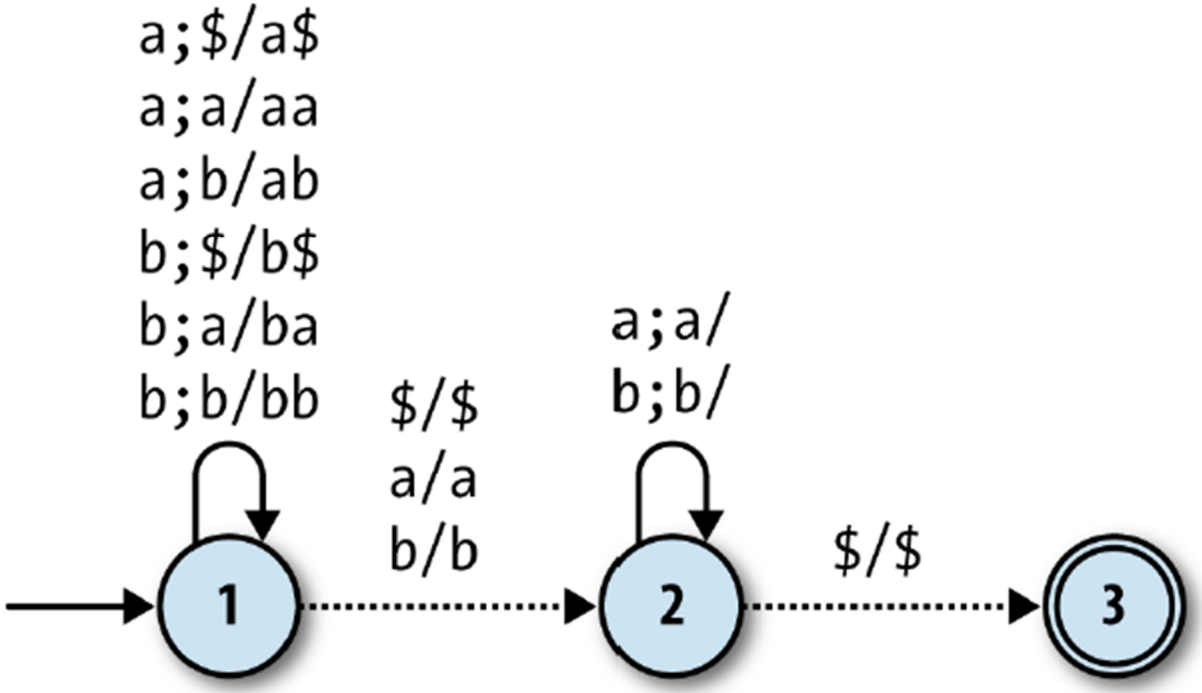
Nondeterministic Pushdown Automaton (NPDA)

- Recognizing palindromes



Nondeterministic Pushdown Automaton (NPDA)

- Recognizing palindromes



Nonequivalence

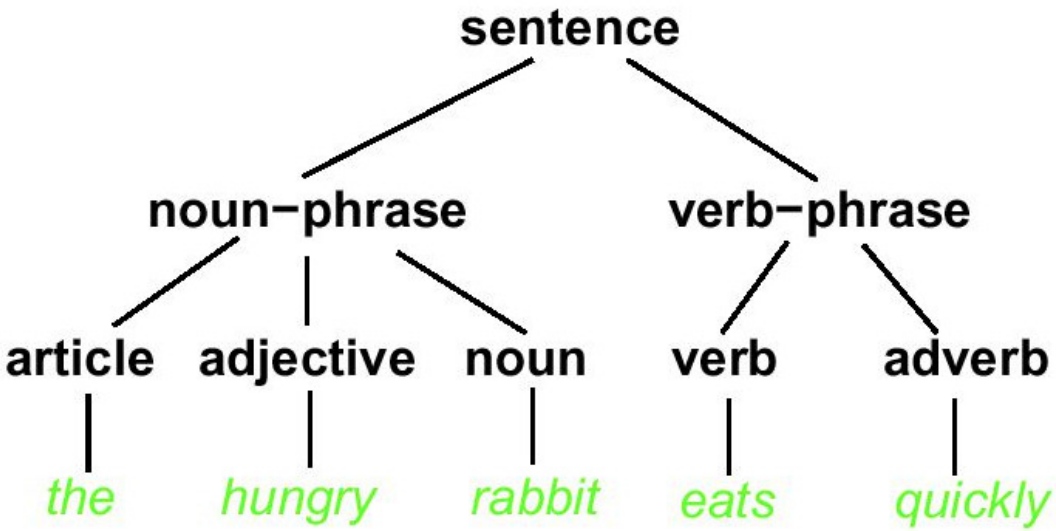
- Is there an algorithm for converting any NPDA to a DPDA?
- No
  - The NFA-to-DFA trick only works because we can use a simple DFA state to represent many possible NFA states
  - To simulate an NFA, we only need to keep track of what states it *could* currently be in, then pick a different set of possible states each time we read an input character, and a DFA can easily do that job if we give it the right rules
  - This trick does not work for PDAs: we cannot usefully represent multiple NPDA configurations as a single DPDA configuration

Nonequivalence

- Is there an algorithm for converting any NPDA to a DPDA?
- No
  - The problem is the stack mechanical limitation
  - An NPDA simulation needs to know all the characters that could currently be on top of the stack, and it must be able to pop and push several of the simulated stacks simultaneously
  - There is no way to combine all the possible stacks into a single stack so that a DPDA can still see the topmost characters and access every possible stack individually

Parsing with Pushdown Automata

- A derivation in the language generated by a context-free grammar can be represented graphically using an ordered rooted tree, called a derivation, or parse tree
- *The hungry rabbit eats quickly*



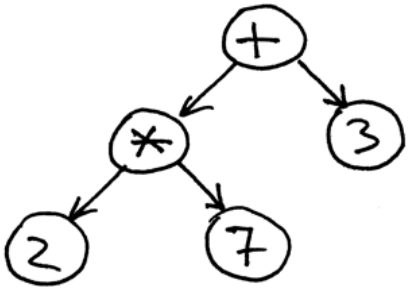
If the production $A \rightarrow w$ arises in the derivation, where w is a word, then the vertex that represents A has as children vertices that represent each symbol in w, in order from left to right.

On the Structure of Programs and Languages

Concrete and Abstract Syntax

- Programs can be represented in their abstract syntax and the concrete syntax forms
- The concrete syntax is the notation with which the user interacts as one edits a program
- It may be textual, symbolic, tabular, graphical, or any combination thereof

On the Structure of Programs and Languages



Concrete and Abstract Syntax

- The abstract syntax is a data structure that represents the semantically relevant data expressed by a program
- It does not contain notational details such as keywords, symbol, white spaces or positions, sizes and coloring in graphical notations

On the Structure of Programs and Languages

Concrete and Abstract Syntax

- The abstract syntax is used for analysis and downstream processing of programs
- A language definition includes the concrete and the abstract syntax, as well as rules for mapping one to the other
- Parser-based systems map the concrete syntax to the abstract syntax

On the Structure of Programs and Languages

Concrete and Abstract Syntax

- Users interact with a stream of characters, and a parser derives the abstract syntax by using a grammar and mapping rules
- The concrete syntax is a mere project/mapping (that looks and feels like text when a textual project/mapping is used). No parsing takes place

On the Structure of Programs and Languages

Concrete and Abstract Syntax

- The abstract syntax of programs are primarily trees of program elements. Each element is an instance of a language *concept*, or concept
- A language is essentially a set of concepts. Every element (except the root) is contained by exactly one parent element

On the Structure of Programs and Languages

Concrete and Abstract Syntax

- Syntactic nesting of the concrete syntax corresponds to a parent-child relationship in the abstract syntax
- There may also be any number of non-containing cross-references between elements, established either directly during editing or by a name resolution phase that follows parsing and tree construction (e.g., `#include <cstdio.h>`)

On the Structure of Programs and Languages

Fragments

- A program may be composed from several program fragments. A fragment is a standalone tree, a partial program
- Conversely, a program is a set of fragments connected by references

On the Structure of Programs and Languages

Languages

- A language consists a set of language concepts and their relationships
- The term concept refers to all aspects of an element of a language, including concrete syntax, abstract syntax, the associated type system rules and constraints as well as some definition of its semantics
- In a fragment, each element is an instance of a concept defined in some language

Parsing with Pushdown Automata

- **Lexical Analysis**
  - Read a *raw string of characters* and turn it into a sequence of *tokens*
  - Each token represents an individual building block of program syntax, like variable name, opening bracket, while keyword, etc.
  - A lexical analyzer uses a language-specific set of rules called a *lexical grammar* to decide which character sequences should produce which tokens
  - This stage deal with messy character-level details like variable-naming rules, comments, white spaces, etc., leaving a clean sequence of tokens for the next stage to consume

Parsing with Pushdown Automata

- **Syntactic Analysis**
  - Read a sequence of tokens and decide whether they represent a valid program according to the syntactic grammar of the language being parsed
  - If the program is valid, the syntactic analyzer may produce additional information about its structure (e.g., a parse tree)

Lexical Analysis

- The lexical analysis stage is usually pretty straightforward
- It can be done with regular expressions (and therefore by an NFA)
- Because it involves simply matching a flat sequence of characters against some *rules* and deciding whether those characters look like a *keyword*, a *variable* name, an *operator*, or whatever else

Syntactic Analysis

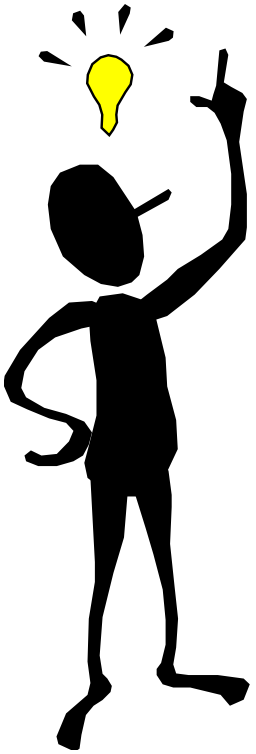
- How to decide whether those tokens represent a syntactically valid program keywords
- Cannot use regular expressions of NFAs because our program's syntax is designed to allow *arbitrary nesting* of brackets
- We know that finite automata are not powerful enough to recognize languages like that
- The use of PDA comes in handy to recognize valid sequences of tokens → next time

Course Project – *A Toy Compiler*

- Write a program (using any programming language) that can "understand" and "execute" some pre-defined commands

```
// 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));
```

- Prepare to present, discuss, and take questions about your design (10–15 minutes) in class during our 2nd executive visit



Thanks ! ☺

Questions ?