Lexical Scanning

COMP360

“Captain, we’re being scanned.”

Spock

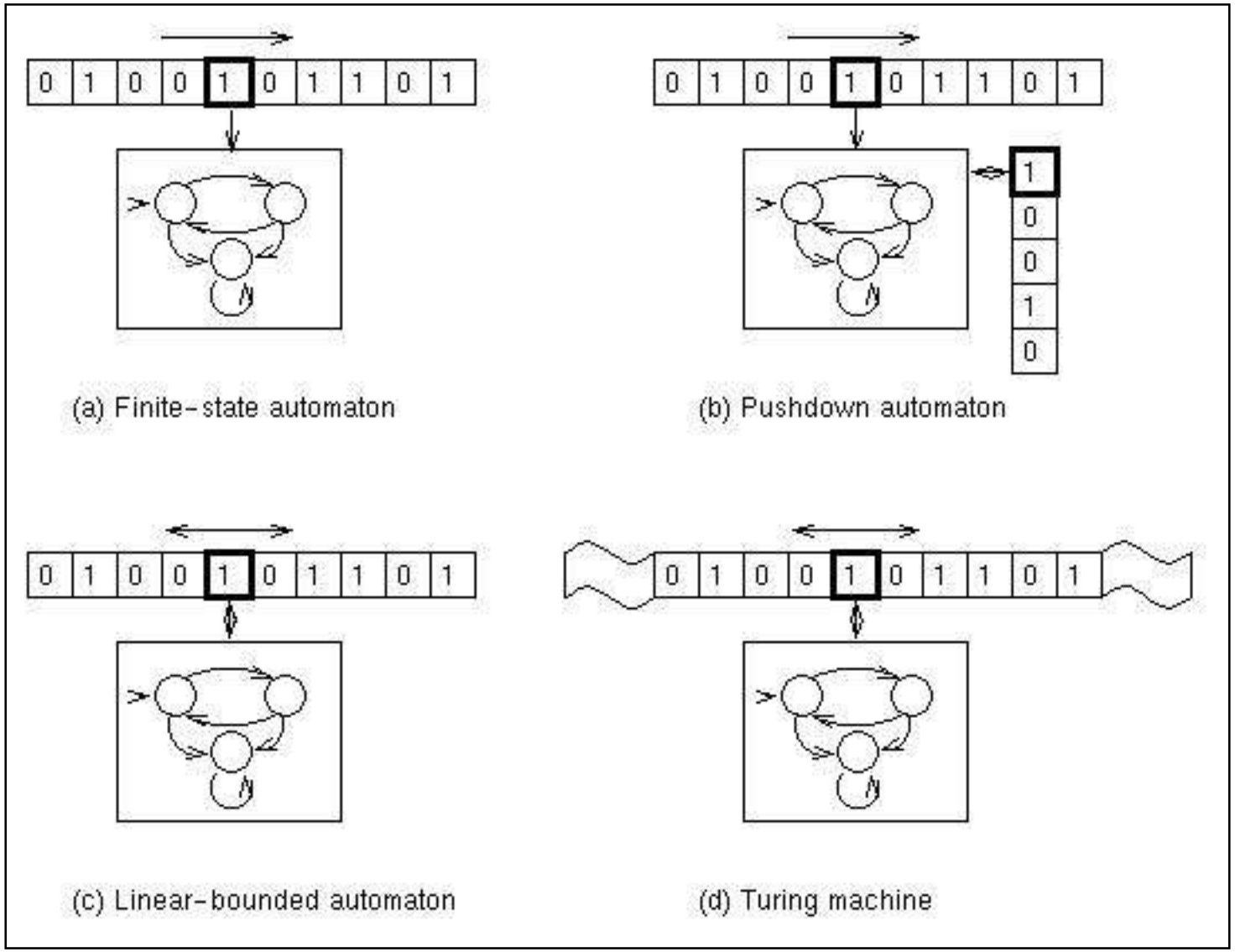
Reading

Read sections 2.1 – 3.2 in the textbook

Regular Expression and FSA Assignment

- A new assignment has been posted on Blackboard
- It is due by 2:00pm on Monday, February 6, 2017
- There are nine questions requiring you to write regular expressions and FSA

Grammar-Machine equivalence



Write a Regular Expression

- Write a regular expression to define a string that begins with exactly two **a**'s and ends with a single **a** and can have any number of **b**'s between them

Possible Solution

- Write a regular expression to define a string that begins with exactly two **a**'s and ends with a single **a** and can have any number of **b**'s between them

aa b* a

Equivalent

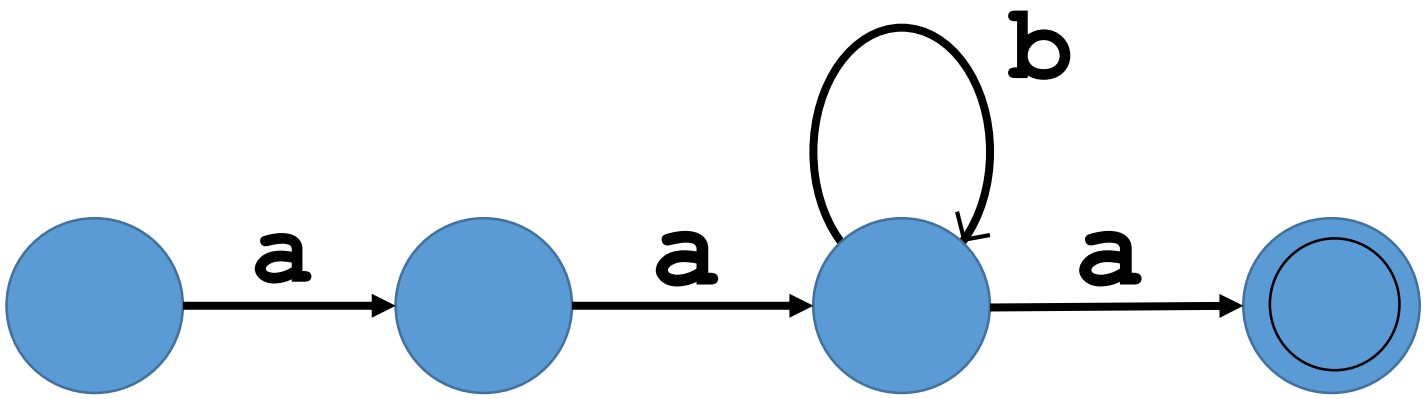
- A FSA and a regular expression can define a regular language
- Regular expressions can be recognized by a regular language

Converting a Regular Expression to a FSA

- Consider the regular expression

aa b* a

- It can be recognized by the FSA



Push Down Automata

- A PDA has a FSA and a stack memory
- The top of the stack, input symbol and FSA state determine what a PDA will do
- A PDA can:
  - Push a new symbol on the stack
  - Pop the top symbol from the stack
  - Change the FSA state

Push Down Automata Languages

- A Push Down Automata (PDA) can recognize context free grammars
- Almost all modern programming languages are context free grammars
- A PDA can “count” things
- Nobody has developed a programming language more complicated than a context free grammar

Defining a Language

- For a program to be able to recognize a member of a language, you have to be able to accurately and unambiguously define the language
- A regular language can be defined by a regular expression
- A context free language can be defined by a BNF

Backus-Naur Form

- Backus-Naur Form or **BNF** can be used to define Context Free Grammars
- Created by John Backus and Peter Naur. Independently created by Noam Chomsky
- BNF is a **metalanguage**, a language used to describe a language

BNF Fundamentals

- Terminal symbols are tokens or symbols in the language. They come from the lexical scanner
- Nonterminal symbols are “variables” that represent patterns of terminals and nonterminal symbols
- A BNF consists of a set of **productions** or **rules**
- A language description is called a **grammar**

BNF Structure

- Nonterminal symbols are sometimes enclosed in brackets to differentiate them from terminal symbols
- I will use **bold font** for terminal symbols and no brackets
- Productions are of the form:

  nonterminal → nonterminals or **terminals**

- Multiple definitions are separated by | meaning OR

  whatever → this | that

BNF to Define a Language

- There must be one nonterminal start symbol which must appear at least once on the left hand side of a rule
- The start symbol defines the language and all other rules follow from it

BNF Example

wloop → **while (** logical **)** stmt

logical → exp relation exp

relation → > | **<** | **==** | **!=**

stmt → *something not defined here*

exp → *something not defined here*

A BNF is used to define a

A. Regular expression
B. Finite State Automaton
C. Context Free Language
D. Push Down Automaton

Compiler’s Purpose

- A compiler converts program source code into a form that can be executed by the hardware
- A compiler works with the language libraries and the system linker

Stages of a Compiler

- Source preprocessing
- Lexical Analysis (scanning)
- Syntactic Analysis (parsing)
- Semantic Analysis
- Optimization
- Code Generation
- Link to libraries

Source Preprocessing

- In C and C++, preprocessor statements begin with a #
- The preprocessor edits the source code based on the preprocessor statements
- **`#include`** is the same as copying the included file at that point with the editor
- The output of the preprocessor is expanded source code with no # statements
- Old C compilers had a separate preprocessor program

Lexical Analysis

- Lexical Analysis or scanning reads the source code (or expanded source code)
- It removes all comments and white space
- The output of the scanner is a stream of tokens
- Tokens can be words, symbols or character strings
- A scanner can be a finite state automata

Syntactic Analysis

- Syntactic Analysis or parsing reads the stream of tokens created by the scanner
- It checks that the language syntax is correct
- The output of the Syntactic Analyzer is a parse tree
- The parser can be implemented by a context free grammar stack machine

Semantic Analysis

- The Semantic Analysis inputs the parse tree from the parser
- Semantic Analysis checks that the operations are valid for the given operands *(i.e. cannot divide a String)*
- This stage determines what the program is to do
- The output of the Semantic Analysis is an intermediate code. This is similar to assembler language, but may include higher level operations

The Java import statement applies to which layer?

A. Source preprocessing
B. Lexical Analysis (scanning)
C. Syntactic Analysis (parsing)
D. Semantic Analysis
E. Code Generation

Optimization

- Most compilers will attempt to optimize the intermediate code
- Some compilers will also optimize after code generation
- There are many optimizations possible such as moving computations out of loops, avoiding redundant loads and stores, efficient use of registers, etc.

Code Generation

- The Code Generator inputs the intermediate language and outputs machine language for the target machine
- The code generator is specific to the machine architecture

Linking and Loading

- While not truly part of the compiler, the libraries provide the functionality that is more than just a few machine language statements
- The linker reads the object files and outputs and executable file

Simple Program

```
/*  This is an example program */
A = Boy + Cat + Dog;
```

After Lexical Scan

A

=

Boy

+

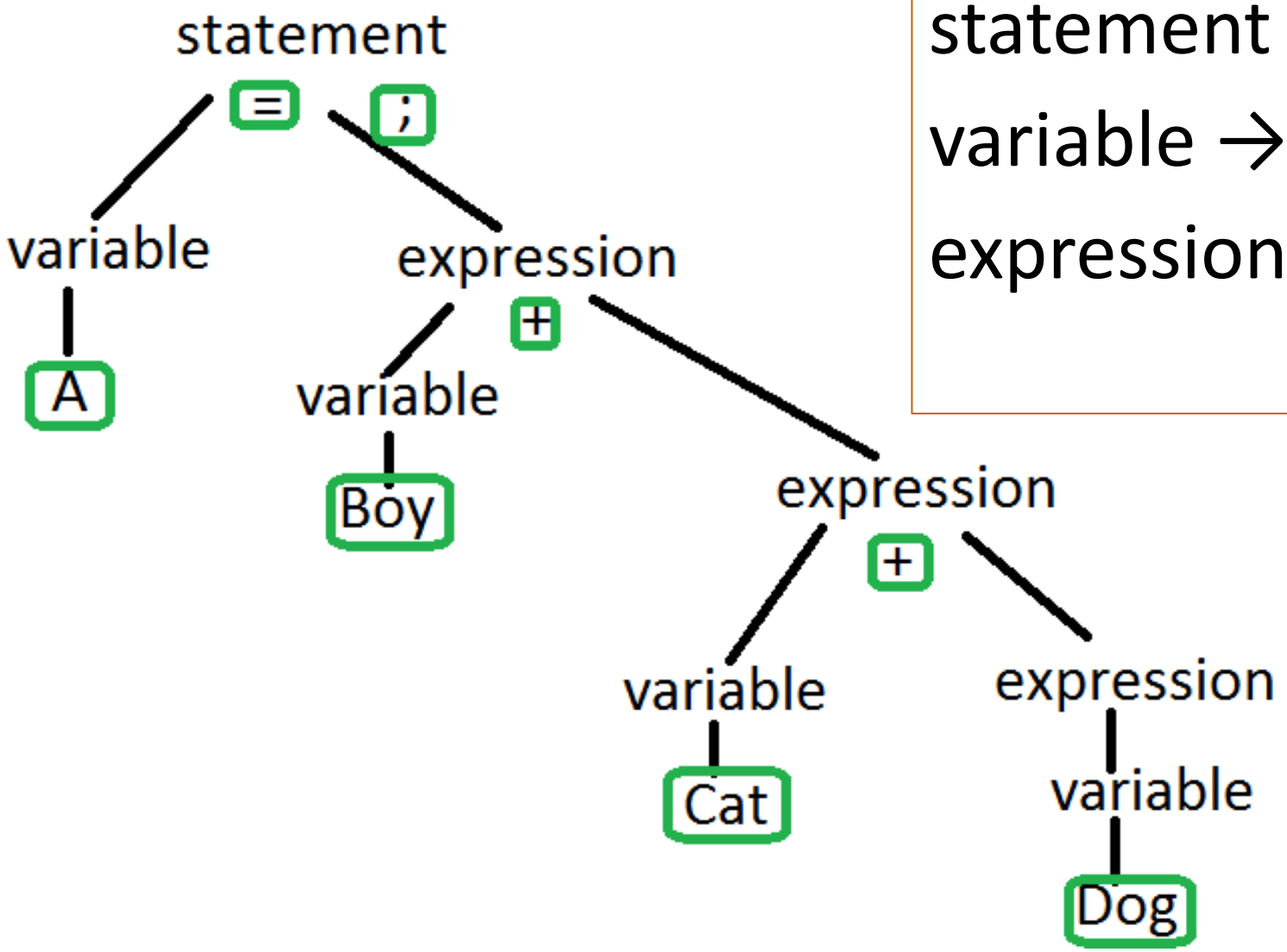
Cat

+

Dog

;

Parsing



statement → variable **=** expression **;**

variable → *string of characters*

expression → variable **+** expression

| variable

Semantic Analysis

- Semantic analysis will check
- Intermediate code defines a series of simple steps that will execute the program

Operation	First Operand	Second Operand	Destination
add	Boy	Cat	temp1
add	temp1	Dog	temp2
copy	temp2		A

Simple Machine Language

- Load register with B
- Add C to register
- Store register in Temp1
- Load register with Temp1
- Add D to register
- Store register in Temp2
- Load register with Temp2
- Store register in A

Optimized Machine Language

- Load register with B
- Add C to register
- Store register in Temp1
- Load register with Temp1
- Add D to register
- Store register in Temp2
- Load register with Temp2
- Store register in A

Symbol Table

- Many stages of a compiler create and reference a symbol table
- The symbol table keeps a list of all of the names used in the program along with information about the name
- To assist debugging, the symbol table can be written into the output object file. This tells debuggers where variables are located

Implementing a FSA

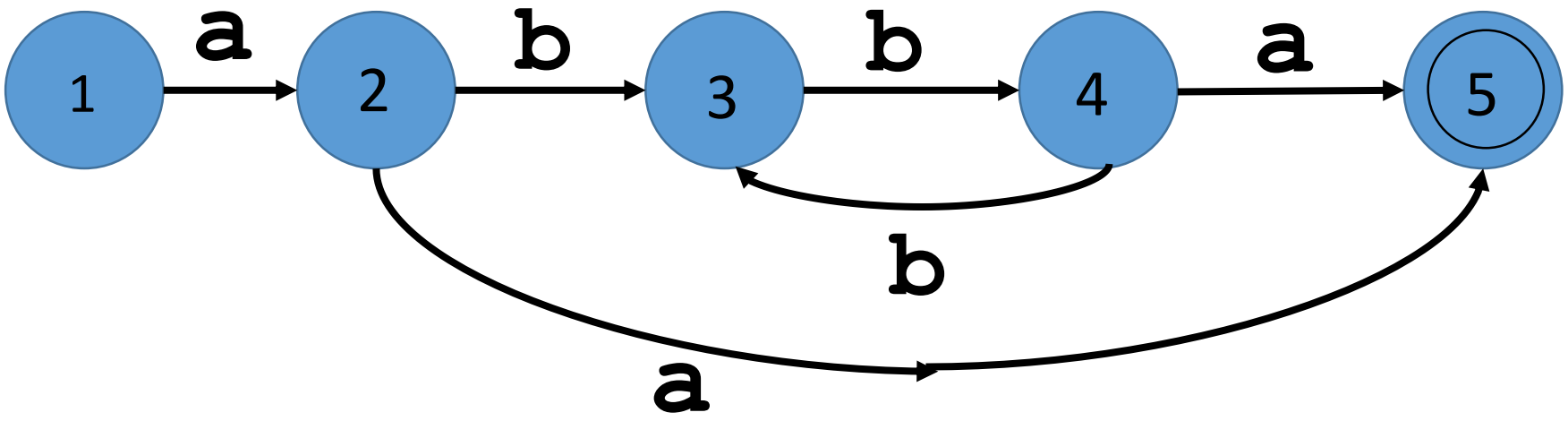
- A FSA graph can be converted to a table
- The table has cells for each state and each input symbol
- In the cell goes the next state if the DFA is in that state and receives that input symbol
- You can consider the state table to be an adjacency table for the graph

Converting an Example FSA

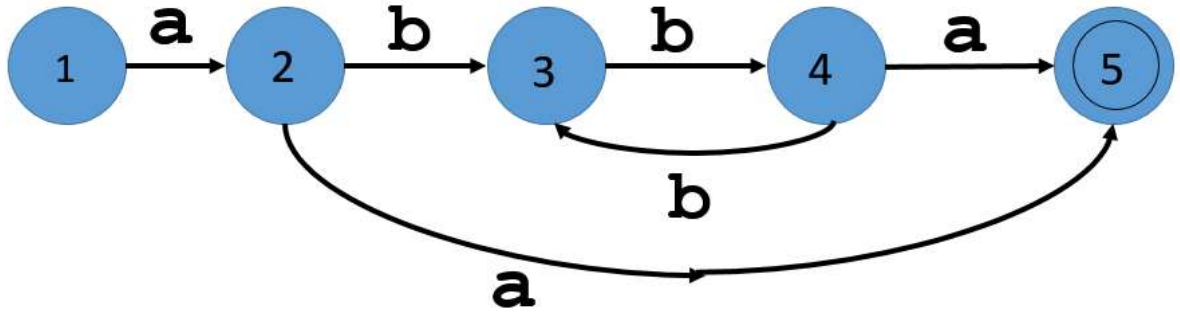
- Consider the regular expression that begins and ends with an **a** and can have an even number of **b**'s between them

a (bb)* a

- It can be recognized by the FSA



Convert the Graph to a Table



	1	2	3	4	5
a	2	5	0	5	0
b	0	3	4	3	0

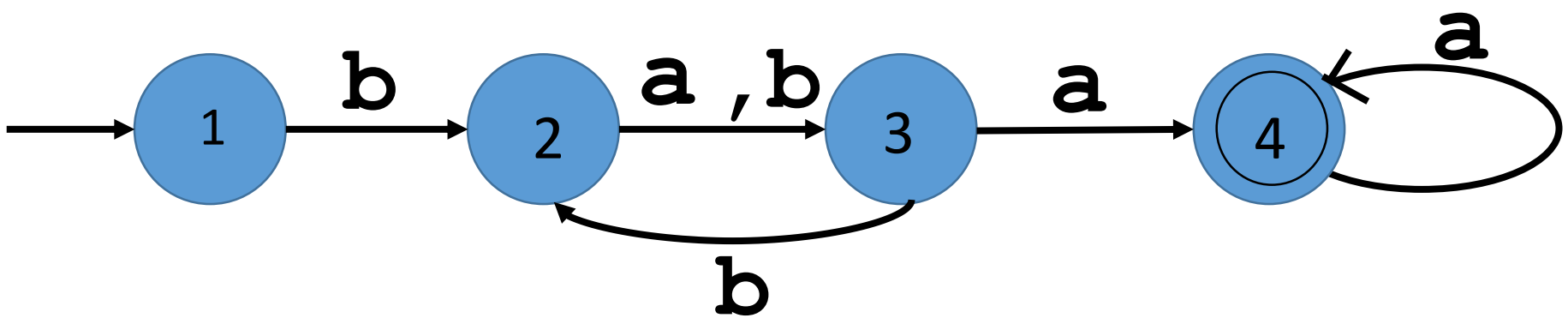
- The states are listed along the top
- The input symbols are along the side
- For that symbol while in that state, the DFA will go to the new state given in the table
- State zero represents a final error state

Draw a DFA for this Regular Language

$$(bb \mid ba)\ a^{+}$$

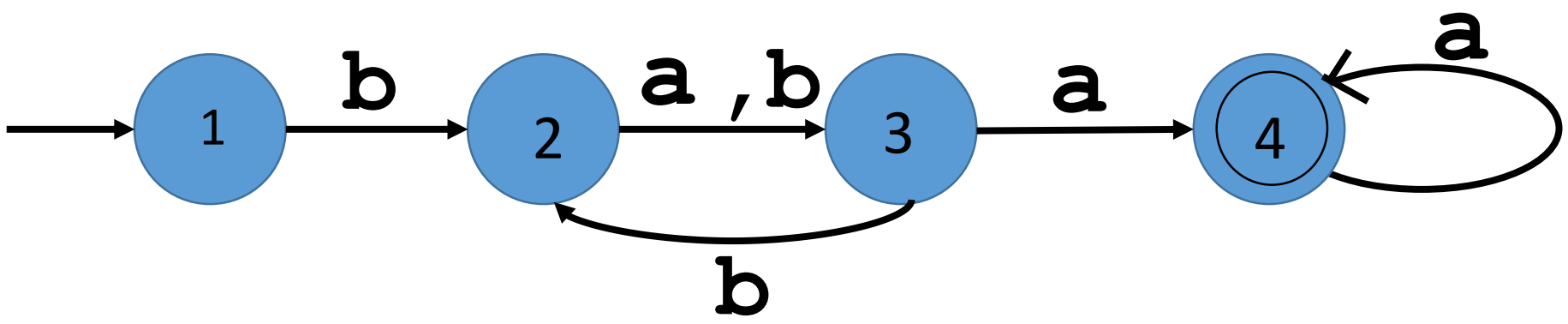
Possible Solution

$$(bb \mid ba)\ a^{+}$$

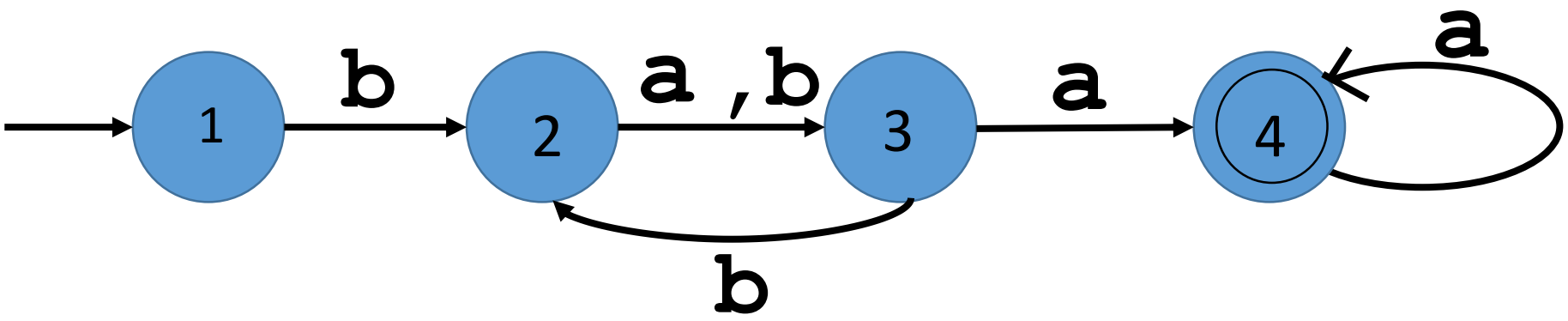


Create a state table for the FSA

$$(bb \mid ba)\ a^{+}$$

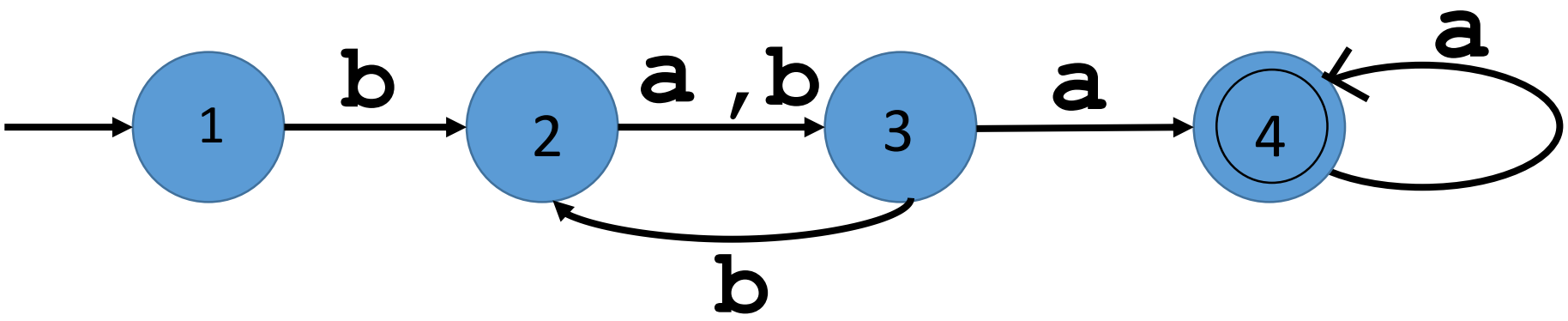


Complete the State Table



	1	2	3	4
a				
b				

Possible Solution



	1	2	3	4
a	0	3	4	4
b	2	3	2	0

Programming a FSA

- It is relatively simple to implement a Finite State Automata in a modern programming language
- This program can be used to recognize if a string conforms to a regular language

FSA Program

```
state = 1
while not end of file {
        symbol = next input character
        state = stateTable[ symbol, state ]
        if state = 0 then error
}
if state is a terminating state, success
```

Grouping Input Symbols

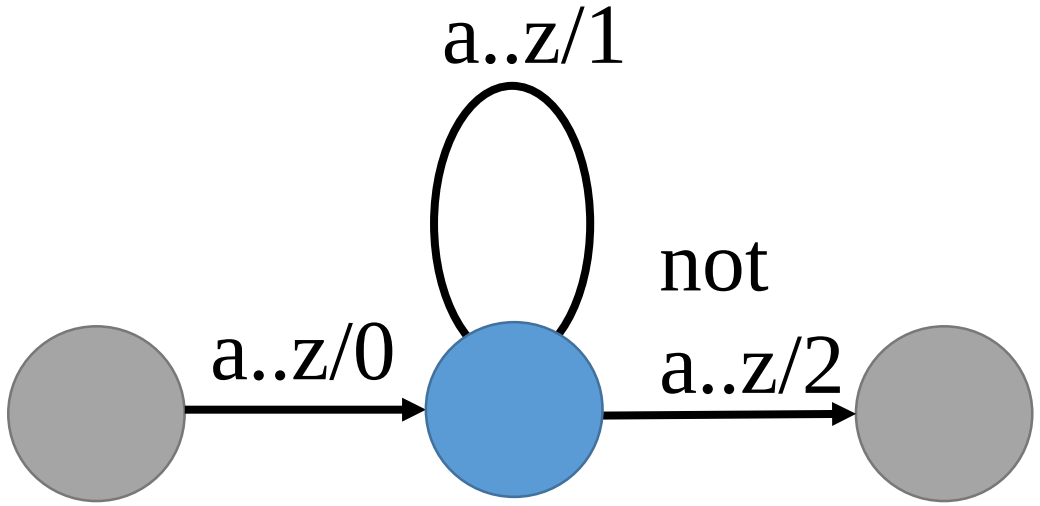
- For many FSA programs, it is useful to create an index value for the input symbols, i.e. a = 0, b = 1
- Often you can have groups of symbols use the same index value, i.e. all letters have the index 2 and all numbers have the index 3

Mealy and Moore Machines

- The FSA we have discussed so far simply determine if the input is valid for the specified language
- An FSA can also produce an output
- A Mealy machine has an output or function associated with each transition or edge of the graph
- A Moore machine has an output or function associated with each state or node of the graph

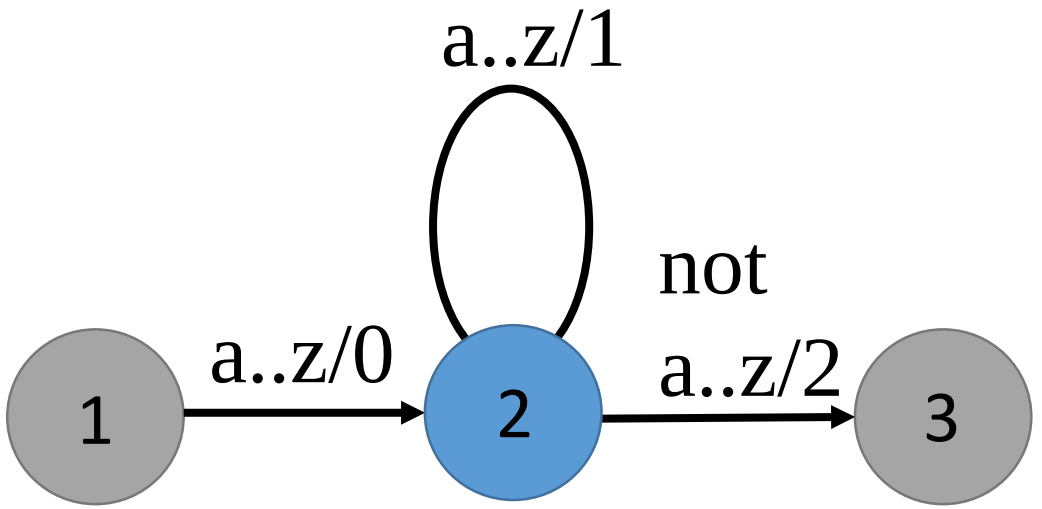
Using Mealy Machines to Create Tokens

- Consider an FSA reading text and creating token of words separated by spaces or punctuation



- 0 = Save character as first letter in a string
- 1 = Save character as next letter in a string
- 2 = Save the string as a token, handle other symbol

Mealy Machine State Table



	1	2	3
a .. z	2/0	2/1	2/0
not a .. z	1/x	3/2	3/x

New state / Output function

Lexical Analysis with a Mealy Machine

- Compilers can use a Mealy machine to scan the source code
- The FSA recognizes and discards comments and white space
- Names, numbers, strings and punctuation are each output as a list of tokens
- A token is an object that contains one unit of the input specifying the value and type

Regular Expression and FSA Assignment

- A new assignment has been posted on Blackboard
- It is due by 2:00pm on Monday, February 6, 2017
- There are nine questions requiring you to write regular expressions and FSA

Reading

Read sections 2.1 – 3.2 in the textbook