# Compiling Regular Expressions

COMP360

*“Logic is the beginning of wisdom, not the end.”*

Leonard Nimoy

# Compiler's Purpose

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

# Output of a Compiler

- Most language systems compile the source into an object file of machine language which is linked into an executable
- Some languages, like Java and C#, are compiled to an intermediate language which is interpreted
- A compiler can output a high level language
- Some systems interpret and execute the source code directly

# Run Time Compilers

- Java and C# compilers create an intermediate language that can be interpreted by a virtual machine
- Most virtual machines contain a “Just In Time” (JIT) compiler that compiles the intermediate language into machine language for efficiency

# 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 (FSA)

# 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
- Language requirements not checked by the syntax are enforced
- 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

# 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

# At which stage will these errors be detected?

```
String  num2go;            // A
int  dog  cat;             // B
dog = myFunc( dog );       // C

int myFunc( int cat, int cow) { … }
```

# Simple Program

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

# After Lexical Scan

A

=

Boy

+

Cat

+

Dog

;

# Parsing

<statement> -> <variable> = <expression>

<variable> -> A | Boy | Cat | Dog

<expression> -> <variable> + <expression>
| <variable>

# Compile A = B + C + D

- Intermediate code

```
Temp1 = B + C
Temp2 = Temp1 + D
A = Temp2
```

# 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
- To assist debugging, the symbol table can be written into the output object file. This tells debuggers where variables are located
- The symbol table can be created by the scanner and updated by all other stages

# Output of Each Stage

- Source preprocessing – expanded source code
- Lexical Analysis – List of tokens
- Syntactic Analysis – Parse Tree
- Semantic Analysis – Intermediate code
- Optimization – Intermediate code
- Code Generation – Object file
- Link to libraries – Executable program

# Machines of a Compiler

- Source preprocessing – simple editing
- Lexical Analysis – Finite State Automata
- Syntactic Analysis – Push Down Automata
- Semantic Analysis
- Optimization
- Code Generation
- Link to libraries

# State Tables

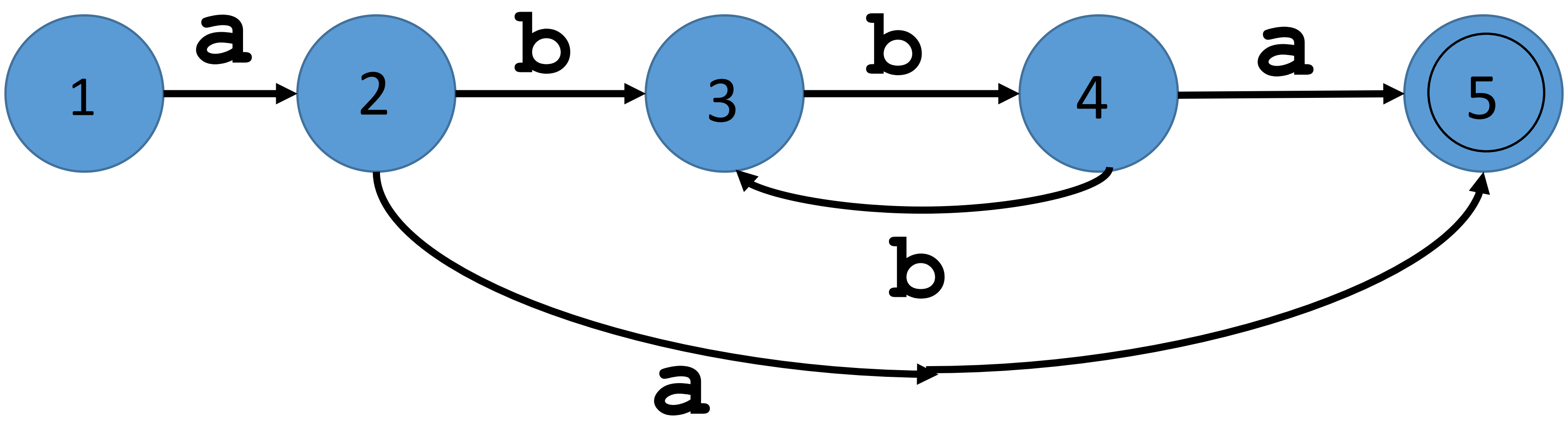
- An FSA graph can be converted to a table
- The table has cells for each state and each input symbol
- In the cell goes the next state if the DFA is in that state and receives that input symbol
- You can consider the state table to be an adjacency table for the graph

# 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 |

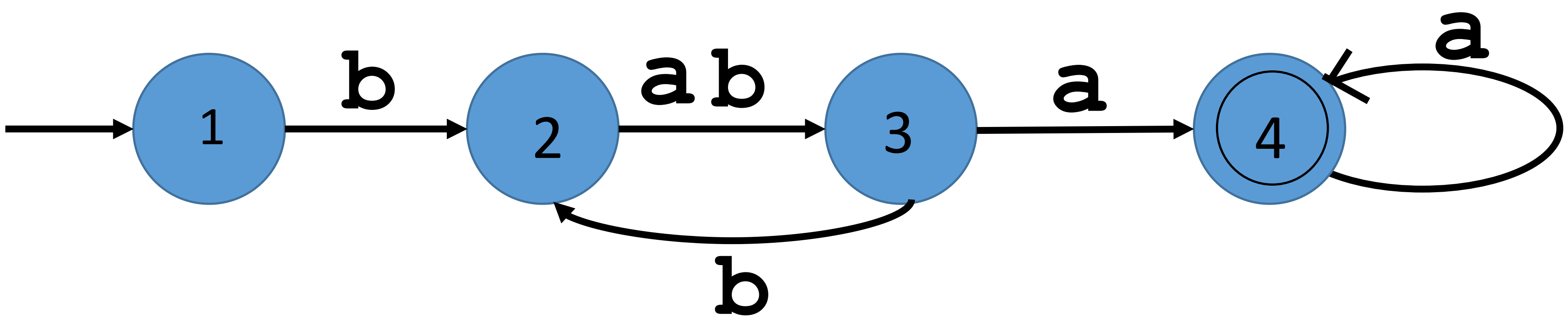
- The states are listed along the top
- The input symbols are along the side
- For that symbol while in that state, the DFA will go to the new state given in the table
- State zero represents a final error state

# Draw a DFA for this Regular Language

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

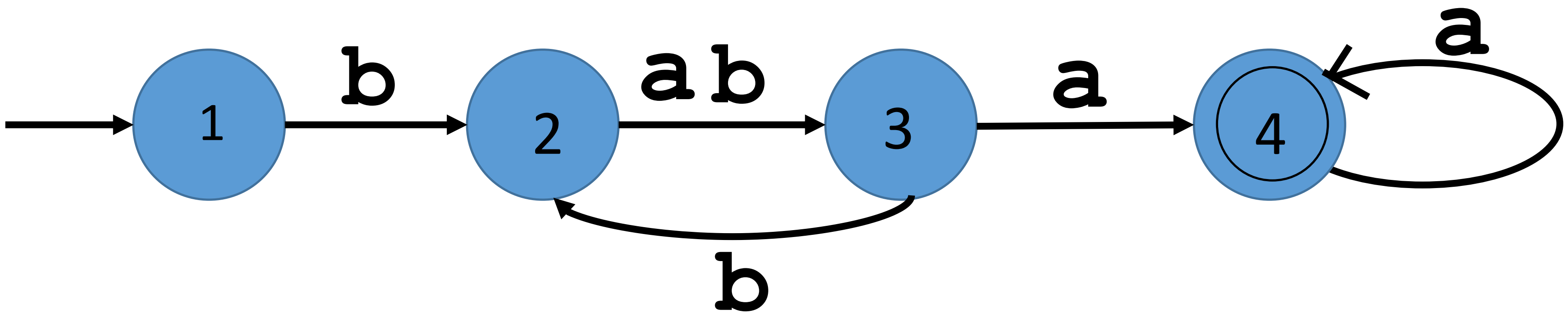
# Possible Solution

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

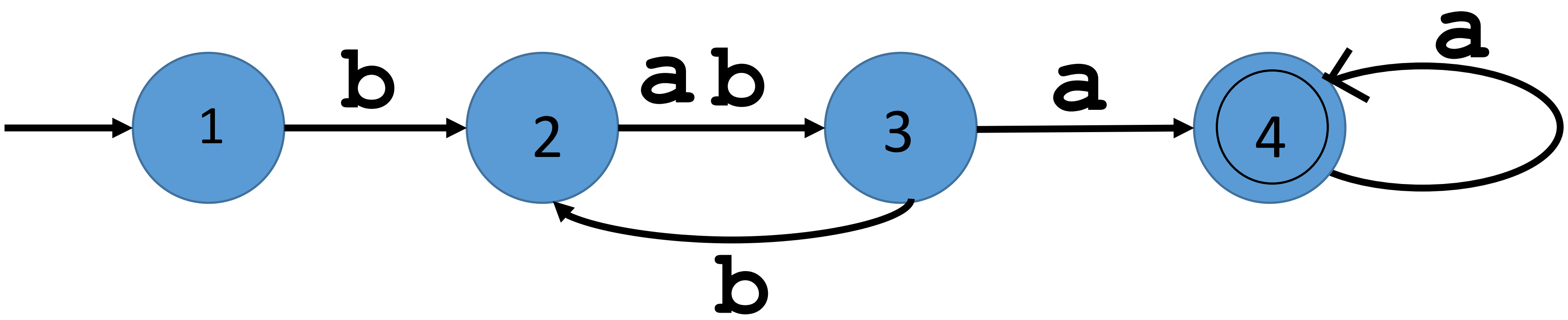


# Possible Solution

**(bb | ba) $a^+$**



# Create a state table for the FSA




| | 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

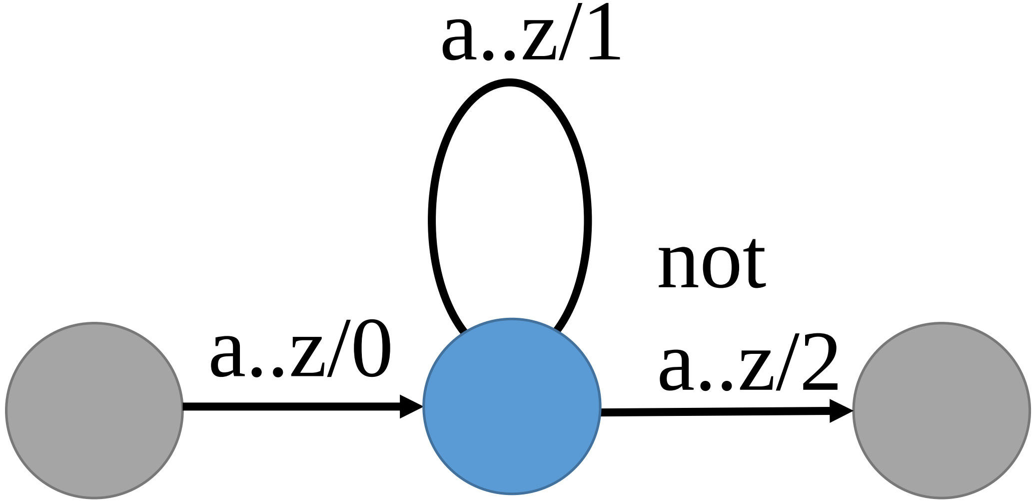
- For many FSA programs, it is useful to create an index value for the input symbols, i.e. a = 0, b = 1
- Often you can have groups of symbols use the same index value, i.e. all letters have the index 2 and all numbers have the index 3

# 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

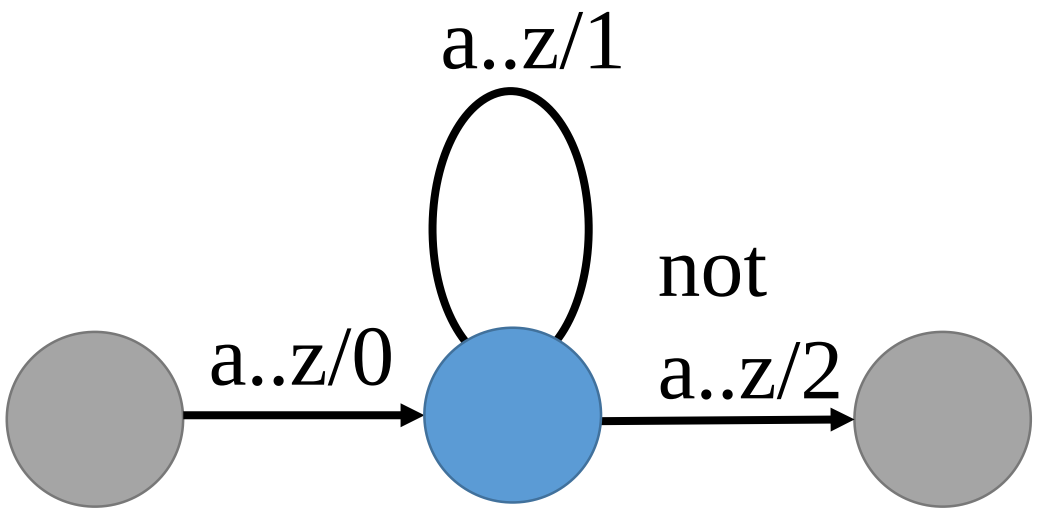
# Using Mealy Machines to Create Tokens

- 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

# Reading

- Read sections 3.1 – 3.3 in the textbook by Wednesday