Regular Expressions & Finite State Machines

Main ideas

Regular expressions / grammars can be expressed with a **finite state machine** (FSM)

- Also called **finite automata** (FA)
- Used to describe and recognize tokens
- Can be deterministic (DFA) or non-deterministic (NFA)

Two related challenges:

- Recognizing the longest substring corresponding to a token
- Separating a lexeme from the rest of the input string

Finite state machine (FSM)

Finite state machine (FSM), also called finite automata (FA), is a state machine that takes a string of symbols as input and changes its state accordingly. It consists of:

- Q Finite set of states
- Σ Alphabet: a finite set of input symbols
- Q_0 An initial start state, $Q_0 \in Q$
- Q_f Set of final states, $Q_f \subseteq Q$
- λ Transition function that describes how to move from one state to another. Defined as: $s \in Q$ and $a \in \Sigma$ implies $\lambda(s, a) = t$ for some $t \in Q$

When a string is fed into the FA, it changes its state for each literal.

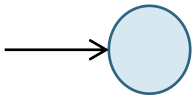
- If the input string is successfully processed and the FA reach its final state, it is ***accepted*** (i.e., the input string is a valid token of the language)
- Languages recognized by FA are the languages described by REs.

FSM represented as a digraph

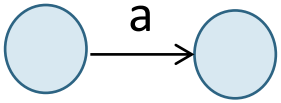
- Each node represents a state; edges represent transitions
- Transitions are labeled with a symbol from the alphabet Σ or the empty string ϵ
- Of all states Q, there is a start state and at least one final (accepting) state
- The language recognized by finite state machine M is denoted $L(M) = \{w \in \Sigma^* \mid (S, w) \rightarrow^* (Y, \epsilon)\}, \text{where Y} \in F$

Example FSM

How FSMs are drawn



Start state

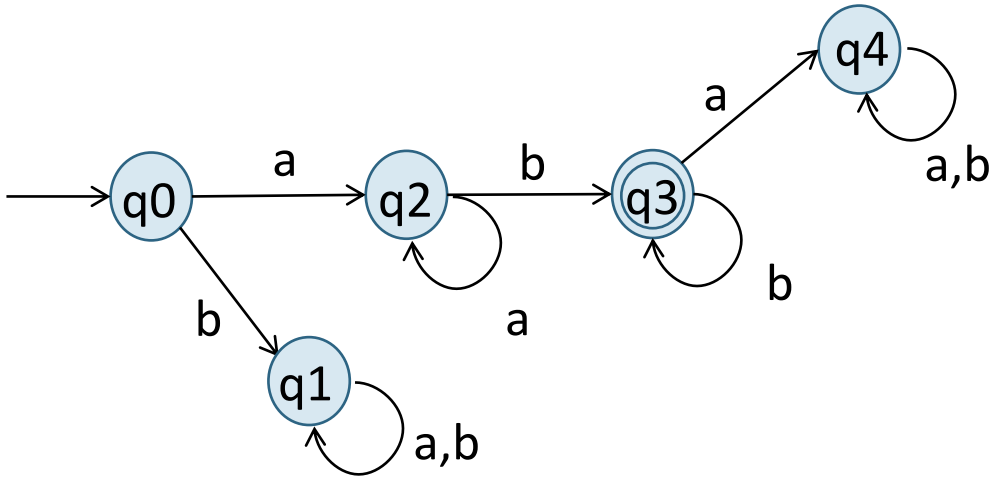


Can only transition from first to next state through the edge if next character read is **a**

Final state

A string is accepted if it can be read from the start state, transition through states, and end at a final state.

Otherwise, it is rejected.



Accepts the strings:

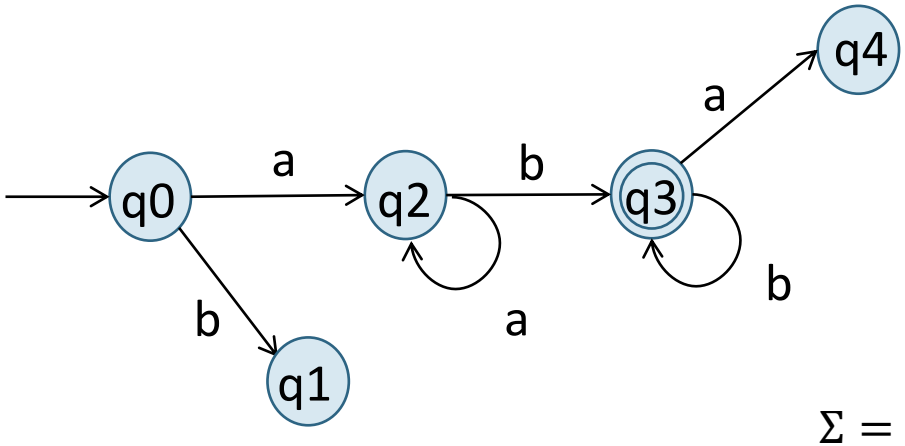
- ab
- aabb
- abbb
- ….

What language does this recognize?

a+b+

Represented as state-transition table

State machine as digraph



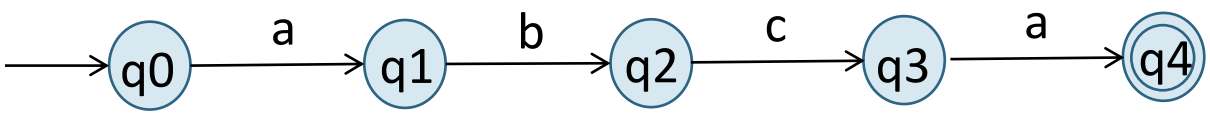
$\Sigma = \{a, b\}$

Can also be represented as a state transition table

	Input	
State	a	b
0	2	1
1	∅	∅
2	2	3
3	4	3
4	∅	∅

Note: Transitions not shown immediately go a null 'reject' state (omitting them is less cluttered and easier to read)

Example with $\Sigma = \{a, b, c\}$



	Input		
State	a	b	c
0	1	Ø	Ø
1	Ø	2	Ø
2	Ø	Ø	3
3	4	Ø	Ø
4	Ø	Ø	Ø

Accepted or rejected?

- Input string: abca
- Input string: ccba
- Input string: abcac

Determinism

A finite automata is deterministic (DFA) or non-deterministic (NFA).

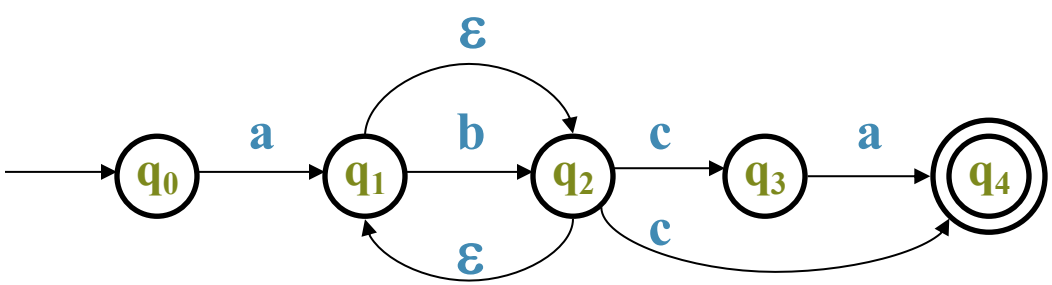
- It is **deterministic** if its behavior during recognition is fully determined by the state it is in and the symbol to be consumed
  - Given an input string, **only one path** may be taken through the FA

- It is **non-deterministic** if, given an input string, more than one path may be taken.
  - One type is ϵ-transitions, which consume the empty string ϵ (no symbols)

Theorem. Any DFA can be expressed as an NFA. Moreover, any NFA can be expressed as a DFA!

Example NFA

$\Sigma = \{ a, b, c \}$



State	Input			
	a	b	c	ε
0	1	∅	∅	∅
1	∅	2	∅	2
2	∅	∅	3,4	1
3	4	∅	∅	∅
4	∅	∅	∅	∅

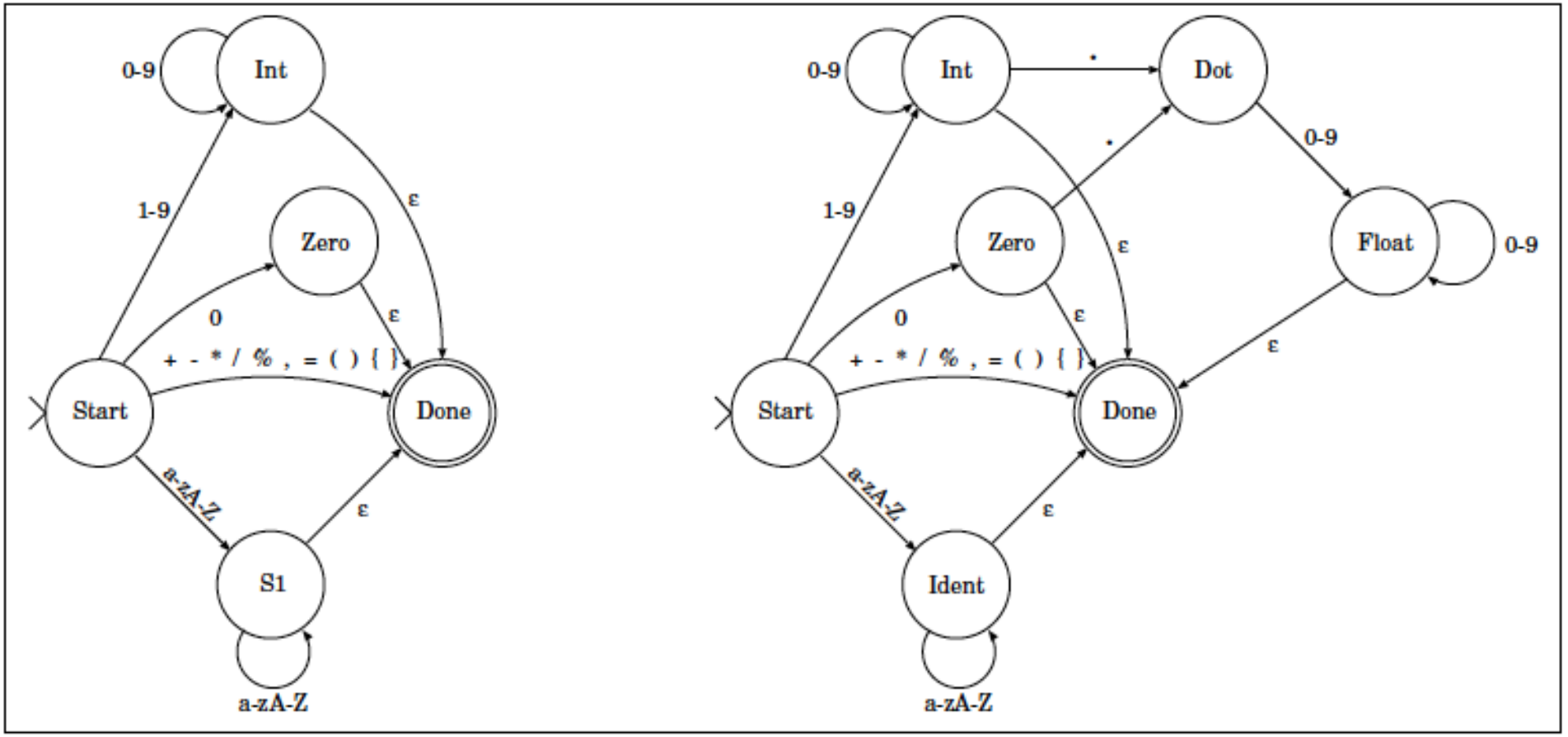
Exercise: This NFA is equivalent to what regular expression?

PDef: Parenthesized Definitions

```
{ float a, a = 3, { int b, b = 4, a = b*a }, a = a+4.0 }
```

Token Class	Regular Expression	Termination Characters
addT	+	any character
subT	-	"
mulT	*	"
divT	/	"
modT	%	"
commaT	,	"
assignT	=	"
lpT	(	"
rpT	)	"
lcbT	{	"
rcbT	}	"
typeT	int | float	non-letter
intT	$0 \mid [1-9][0-9]^*$	non-digit
fltT	$(0 \mid [1-9][0-9]^*) \,.\, [0-9]^+$	non-digit
identT	$[a-zA-Z]^+$	non-letter

FSM for PDef


0-9
Int
1-9
Zero
ε
0
+ - * / % , = ( ) { }
Start
Done
a-zA-Z
S1
ε
a-zA-Z
0-9
Int
.
Dot
.
0-9
1-9
Zero
ε
Float
0-9
0
ε
+ - * / % , = ( ) { }
ε
Start
Done
a-zA-Z
Ident
ε
a-zA-Z

Theory to Practice

- Need to represent the states, represent transitions between states, consume input, and restore input
- Create an enumerated type whose values represent the FSM states: Start, Int, Float, Zero, Done, Error, …
- Keep track of the current state and update based on the state transition

```
state = Start;
while (state != Done) {
   ch = input.getSymbol();
   switch (state) {
        case Start: // select next state based on current input symbol
        case S1:    // select next state based on current input symbol
        ..
        case Sn:    // select next state based on current input symbol
        case Done:  // should never hit this case!
   }
}
```

```
while (state != StateName.DONE_S) {
    char ch = getChar();
    switch (state) {
        case START_S:
             if (ch == ' ') {
                  state = StateName.START_S;
             }
             else if (ch == eofChar) {
                  type  = Token.TokenType.EOF_T;
                  state = StateName.DONE_S;
             }
             else if ( Character.isLetter(ch) ) {
                  name += ch;
                  state = StateName.IDENT_S;
             }
             else if ( Character.isDigit(ch) ) {
                  name += ch;
                  if (ch == '0') state = StateName.ZERO_S;
                  else state = StateName.INT_S;
             }
             else if (ch == '.') {
                  name += ch;
                  state = StateName.ERROR_S;
             }
             else {
                  name += ch;
                  type = char2Token( ch );
                  state = StateName.DONE_S;
             }
              break;
```

FSM Practice

Join your team to work through the exercises
Each individual will submit docx file to Moodle

@mention me if questions on practice or environment setup