ENGR110 T2 2017

Engineering Modeling and Design.
Introduction to models of computation - FSM

Arthur Roberts

School of Engineering and Computer Science
Victoria University of Wellington

Intro to the Models of Computation: Finite State Machines

This part of the course is NOT about clever coding. On the contrary, we ask you to keep coding part as simple/stupid as possible(KISS).

But your code should be STRUCTURED. It is OK if it is big.

Do not jump straight into coding, think and draw your program flow first.

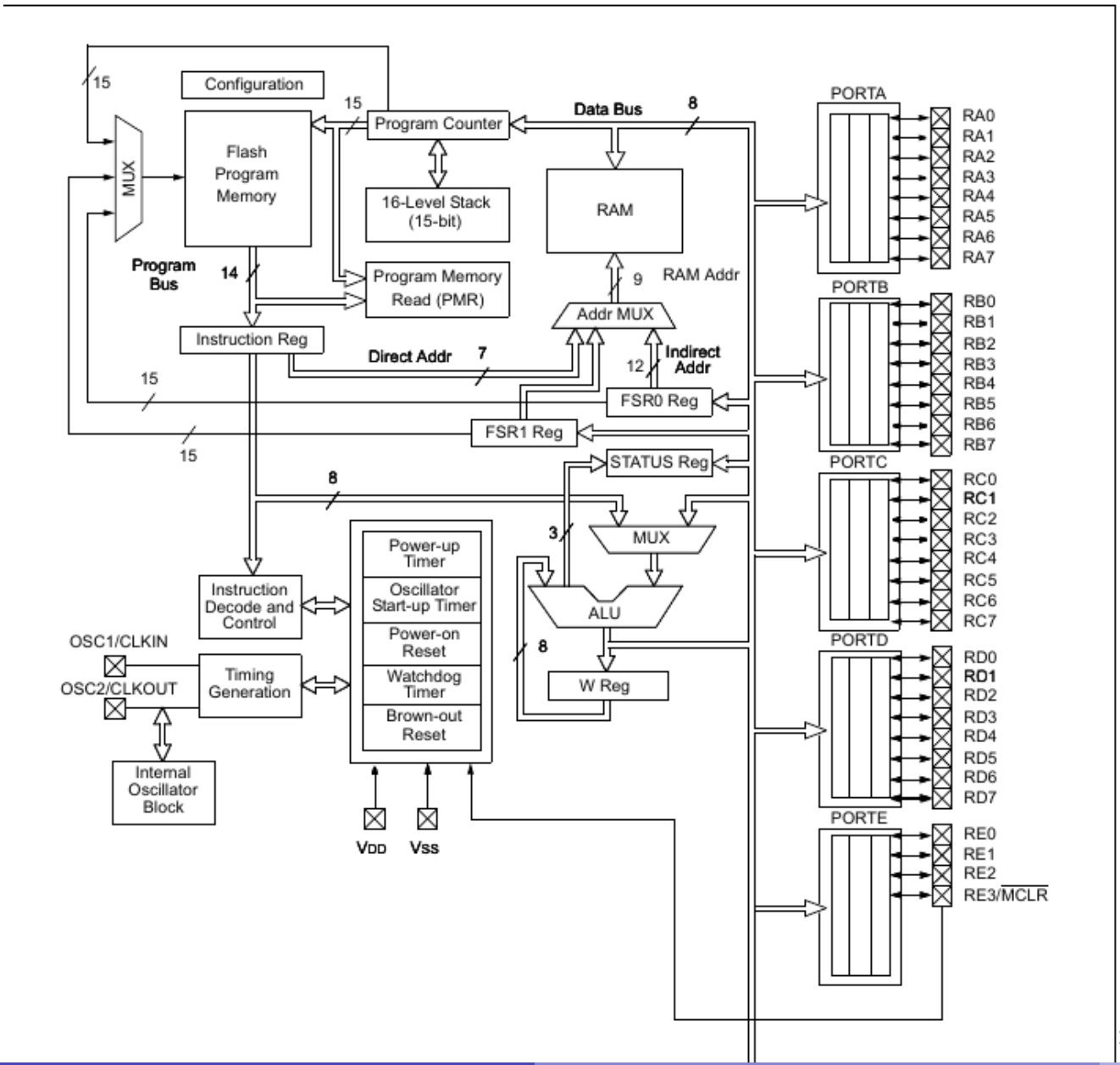
EVERYTHING SHOULD BE MADE AS SIMPLE AS POSSIBLE, BUT NOT SIMPLER—ALBERT EINSTEIN

Workflow

To make the mess requires no effort. It happens by default.
To this things simple is not easy.

1. Understand the task
2. Turn away from computer and design your algorithm on paper. Should take 70-80% percent of your time. **Hard part.**
3. Type in the code. There are strict/simple rules for converting design to code.
4. You can have typos, but if 2 is done properly you will have no bugs.

Why KISS? Computers are BIG



Block-diagram of PIC16F1937 processor produced by **Microchip**.
Each of these rectangles contains thousands of logic gates.
And it is not even remotely as complicated as CPU.

Why KISS? Programs are messy

```
var drawLine =
d3.behavior.drag()
.on('dragstart',function(d){
        d3.event.sourceEvent.stopPropagation();
        force.stop();
        circle = {'x':parseInt(this.getAttribute('cx')),
                  'y':parseInt(this.getAttribute('cy'))};
        parent = {'x': d3.select('#' + d.parent.id)[0][0].__data__.x,
                  'y': d3.select('#' + d.parent.id)[0][0].__data__.y};
        circleAbsPosition = {'x' : circle.x + parent.x,
                  'y' : circle.y + parent.y};
        allLinks.append('line')
        .attr('class','draw')
        .attr('x1',circleAbsPosition.x)
        .attr('y1',circleAbsPosition.y)
        .attr('x2',circleAbsPosition.x)
        .attr('y2',circleAbsPosition.y)
        .attr('stroke', linkColor)
        .attr('stroke-width', linkWidth)
        .style('marker-start', 'url(#start)')
        .style('marker-end', 'url(#end)');
})
.on('drag',function(d){
        lastDragPosition = {'x':d3.event.x + d3.select('#' + d.parent.id)[0][0].__data__.x,
                  'y':d3.event.y + d3.select('#' + d.parent.id)[0][0].__data__.y};
        d3.select('.draw')
        .attr('x2', lastDragPosition.x)
        .attr('y2', lastDragPosition.y);
})
.on('dragend',function(d){
        d3.select('.draw').remove();
        d3.selectAll('.nodeGroup').selectAll('.input,.output').each(
        function(){
                c =
```

Model of the computer

If something is too complex to keep it in mind when working with it it cam be good idea to create an **abstraction** of it.
We do not mean "abstract" as something not relevant and hard to grasp. On the contrary, abstraction should be simpler and easier to work with.
We ignore the details and concentrate on big picture.
Forget for a while about **int i = 0**, **for(i = -20)** and likes. That stuff can be easily (hm...), automated (that will be your project).
Simplest model of computation is **Finite State Machine/Automata**.

Finite State Machines

What are FSMs?

- A way of thinking about certain kinds of problems.
- A way of describing/analysing/modeling systems.
- A way of designing solutions to a wide range of engineering problems.

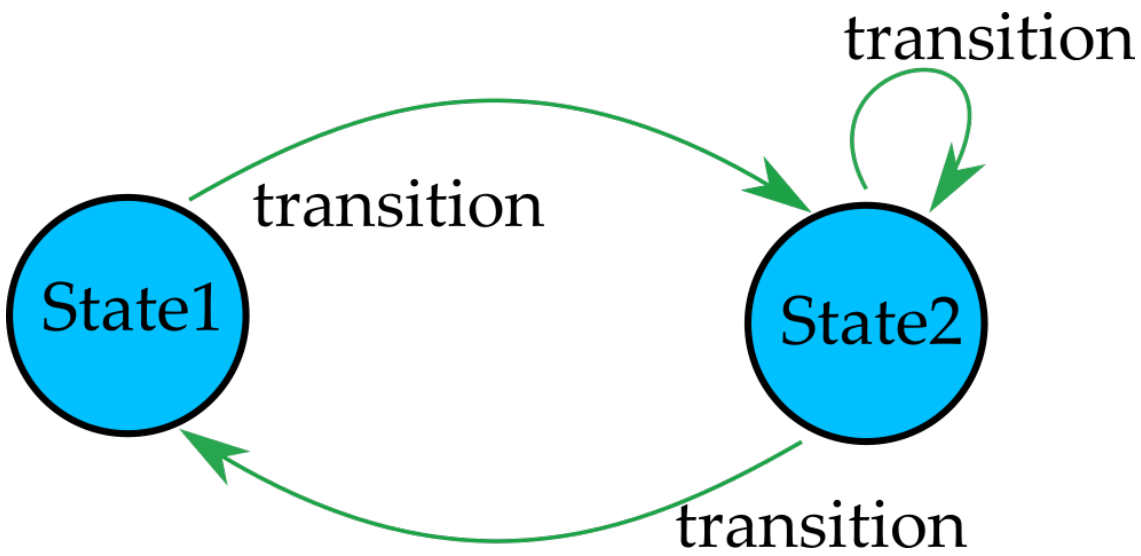
For example:

- Regular expression/natural language processing.
- Models or controllers of physical devices
- Games.
- Communication protocols.
- Analysing and designing UI.

States and Transitions

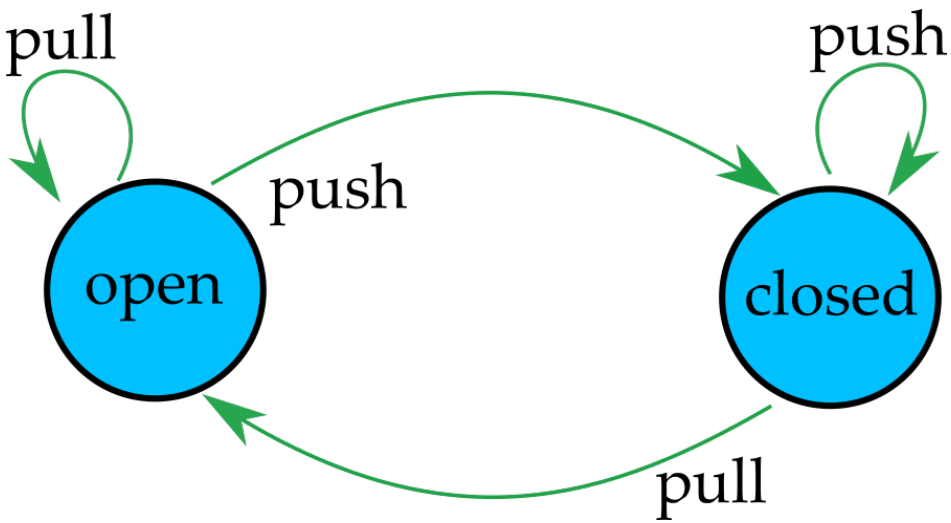
If a system has a finite number of states then you can list them and identify each of the possible things that could happen in each state

- Labelled circles for states.
- Labelled arrows for transitions.



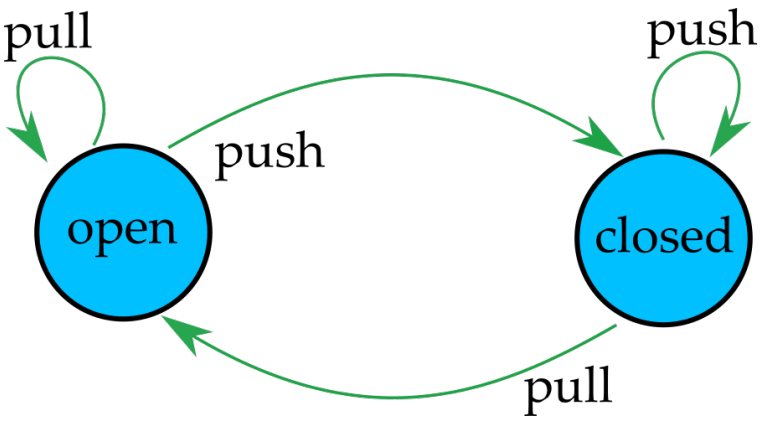
FSM example: a door

- states: open, closed.
- transitions: pull, push.



This representation is called **transition graph**.

What is the code?




```
string state="open";
string input;
\\read input
 if (state == "open"){
 //in C++ you can do
 //strings comparison
    if (input=="push"){
      state = "closed";
   }
    if (input=="pull"){
      state = "open";
   }
 }
 if (state == "closed"){
    if (input=="push"){
      state = "closed";
    }
    if (input=="pull"){
       state = "open";
    }
  }
```

Question: What is wrong here? Hint: follow it line by line...

What type should be **state** and **input**?

```
string state="open";
string input;
\\read input
if (state == "open"){
//in C++ you can do
//strings comparison
 if (input=="push"){
   state = "closed";
 }
 if (input=="pull"){
    state = "open";
   }
}
```

- Anything as long as you can use **equality** operator ==
- If using **if..else** state variables can be strings (which is more descriptive), can be **int** as well.
- There are different ways to program the FSM

Another option - **switch** operator

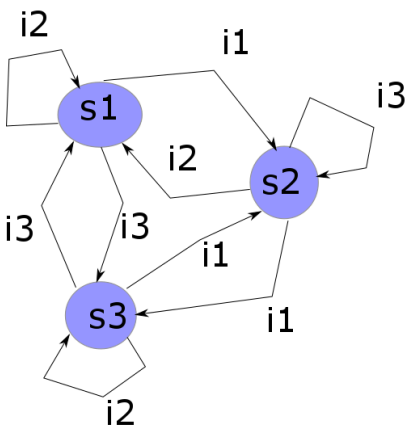
Case is equivalent to sequence of **if** statements.

```
#include <iostream>
using namespace std;
int main(){
 int a = 1;
 switch(a) {
  case 1: cout<<"1"<<endl;
  //break;
  case 2: cout<<"2"<<endl;
  break;
  case 3: cout<<"3"<<endl;
  break;
  default:
 }
}
```

a can be only integer.

- One of the **case** branches is taken, depending upon value of **a**. Branch sequence executes until **break**
- When break is encountered execution jumps out of switch operator.
- If value of a not found in **case**s - default branch is taken
- Watch out for the **break**s.

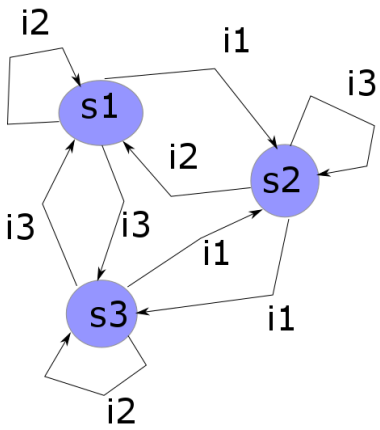
It is structured code




```
program:
  state ==s1:
    input ==i1: state=s2
    input==i2: state=s1
    input==i3: state=s3
  state == s2:
    input==i1: state=s3
    input==i2: state=s1
    input==i3: state=s2
  state ==s3:
    input==i1: state=s2
    input==i2: state=s3
    input==i3: state =s1
```

- Code follows well-defined **pattern**. If you got picture to the left correct - writing the code does not require any thinking.
- If nothing should happen in the particular state for some signal(for state1 input2 has no effect) - this parts of code can be taken out.

It runs forever...Or until input is available



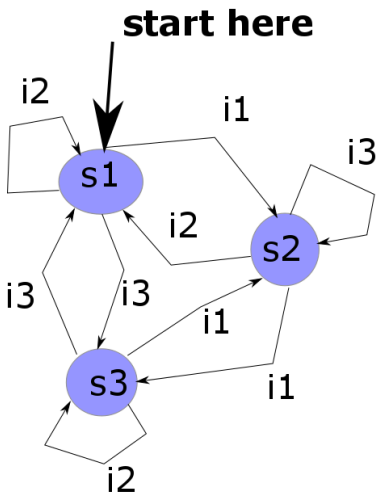

```
program:

while(input){
  //read input
  state ==s1:
    input ==i1: state=s2
    input==i3: state=s3
  state == s2:
    input==i1: state=s3
    input==i2: state=s1
  state ==s3:
    input==i1: state=s2
    input==i3: state =s1
}
```

FSM code runs forever or until there is no more input available. Note **while()** loop. Depends on application.

Figure: Shown is NOT C++ code. It is pseudo-code.

It should start somewhere - in some state

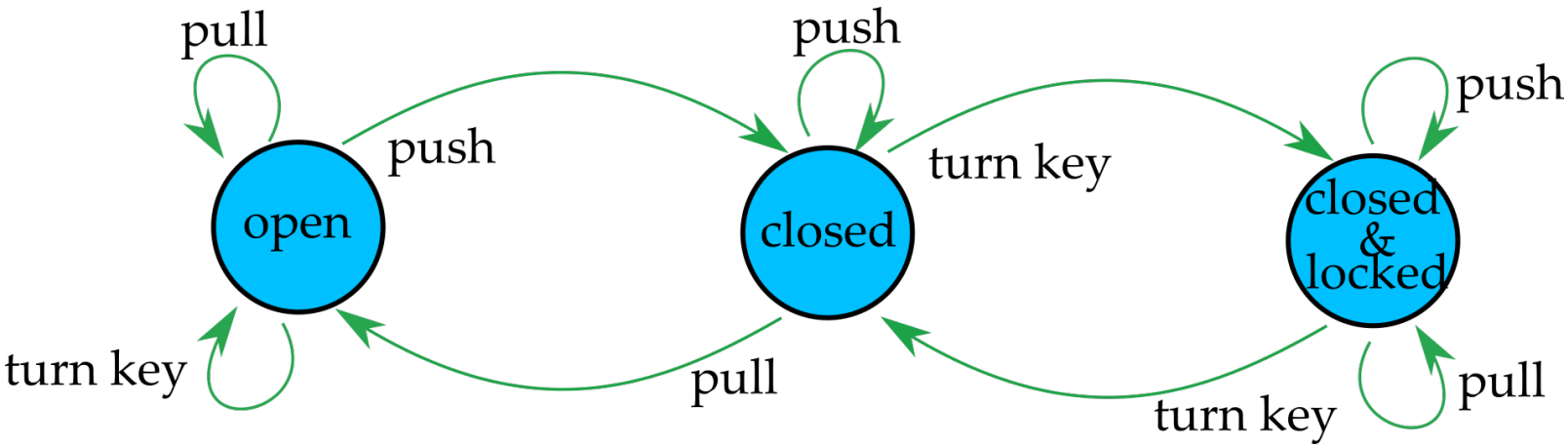



```
program:
state = s1
while(input){
  //read input
  state ==s1:
    input ==i1: state=s2
    input==i3: state=s3
  state == s2:
    input==i1: state=s3
    input==i2: state=s1
  state ==s3:
    input==i1: state=s2
    input==i3: state =s1
}
```

Before entering while loop value of variable **state** is set to **initial** state of the program. Indicated with **arrow coming from nowhere**.
Back to examples.

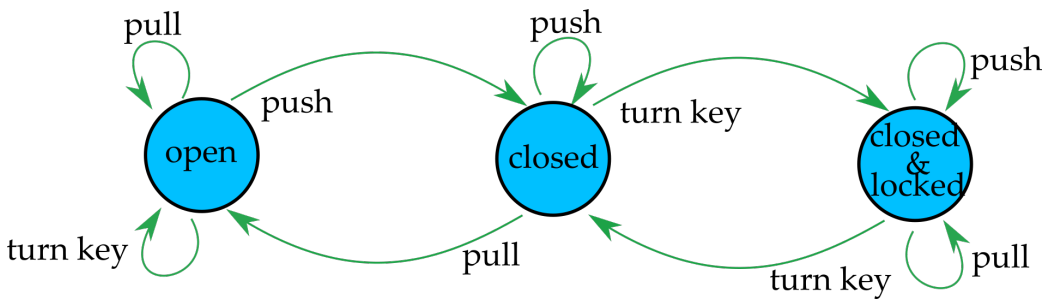
A more complex door

- states: open, closed, closed and locked.
- transitions: pull, push, turn key.



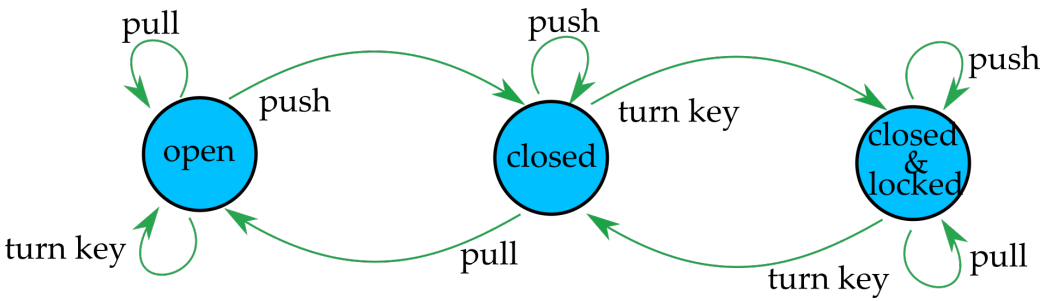
A more complex door in table form

Current State	Input	Next state
Open	pull	Open
Open	push	
Open	turn key	
Closed	pull	
Closed	push	
Closed	turn key	
Closed and Locked	pull	
Closed and Locked	push	
Closed and Locked	turn key	



A more complex door in table form

Current State	Input	Next state
Open	pull	Open
Open	push	Closed
Open	turn key	Open
Closed	pull	Open
Closed	push	Closed
Closed	turn key	Closed and Locked
Closed and Locked	pull	Closed and Locked
Closed and Locked	push	Closed and Locked
Closed and Locked	turn key	Closed

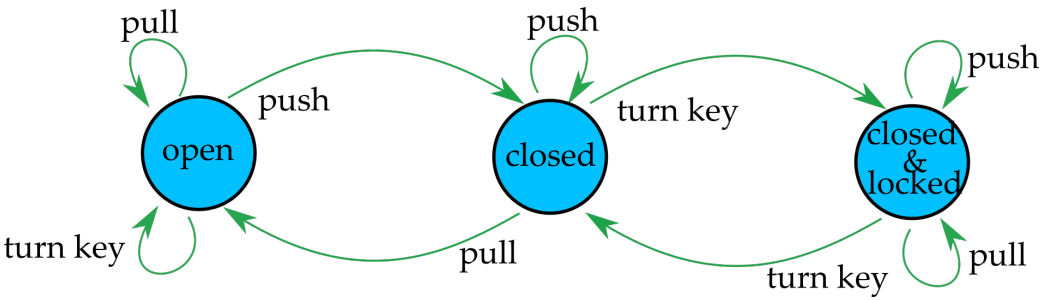


We can see the sequence here.

Another representation of the FSM: an array

Table contains next state of the FSM.
Column - current state. Row - input.

	open, [0]	closed[1]	closed_locked [2]
push[0]	open	closed	closed_locked
pull[1]	open	closed	closed_locked
turn key[2]	open	closed	closed_locked



Abstraction

Finite State Machines are **abstractions**

- Simplification.
- Captures significant aspect of behaviour.
- Omits unimportant details.
- Describes system at particular level.

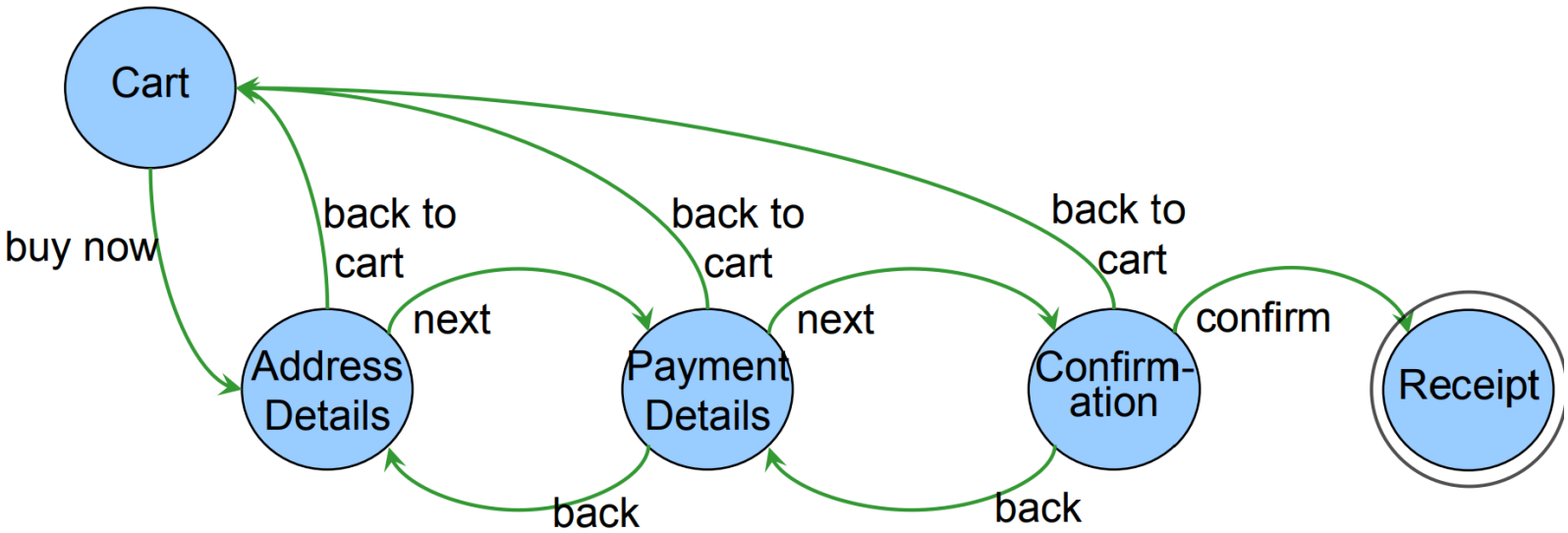
What did we omit about:

- the door? (how far open, how big, what kind of key...)

Another example

An Ecommerce website

- states: series of pages in purchase sequence.
- transitions: the button; buy now, next, back, confirm, back to cart.



Useful first model when designing webpages and navigation.

E-commerce again

If you are running this website what is important for you? User paid you.

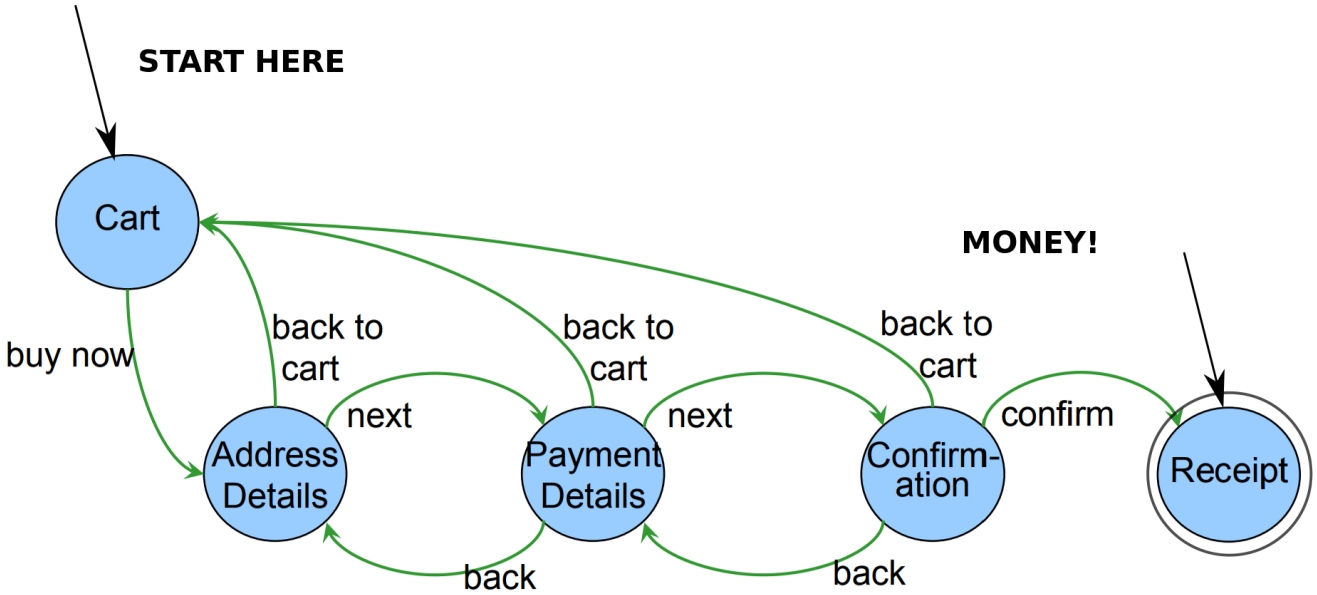
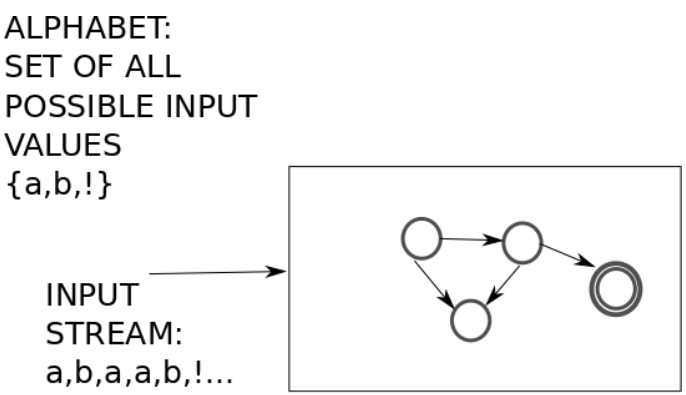


Figure:

One state is more important. It is called **accepting** state and is indicated with double circle.

Acceptors

Major application of FSM is to build **acceptors** - software(or hardware) which indicates that certain pattern happened in input data.

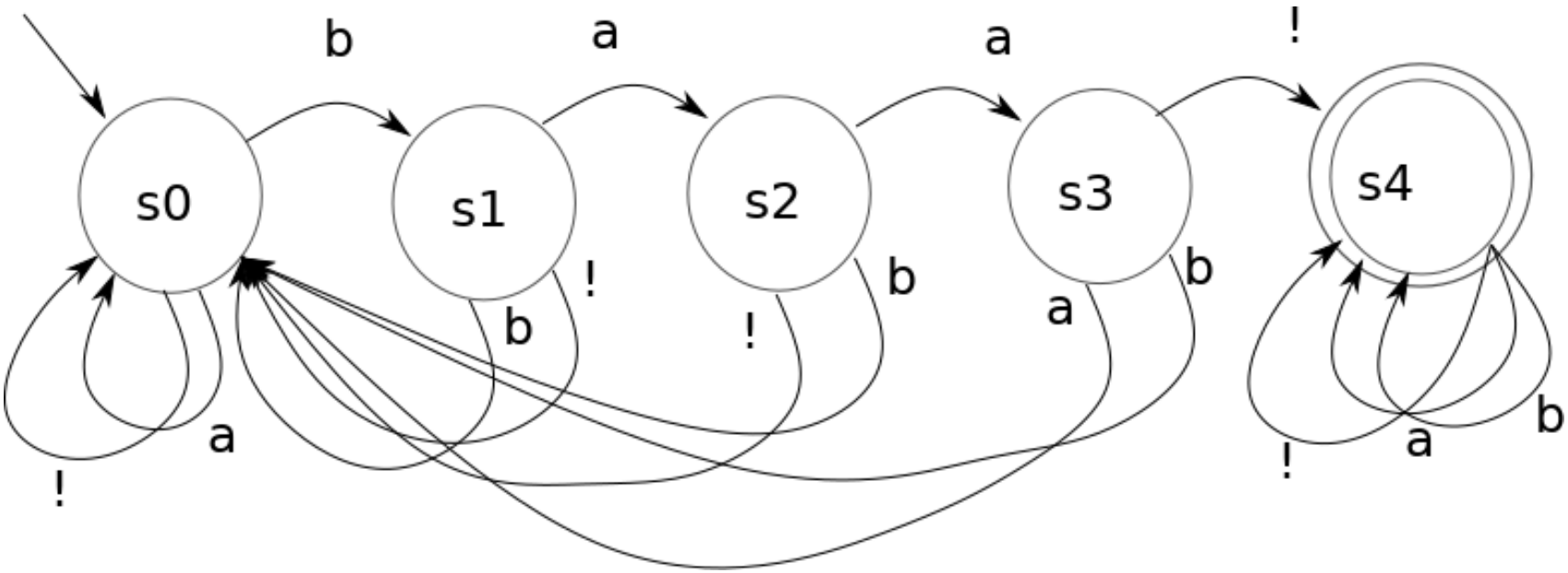


- There is a sequence of different input signals (say, characters) coming to FSM
- There can be only certain characters in this stream. As an example: a,b,! - and nothing else
- There finite number of characters in the stream
- If certain sequence of characters happens to be present in the stream - FSM goes to **accepting** state
- If after input stream finishes FSM is not in accepting state - sequence was **rejected**

How FSM can be used here?

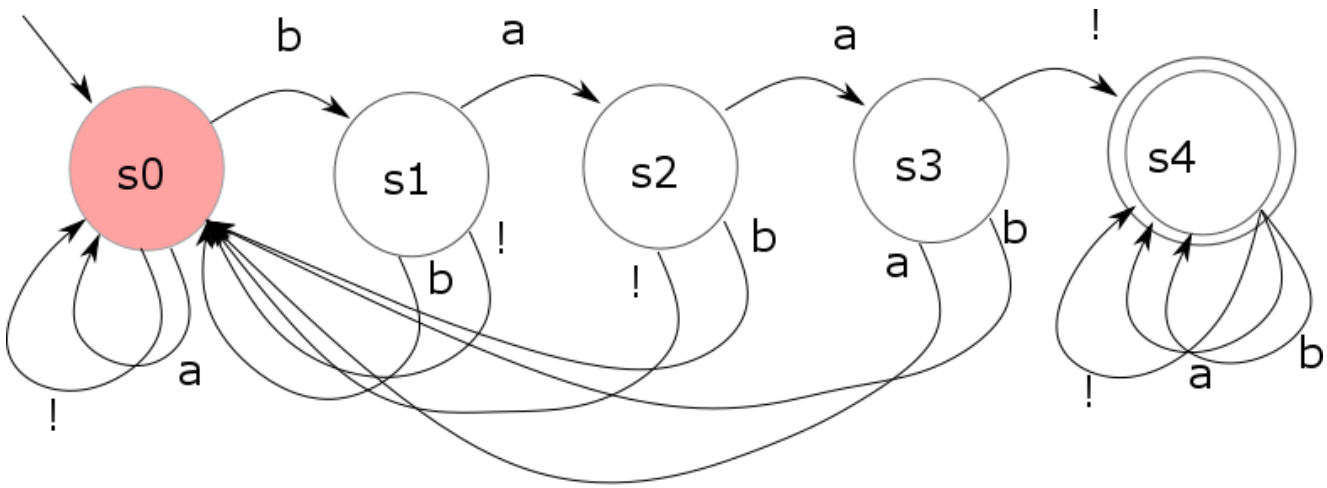
Sheep detector

We want to detect that sequence **baa!** was present - in exactly this order.



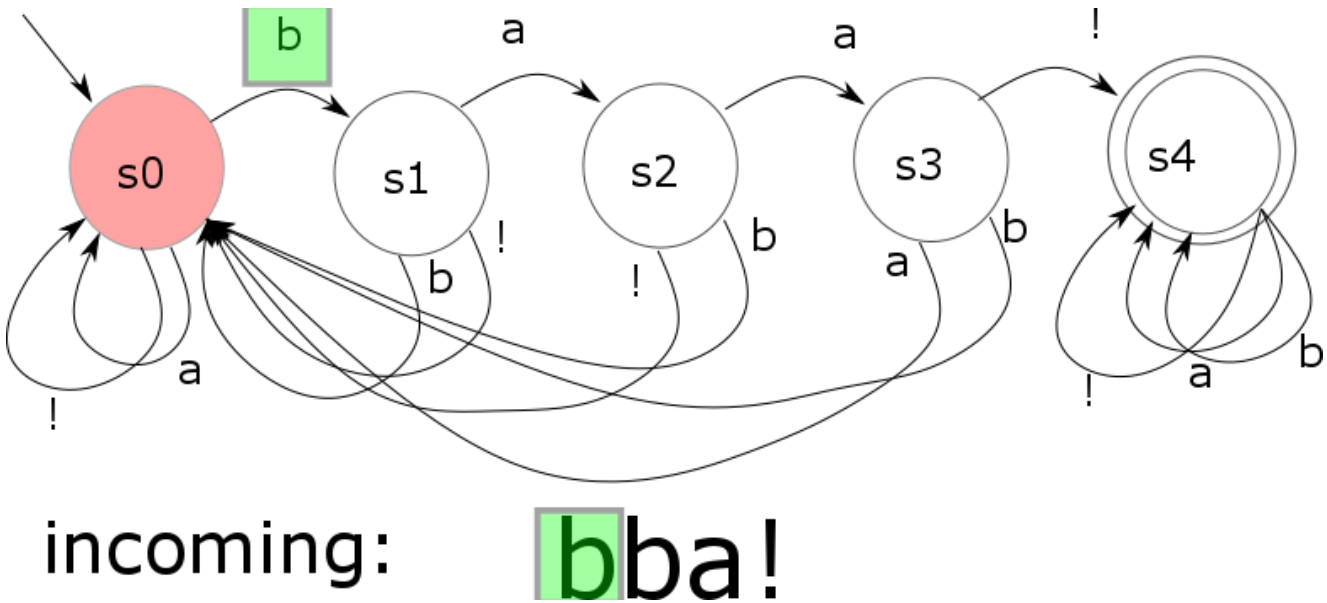
Each new character from the stream drives FSM to next state until accepting state is reached. FSM does not accept as soon as it reaches accepting state. It accepts if it **ends** in an accepting state.

Sheep detector - start

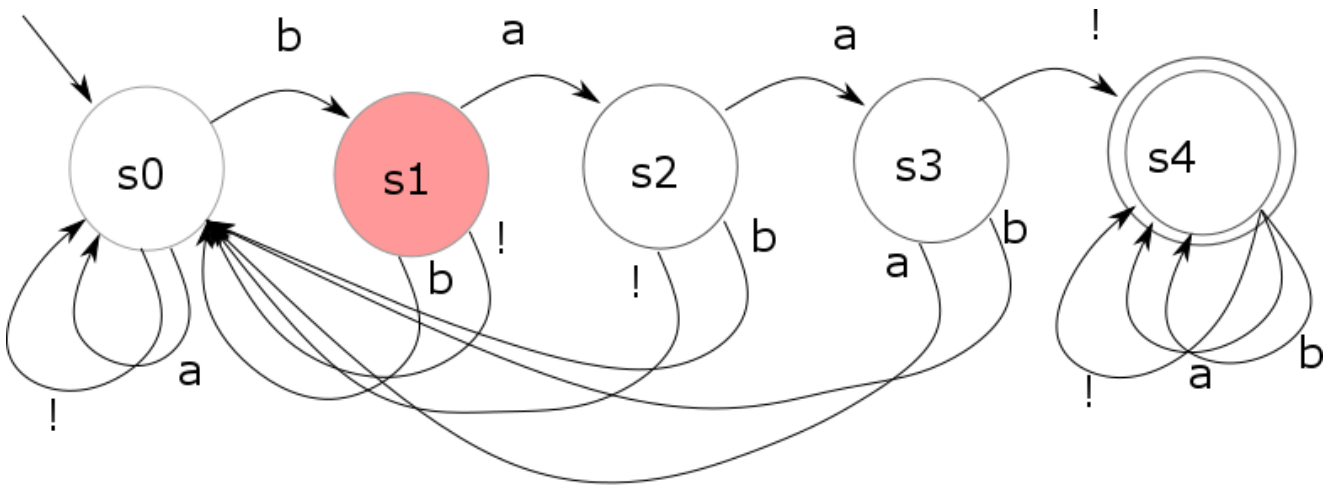


incoming: bba!

Sheep detector - first symbol in

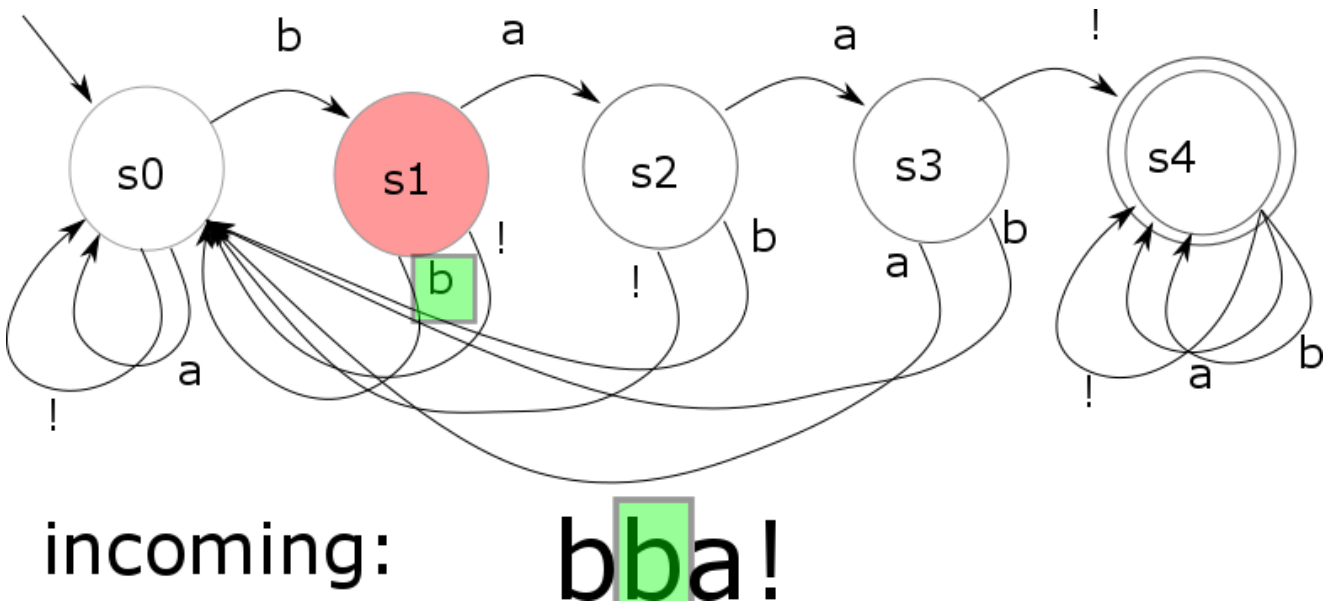


Sheep detector - after first symbol

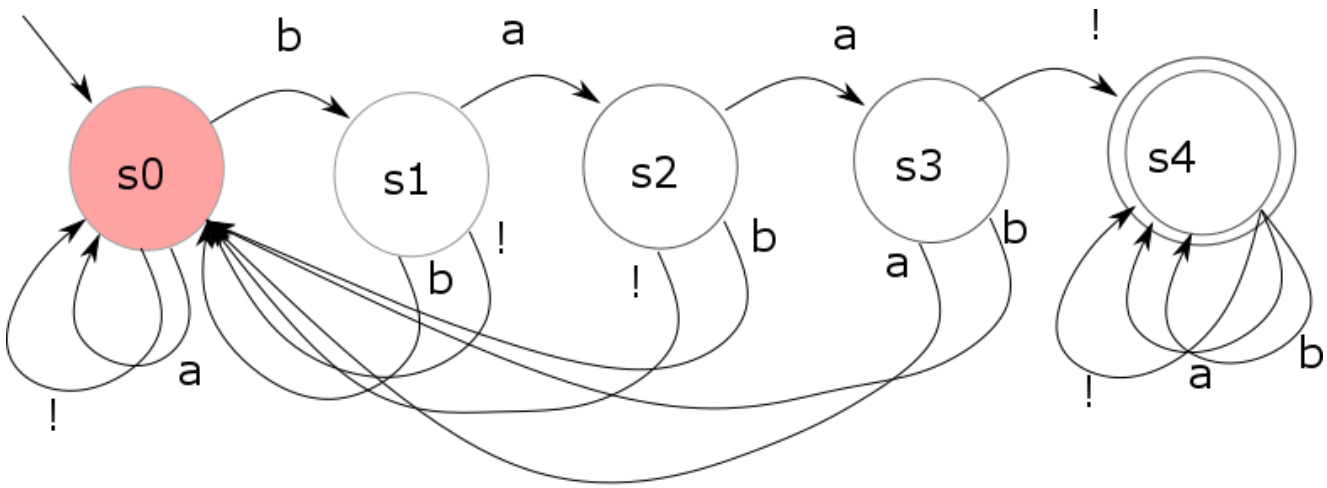


incoming: bba!

Sheep detector - after first symbol



Sheep detector - after first symbol



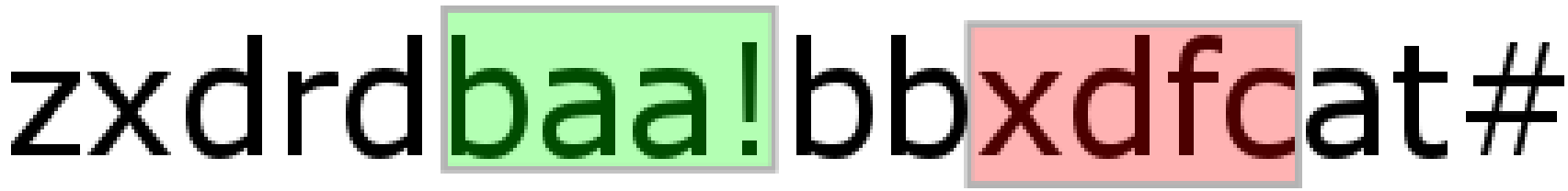
incoming: bba!

Can we just code it?

This particular example is not too hard to code.

```
bool accepted = false;
if ((stream[i]=='!")&&(stream[i-1]=='a")&&
    (stream[i-2]=='a")&&(stream[i-3]=='b") )
{
  aceepted = true;
}
```

and run this check in sliding window.

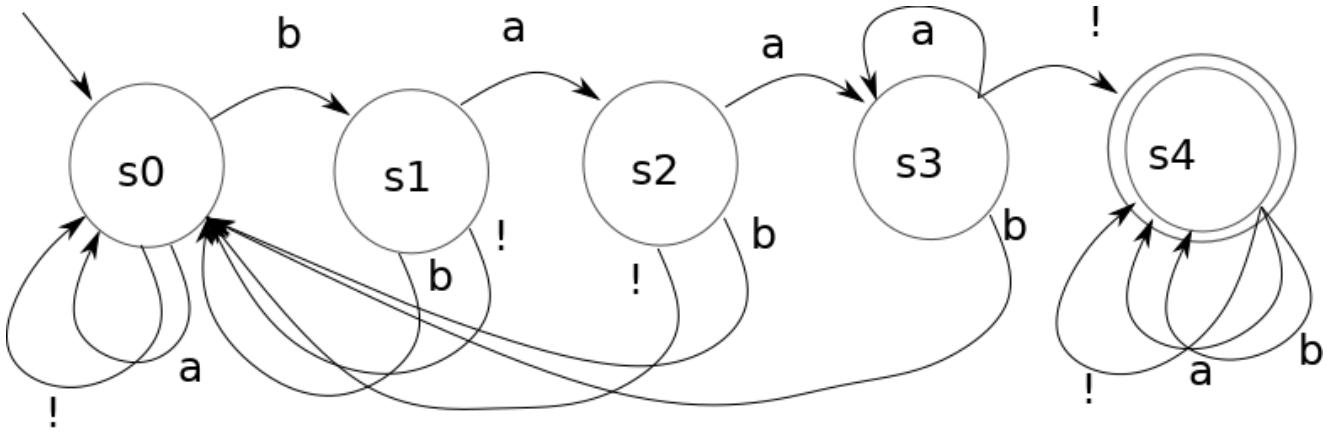


What about a bit different one: Detect that **baaaaa!** was present.
Number of **a**s can be different, but there should be at least two.
Still not impossible to program:

- Detect **!** in input stream
- Check that there are 2 **a**s before it
- If there is **b** in front - success. If there is an **a** - keep checking. If there is an **!** - failure, start again.

Not impossible without using FSM but hard.

And in state machine format:



That would be practically same code (same structure, anyway).

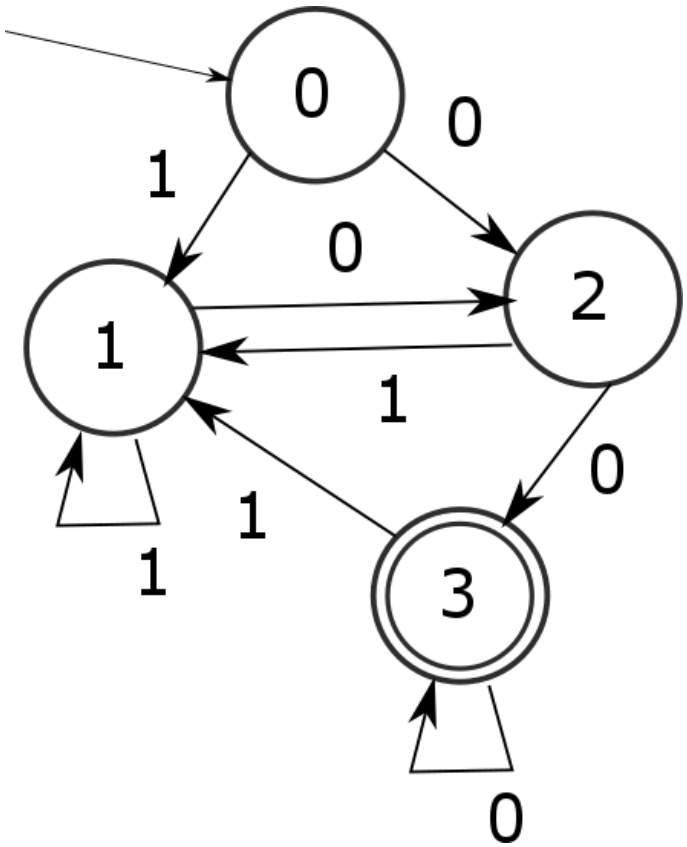
What to do if we have multiple transitions from the state?

Looking at FSM at previous page, state s2.
There are only three letters in the alphabet: **a,b,!**. If input is **a** FSM should move to s3. In **all** other cases it should move to s0.
Way to implement that:

```
\\...
if (state == "s2"){
    if (input == 'a'){
        state = "s3"; // 'a'
    } else{
        state = "s0"; // all other cases: 'b','!'
    }

}
\\...
```

Question



Alphabet is {0,1}.
Accepting state is 3.
Which input stream will be accepted?

1. 0101
2. 0111
3. 1100
4. 1001

Coding it: Using strings in C++

Strings (which are sequences of characters) in C++ can be treated as **vector**s.
Do not forget to include the string library.

```
#include <string>
```

And the vector library.

```
#include <vector>
```

As usual, you can find the manual at
`www.cplusplus.com/reference/string/string`

Programming: Working with characters: C++ functions to decide if character is letter or digit

```
#include <iostream>
#include <ctype.h>
using namespace std;

int main(){
int i=0;
char str[]="Cr67+";
while (str[i]){
if (isalpha(str[i])) cout<<"character "<<str[i]<<" is alphabetic"<<endl;
else cout<<"character "<<str[i]<<" is not alphabetic"<<endl;
i++;
}
i=0;
while (str[i])
{
if (isdigit(str[i])) cout<<"character "<<str[i]<<" is digit"<<endl;
else cout<<"character "<<str[i]<<" is not digit"<<endl;
i++;
}
i=0;
while (str[i])
{
if (isalnum(str[i])) printf ("character %c is alphanumeric\n",str[i]);
else printf ("character %c is not alphanumeic\n",str[i]);
i++;
}
}
```

Working with characters to the string

If you want add character to the string - use **push_back()**.
To delete all characters from the string - use **clear()**.
Example:

```
#include <iostream>
#include <string>
using namespace std;
string str;
int main(){
  char ch = 'a';
  cout<<"  str="<<str<<endl;
  str.push_back(ch);
  cout<<"  str="<<str<<endl;
  str.push_back('b');
  cout<<"  str="<<str<<endl;
  str.clear();
  cout<<"  str="<<str<<endl;
}
```

- isalphanum(): checks whether c is either a decimal digit or an uppercase or lowercase letter.
- isdigit(): is the character a digit
- isalpha(): is character a letter

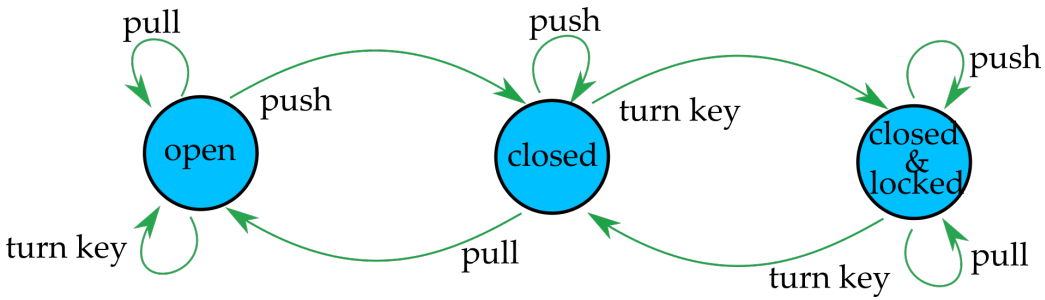
If you want insert the double quotation mark **"** into C++ string (it is indication of the string end) - use **escape sequence** - backslash + double quotation mark.
If you want to insert one string inside another you can use << operator.

```
std::ostringstream new_string;
string a = "s0";
new_string<<"  state = \""<<a<<"\";";
// result is     state = "s0";
```

FSM as an array - repeat slide

Table contains next state of the FSM.
Column - current state.
Row - input.

	open, [0]	closed[1]	closed_locked [2]
push[0]	open	closed	closed_locked
pull[1]	open	closed	closed_locked
turn key[2]	open	closed	closed_locked



More compact FSM code

Listing 1: Matrix.cpp

```
#include <iostream>
using namespace std;
int next_state[2][2] = {{0,0},{1,1}};

int main(){

 int state = 0;
 int input = 0;
 while(1){
   cout<<"state before="<<state<<" Enter input";
   cin>>input;
   state = next_state[input][state];
   cout<<endl;
   cout<<"state after="<<state<<endl;
 }
}
```

More compact FSM code

Code in table form is much more compact, but there is a drawback - all names of states and inputs should be integers:

```
state = next_state[input][state];
```

There are ways around it, like **map**, but it is beyond our scope.

Another type of the FSM - transducer

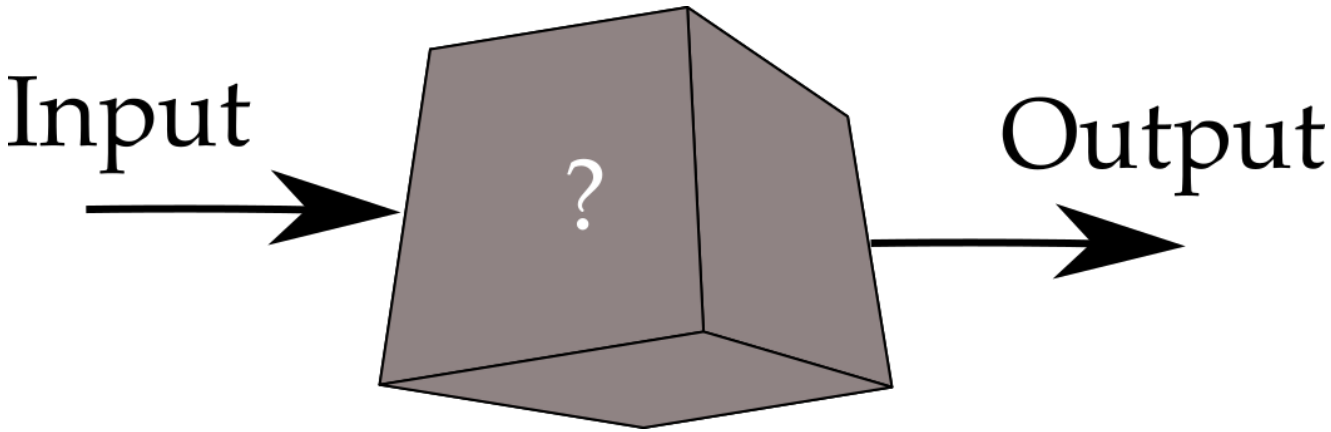


Figure: System takes an input and produces an output

It is a convention to indicate an output with**->**.

Better door, turnstile- you have to pay

- Usually locked
- If you put ticket in - unlocks
- If you push it when unlocked - turns and locks
- Turnstile inputs:
  - Ticket inserted
  - Bar pushed
- Turnstile **outputs** - new bit of FSM:
  - Lock the bar

Control system for turnstile

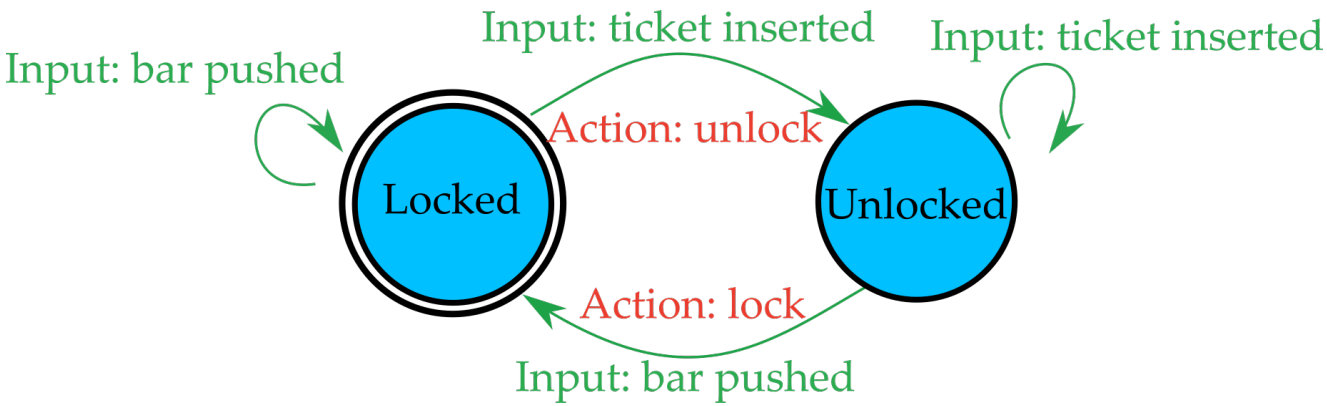


Figure: Outputs shown in red

Another FSM with output - alarm clock

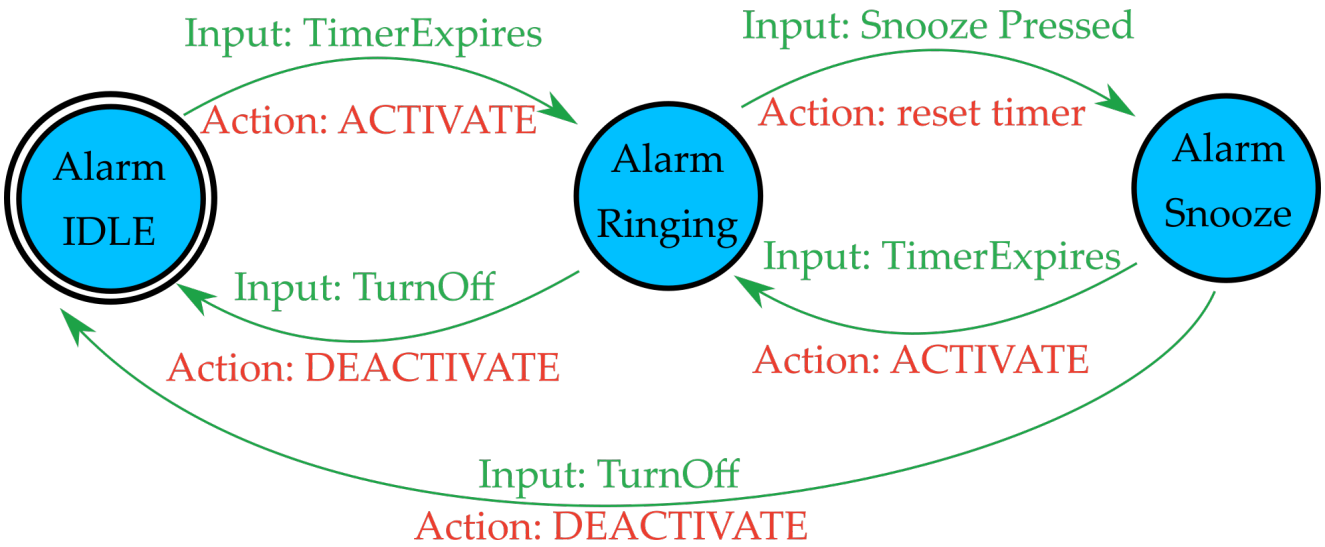


Figure: Outputs shown in red

short **baaaa!**

We want to cut out all repeating **a**s except 2. If input is **bbbaaaaa!** we want to cut to **bbbaa!** and pass result to the output.

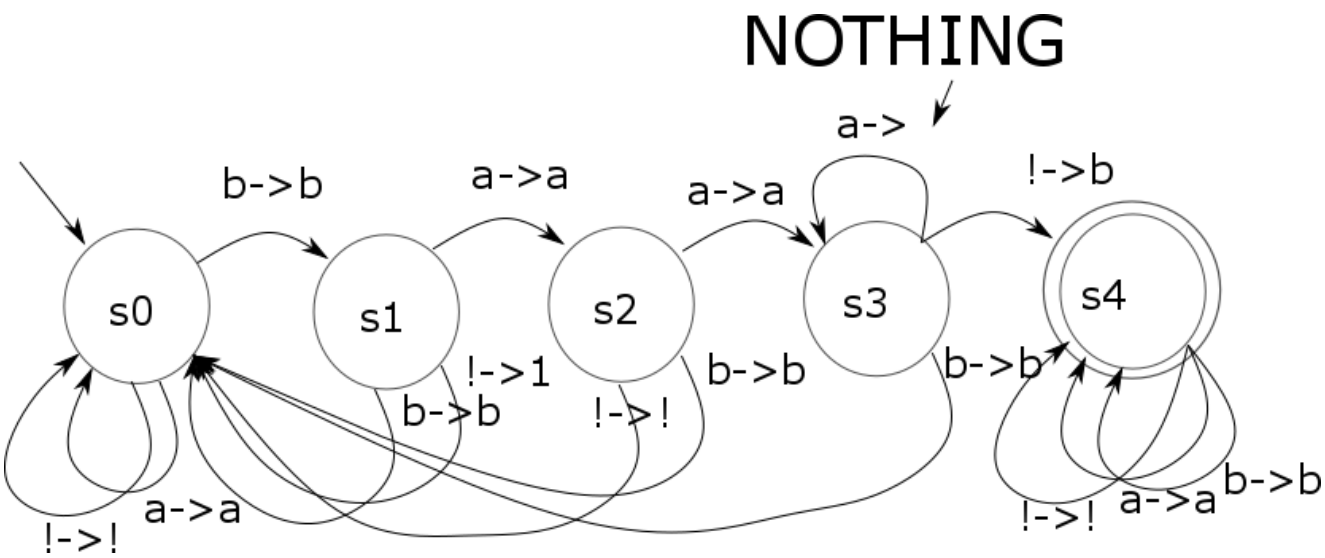


Figure: In all states but s3 input is passed to the output

Notice that there is no output on the looped transition in state s3.

FSMs in hardware design

Binary adder. Similar to ENGR123.
Takes two binary input streams and produces one output binary stream which is sum (binary) of input streams.

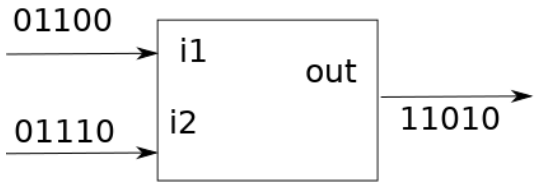


Figure:

i1	0	1	1	0	0
i2	0	1	1	1	0
Out	1	1	0	1	0
Carry	0	1	1	0	0

Binary sum rules:
0+0=0
0+1=1
1+0=1
1+1=10

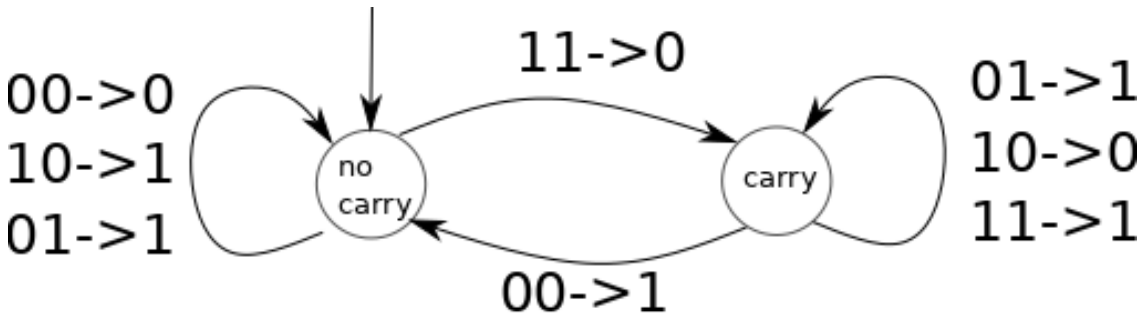


Figure: Output shown with **->**

i1	0	1	1	0	0
i2	0	1	1	1	0
Out	1	1	0	1	0
Carry	0	1	1	0	0

- We are in **nocarry** state and received 0,1. 0+1+0 = 1 (and no carry)
- We are in **carry** state (1 from carry should be added) and we receive 1,0 :1+0+1 = 10 - 0 output , keep the **carry** state.

Programming application of transducers is often to generate sequences of the output.

From programming perspective - it is generating the output function for given input sequence.

In simplest case (as it is in Assignment) output is the screen (or file, as it is in the Project).

Pure State Machine:

- All information is stored in **state**s. There is no memory to store anything.
- Transition is initiated by input signal only.
- There is nothing else: no variables, no conditional checks

Pure State Machine is very limited, not to say useless.
Why?

FSM explosion

We need to implement controller for microwave oven.
There are 3 buttons:

- **+** -increment timer [seconds]
- **-** - decrement timer
- **start** - microwave turns on

There is a timer. It produces **tick** input every second.

Implementing microwave controller as pure State Machine.

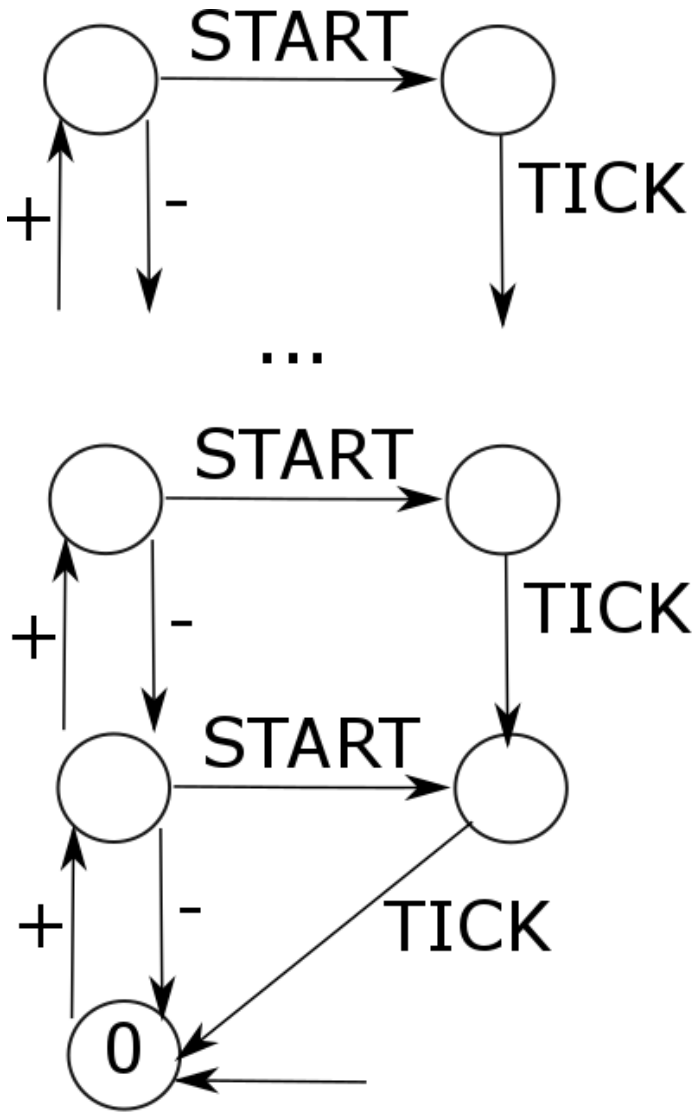


Figure:

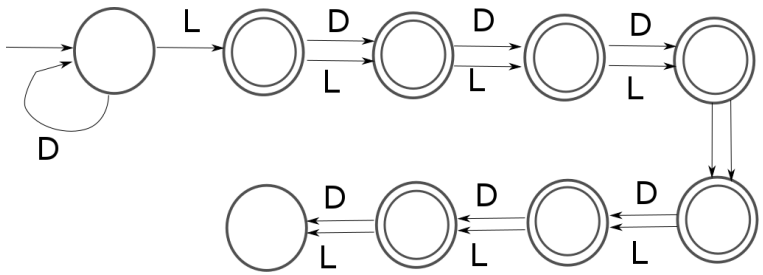
- Implemented as pure State Machine this controller requires twice as many states as there are seconds in max microwave heating time
- State Machine **explodes** - too many states

Extended FSM - one with variables to store the information

- Pure FSM describes both control (program flow) and data.
- Only information stored - what is state of FSM at this moment
- Using variables to represent the data leaves us with control (program flow, execution sequence) alone. It makes FSM simpler.

It explodes!

In some (old) programming languages name of the variable, made out of can be no longer than, say, 7 characters. Indication of the string containing variable name end can be whitespace, for example. We are not concerned with termination here.



If input stream (string) contains 8 or less letters or digits and starts with the letter then machine is in accepting state when string finishes. Machine will work but is rather bulky. We can use variable make it smaller.

State machine with variables

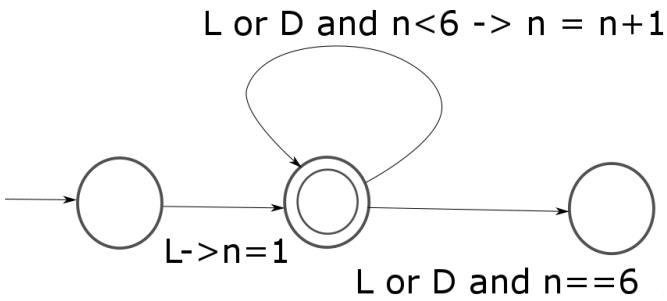


Figure:

- Variable **n** how far into variable name we are
- Structured transitions
- transition takes place only if **guard**s are satisfied.

There are several new things introduces here. First, there is a variable and assignment to it. Second, our transitions are more complex. Guard if logical operator:

```
//  ...
if (state=="s0")&&(n<6)) //
// do something for this situation
```

What is the code for FSM with variables?

```
string state="open";
string input;
int nOpen = 0; \\here is the variable
while(1){
  \\read input
    if (state == "open"){
      if (input=="push"){
        state = "closed";
      }
     if (input=="pull"){
       state = "open"; nOpen++; // variable used
     }
    }
}
```

Example of structured transitions - balanced parentheses

We have string composed of opening and closing brackets **(** and **)** and nothing else.
Anything else indicates end of the string. String **(()())** and **((()))** are accepted. We introduce the variable **n** - number of un-matched opening brackets **(** in the string.

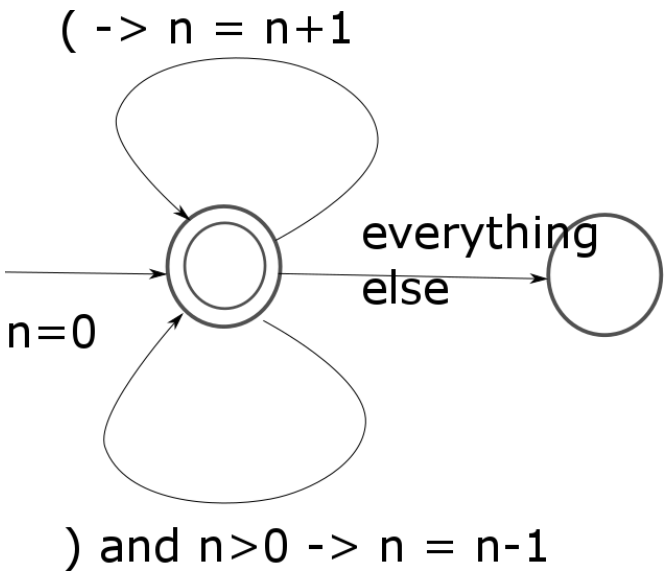


Figure:

A bit more useful thing

FSM which calculates the value of arithmetic expression.

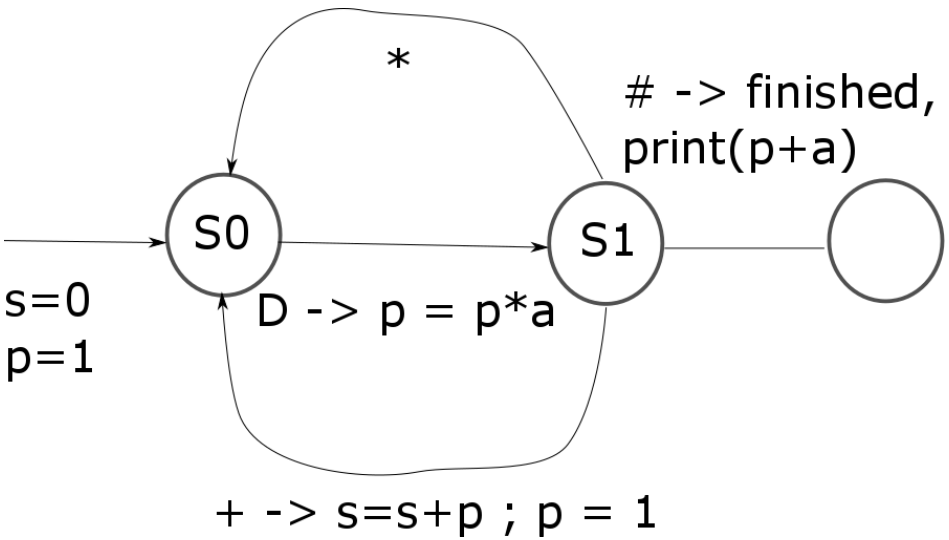


Figure:

- There are two variables s and p
- Variable p contains the value of the current term
- s contains value excluding current term
- If machine is in state S1 - it can terminate processing
- Output is

A bit more useful thing FSM calculator

FSM which calculates the value of arithmetic expression.
We want to calculate:
3*4+6#. Hash indicates the end of the expression
Reading the string from left:

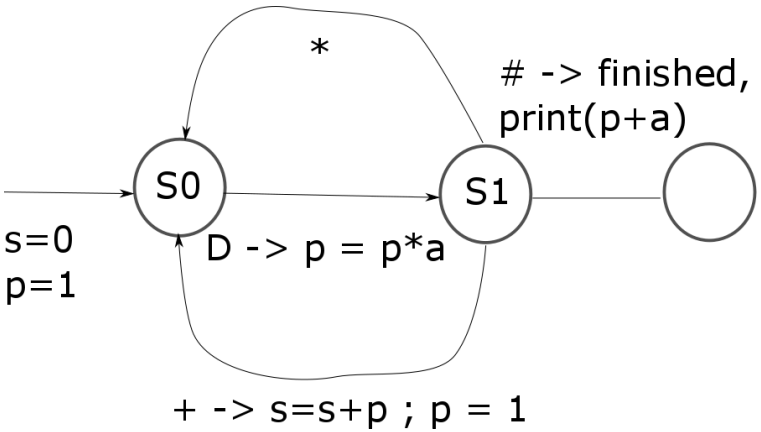
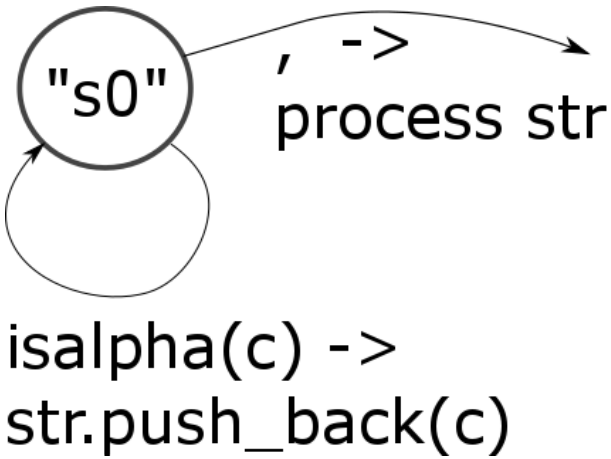


Figure:

INPUT	STATE	STATE	s	p	s+p
none	s0	s0	0	1	1
3	s0	s1	0	3	3
*	s1	s0	0	3	3
4	s0	s1	0	12	12
+	s1	s0	12	1	13
6	s0	s1	12	1	13

Functions called on transitions

If variables are used we can call functions when transitions happen. For example, we want to keep adding characters to the string until there is comma , in the input stream.



Listing 2: "Pseudocode'

```
if (state == "so"){
    if (isalha(in_char) {
        str.push_back(in_char);
    }
}
```

What FSMs are used for?

- User interfaces
- Control systems
-