Automata and machines

Paolo Burgio
paolo.burgio@unimore.it

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
High Performance Real Time Lab

First, solve the problem. Then, write the code.

– John Johnson

Industrial embedded systems

What they do

- Monitor physical properties of the system/plant (via *sensors*)
- Might perform some control, or part of, control algos
- Via *actuators*

Control can be

- Continuous in time
- Discrete in time

➔ Control theory

Industrial controls in a nutshell

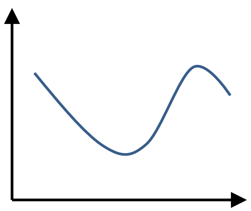
In their generic form,

$$F: \{ S, I \} \rightarrow \{ O \}$$

computed ...when?

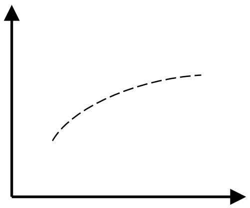
If continuous

- Physical properties and actuators are continuous in time
- *F(t)* continuous
- Combinatorial logic/analogic systems



If discrete

- Computed at pre-determined instance in time
- Event-driven (e.g., timeout, interrupt)
- Sequential logics/digital systems



Finite state automations for discrete controls

E.g., an elevator, reacts to multiple events

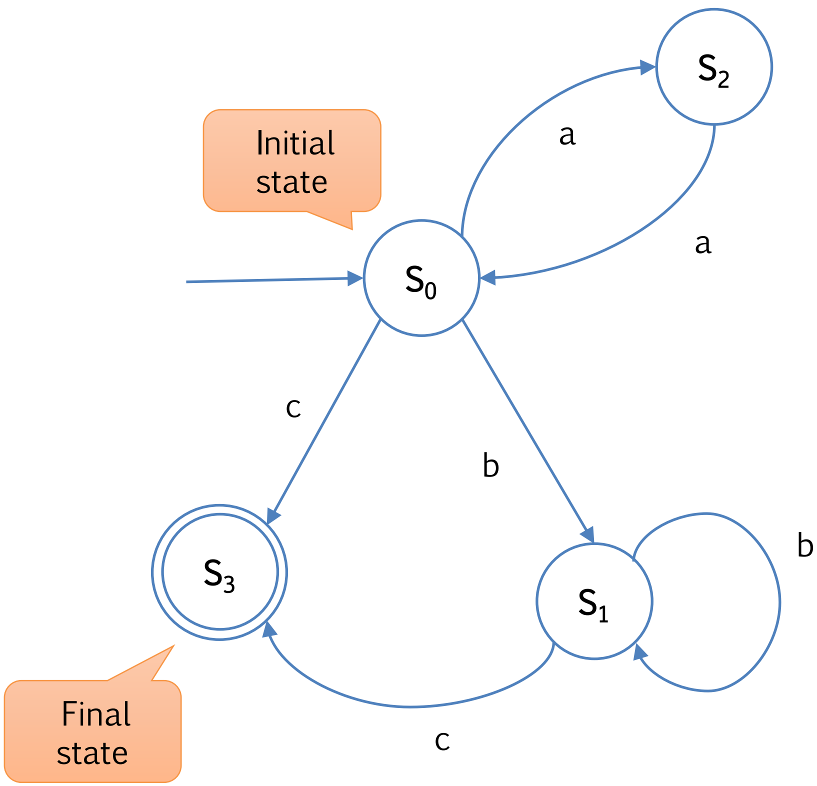
- Typically in idle state
- If you are press the button, the door opens
- You select the floor, doors close
- Then, it reaches the floor (feat. velocity control)
- Then, it opens the door, which subsequently closes after X seconds

This behavior is controlled by a **finite state automations/machine**

Finite State Automations/Machines

Problem

- Identify even sequences of *a* (even empty), followed by one, or more, or no, *b*, ended by *c*



Given an <u>alphabet</u> *V*,

…that identifies a language (we'll see)..

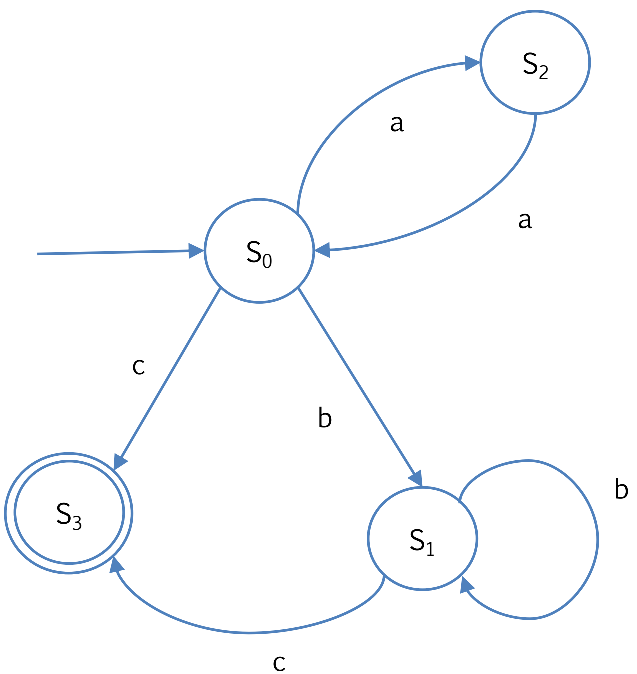
define FSA as

- S: a non-empty states set
- $s_0 \in S$: initial state
- $S_f \subseteq S$: final states set
- $t: S \times V \rightarrow S$: states transaction func

FSMs and languages

Let V* = *{ v, w, … }* contain all the combinations of words using V symbols

- Including the empty word ε
- For instance, ac, aabbc, abbabbbc belong to V*
- (note that, we can associate words in be to inputs, or combination of them)



A **language** L is a subset of V*

(abbabbbc does **not** belong

to L, as previously defined)

"Identify even sequences of a (even empty),

followed by one, or more, or no, b, ended by c"

State transaction function

- $t(s_0, b) = s_1 \quad | \quad s_0 \xrightarrow{b} s_1$
- s_y is <u>reachable</u> by s_x if there exists a path from s_x to s_y
  - a combination of alpabet symbols **l** (letters in our case)

$$t: S \, x \, V \xrightarrow{b} S$$

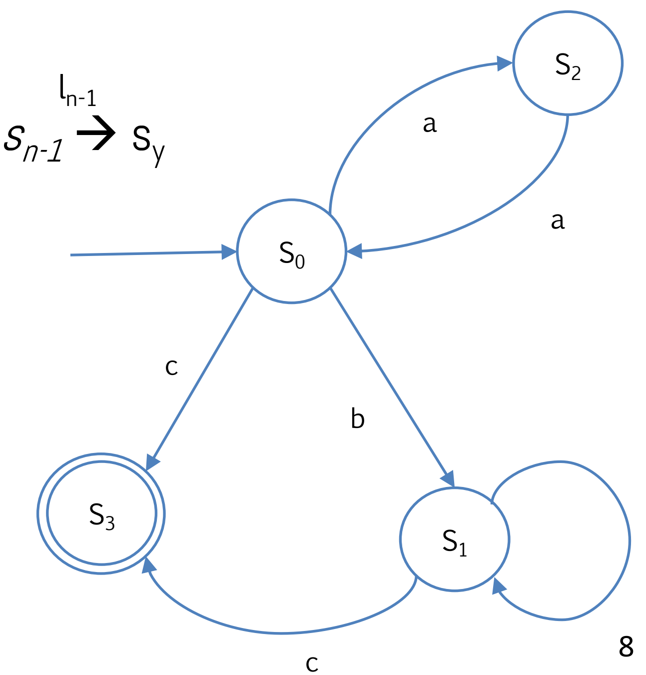
$$\rightarrow^* : \quad S \, x \, V^* \, x \, S : \quad s_x \xrightarrow{w^*} s_y$$

iff

$$w = l_1 l_2 \ldots l_n \qquad \exists \; s_{1,} s_{2\,, \ldots,} s_n : s_x \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots s_{n-1} \xrightarrow{l_{n-1}} s_y$$

$$s_2 \xrightarrow{w^*} s_1$$

$$w = aaab \qquad \exists \; s_{1,} s_{2\,, \ldots,} s_n : s_2 \xrightarrow{a} s_0 \xrightarrow{a} s_2 \xrightarrow{a} s_0 \xrightarrow{b} s_1$$



Exercise

Let's
code!

Implement the automata that understands whether a words is from L

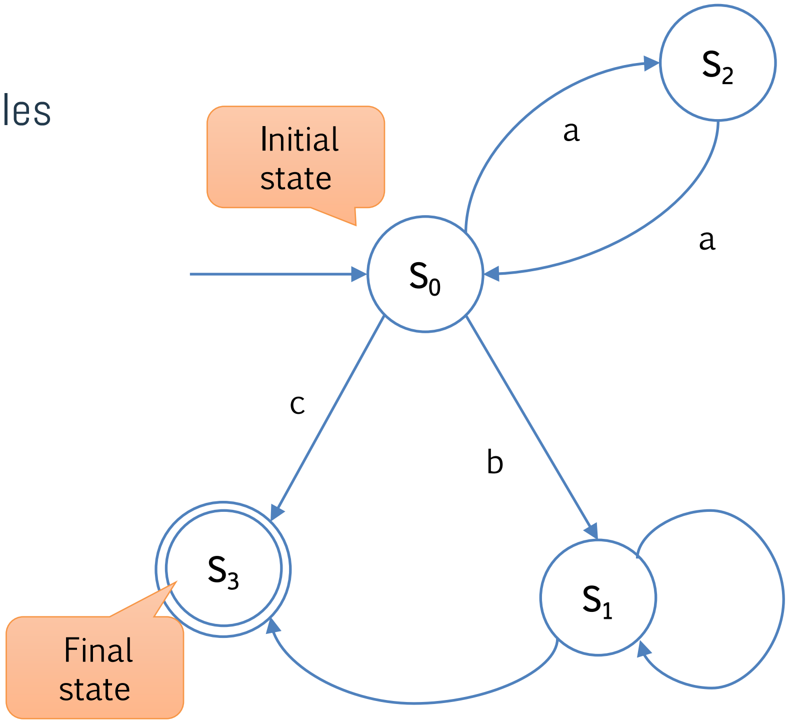
"Identify even sequences of a (even empty),

followed by one, or more, or no, b, ended by c"

Use the language that you want

- You just need IFs, CASE-SWITCH, recursion, tables
- Receive the target word from stdin
- Hint: start simple...

What's missing?



Exercise

Let's code!

Implement the automata that understands whether a words is from L

"Identify even sequences of a (even empty),

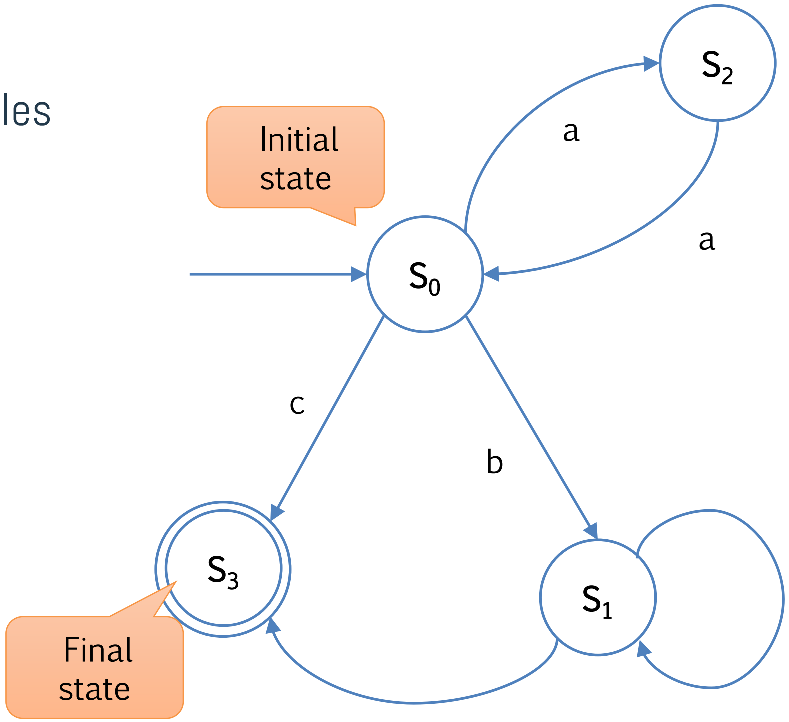
followed by one, or more, or no, b, ended by c"

Use the language that you want

- You just need IFs, CASE-SWITCH, recursion, tables
- Receive the target word from stdin
- Hint: start simple...

What's missing?

- In case of error => default error state
- Typically implicit in state diagrams



Grammars

- A standard way of representing languages (Noam Chomsky, 1950)

$$G = < VT, VN, P, S >$$

VT and VN disjoint

$VT \cap VN = \emptyset$

- VT : terminal symbols $\subseteq$ V
- VN : non-terminal symbols $\subseteq$ V (aka: syntax categories)
- P : production rules $P \subseteq VN \times (VN \cup VT)$
- $S \in VN$: initial symbol

VT and VN are V

$VT \cup VN = V$

A language L_G generated by grammar G is the set of V* elements derived by start symbol S through productions in P

Backus-Naur Form

- Productions rules have form

$$\alpha ::= \beta \;,\; \alpha \in VN \;\; \beta \in V$$

- $x \in VN$ have the form <name>
- | specifies an option

```
VT = { il, gatto, topo, sasso, mangia, beve }

VN = { <frase>, <soggetto>, <verbo>, <compl-ogg>, <articolo>, <nome> }

S = <frase>

P = {
 <frase> ::= <soggetto> <verbo> <compl-ogg>
 <soggetto> ::= <articolo> <nome>
 <articolo> ::= il
 <nome> ::= gatto | topo | sasso
 <verbo> ::= mangia | beve
 <compl-ogg> ::= <articolo> <nome>
}
```

Automata states

Another example

- Natural numbers

```
VT = { 0, 1, …, 9 }

VN = { <num>, <cifra>, <cifra-non-nulla> }

S = <num>

P = {
 <num> ::= <cifra>|<cifra-non-nulla>{<cifra>}
 <cifra> ::= 0|<cifra-non-nulla>
 <cifra-non-nulla> ::= 1|2|3|4|5|6|7|8|9
}
```

"Recursion" Extended BNF

- **Challenge**: extend it with sign (+, -)!

Another example: solution

- Natural numbers

```
VT = { 0, 1, …, 9, +, - }

VN = { <int>, <num>, <cifra>, <cifra-non-nulla> }

S = <int>

P = {
 <int> ::= [+|-] <num>
 <num> ::= <cifra> | <cifra-non-nulla> {<cifra>}
 <cifra> ::= 0 | <cifra-non-nulla>
 <cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
}
```

- **Challenge**: extend it with sign (+, -)!

In reality...

- We only give production rules: VN, VT, S are implicitly defined..

```
P = {
 <frase> ::= <soggetto> <verbo> <compl-ogg>
 <soggetto> ::= <articolo> <nome>
 <articolo> ::= il
 <nome> ::= gatto | topo | sasso
 <verbo> ::= mangia | beve
 <compl-ogg> ::= <articolo> <nome>
}
```

```
Let's
code!
```

Want to try?

- Implement a machine that recognizes whether a sentence (aka: **a word of the Language L**) is legal for that language
- ("our" words are symbols of L)

Chomsky classification

- 4 types of grammars, with increasing constraints on production rules structures

Type 0

- No restriction on productions
- Phrases can even become shorter!

Type 1 grammars/languages

- *Context-sensitive*
- Production must be in the form

$$x\ A\ y \rightarrow x\ \alpha\ y$$

where

$$x, y, \alpha \in (VT \cup VN)^*,\ A \in VN,\ \alpha \neq \varepsilon$$

- A can be replaced with α only if in the context of (surrounded by) x and y
- Phrases never get shortened
- $\alpha \longrightarrow \beta$ con $|\beta| \geq |\alpha|$

Type 2 grammars/languages

- *Context-free*
- Production must be in the form

$$A \rightarrow \alpha$$

where

$$\alpha \in (VT \cup VN)^*,\ A \in VN$$

- α can be ϵ
- A can always be replaced with α

Type 3 grammars/languages

- *Regular*
- Production must be in the **linear** form

$A \rightarrow \alpha$	$A \rightarrow \alpha$
$A \rightarrow B\,\alpha$	$A \rightarrow \alpha\, B$

where

$\alpha \in VT^*$, $A, B \in VN$

- α can be ϵ
- Either left, or right linear: not both in the same grammar

...and..?

We can build specific machines to recognize/process specific grammar Types

- Type 0 => Turing machine (if L(G) is recognizable)
- Type 1 => **Turing machine** with constrained tape length
- Type 2 => Finite state automations with stack (**Push down automations**)
- Type 3 => **Finite state automations**

Hierarchy of machine types

- Base (combinatorial) machine
- Finite state machines – FSM
- FSM with stack (PDA)
- Turing machine

Base combinatorial machine

< I, O, mfn >

I : (finite) set of Input symbols

O : (finite) set of output symbols

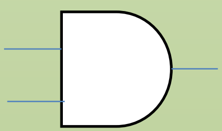
mfn: I → O machine function

- E.g., Logical ports, gates
- Suitable for continuous control
- Non suitable if you need state/memory
  - Need to model all possible cases!

Example: logical AND

I = { {0,1} × {0,1} }

O = {0,1}



mfn defined by a table

	0	1
0	0	0
1	0	1

Finite state machine

`< I, O, S, mfn, sfn >`

`I` : (finite) set of Input symbols

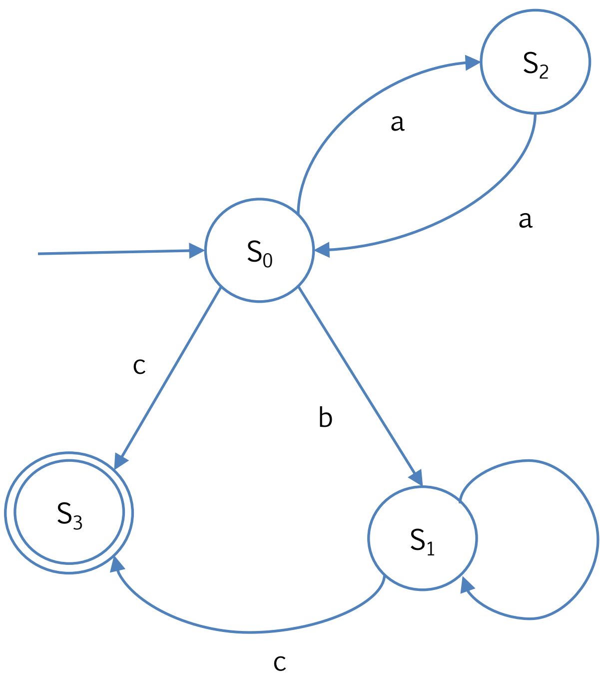
`O` : (finite) set of output symbols

`S` : (finite) set of states

`mfn: I x S → O` machine function

`sfn: I x S → S` state function

- Partly already seen
- Has memory
- Memory is a limitation



Finite state machine with stack

< I, O, A, S, mfn, sfn >

`I` : (finite) set of Input symbols

`A` : (finite) set of stack alphabet symbols

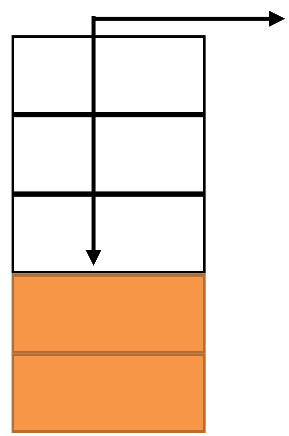
`O` : (finite) set of output symbols

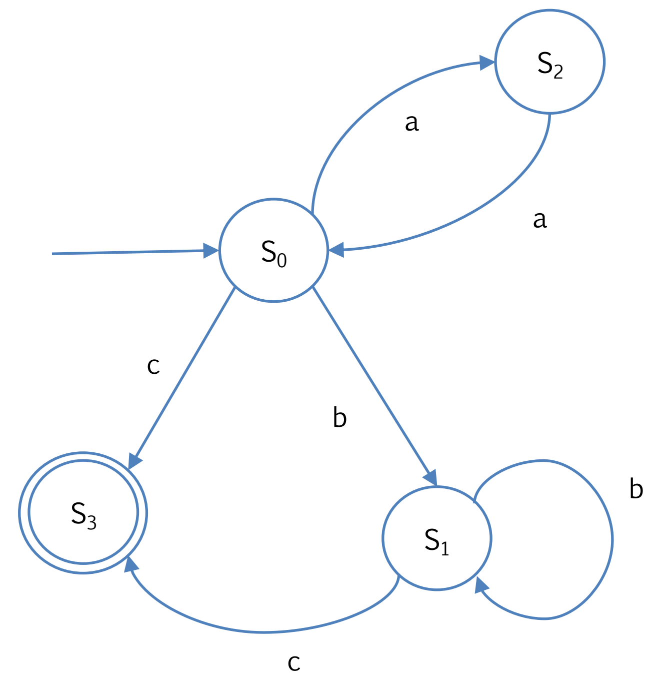
`S` : (finite) set of states

`mfn: I x S x A` → `O` machine function

`sfn: I x S x A` → `S` state function

- Also known as Push-Down Automata (PDA)
- Uses a stack
- We'll see them…





Turing machine

`< A, S, mfn, sfn, dfn >`

`A` : (finite) set of in/out symbols

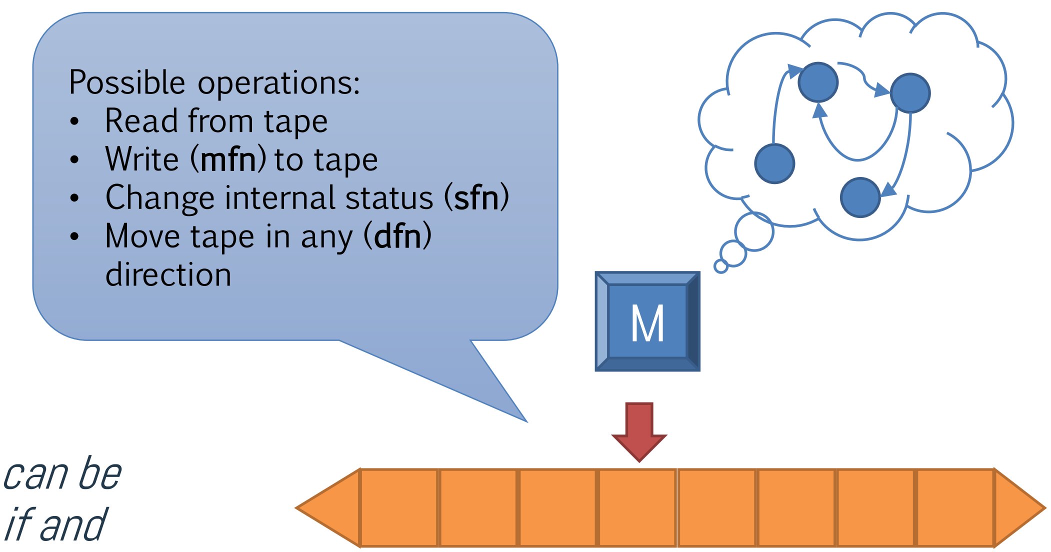
`S` : (finite) set of states

`mfn`: `A x S` → `A` machine function

`sfn`: `A x S` → `S` state function (inc. `HALT`)

`dfn`: `A x S` → `{ left, right, none }` direction function

- Unlimited memory



Church-Turing thesis

A function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine

A Universal Turing Machine

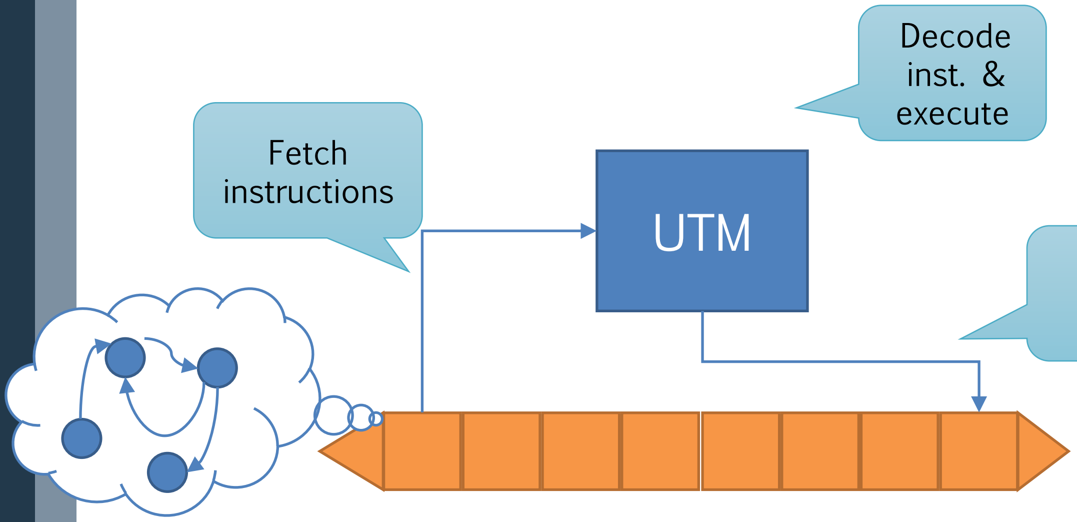
- In TM, the algorithm is inside the machine M, we write results in the tape

What if instruction as well is <u>in</u> the tape?

- We have a programmable machine, with a memory
- .....does this remind something?

Which are the catch? What do we miss?

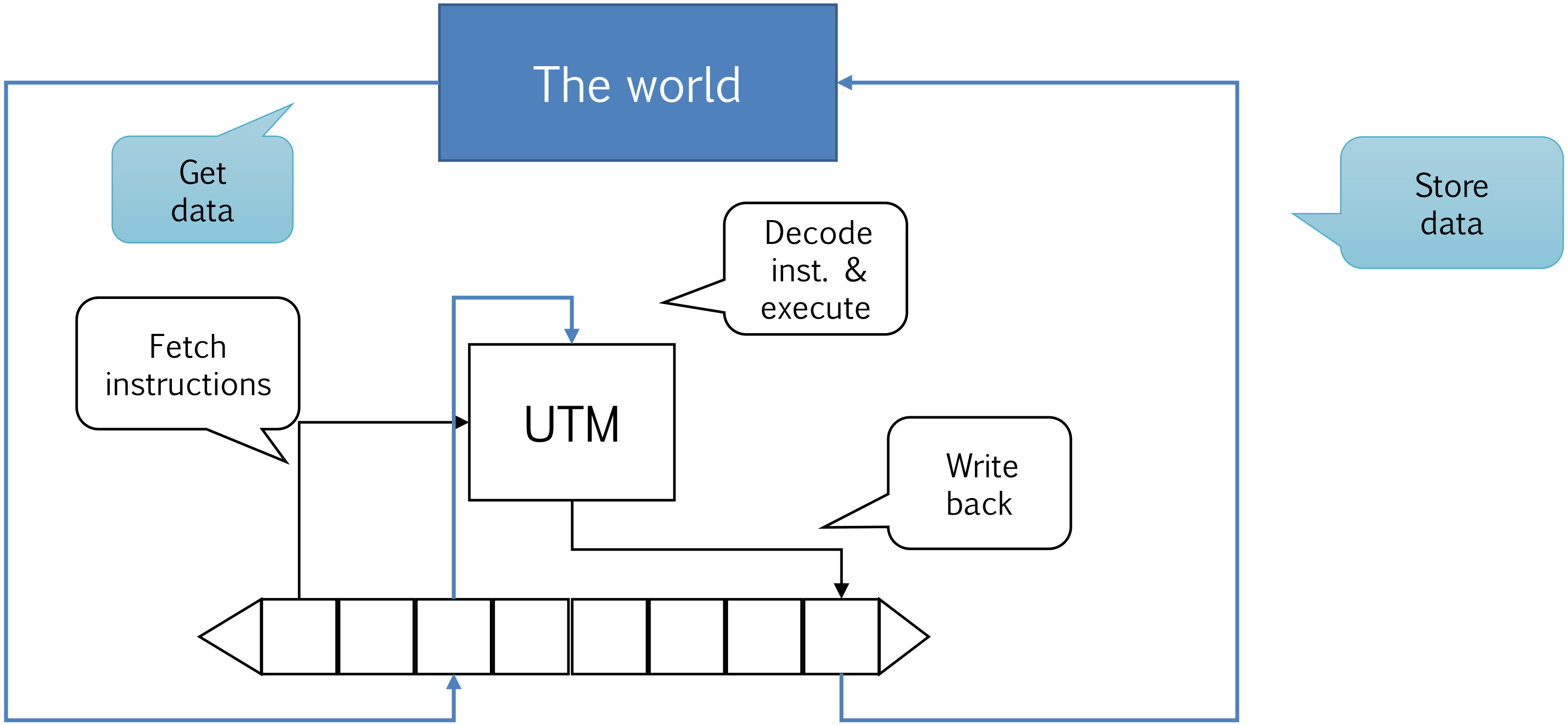
- Ok, the infinite tape makes it infeasible
- ..but what else?



The Van Neumann Machine

We also need to model the interaction with the environment!

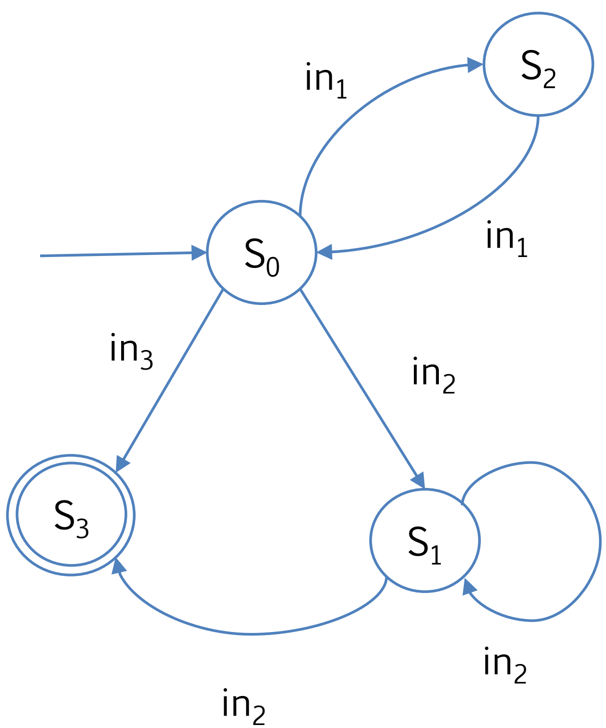
- Aka: I/O (HD/SSD is also I/O)
- Where data comes from!
- It is a real machine: we can **build** it



How to implement a FSM

A generic FSM

- Till now, we only saw machines that can recognize a **word** from a language
  - I say "word", you might want to understand "sentence"
- Let's now see how a machine can actually **produce** an output



The Machine of Mealy

- When crossing an edge, produce an output

`< I, O, S, mfn, sfn >`

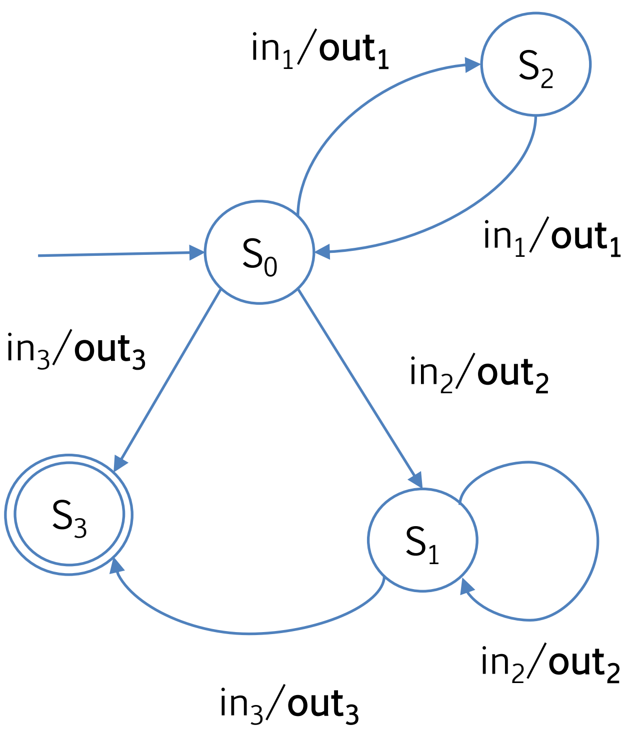
`I` : (finite) set of Input symbols

`O` : (finite) set of output symbols

`S` : (finite) set of states (s_0 initial state)

`mfn: I x S` → `O` machine/output function

`sfn: I x S` → `S` state transition function



The Machine of Moore

- When in a state an edge, produce an output

`< I, O, S, mfn, sfn >`

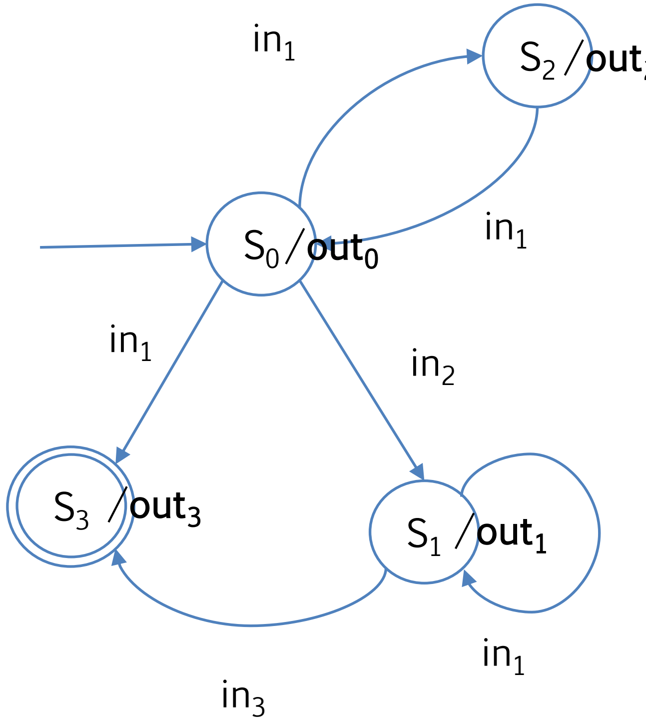
`I` : (finite) set of Input symbols

`O` : (finite) set of output symbols

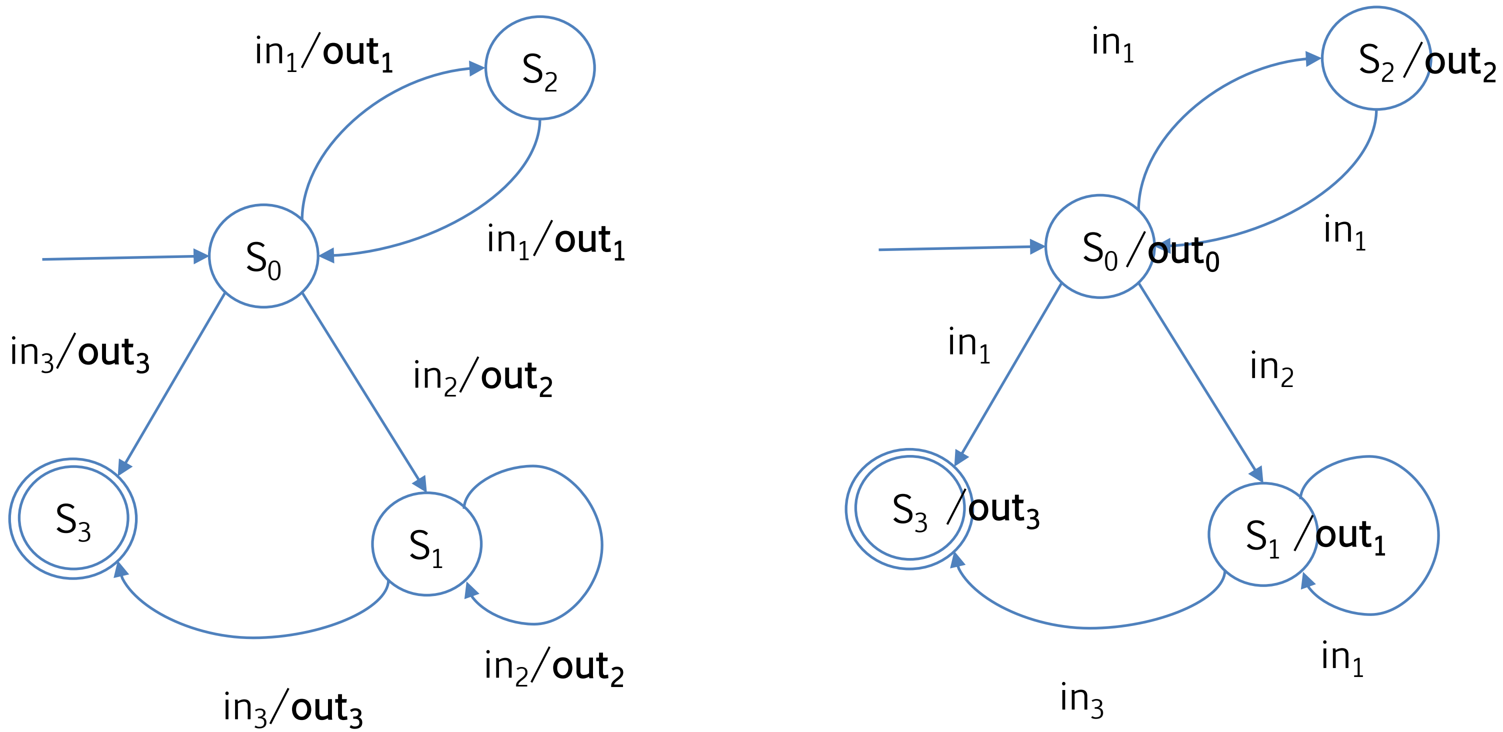
`S` : (finite) set of states (s_0 initial state)

`mfn`: `S` → `O` machine/output function

`sfn`: `I x S` → `S` state transition function



What's the difference?



What's the difference?

Mathematically equivalent

- One can be transformed in another

..but..

- Mealy can potentially have different outs, to different inputs/transitions
  - Less states, if output depends on inputs one can add an edge to the machine
- Moore potentially keeps the output stable for all the state
  - Moore requires more states, in case out depends on input and not only on state

Exercise

Let's code!

› Implement the automata that understands whether a words is from L

"Identify even sequences of a (even empty),

followed by one, or more, or no, b, ended by c"

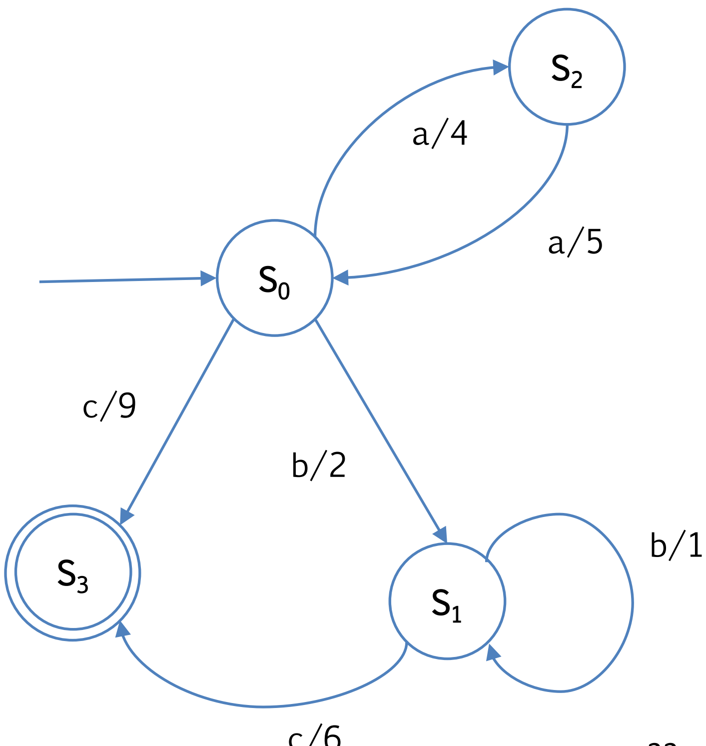
› ..and writes the corresponding number (I choose them randomly)

› Mealy? Moore? You choose

– Here, I show Mealy

Hint

› If not already done, use tables for state/output transactions



What else?

Several tools to support the design

- Matlab Stateflow, UML

Several grammar interpreters to rely the burden of writing FSM code

- FSF's GNU Bison – Included in GCC
- YACC – Yet Another Compiler-Compiler

GNU Bison

Converts a context-free grammar into a deterministic LR parser (but not only) in C

- Recognizes correct sentences from a grammar
- https://www.gnu.org/software/bison/

Input format: Bison grammar files

```
%{
  Prologue
%}

Bison declarations

%%
Grammar rules
%%

Epilogue
```

Bison prologue

C-style code that will be appended at the beginning of the generated file

- Useful for defining macros, includes, headers..
- `ptypes.h` contains Bison internal data structures: trees, tokens…

```
%{
  #define _GNU_SOURCE
  #include <stdio.h>
  #include "ptypes.h"
%}


%union {
  long n;
  tree t;  /* tree is defined in ptypes.h. */
}


%{
  static void print_token (yytoken_kind_t token, YYSTYPE val);
%}
```

Grammar rules

- Like-BNF syntax
- Can also include (C) language-specific rules

```
// results => non-terminal;
// components => any
result: components…;

// C statement
{C statements}


// Multiple rules
result:
  rule1-components…
| rule2-components…
…
;

// recursive rule
expseq1:
  exp
| expseq1 ',' exp
;
```

Example - Reverse-polish notation calculator

rpcalc.y

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM             { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        | exp exp '*'     { $$ = $1 * $2;    }
        | exp exp '/'     { $$ = $1 / $2;    }
      /* Exponentiation */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'         { $$ = -$1;        }
;
%%
```

Example - Reverse-polish notation calculator

"A complete input is either an empty string, or a complete input followed by an input line"

rpcalc.y

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM             { $$ = $1;           }
        | exp exp '+'     { $$ = $1 + $2;      }
        | exp exp '-'     { $$ = $1 - $2;      }
        | exp exp '*'     { $$ = $1 * $2;      }
        | exp exp '/'     { $$ = $1 / $2;      }
      /* Exponentiation */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'         { $$ = -$1;          }
;
%%
```

Example - Reverse-polish notation calculator

rpcalc.y

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM             { $
        | exp exp '+'     {
        | exp exp '-'     {
        | exp exp '*'     {
        | exp exp '/'     {
      /* Exponentiation */
        | exp exp '^'     {
      /* Unary minus    */
        | exp 'n'         {
;
%%
```

"Can be either a newline, or an expression followed by a newline"

Also, speficies an **action** that prints this value (`exp`, indicated by *$1*)

Note: we use language-specific features and libraries, such as `printf` (in prologue, I included `stdio.h`)

Example - Reverse-polish notation calculator

rpcalc.y

Multi-rules expression (“pure” numbers + six arithm operators)

Actions specify how to translate it in C

- $$ => result
- $1, $2 => operators
- (remember to `#include math.h` ☺)

```
                                                   ; }
;

exp:      NUM             { $$ = $1;          }
        | exp exp '+'     { $$ = $1 + $2;     }
        | exp exp '-'     { $$ = $1 - $2;     }
        | exp exp '*'     { $$ = $1 * $2;     }
        | exp exp '/'     { $$ = $1 / $2;     }
      /* Exponentiation */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'         { $$ = -$1;         }
;
%%
```

Exercise (optional)

Let's code!

Write a parser for the following grammar using Bison

```
P = {
 <frase> ::= <soggetto> <verbo> <compl-ogg>
 <soggetto> ::= <articolo><nome>
 <articolo> ::= il
 <nome> ::= gatto | topo | sasso
 <verbo> ::= mangia | beve
 <compl-ogg> ::= <articolo> <nome>
}
```

Event driven Systems

Event driven systems

A system that reacts from external stimula

- Instantly?
- Aka: Cyber-Physical Systems (CPS)

Can be

- Synchronous
- Asynchronous

Synchronous (Active polling)

- Infinite loop

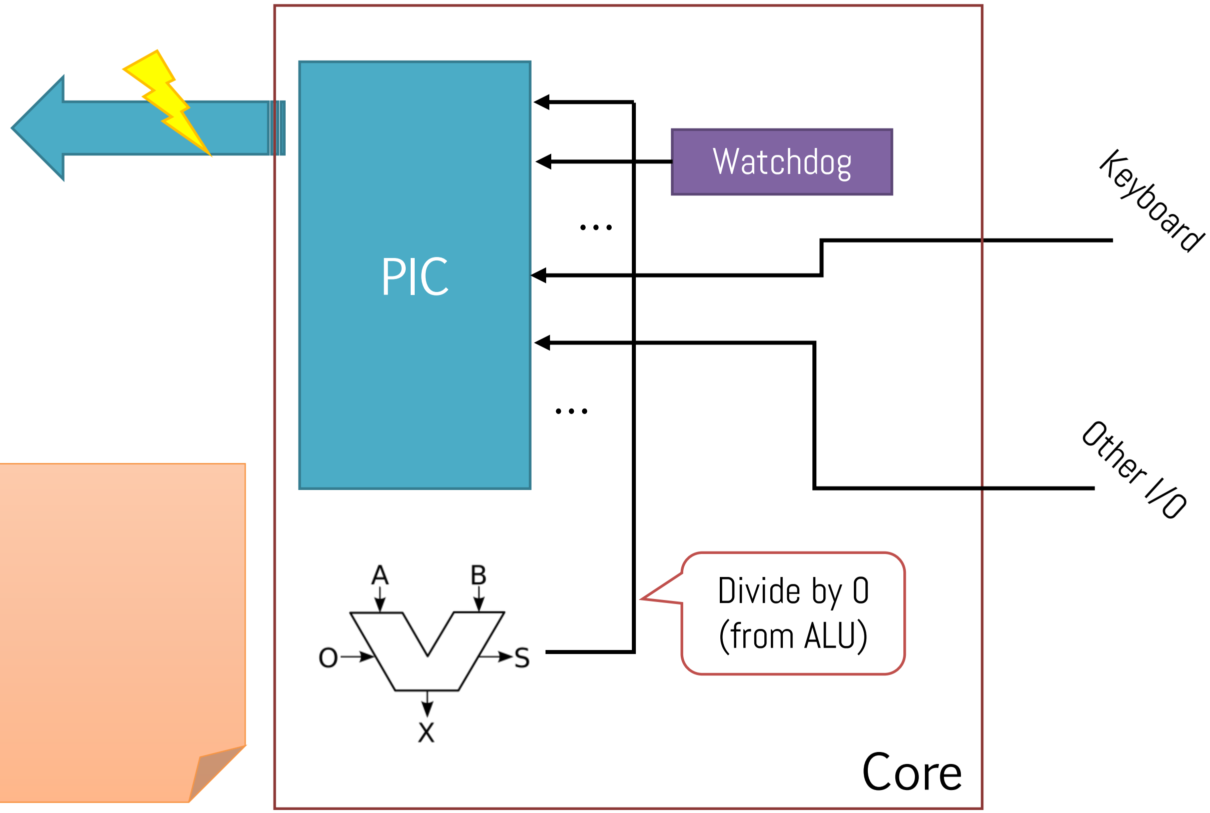
```c
char c;
while (c != EXIT_VALUE)
  c = readC();

// We can go, now
```

- **Pros**: extremely fast and reactive
- **Cons**: waste of resources as one core is busy
  - Possible workaround: insert a sleep

Asynchronous (Interrupt Service Routine)

- Programmable interrupt controller (hierarchy)

```
top_half() {
  // ...
}
```



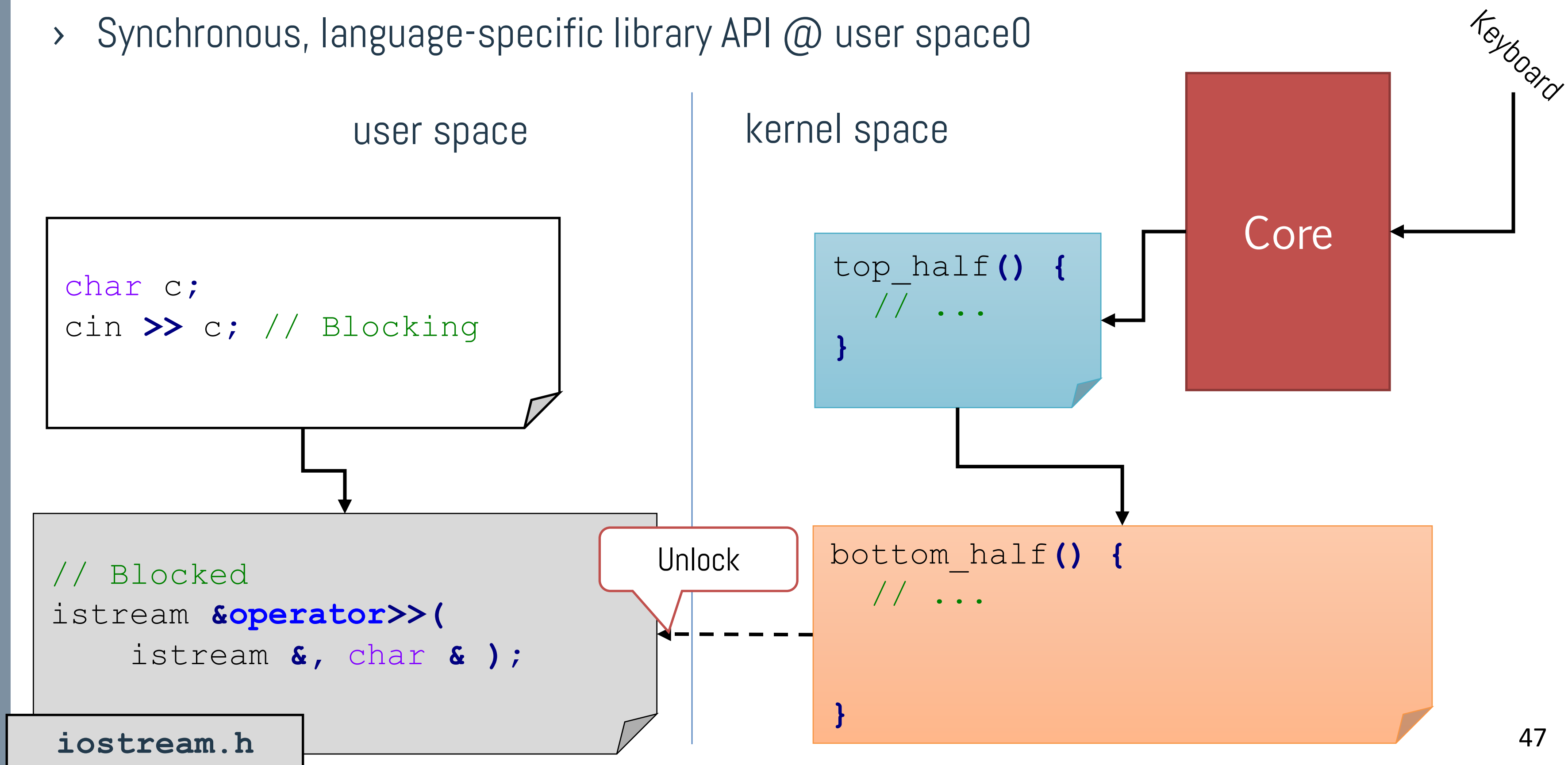

```
bottom_half() {
  // ...
}
```

- **Pros**: "pay-as-you-go"
- **Cons**: takes more time to issue a ISP

...a mix of the two

Keyboard management in a General-Purpose system

- GNU/Linux
- ISP with bottom-half and top-half @ kernel space
- Synchronous, language-specific library API @ user space0

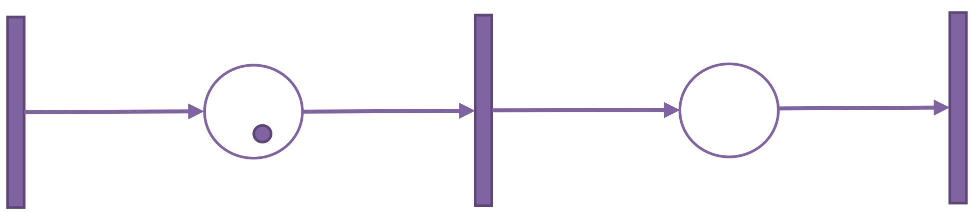


Petri nets

Petri nets - definition

A directed bipartite graph

- **Transitions** triggered by **events** (*bars*)
- **Places**, i.e., conditions (*circles*)
- *Arcs* connect only places to transitions (or vice-versa), and specify which places are **pre- or post-conditions** for events
- Every place collects **tokens** (*dots*) which might trigger an event (if multiple events are triggered in the same net, which fires first is non-deterministic

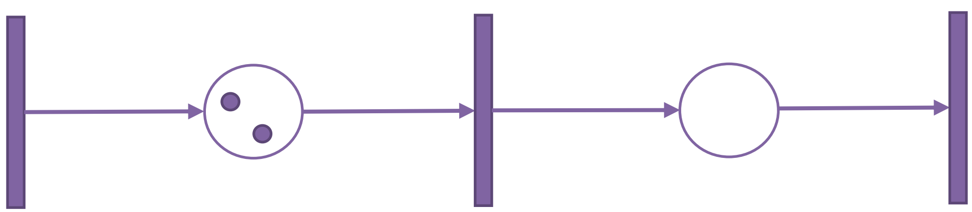


Model distributed systems, discrete events dynamic systems

Petri nets - definition

A directed bipartite graph

- **Transitions** triggered by **events** (*bars*)
- **Places**, i.e., conditions (*circles*)
- *Arcs* connect only places to transitions (or vice-versa), and specify which places are **pre- or post-conditions** for events
- Every place collects **tokens** (*dots*) which might trigger an event (if multiple events are triggered in the same net, which fires first is non-deterministic

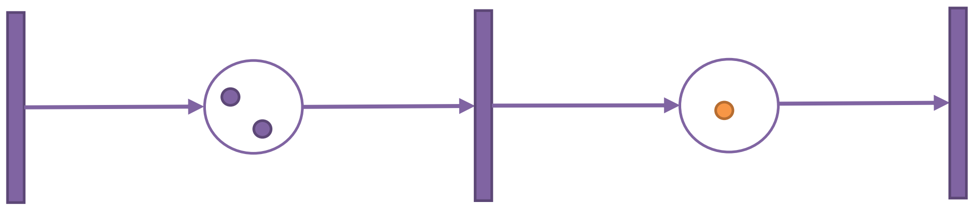


Model distributed systems, discrete events dynamic systems

Petri nets - definition

A directed bipartite graph

- **Transitions** triggered by **events** (*bars*)
- **Places**, i.e., conditions (*circles*)
- *Arcs* connect only places to transitions (or vice-versa), and specify which places are **pre- or post-conditions** for events
- Every place collects **tokens** (*dots*) which might trigger an event (if multiple events are triggered in the same net, which fires first is non-deterministic

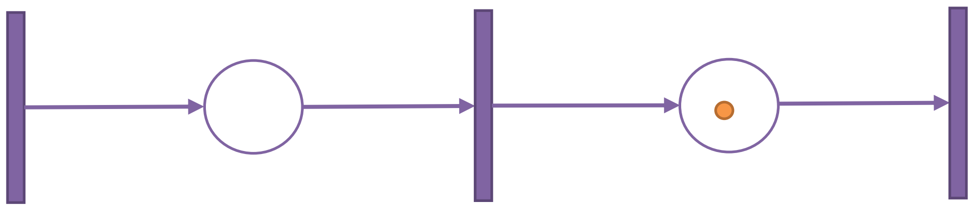


Model distributed systems, discrete events dynamic systems

Petri nets - definition

A directed bipartite graph

- **Transitions** triggered by **events** (*bars*)
- **Places**, i.e., conditions (*circles*)
- *Arcs* connect only places to transitions (or vice-versa), and specify which places are **pre- or post-conditions** for events
- Every place collects **tokens** (*dots*) which might trigger an event (if multiple events are triggered in the same net, which fires first is non-deterministic



Model distributed systems, discrete events dynamic systems

(Marking) Petri net - formalism

A tuple (S, T, W, M_0)

- S: finite set of states
- T: finite set of transitions
- $W: (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ multiset of arcs
- M: (marking) a mapping $S \rightarrow \mathbb{N}$ that assigns to each place a number of tokens
- M_0: initial marking

Subject to:

- S and T are disjoint
- No arc can connect two states or two transitions among them

How they execute

- firing a transition t in a marking M consumes $W(s, t)$ tokens from each of its input places, and produces $W(t, s)$ tokens in each of its output places
- a transition is *enabled* (it may fire) in M if there are enough tokens in its input places for the consumptions to be possible, i.e. if and only if $\forall s : M(s) \geq W(s, t)$

How to run the examples

Let's
code!

- Find them in `Code/` folder from the course website

For C++: compile

- `$ gcc code.cpp -o code -Wall -lstdc++`

Run (Unix/Linux)

```
$ ./code
```

Run (Win/Cygwin)

```
$ ./code.exe
```

References

Course website

- http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- Alessandro Fantechi, «Informatica Industriale», Città Studi Edizioni
- For interrupts
  - Robert Love, «Linux kernel development», Pearson
- For GNU Bison
  - http://dinosaur.compilertools.net/bison/bison_5.html
- A "small blog"
  - http://www.google.com