Dpt IMA
Informatique Fondamentale S8

Laure Gonnord et Julien Forget
`http://laure.gonnord.org/pro`

Informatique Fondamentale - IMA S8 - Mars - Avril 2001

Contenu :

- Transparents du cours 1 : automates finis, expressions régulières, notion de langage.
- Transparents du cours 2 : automates à piles, grammaires et langages hors contexte.
- Transparents du cours 3 : automates à compteurs, programmes, machines de turing et décidabilité

Enseignants S8 :

- Cours et TD : Laure Gonnord
- Tutorat : Laure Gonnord et Julien Forget

Emails des enseignants : Prenom.Nom@polytech-lille.fr

Informatique Fondamentale IMA S8

Cours 1 - Intro + schedule + finite state machines

Laure Gonnord
`http://laure.gonnord.org/pro/teaching/`
`Laure.Gonnord@polytech-lille.fr`

Université Lille 1 - Polytech Lille

March/April 2011

Until now

During S5,S6,S7, IMA students have learnt :

- (programming stuff) C programing and compiling,
- (conception stuff) algorithm design and encoding,
- (microelec) circuits and embedded systems hardware,
- (auto) design flows of industrial processes

▶ Solved (computation) *problems* by ad-hoc solutions.

Until now 2/2

But :

- No general scheme to design algorithms.
- (manual) evaluation of the *cost* of our programs.
- worse no assurance of correctness.
- even worse it there always a solution ?

▶ All these problems will be addressed in this course

Schedule

- Finite state machines (regular automata), regular languages. Notion of non determinism. Link with circuits.
- I/O automata, stack automata and grammars. Link to “simple” languages.
- Counter automata, Turing machines and undecidable problems. Link to “classical” programs.
- Graphs and classical problems/algorithms on graphs.
- Compiler Construction : front end + classical static analysis.
- Compiler Construction : code generation + classical dynamic analysis.

I - Regular languages and automata

What for ?

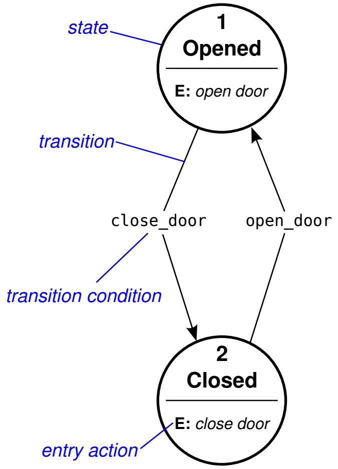
We want to model :

- the behaviour of systems with behavioural **modes**.
- the behaviour of (Boolean) circuits.
- sets of words.

▶ A finite representation.

Example

Opening/Closing door - Source Wikipedia.

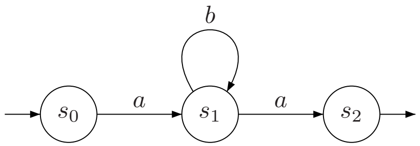


General definition

Finite state machine (FSM) or **regular automata**

- States are labeled, and there are finitely many ($s \in Q$)
- Initial state(s) ($i \in I$) and accepting (terminating/finishing) states ($t \in F$)
- Transitions are finitely many.
- Transitions are labeled with **letters** ($a \in A$).

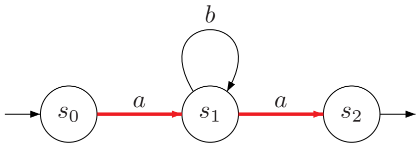
▶ The transition function is $\delta : Q \times A \to Q$.



Accepted language 1/2

Accepted word

A **word** w on the alphabet A is **accepted** by the automaton iff there exists a **finite** path from an initial state to an accepting state which is labeled by w.



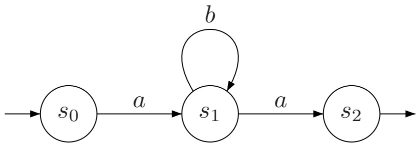
▶ $w = aa$ is accepted/recognised.

Accepted language 2/2

Accepted language

The **accepted language** of a given automaton $\mathcal{A}$ is the set of all accepted words and is denoted by $\mathcal{L}(\mathcal{A})$.

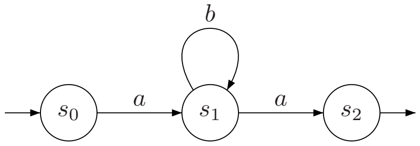
▶ Remark : it can be infinite.



▶ $\mathcal{L}(\mathcal{A}) = \{ab^k a, k \in \mathbb{N}\}$.

Data Structure for implementation

Problem : how to **encode** an automata ?



The transition function can be (for instance) encoded as a **transition table** :

	s_0	s_1	s_2
s_0	$--$	a	$--$
s_1	$--$	b	a
s_2	$--$	$--$	$--$

Goal

Problem : how to describe languages easily in a textual “linear” way ?

▶ use **regular expressions**.

Regular expressions

Regular expression (recursive def)

A **regular expression** e on the alphabet A is defined by induction. It can be of any of the following kinds :

- the empty word ε
- a letter $a \in A$
- a choice between an expression e_1 and another expression e_2 : $e_1 + e_2$
- two successive expressions : $e_1 \cdot e_2$
- 0,1 or more successive occurrences of e_1 : e_1^*.

Example with $A = \{a, b, c, d\}$: $e = a \cdot (b + c \cdot d)^*$

Regular expression vs word

A regular expression “encodes” the form of a word. For instance :

$$i \cdot m \cdot a \cdot (3 + 4 + 5)$$

(on the alphabet $\{i, m, a, 3, 4, 5\}$) describes all words beginning by the prefix “ima” and finishing by one of the numbers $3, 4$ or 5.

▶ A regular expression describes a **language**

Regular language

Regular language

Given a regular expression e, $\mathcal{L}(e)$ denotes the set of words (the **language**) that are described by the regular expression e.

Example with $A = \{0, \ldots, 9\}$:
$e = (1 + 2 + \ldots + 9) \cdot (0 + 1 + 2 + \ldots + 9)^*$. What is $\mathcal{L}(e)$?

Linux world

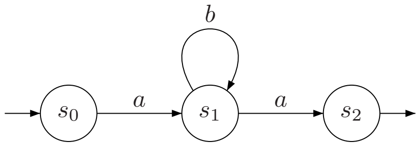
Some commands use (extended) regular expressions (regexp) :

- `ls *.pdf` lists all pdfs of the current directory
- `grep ta*.* *.c` find all lines in .c files that contains words that begin with t + some a’s.
- `sed ’’s ta*.*/toto/g’’` file replace all occurrences...

Relationship between automata and languages - 1/3

First, some experiments.

Given the following automata $\mathcal{A}$, are you able to give a regular expression e such that $\mathcal{L}(e) = \mathcal{L}(\mathcal{A})$?



- $e = ?$

Relationship between automata and languages - 2/3

And the converse :

Given the following regular expression $e = b^* \cdot (c + a)^*$, are you able to give an automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(e)$?

Relationship between automata and languages - 3/3

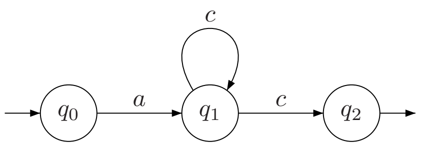
General result - Kleene Theorem

The regular languages are exactly the languages that are described by finite automata.

- What we've done before is always possible.

Goal

Sometimes some info lacks to make a choice between two transitions :



- From state q_1, there is a **non deterministic choice** while reading c : either go to state q_2 or stay in q_1.

Definition

Non deterministic FSM

- A deterministic automaton is $\mathcal{A} =< A, Q, I, F, \delta >$ with

$$\delta : A \times Q \to Q$$

- A **non deterministic** automata is the same with :

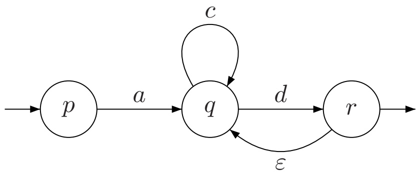
$$\delta : A \times Q \to P(Q)$$

- A **non deterministic** automata with ε-transitions is the same with

$$\delta : (A \cup \{\varepsilon\}) \times A \to P(Q)$$

Example

A non deterministic automata with ε-transitions :



- Then $\mathcal{L}(\mathcal{A}) = a \cdot (c^* \cdot d)^*$.

Find the associated language

Goal : Given $\mathcal{A}$ an automaton, find the associated language.

- Exercises !

Construction of automaton from a regular expression

Goal : Given a regular expression, construct a regular automaton that "**recognises**" it.

- Exercises !

Determinisation

Goal : transform a non deterministic automaton into a deterministic one.

▶ Exercises !

Other Algorithms

In the literature you will easily find :

- algorithms to eliminate ε transitions without determinising (ε closure) ;
- algorithms to minimise automata (the number of states) ;
- algorithms to use automata to find words in a text ;
- algorithms to test language inclusion (if they are regular)
- . . .

Non regular languages

Important result

There exists some non-regular languages.

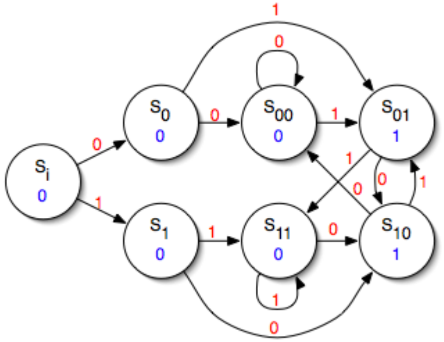
Examples of non-regular languages :

- $\{a^n b^n, n \in N\}$.
- palindromes on a non-singleton alphabet.
- $\{a^p, p$ prime$\}$

▶ There exists a quite systematic way to prove that a given language is not regular (Pumping Lemma).

Moore and Mealy Machines - 1

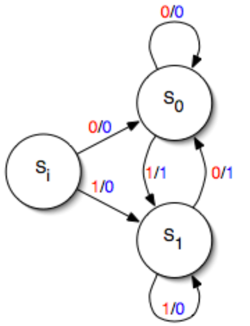
Moore : I/O machine whose output values are determined solely by the current state :



source Wikipedia

Moore and Mealy Machines - 2

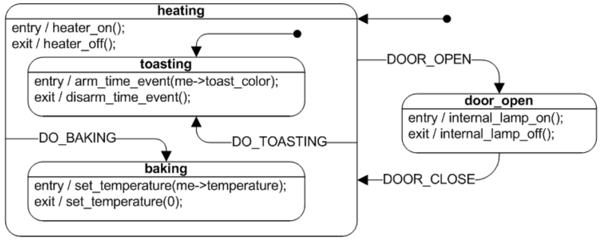
Mealy : output values are determined both by the current state and the value of the input.



source Wikipedia

UML Implementation of FSMs

UML variants of FSMs are **hierarchical**, react to messages and call functions.



source Wikipedia

Hardware Implementation of FSMs

It requires :

- a register to store state variables
- a block of combinational logic for the state transition
- (optional) a block of combinatorial logic for the output

Summary

Regular Automata or **Finite State Machines** are :

- Acceptors for regular languages. But some languages are not regular.
- Algorithmically efficient.
- Useful to describe (simple) behaviours of systems.
- Closely linked to circuits.

Informatique Fondamentale IMA S8

Cours 2 - Stack automata + Grammars

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

March/April 2011

II - Stack automata and Grammars

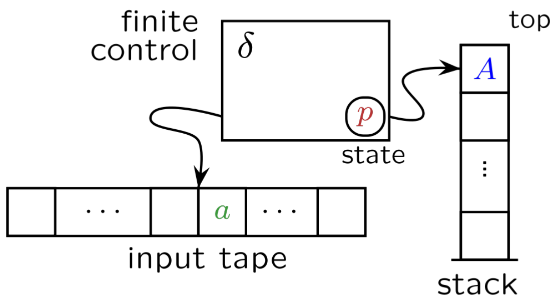
What for ?

Express more than regular languages !

- Give a way for automata to “**count**”, to “have a memory”.

Example

(source : Wikipedia)



General definition

Stack/P automata

- States (Q), initial state (q_0), finite states (F).
- Two alphabets (one for read : Σ one for stack : Γ)
- $\gamma_0 \in \Gamma$ the end of stack character.
- Transitions are finitely many.
- A **stack** to write into.
- Transitions use the stack.

▶ The transition function is :

$$Q \times (\Sigma \cup \varepsilon) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$$

How it works ? 1/3

Configuration

A **configuration** is a triple (q, w, α) where :

- q is the current state
- $w \in \Sigma^*$ the part of the input tape which is not yet read
- $\alpha \in \Gamma^*$ the current stack word

Two different conditions for accepting words :

- “**accepting state**” : the read word leads to a configuration of the form (q, ε, α) where $q \in F$ (with any α)
- “**empty stack**” : ... $(q, \varepsilon, \gamma_0)$ (with any q, but γ_0 is the “empty stack” character.

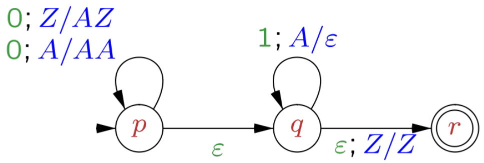
How it works ? 2/3

Given a configuration (q, w, α), the next one is computed by the help of the transition function $Q \times (\Sigma \cup \varepsilon) \times \Gamma \to \mathcal{P}(Q \times \Sigma^*)$:

- **enabled transitions** : those of the form $(q, a, A) \to (q', A')$ where a is the next letter to read on the input tape (or ε), and A the letter on top of the stack : $w = aw'$ and $\alpha = A\alpha'$.
- Pick one enabled transition, and compute the next configuration $(q', w', A'\alpha')$.

How it works ? 3/3

(source : wikipedia)



- Try to derive 0011 and 00111 !
- The PDA recognises $\{0^n1^n|n \in \mathbb{N}\}$ by accepting state.

Important theoretical results on PDA

- Important Stack automata are non deterministic !
- The two acceptance criteria define the same class of languages

---

Goal

Problem : Express languages with the same expressivity as stack automata.

- use **grammars**

General grammars

Grammar rule

A **grammar** rule (production rule) is of the form

$$w \longrightarrow w'$$

where w and w' are words.

A grammar is a set of rules.

Grammars

Grammar

A **grammar** is composed of :

- A finite set N of non terminal symbols
- A finite set Σ of terminal symbols (disjoint from N)
- A finite set of production rules, each rule of the form $w \to w'$ where w is a word on $\Sigma \cup N$ with at least one letter of N. w' is a word on $\Sigma \cup N$.
- A start symbol $S \in N$.

Grammars

Example :

$$S \to aSb$$

$$S \to \varepsilon$$

is a grammar with $N =$ .... and ....

Associated Language

Derivation

G a grammar defines the relation :

$$x \Rightarrow_G y \text{ iff } \exists u, v, p, q x = upv \text{ and } y = uqv \text{ and } (p \to q) \in P$$

► A grammar describes a **language** (the set of words on Σ that can be derived from the start symbol).

Examples

$$S \to aSb$$

$$S \to \varepsilon$$

The grammar defines the language $\{a^n b^n, n \in \mathbb{N}\}$

$$S \to aBSc$$

$$S \to abc$$

$$Ba \to aB$$

$$Bb \to bb$$

The grammar defines the language $\{a^n b^n c^n, n \in \mathbb{N}\}$

- Exercises

Context-free grammars

Context-free grammar

A **CF-grammar** is a grammar where all production rules are of the form $N \to (\Sigma \cup N)^*$.

Example of CF-grammar

$$S \to S + S | S * S | a$$

The grammar defines a language of arithmetical expressions.

- Notion of **derivation tree**.
Draw a derivation tree of a*a+a, of S+S !

Relationship between stack automata and grammars

General result

The context-free/algebraic languages are exactly the languages that are described by stack automata.

- The proof is not difficult.
- Exercises

Some other results/definitions

- Regular languages are algebraic languages, but not the converse.
- There exists normal forms for algebraic grammars
- A grammar can be ambiguous.

Summary

Stack Automata or **PushDown Automata** are :

- Acceptors for context-free languages. But some languages are not context free.
- Non deterministic.

▶ `http://en.wikipedia.org/wiki/Pushdown_automaton` for useful pointers

Informatique Fondamentale IMA S8

Cours 3: Counter automata, Turing Machines and decidable problems

Laure Gonnord
`http://laure.gonnord.org/pro/teaching/`
`Laure.Gonnord@polytech-lille.fr`

Université Lille 1 - Polytech Lille

March/April 2011

III - Counter automata, Turing Machines and decidable problems

---

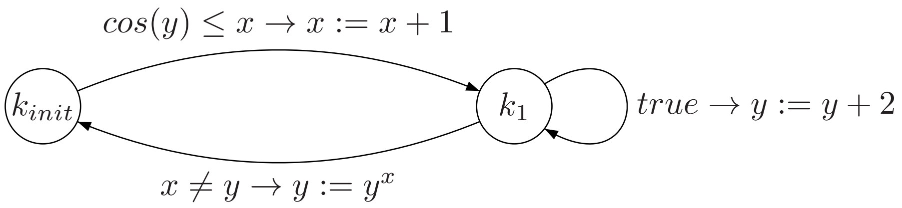
1. Counter automata
2. Programs
3. Turing Machines
4. Decidability - Complexity

What for ?

Express more than context-free languages !

- Give a way for automata to “**count** more” : include **variables**.

Example



General definition

Counter Automata

- A finite number of **counters** (variables)
- A finite number of **control points**
- Transitions between them that operate on counters.

▶ The transition function is of the form $g \rightarrow a$ (g is the **guard**, a is the **action**)

How it works ? 1/3

State

A **state** is (q, σ)

- q is the current control point
- $\sigma : Var \rightarrow Val$ is a function that assigns a value (a real one or $\perp$) to all counters.

▶ Non deterministic/No notion of acceptance/Notion of **reachability**.

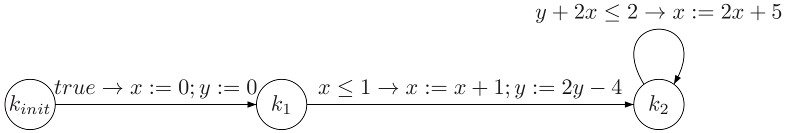
How it works ? 2/3

Given a state (q, σ), the next one is computed by the help of the transition function :

- **enabled transitions** are transition of the current control points where the current valuations of variables satisfy the guard.
- Pick one enabled transition, and compute the next state : (q', σ'). The new values for the variables are computed w.r.t. the action.

How it works ? 3/3

Example of an affine automata :

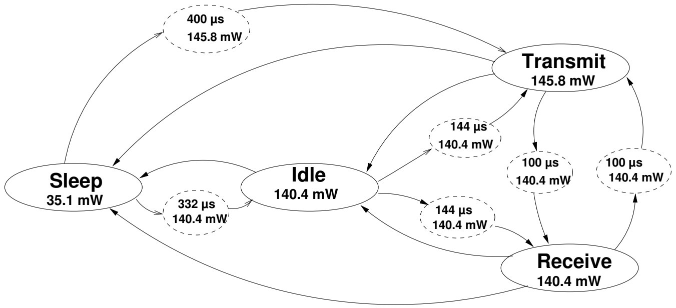


- ▶ Compute successive states.
- ▶ Exercises.

What kind of systems ?

These automata are used to encode, for example :

- Simple systems (coffee machine, ...) **specifications**
- Energy consumption of sensors :



What for ?

Some classical problems :

- Automatically finding invariants
- **Reachability** analysis
- Formula proving.
- (deterministic) Code generation.

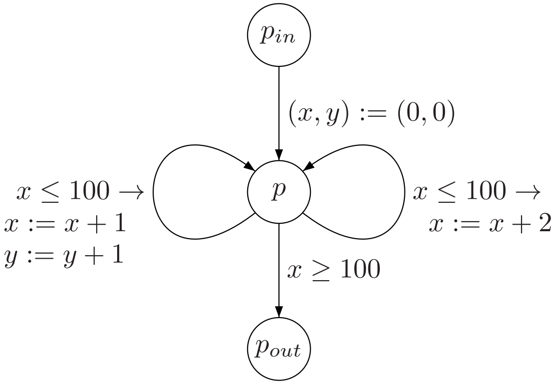
---

1. Counter automata
2. **Programs**
3. Turing Machines
4. Decidability - Complexity

Expressing programs as counter automata

An **example** :

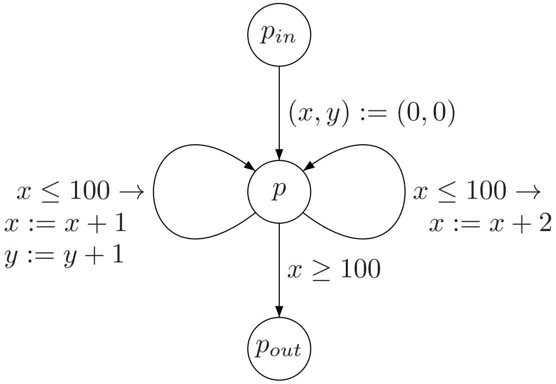
```
x:=0;y:=0
while (x<=100) do
  read(b);
  if b then
         x:=x+2
       else begin
         x:=x+1;
         y:=y+1;
       end;
  endif
endwhile
```



- ▶ Some approximations are made (arrays, Boolean, . . .)

Expressing properties of programs

Some (**safety**) properties of programs can be expressed inside the counter automaton :



- ▶ Encode the fact that $state = p_{out} \wedge y > 100$ is “bad”
- ▶ Some modifications of the automaton can be automatically done.

Expressing properties of programs - 2

Automatically deriving bad states :

```
int j;
char user[USERSZ];

for(j = 0; line[j] != EOS; ++j)
  if (!strchr("-", line[j]))
    break;

  if(j == J && line[j] == ’ ’) { /* long list */
    /* BUG! No bounds check. */
     assert(USERSZ>=N-j,"badstate");

     r_strcpy (user, line + j);
  }
```

- ▶ “badstate” is reachable.

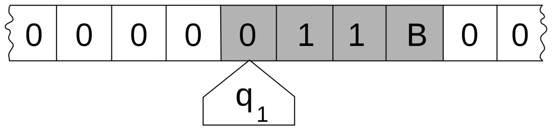
Expressing programs as counter automata

- **Pros** : finding and proving program properties
- **Cons** : how do we define the operations ? the fact that integers are encoded on bits ?

- ▶ There is a need for a more accurate (not specialised) model !

1. Counter automata
2. Programs
3. **Turing Machines**
4. Decidability - Complexity

Turing Machine, def - 1



Turing Machine, Turing, 1935

- A **tape** (infinite in both sides) : the memory. (it has a working alphabet which contains a **blank** symbol).
- A **head** : read/write symbols on the tape.
- A finite transition table (or function)

▶ A PushDown automaton which is more flexible.

image credits : Wikipedia.org

Turing Machine, def - 2

Turing Machine elements

- States (Q), initial state (q_0), final (accepting) states (F).
- One alphabet Γ for the tape.
- $b \in \Gamma$ the blank letter.
- Transitions are finitely many : read the letter under the head, and then :
  - Erase a symbol or write one under the head
  - Move the head (read, write, or stays)
  - The “state” can be modified.

▶ The transition function is :

$$Q \times \Gamma \times \to Q \times \Gamma \times \{L, R, S\}$$

An Example

TM that decides if x is even : (final State : q_4)

State/char	0	1	B
q_1	(q_1,0,R)	(q_1,1,R)	(q_2,B,L)
q_2	(q_3,B,L)	(q_4,B,L)	—

Play on $BBBBBBB11BBBBB$ and $BBBBBBB10BBBBB$

▶ A **configuration** is a tuple (word on the tape, position of the head, state).

Adapted from : `http://www.madchat.fr/coding/algo/algo_epfl.pdf` slide 4

Another Example

Demo of a TM recognising a language : $a^n b^n c^n$

Program found here :

`http://www.cs.columbia.edu/~zeph/software/BJDweck/`

General results 1/2

There exists Turing machines for the following languages :

- palindromes
- $a^n b^n c^n$ (non algebraic language)
- a^i with i prime (non algebraic)
- a^{n^2}, with $n \geqslant 0$

▶ Turing machines are more powerful than all other models (we have seen yet)

Decidable Languages

A language that is recognised by a Turing Machine is said to be decidable.

Accepting or computing

A TM can also compute functions

TM that writes 1 if x is even, 0 else (q_6 is final state) :

State/char	0	1	B
q_1	(q_1,0,R)	(q_1,1,R)	(q_2,B,L)
q_2	(q_3,B,L)	(q_4,B,L)	—
q_3	(q_3,B,L)	(q_3,B,L)	(q_5,1,R)
q_4	(q_4,B,L)	(q_4,B,L)	(q_5,0,R)
q_5	—	—	(q_6,B,R)

Play on $BBBBBBB11BBBBB$ and $BBBBBBB10BBBBB$

Another Example

Demo of a TM computing the subtraction.

General results 2/2

There exists Turing machines for the computation of :

- $x \mapsto x+1$
- $(x,y) \mapsto x+y$
- $x \mapsto x^2$
- all the functions you are able to write on computers

Computable functions

A function (defined for all its input) that is computable by a Turing Machine is said to be TM computable

▶ A Model of what can be computed with machines. (Church Thesis)

Decidable - Semi-decidable languages

If L is a language, L is **decidable** if there exists a Turing Machine (or an algorithm) that outputs for all w :

- 1 if $w \in L$
- else 0.

Semi-decidable :

- 1 if $w \in L$
- else does not terminate

▶ equivalent definition for problems.

Complexity

Link with computational complexity :

- The number of steps in the execution of a TM gives the **complexity in time**,
- The number of seen squares in the execution of a TM gives the **complexity in space**.

Examples of undecidable problems

- The halting problem for TM or counter automata (for more that 2 counters)
- Given a program, does it loops ?
- Is a given algebraic expression (with log, *, exp, sin, abs) equal to 0 ? (Richardson, 1968)
- The 10^{th} Hilbert Problem (Diophantine equations)

Summary

Turing Machines :

- A model closed to **programming languages**.
- Non deterministic.
- Acceptors for decidable languages. But some languages are not decidable !
- (equivalently) Computes solutions to problems. But ...

Important fact

Some common problems are undecidable !

Counter automata :

- are a more simpler model
- have the same power of expression as Turing Machines.
- Reachability is **undecidable** too. But we can do approximations.