Decidability / Undecidability

To discuss decidability/undecidability we need Turing-machines and to discuss Turing-machines we need formal languages, and strings and alphabets. And a bit more…

Our Turing machines are defined as M = (Q,Σ,Γ,δ):

- Q - the states (includes start- and stop state),
- Σ - input alphabet,
- Γ - tape symbols (includes b [blank]),
- δ - the transition function.

Turing-machines work on the tape string. When the machine starts input is written with the alphabet Σ, when it stops output is written with the alphabet Γ. (Σ must therefore be a subset of Γ, input can be written on the tape.)

Turing-machine: http://www.youtube.com/watch?v=E3keLeMwfHY

Decidability / Undecidability

Turing machine (general)

input: string written in the alphabet Σ (the input alphabet),
output: string written in the alphabet Γ (the tape alphabet).

So a Turing-machine M is really a function from strings in Σ* to strings in Γ*, we write:

$$M: \Sigma^* \rightarrow \Gamma^*$$

Which means something like output = M(input) with input$\in\Sigma^*$, output$\in\Gamma^*$.

We often want our Turing-machines to answer only YES or NO (Y/N):

Turing-machine (YES/NO)

input: string written in the alphabet Σ (the input alphabet),
output: Y or N, Y and N must then be in Γ (the tape alphabet).

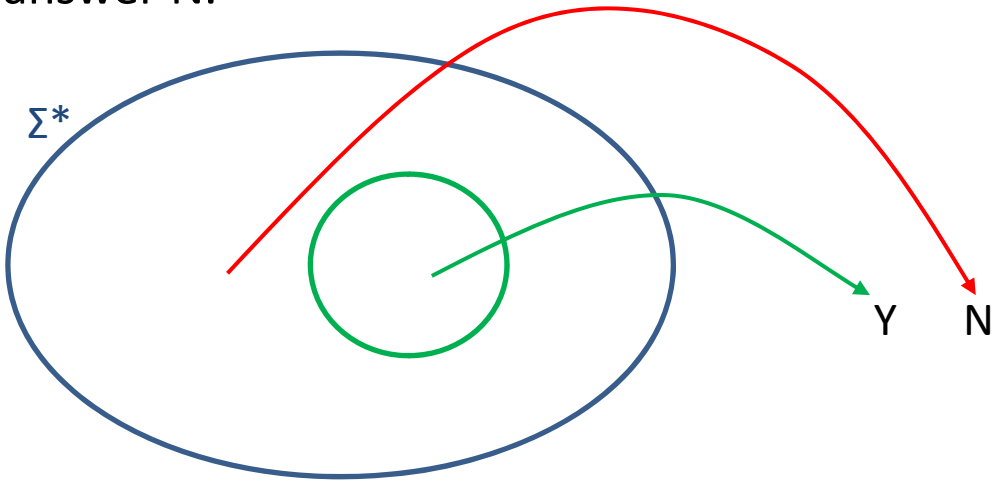
$$M: \Sigma^* \rightarrow \{Y,N\}$$

Decidability / Undecidability

We now limit ourselves to descision problems (answer YES or NEI).

When we use Turing machines to solve decision problems, we really calculate functions of the type $M: \Sigma^* \rightarrow \{Y,N\}$.

Then ther will be some strings in Σ^* that give the answer Y, and some that give the answer N.



- Formal language
- YES-instances (and NO-)
- Decision problem
- Decide

Decidability / Undecidability

- A formal language L is a set of strings (over a given alphabet).
- The strings (usually) have interpretation, e.g. Hamiltonian graphs.
- Formal languages correspond to problems, answering whether or not a given string is *in* the language.

Looking at the example with Hamiltonian graphs:

- The formal language consists of all (strings describing) Hamiltonian graphs,
- the corresponding decision problem is deciding whether or not a given graph is Hamiltonian (answer YES or NO),
- And the language can be decided by a Turing- machine (probably not in polynomial time, though).

The strings (the input) we answer YES to are called YES-instances (or positive instances), the ones we answer NO to for NO instances (negative). The formal language defines what are YES-instances for the corresponding problem; the rest of the strings possible to make with the input alphabet become NO-instances.

3b

Is the following language decidable?

$L_1 = \Sigma^*$ (where Σ^* is the set of all possible strings over the alphabet Σ).

We have to decide whether or not our input string is made only with symbols from Σ. (I'm assuming Σ is something other than the input alphabet – then all strings would be legal; but that *is* a possible interpretation, just answer YES.)

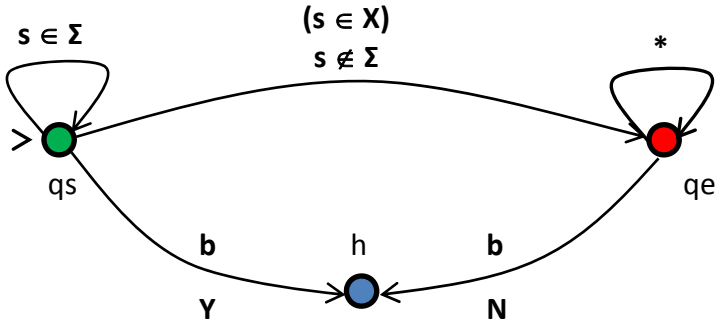
```
PROC check (string S)
{
   BOOLEAN isOK = TRUE
   FOR <all s in S> DO
   {
      IF <s ∉ Sigma> THEN
         isOK = FALSE
   }
   RETURN(isOK)
}
```

L_1 is decidable.

ML1 = (Q,Σ',Γ,δ)

Σ' = Σ U X (the machine input alphabet Σ' is Σ U X , where X are the illegal lsymbols)
Γ = Σ' U {b} (we can also write blank on the tape, but need no more symbols here)
Q = {qs, qe, h} (three states are enough, the start state s is also the OK-state.)

δ =	(qs, s ∈ Σ) → (qs,b,r)	Read symbol in Σ, stay in qs (OK), overwrite with blank, go to next.
	(qs, s ∉ Σ) → (qe,b,r)	Read symbol not in Σ, go to qe (not OK), overwrite, og to next.
	(qs, b) → (h,Y,-)	Reached the end (read blank), is in qs, stop with YES.
	(qe, s ∈ Σ') → (qe,b,r)	Whatever we read (except b) we stay in qe when we have error.
	(qe, b) → (h,N,-)	Reached end with error, stop with NO.



3b

Is the following language decidable?

$L_2 = \{M \mid M \text{ decides } L_1\}$.

This means that we have a language L_2 consisting of all machines for L_1 (that we just showed decidable).

L_2 is *not* decidable. Looking t Turing-machines as functions it is generally not possible to answer questions about the function (here: whether the function decides L_1).

We use the standard reduction from HALTING and tansform a HALTING-instance (M,x) to an L_2-instance (M') in the following manner:

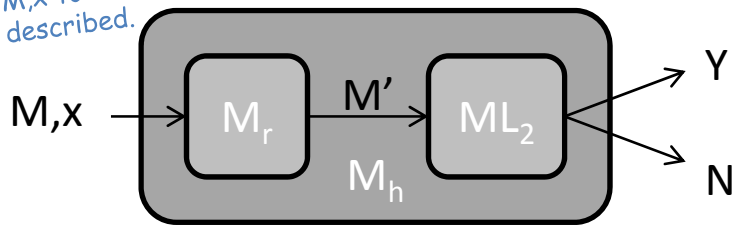
	HALTING	⟶	**L_2**
input:	M,x		M'

```
MACHINE M'(input)
{
    simulate M on x
    ML1(input)
}
```

M' is now an L_2-instance, made such that:

- M' contains a hard-coded simulation of M on x. We do not care about the output from this simulation and it does not modify the input to M'.
- After the simulation we copy in the code for ML1 (the machine we just made for L_1) which then works on the input to M'.

M_r does the transformation from M,x to M', that is, builds M' as described.



3b

So why does this work? What have we done

We have shown how to translate all HALTING-instances (all possible HALTING-input) to L_2-instances (by making M'). The Transformation translates YES-instances to YES-instances, and NO-instances to NO-instances, so that the answer we get from a machine for L_2 will be the same as from one for HALTING (on corresponding instances).

That a YES-instance for HALTING becomes a YES-instance for L_2 is easy to see: the simulation (of M on x) will stop, and M' *is* practically then ML1 (a machine for L_1).

That a NO-instance for HALTING becomes a NO-instance for L_2 is even easier: the simulation never stops, and we never get to the ML1-part, M' runs in an infinite loop and isn't a machine for anything (other than looping…).

To be able to answer whether or not a machine (of type M') is a machine for **L_1**, we have to be able to answer whether it halt or not. We can't. **L_2** is therefore undecidable. (ML_2 in the schematic drawing of the reduction in the previous slide cannot exist, because M_h does not exist.)

But, can't we…?

No

Can't we just put «Simulate M on x» in front of anything helst and prove that it is undecidable? Even if it really is decidable??

No. That is, you can put «Simulate M on x» in front of whatever you like, but that does *not* that this whatever is undecidable. (Look at what we did with L_1 and L_2).

Let us take another example, of something decidable – checking whether a Turing machine has only *one* transition rule.

The machine (program) M_{ex} below can answer this:

```
MACHINE M_ex ( MACHINE M_i = (Q_i,Σ_i,Γ_i,δ_i) )
{
    IF | δ_i | = 1 THEN
        RETURN(Y)
    ELSE
        RETURN(N)
}
```

Checks if the machine M_i in input haa *one* transition rule.

But, can't we …?

No

And then just power along with the standard reduction…

	HALTING	⟶	$L_?$
input:	M,x		M'

```
MACHINE M' (M_i)
{
    simulate M på x
    M_ex (M_i)
}
```

- YES-instances of HALTING become machines M' that answer wheter or not the machine it gets as input has only one transition rule (by using the code for M_{ex}),
- NO-instances of HALTING become machines M' that never answer.

What languages really became decidable and undecidable here now?
And what machines belong to what languages?

Language	Machine	Decideability
L_{ex} = {M | M has *one* transition rule}	M_{ex} , and partly M'	Decidable (by M_{ex}).
$L_?$ = {M | M decides L_{ex}}	$M_?$ (hypothetical)	Undecidabe.

We get one more "level" when we use the reduction:

- We can answer whether a machine only has one transition rule – that is decidable.
- But we can *not* decide whether or not a machine answers whether or not another machine consists of only one transition rule – *that* is undecidable.

It is the language machine M' is an instance for that is undecidable. To see whether or not M' is a machine foe L_{ex} , we must be able to decide if it stops. It is not L_{ex} that suddenly becomes undecidable, but $L_?$ That consists of machines for L_{ex} . It is deciding whether something is a machine for L_{ex} that becomes undecidable, not L_{ex} itself.

3b

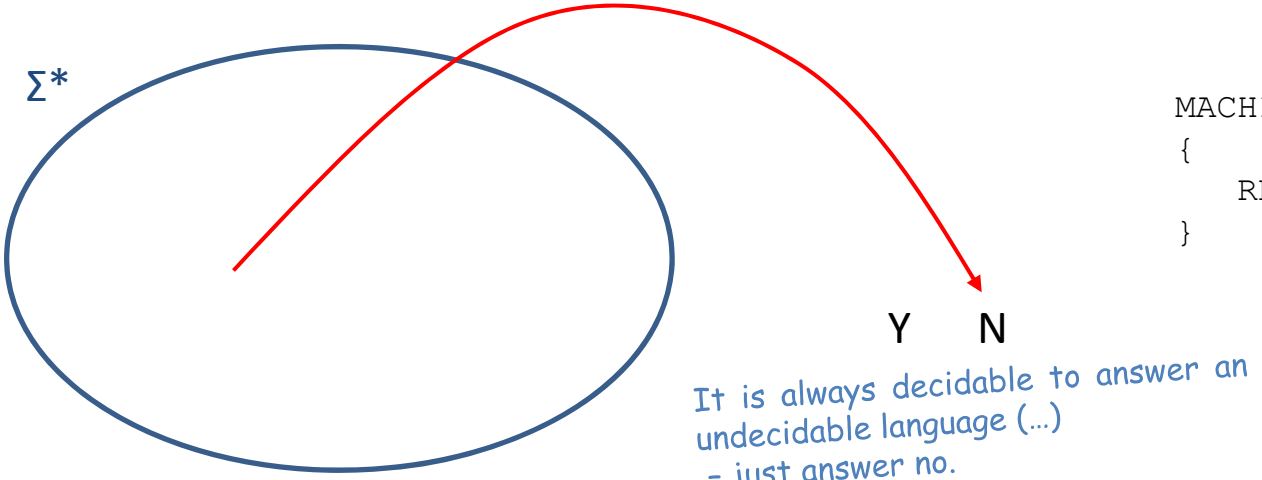
Is the following language decidable?

$L_3 = \{M \mid M \text{ decides } L_2\}$.

This means that we have a language L_3 that consists of all machines for L_2 (that we just showed undecidable).

L_3 ***is*** decidable. It is easy to design a machine for an undecidable language (When we know that the language is undecidable). An undecidable language has *no* machines. The formal language L_3 is empty $L_3 = \varnothing$, and can be decided by a machine that always answers NO.

If someone comes to you with a Turing machine and asks wheter or not it is a machine for HALTING, you can always say no. Similarly for **L_2**, you can always say no for all input. It is surely not a machine for **L_2** this person has.



It is always decidable to answer an undecidable language (...) - just answer no.

```
MACHINE ML3(M_i)
{
    RETURN(N)
}
```

ML3 decides L_3.

3b – alt. interpretation

If we assume that all input symbols are legal for L_1 (that the Σ in the exercise text really *is* the input alphabet to a Turing machine M = (Q,Σ,Γ,δ)), the whole exercise gets a nice symmetry…:

L_1: decidable

```
MACHINE ML1(S)
{
   RETURN(Y)
}
```

L_2: undecidable HALTING $\propto$ L_2 (as before)

L_3: decideble

```
MACHINE ML3(M)
{
   RETURN(N)
}
```