COMP1600, week 10:

Turing Machines

convenors: Dirk Pattinson, Pascal Bercher

lecturer: Pascal Bercher

slides based on those by: Dirk Pattinson

(with contributions by Victor Rivera and previous colleagues)

Semester 2, 2024

Australian
National
University

Overview of Week 10

- Introduction
- Turing Machines: Basic Definitions
- Examples for Turing Machines
- TMs vs. Programming and Common Idioms
- Languages of TMs and the Chomsky Hierarchy
- TM Expressivitiy, again
- Universal Turing Machines

Introduction

Computing as Profession

Photo c/o Early Office Museum Archives

(Curious? Search for "Computer (occupation)")

A model for 'computers'

Computing in 1936

- computers not machines, it was a *profession*.
- Alan Turing's contribution: mathematical *definition* of what computers can do (= of what computation is).
- In the 1936 paper "On computable numbers, with an application to the *Entscheidungsproblem*"
  - solved long standing open problem posed by D. Hilbert and W. Ackermann in 1928: "the Entscheidungsproblem" (German for Decision Problem)
  - changed the world (of mathematics, and computing)

A model for 'computers'

Computing in 1936

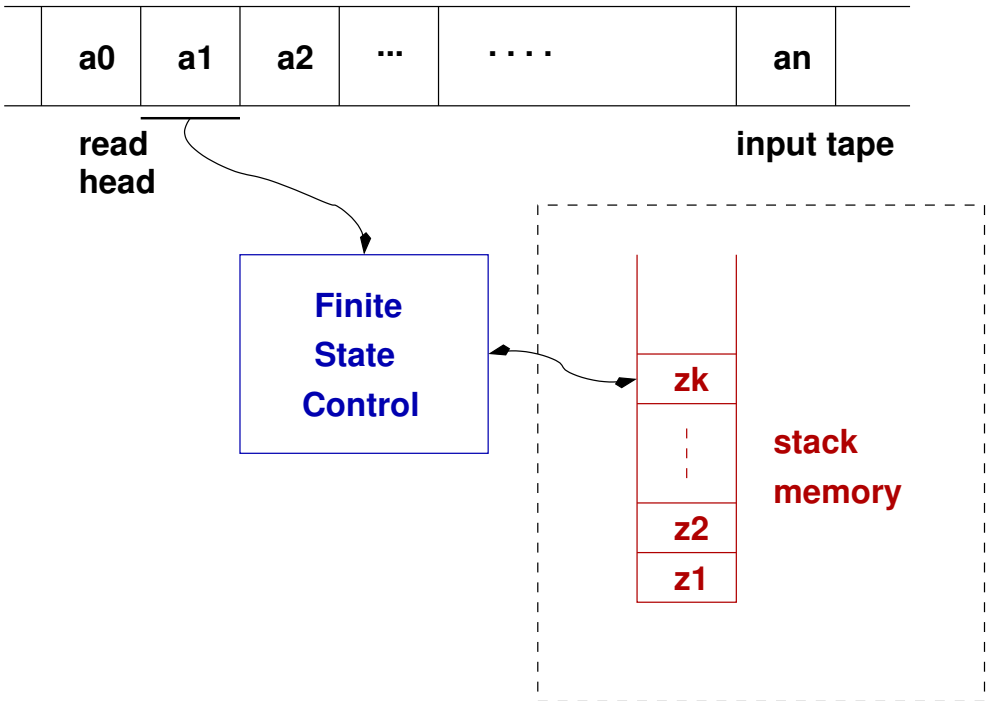
- computers not machines, it was a *profession*.
- Alan Turing's contribution: mathematical *definition* of what computers can do (= of what computation is).
- In the 1936 paper "On computable numbers, with an application to the *Entscheidungsproblem*"
  - solved long standing open problem posed by D. Hilbert and W. Ackermann in 1928: "the Entscheidungsproblem" (German for Decision Problem)
  - changed the world (of mathematics, and computing)

Turing's model today

- no computer has been built that is *more* powerful than Turing's model
- no other formalism was ever found to be more expressive
- Turing has discovered the *essence of computation*

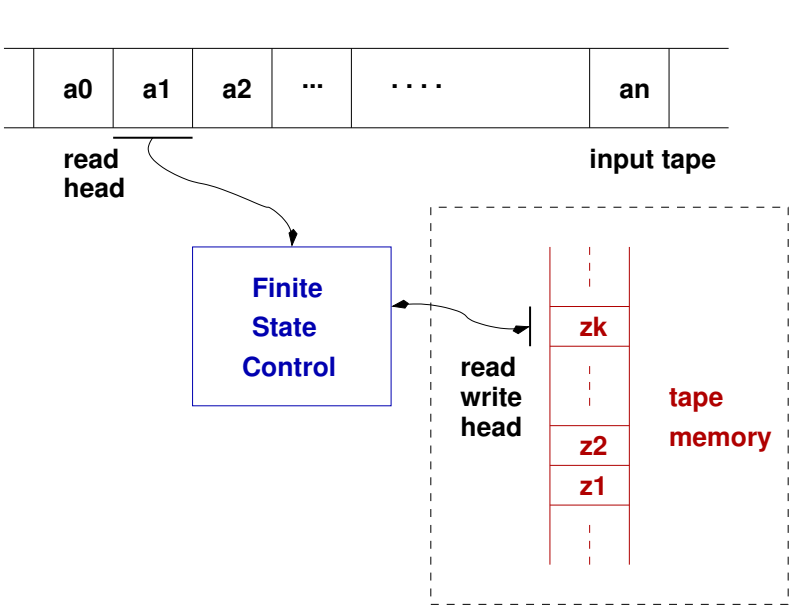
Turing Machines: Basic Definitions

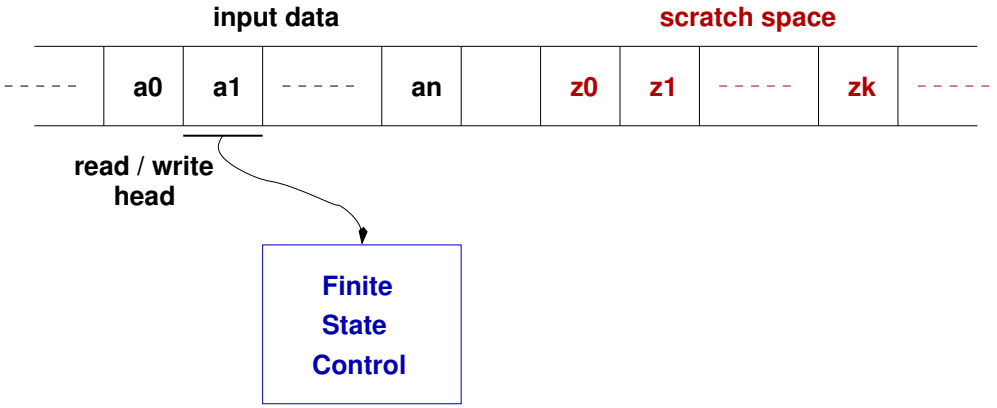
Push Down Automata, reloaded



A PDA with its auxiliary store is *almost* a whole computer, except we can only directly access the symbol on the top of the stack.

Turing Machines





Generalisation from PDA to Turing machine

- replace *stack memory* (previous slide) by *tape memory* (this slide)
- for simplicity, don't use separate tape memory (cf. left graphic), but re-use tape holding the input word (cf. right graphic)
- can access and change *arbitrary* symbols on tape by moving tape head

Fundamental Properties

Basic Properties/Definitions

- For now: Deterministic Turing Machines (we add non-determinism later)

Fundamental Properties

Basic Properties/Definitions

- For now: Deterministic Turing Machines (we add non-determinism later)
- If there is no transition possible in a state, the TM *halts*
  - Notice how for other machines that implied rejection
  - Here it could be both or neither, it just means we stop ("halt")

Fundamental Properties

Basic Properties/Definitions

- For now: Deterministic Turing Machines (we add non-determinism later)
- If there is no transition possible in a state, the TM *halts*
  - Notice how for other machines that implied rejection
  - Here it could be both or neither, it just means we stop ("halt")
- If a TM reaches a *final* state, it accepts the word (i.e., the original tape content). Notice that:
  - we don't require anymore that the entire word was "processed" (as it's not fed letter by letter anymore but simply written on the tape initially).
  - we don't require halting after a word was accepted

Fundamental Properties

Basic Properties/Definitions

- For now: Deterministic Turing Machines (we add non-determinism later)
- If there is no transition possible in a state, the TM *halts*
  - Notice how for other machines that implied rejection
  - Here it could be both or neither, it just means we stop ("halt")
- If a TM reaches a *final* state, it accepts the word (i.e., the original tape content). Notice that:
  - we don't require anymore that the entire word was "processed" (as it's not fed letter by letter anymore but simply written on the tape initially).
  - we don't require halting after a word was accepted
- A word is rejected iff it is not accepted. Note that this implies:
  - rejection can happen because it halts in a state without accepting the word
  - it rejects because it loops forever (without accepting it)

Fundamental Properties

Basic Properties/Definitions

- For now: Deterministic Turing Machines (we add non-determinism later)
- If there is no transition possible in a state, the TM *halts*
  - Notice how for other machines that implied rejection
  - Here it could be both or neither, it just means we stop ("halt")
- If a TM reaches a *final* state, it accepts the word (i.e., the original tape content). Notice that:
  - we don't require anymore that the entire word was "processed" (as it's not fed letter by letter anymore but simply written on the tape initially).
  - we don't require halting after a word was accepted
- A word is rejected iff it is not accepted. Note that this implies:
  - rejection can happen because it halts in a state without accepting the word
  - it rejects because it loops forever (without accepting it)

Language Definitions.

- As always, the language of a Turing Machine is the words accepted by it.
- Languages accepted by a Turing Machine are called *recursively enumerable*.

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts.

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.
- Others even feature rejecting states (similar to accepting states).

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.
- Others even feature rejecting states (similar to accepting states).
- Others have multiple tapes: some with one head globally, some per tape.

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.
- Others even feature rejecting states (similar to accepting states).
- Others have multiple tapes: some with one head globally, some per tape.
- Others have a semi-infinite tape (infinite only to one side).

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.
- Others even feature rejecting states (similar to accepting states).
- Others have multiple tapes: some with one head globally, some per tape.
- Others have a semi-infinite tape (infinite only to one side).
- Others can't let the head stop (it always must move left or right).

We stick with the definitions provided before (without specific reason).

Disclaimer

Many different definitions of TMs can be found, all of which are equivalent.
For example:

- It is common to assume that final states have no outgoing transitions. This would imply that the machine always halts after accepting a word. (We don't make this restriction, but when creating a TM, it makes sense!)
- Other definitions accept a word iff the Turing Machine halts..
- Even "rejection"can be defined differently. In some definitions, rejection implies termination (we do not define it that way). Then, acceptance and rejection are not "complete", i.e., there are more cases.
- Others even feature rejecting states (similar to accepting states).
- Others have multiple tapes: some with one head globally, some per tape.
- Others have a semi-infinite tape (infinite only to one side).
- Others can't let the head stop (it always must move left or right).

We stick with the definitions provided before (without specific reason).

Interested in equivalence proofs (of some of these alternative definitions)?
$\rightarrow$ Take Theory of Computation! (COMP3630/COMP6363)

Output

Turing Machines as Computing Devices

- TMs can calculate *any* computable function. (So we think.)
- Input: a string written onto the tape before the machine starts.
- Output: whatever is left on the tape when the machine halts.

Output

Turing Machines as Computing Devices

- TMs can calculate *any* computable function. (So we think.)
- Input: a string written onto the tape before the machine starts.
- Output: whatever is left on the tape when the machine halts.

Note that we usually never care what's written on the tape after accepting (i.e., the "output"). We only care for the language accepted by a Turing Machine.

Turing Machine – Formal Definition

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, \Lambda, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;

Turing Machine – Formal Definition

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, \Lambda, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;
- Γ is the set of **tape symbols** (everything that might ever be on the tape);
- $\Lambda \in \Gamma \setminus \Sigma$ is the **blank symbol**;

Turing Machine – Formal Definition

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, \Lambda, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;
- Γ is the set of **tape symbols** (everything that might ever be on the tape);
- $\Lambda \in \Gamma \setminus \Sigma$ is the **blank symbol**;
- $\Sigma \subseteq \Gamma$ is the set of **input symbols**;
- δ is a (partial) **transition function**

$$\begin{array}{rcl} \delta \ : \ S \times \Gamma & \rightharpoonup & S \times \Gamma \times \{L, R, S\} \\ (\text{state}, \text{tape symbol}) & \mapsto & (\text{new state}, \text{new tape symbol}, \text{direction}) \end{array}$$

The direction tells the read/write head which way to go next:
Left, Right, or Stay/Stop. (Stopping the head is different from halting.)

Running a TM

Initialisation.

- Some input (a finite string over Σ) is written on the tape;
- all other infinitely many tape cells are blank – Λ;
- the read/write head sits over the left-most cell of the input (or over any Λ if the input is ϵ);
- we start in the start state s_0.

Running a TM

Initialisation.

- Some input (a finite string over Σ) is written on the tape;
- all other infinitely many tape cells are blank – Λ;
- the read/write head sits over the left-most cell of the input (or over any Λ if the input is ϵ);
- we start in the start state s_0.

Running.

- In a cycle: read symbol and execute action (state-dependent): change state / write / move head
- Until a final state is reached (or the machine gets stuck). Wait, when can we stop?

Running a TM

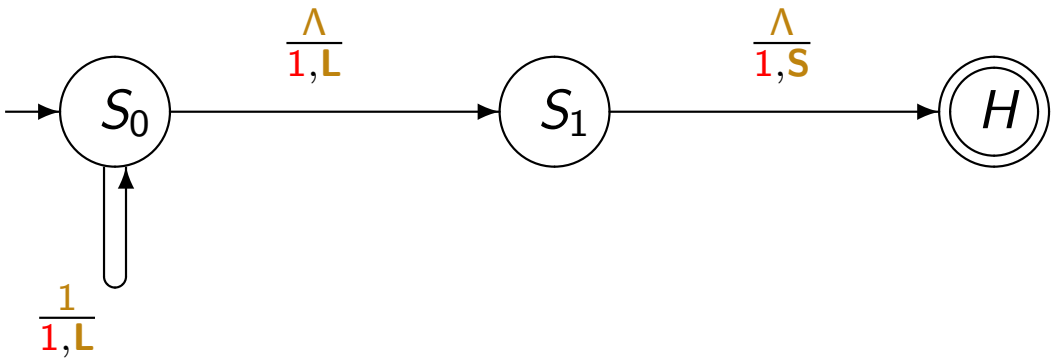
Initialisation.

- Some input (a finite string over Σ) is written on the tape;
- all other infinitely many tape cells are blank – Λ;
- the read/write head sits over the left-most cell of the input (or over any Λ if the input is ϵ);
- we start in the start state s_0.

Running.

- In a cycle: read symbol and execute action (state-dependent): change state / write / move head
- Until a final state is reached (or the machine gets stuck).
  Wait, when can we stop? Depends on what you want to do:
  - Compute some output? Stop once it halts.
  - Accept a word? Stop after acceptance. (Usually, that's what we want.)
    To clarify: the machine continues until it halts, but "your job" is done!

Graphical Representation of the Transition Function



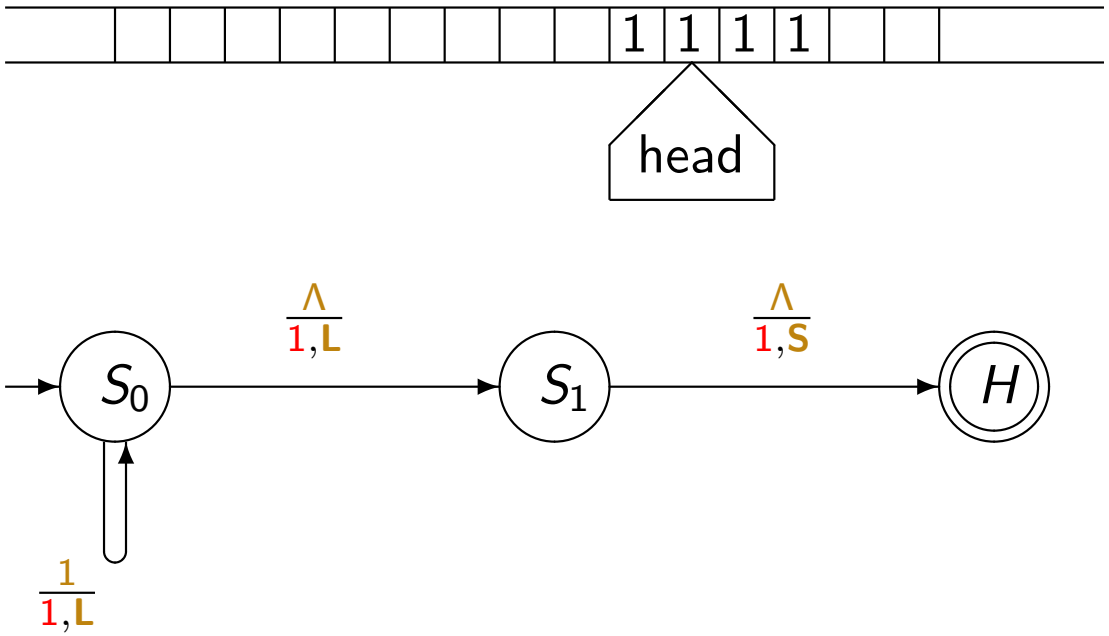
(Like in FSAs and PDAs, annotate transition edges with commands for accessing tape.)

Convention.

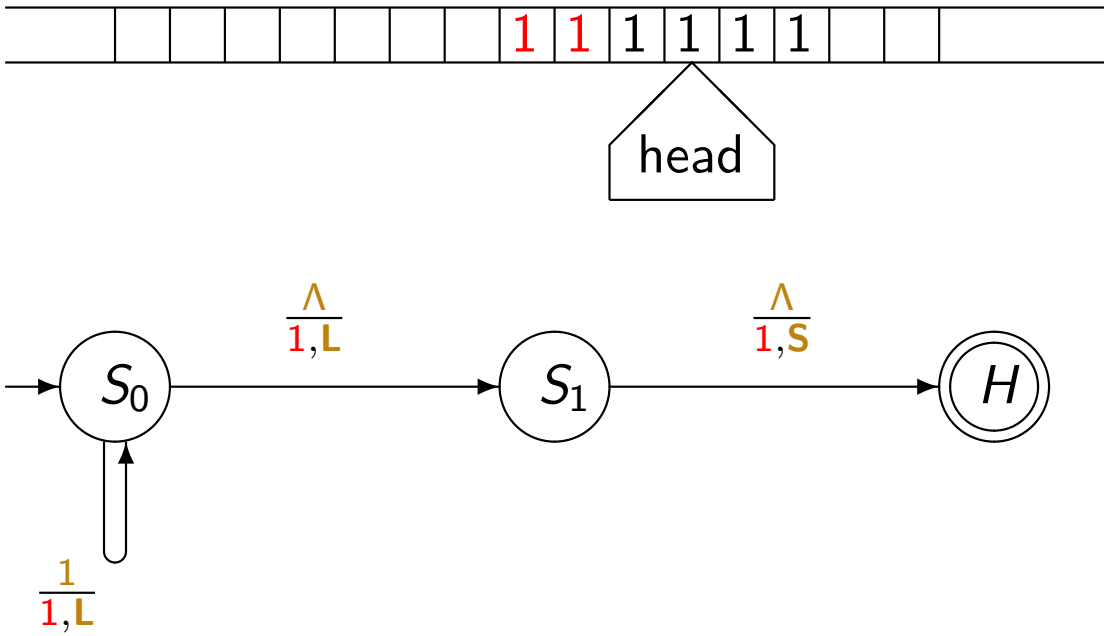
- Numerator: *symbol read from tape*.
  - Λ means the tape is blank at that position.
- Denominator: *symbol written* / *direction of head movement*.
  - direction one of L, R, S for Left, Right, Stay.
- There are also other conventions; but the meaning should always be clear.

Examples for Turing Machines

What does it do?

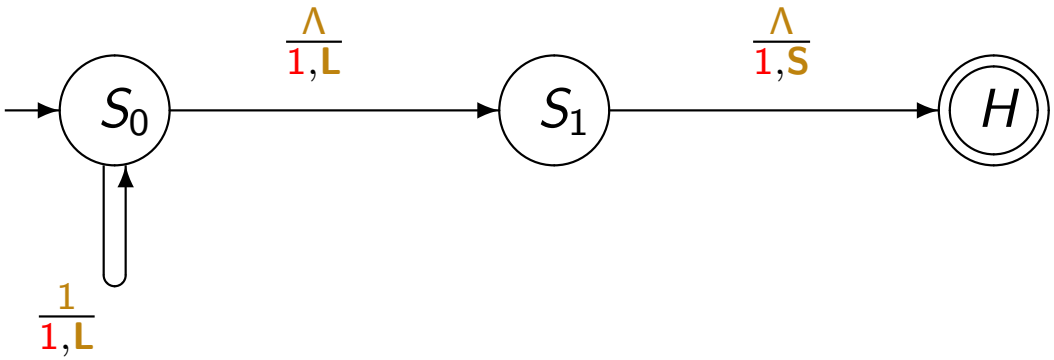


What does it do?



- Adds two to a unary number!
- Assume the head starts over the input data. (Usually we assume we start on the left-most symbol; this time we don't for the example!).
- First phase scans left.
- Second phase writes two extra *1*s.

The Convention for Errors



TMs getting Stuck

- suppose head starts right-most and contains a token other than 1.
- TM would halt in state S_0, as there is no arc telling us what to do if we meet such a token (this job would be done by a rightwards scan).
- this is an error – the input is *rejected*.
- S_0 *not* an accepting state!

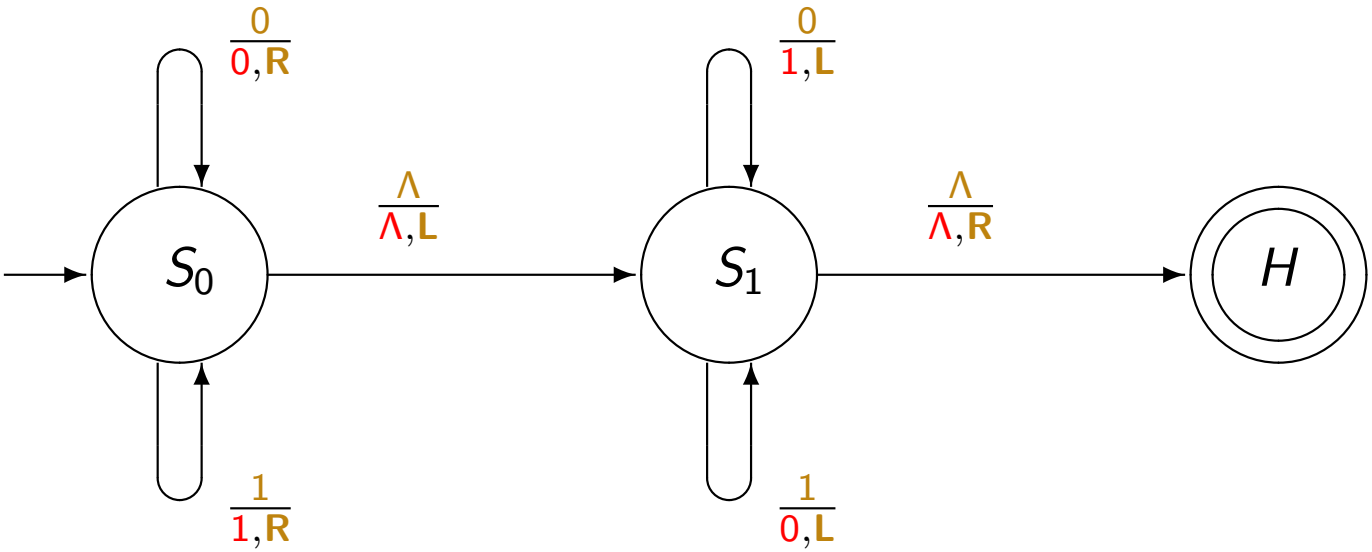
Language. The TM accepts:

- precisely $\{1^n \mid n \in \mathbb{N}\}$ if head starts right-most (we don't!).
- precisely $\{\varepsilon\} \cup \{1\alpha \mid \alpha \in \Sigma^*\}$ if heads starts left-most (we do!).

Alternative Formulation (not used here)

- could add an error state that the machine transitions to
- error state *not* accepting

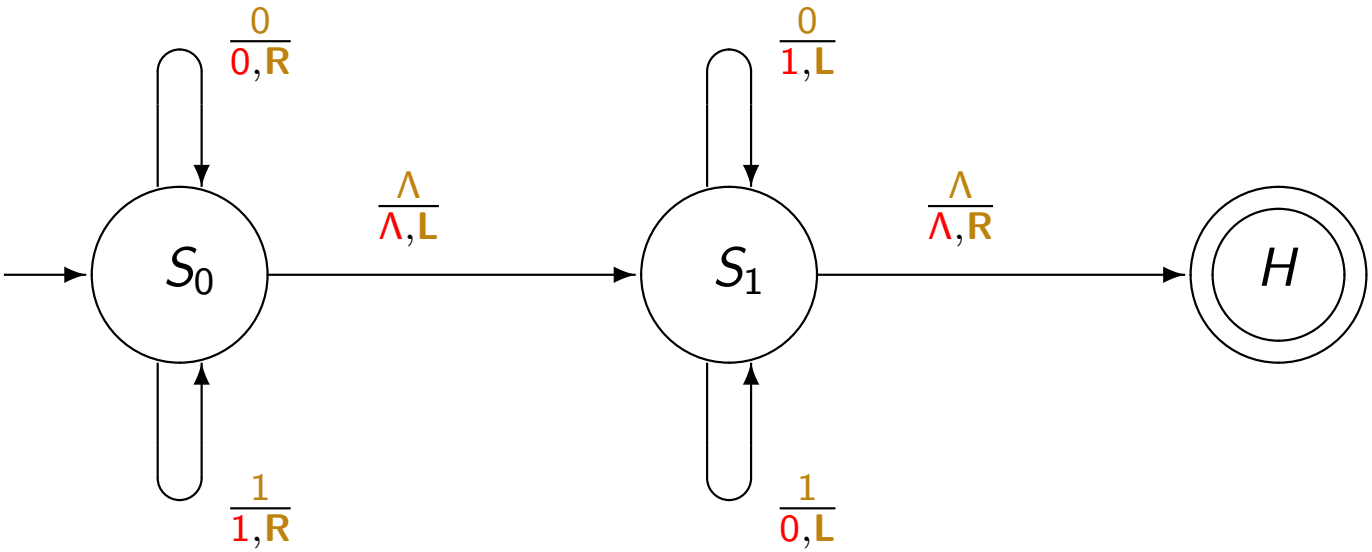
What does this one do?



Q.

- Do you see two phases?
- What does each phase accomplish?

What does this one do?



Q.

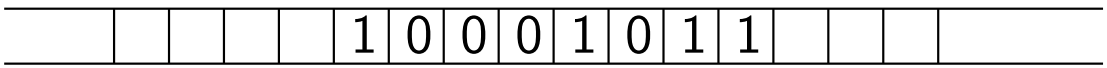
- Do you see two phases?
- What does each phase accomplish?

A.

- Phase 1: initialisation.
- Phase 2: computation, in this case, complement a binary number.

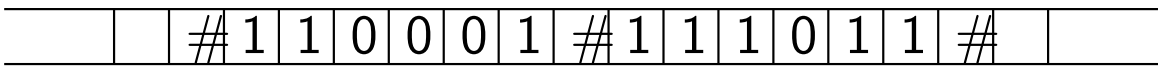
Harder Problems?

- Incrementing a binary number



You should try this!

- Adding numbers - need terminators

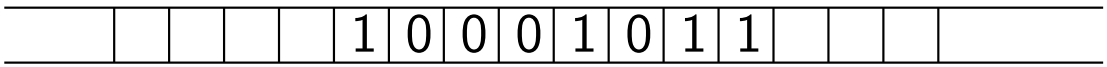


Convenient to write the result before the data.

- Multiplication - and so on!

Incrementing a binary number

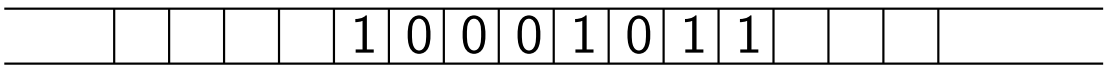
Example. Number increment



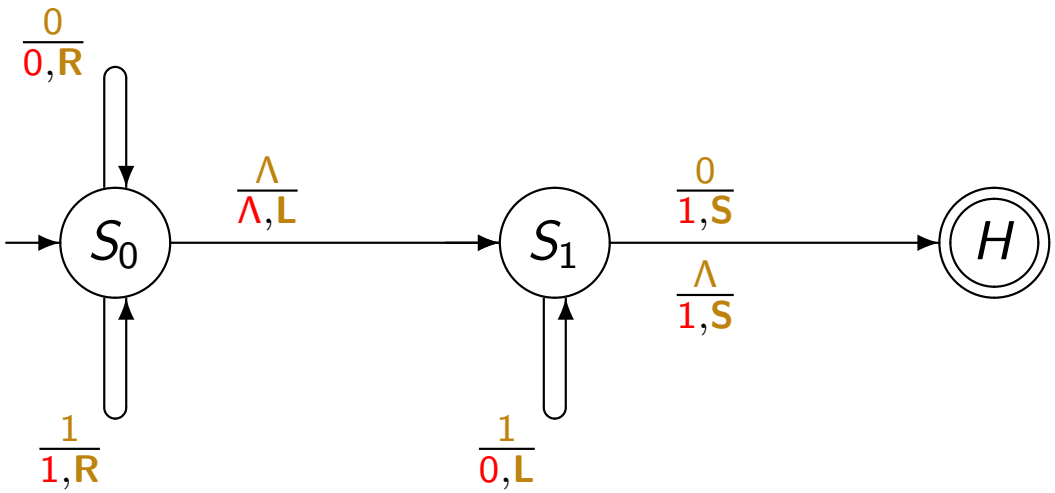
Solution:

Incrementing a binary number

Example. Number increment

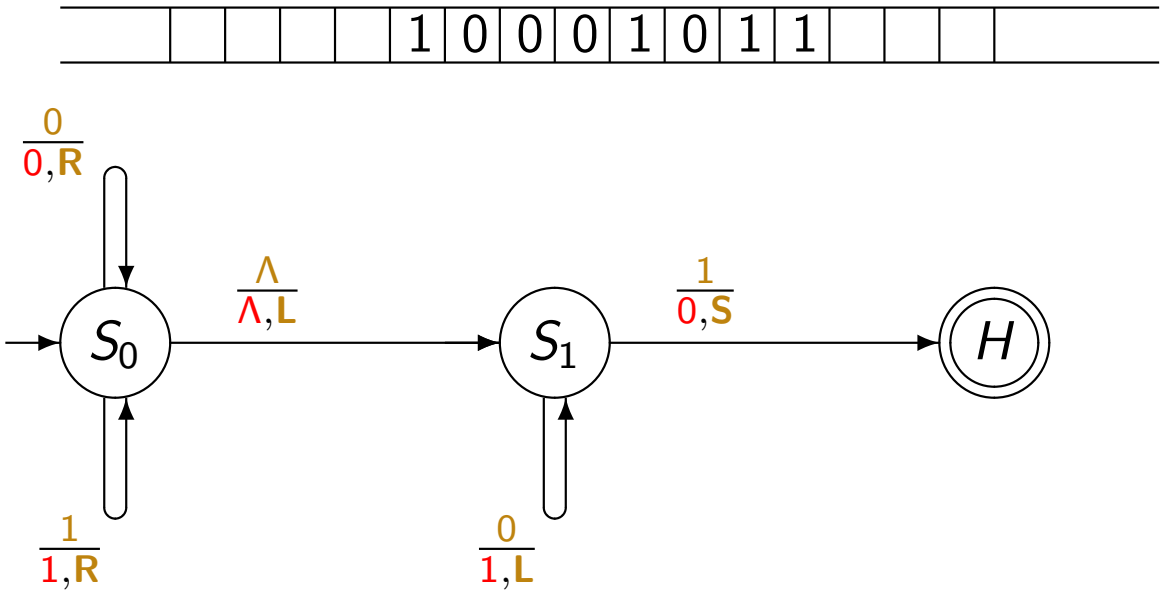


Solution:



Decrement

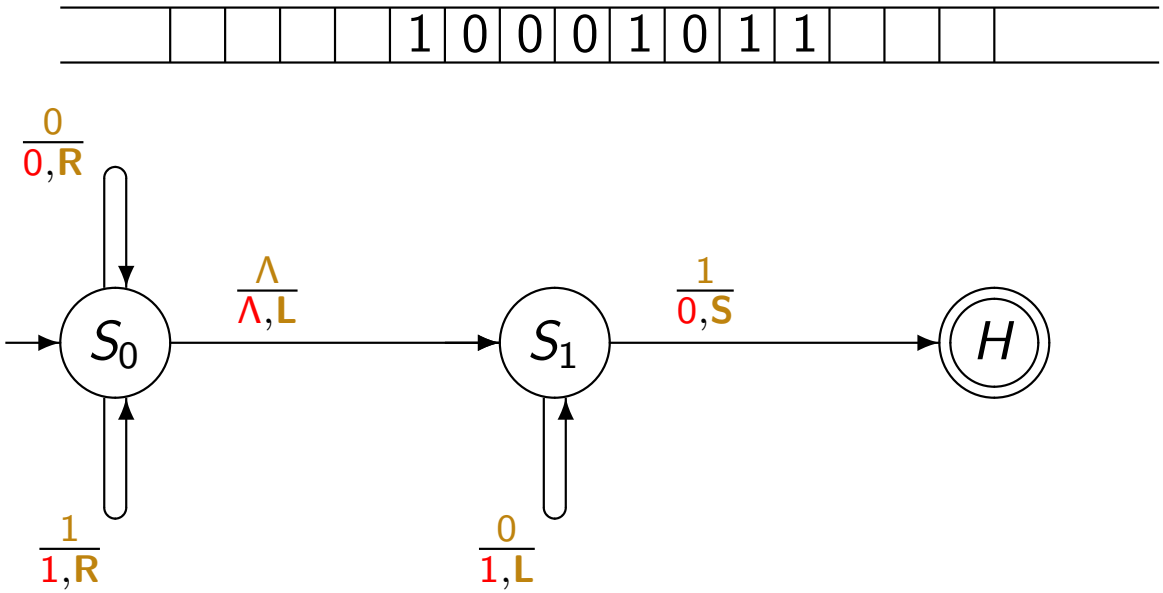
Example. Number decrement (similar)



Q. What happens if the input number is zero (e.g., 000)?

Decrement

Example. Number decrement (similar)



Q. What happens if the input number is zero (e.g., 000)? **A.:**

- First, it finds the right-most zero,
- then, the left-most one,
- then, for the blank on its left, no transition is defined.

How to add two numbers?

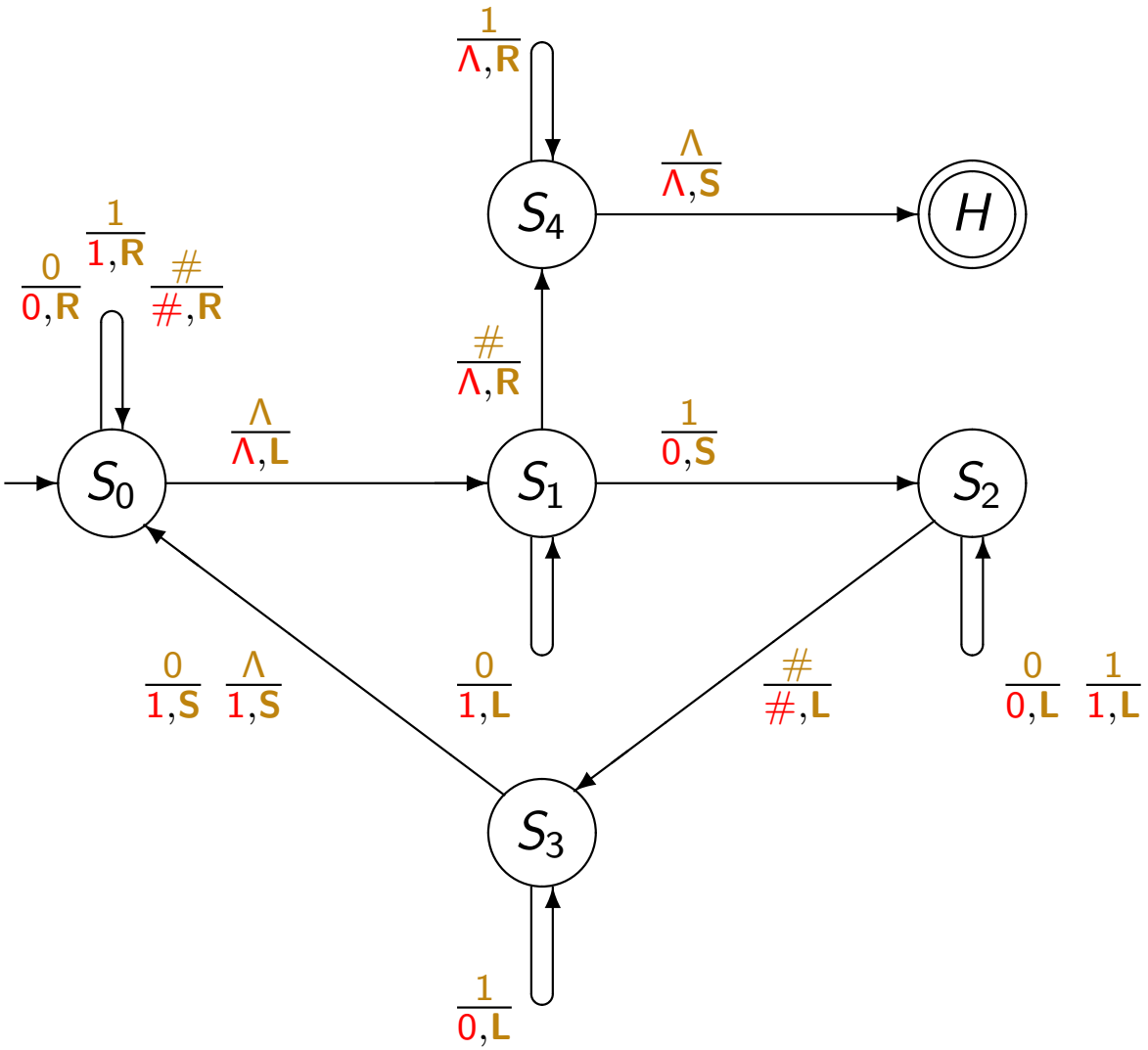
Input. Binary numbers separated by #, say

$$\overbrace{10100101}^{n}\#\overbrace{100101010}^{m}$$

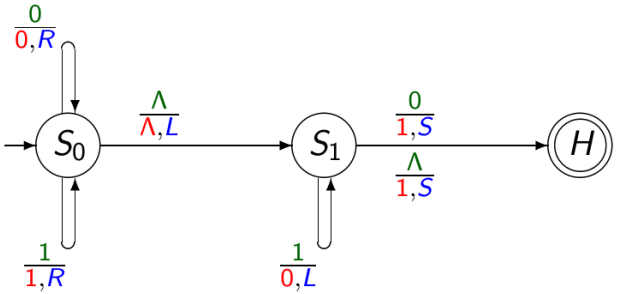
Operation.

- Go back and forth between m and n, decrementing one (until this fails) and incrementing the other.
- decrement m, and increment n, because n will expand leftwards.
- m gets changed to $00\ldots0$, n is replaced by the sum.
- Finally, delete the $\#00\cdots0$ on the right.

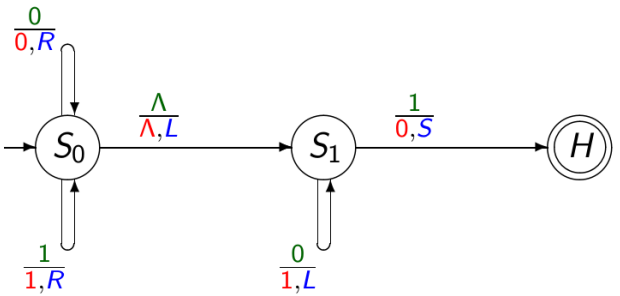
How to add two numbers? cont'd



Increment:



Decrement:



How to add two numbers? cont'd – again!

Just another short explanation on how the previous TM works:

- It works in phases. Before we start we have a tape $n\#m$.
  Then, after each round we have:
  1. $n+1\#m-1$ (after first pair of increment and decrement)
  2. $n+2\#m-2$ (after second pair of increment and decrement)
  3. $n+3\#m-3$ (after third...)
  4. ...
  5. $n+m\#0$ (after m pairs of increment and decrement)
- What happens in round $m+1$? Since we start with decrement, the second string is $0\ldots0$ before it gets decremented.
- So state S_1 will eventually find a $\#$, and then the string will be $n+m\#1\ldots1$. Then enter state S_4 to delete the right string $\#1\ldots1$.
- Now you can also see why we started to *decrement* instead of incrementing first! (Otherwise we might have to decrement the first string again in case m turns out to be 0.)

How to multiply two numbers?

Input. as for addition

$$\underbrace{}_{p}\#\underbrace{10100101}_{n}\#\underbrace{100101010}_{m}$$

Operation.

- Repeatedly decrement m (until this fails) and add n to p (p is initially blank)
- Must modify the addition routine to **not** erase the number n being added.

Modification of addition routine

- Two new tape symbols, $0'$ and $1'$.
- Before each addition stage, change all the 1s in n to $1'$.
- When decrementing n, swap 0s and 1s as usual, but keep the primes.
- When finished adding n to p, go back and use the primes to restore n.

Observation.

- this is *very tedious* – but the model is simple and easy to analyse
- tricks that you see here are typical

TMs vs. Programming and Common Idioms

Programming Issues – Data

Data Types and Gadgets

- not present in the model
- but can be simulated ...

Numbers.

- Usually use unary or binary for integers.

Vocabulary.

- Can be arbitrary, but $\{0, 1\}$ suffices. Characters are represented as strings of bits.

Variables, Arrays, Files

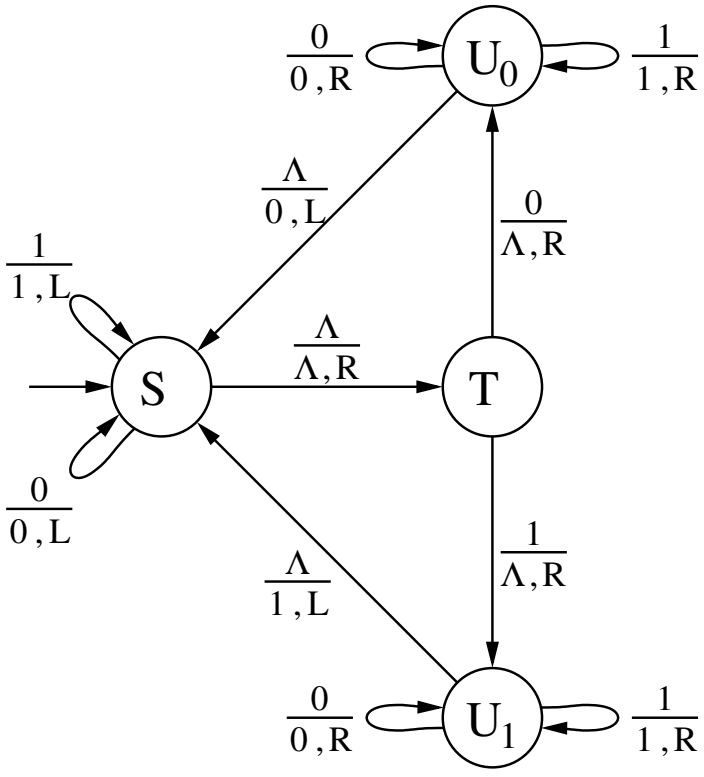
- Use markers on the tape to separate values.

Programming Issues – Control

Common Idioms.

- Scan to blank, or to find, insert, delete a symbol.
- Use control states to remember information
  In particular, we often need to "remember" a symbol, to write it elsewhere: this typically requires a set of states, one per symbol
- Composition
  If you have a TM to multiply by 3 and one to multiply by 5, put them together to multiply by 15.
- Decisions (conditional computation)
  As we have seen, we can branch on 0 or 1 (or T or F).
- Loops — of course.

Using States to Remember a Tape Symbol



Given a string of 0 or 1 surrounded by blanks, this machine repeatedly forever erases the leftmost bit, and writes it on the right hand end. (Not so useful, but illustrates the point)

We use the choice of states U_0 or U_1 to remember which symbol has been erased and is to be written

Languages of TMs and the Chomsky Hierarchy

The Chomsky Hierarchy, Revisited

Each grammar is of a *type*: (There are *lots* of intermediate types, too.)

Unrestricted: (type 0) no constraints, i.e., all productions $\alpha \to \beta$

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right*, $|\alpha| \leq |\beta|$.

- Note 1: There are other equivalent definitions which don't restrict the length
- Note 2*: If $\epsilon \in L$ should be allowed, we are allowed $S \to \epsilon$, but then we don't allow S to occur on any right-hand side.

Context-free: (type 2) the left of each production must be a *single non-terminal*.

Regular: (type 3) As for type 2, but the right of each production is further constrained (details to come).

Expressivity of Turing Machines

Theorem. Any language that is generated by a grammar (i.e., type 0) can be recognised by a Turing machine.

Proof (Sketch)

- write the *start symbol* S onto the tape (say, right of our input)
- search through all possible derivations from S
- each time we reach a *word*, check whether it matches the input (recall that we want to check whether a word lies in $L(G)$)

Acceptance.

- if the grammar generates the input, we accept
- if the grammar does *not* generate the input, we may loop forever (and not accept); this can't be prevented for some type 0 grammars (you should understand why next week), for all others the non-termination can be prevented

Expressivity of Turing Machines cont'd

Unrestricted Grammars. Have productions of the form $\alpha \to \beta$, where β is arbitrary, and α contains at least one non-terminal.

Theorem. For any TM, there exists a grammar that generates *precisely* the words that the TM accepts. (With the last theorem, this makes TMs *exactly* as expressive as type 0 grammars.)

Expressivity of Turing Machines cont'd

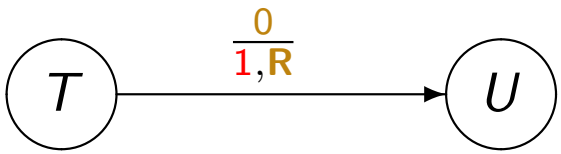
Unrestricted Grammars. Have productions of the form $\alpha \to \beta$, where β is arbitrary, and α contains at least one non-terminal.

Theorem. For any TM, there exists a grammar that generates *precisely* the words that the TM accepts. (With the last theorem, this makes TMs *exactly* as expressive as type 0 grammars.)

Proof (Sketch)

- non-terminals (of the grammar) are states of the TM
- run TM "backwards" (we are interested in inputs, not outputs)

TM Transition.



Grammar Production. $1U \to T0.$

(Details missing, e.g., how to handle blanks)

TM Expressivity, again

Can real computers simulate Turing Machines?

Differences between computers and Turing machines

- A Turing Machine has an *infinite tape*.
- "real" computers have finite memory
- physical devices *necessarily* have finite memory. They're more like finite automata. (Though "programmable" like universal TMs.)
- . . . but the number of states can be very, very large.
  How big is $2^{4294967296}$? (The exponent equals 2^{32})
  (FYI: there are approx $10^{80} \approx 2^{265}$ atoms in the observable universe)
- physical computers are an *approximation* of TMs.
  (It works as long as the memory is not insufficient.)

Computers & Programming Languages

Observation.

- no computer ever invented could do things that a TM can't do
- no programming language (PL) can do more than a TM
- back-and-forth translation TM $\leftrightarrow$ PL

Common Terminology.

A programming language that can compute every function that can be computed by a Turing machine is called *Turing complete*.

Examples.

- The languages that you know: Haskell, Java, Python, ...
- even the simple while language that we used for Hoare logic
- implement TM simulator in your favourite programming language
  Or online! *https://turingmachinesimulator.com*
  Invitation: construct some of ours and share the URL with us!

Church-Turing Thesis

Church-Turing Thesis.

> If a function is computable, then it can be calculated by a Turing machine.

Equivalent Formulation.

- if a problem can be solved by an algorithm, then it can be solved by a Turing machine.

Church-Turing Thesis

Church-Turing Thesis.

> If a function is computable, then it can be calculated by a Turing machine.

Equivalent Formulation.

- if a problem can be solved by an algorithm, then it can be solved by a Turing machine.

This is a **Thesis.**

- could also be regarded a definition (of computation/computability)
- can never be *proved*: what does "computable" mean?
- however, there's lots of *evidence*

Evidence.

- all *other* definitions of the term computable give the same class of computable functions
- there are many: λ-calculus, register machines, while programs, etc.

Argument 1: More power is useless

Stability of the TM Model.

- adding 'features' doesn't make more functions computable

Multi-tape.

- have extra tapes to store data
- easier to program, but no extra "power"
- (single-tape can simulate multi-tape)

Multi-head.

- have more than one head, heads can move independently
- heads can access multiple symbols at once
- again, no extra power

Non-determinism. (as for NFAs)

- tm can make one of several possible next moves
- tm can "guess" the right next move
- *may* make it faster, but cannot compute more functions.

Argument 2: Other Models are not more Powerful

Many Models of Computation

- plenty of *different* definitions of what computable may mean
- different purposes, different contexts

Main Insight.

- *any* (reasonable) model of computation can compute *precisely* the same functions as the TM model.
- "reasonable" in the sense of modelled on mechanical computation

Examples. Grammars, Lambda-Calculus (Church, 1932), Post-Systems (Post, 1939), Register Machines, ...

Doesn't include

- models based on physical phenomena
- ...or biology, or ...

Example: λ-calculus is not more powerful

Example. The (untyped) λ-calculus

- proposed by Alonzo Church (the Church in the Church-Turing thesis)
- in 1932, even before Turing's paper
- Rosser showed that both notions are equivalent

Equivalence. (Rosser, 1939)

- if a function is computable by a Turing machine, then it is computable in the λ-calculus
- . . . and vice versa
- simulation of the respective formalism in the other approach.

Example: John Conway's Game of Life (GoL)

Game of Life.

- infinite 2D grid, finitely many cells marked *alive*, all others are *dead*

Rules of the Game. iterate through generations

- live cells with < 2 alive neighbours die *(under-population)*;
- live cells with > 3 alive neighbours die *(over-population)*;
- dead cells with $= 3$ alive neighbours come alive *(reproduction)*;
- all other cells stay as they are.

Emergent Behaviour.

- analogy of complex behaviour emerging from simple rules
- Visualization of GoL (and TMs and more) in ≈30 minute video "*Math's Fundamental Flaw*" by Veritasium: *https://www.youtube.com/watch?v=HeQX2HjkcNo* (from 0:58)
  I also recommend this video for Logic and undecidability!

Example: John Conway's Game of Life (GoL) cont'd

From TM's to Conway's Game

- can implement Game of Life on a Turing machine
- lots of coding, in particular 2D grid onto 1D tape

From Conway's Game to TMs (Paul Rendell, 2011)

- showed that GoL can simulate Turing machines
- comes down to clever choice of initial configurations see `http://rendell-attic.org/gol/turing_js_r.gif` or, again, *https://www.youtube.com/watch?v=HeQX2HjkcNo*

Universal Turing Machines

Universal Turing Machines and Turing Completeness

- So far, TMs were "one job computers", i.e., not programmable!

Universal Turing Machines and Turing Completeness

- So far, TMs were "one job computers", i.e., not programmable!
- We can construct a TM that first reads a description of some other TM and then simulates it. This is a *universal* TM.
- Any computing device which can simulate a universal Turing Machine is also called *universal* or *Turing Complete*.

Coding a TM onto tape

Coding of a TM as binary strings

- can be written onto a tape
- just code the transition function

States are ordered $S_1, S_2, \ldots, S_n$, where S_1 is the start state and S_2 the unique final state (this works for our example, but in general we had to specify which states are final states, e.g., by listing their numbers).

Tape Symbols are ordered X_1, X_2, X_3 where X_1 is 0, X_2 is 1, and X_3 is Λ. (Can be extended if we have more symbols.)

Directions L, R, S as D_1, D_2, D_3 respectively.

Transitions $\delta(S_i, X_j) = (S_k, X_l, D_m)$ mapped to

$$0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$$

$(i \in \{1, \ldots, n\}, j, l, m \in \{1, 2, 3\})$
(the 0s carry information, the 1s act as separators.)

Coding a TM onto tape ctd.

Coding of all transitions. A TM with transitions $C_1, \ldots, C_n$ is coded as

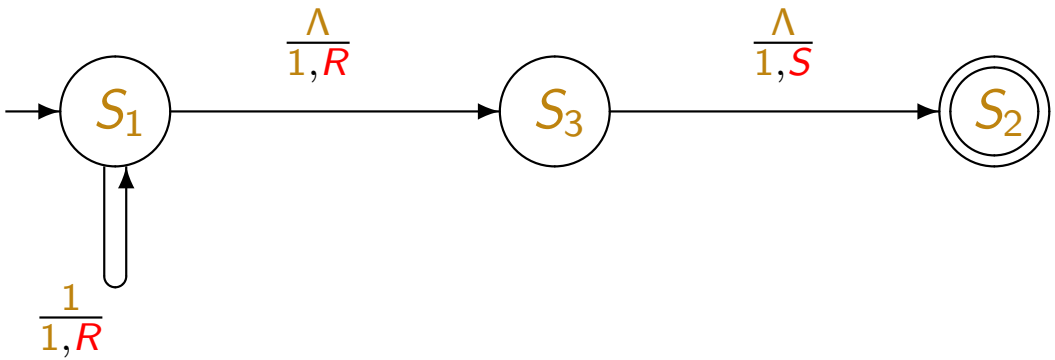
$$C_1\ 11\ C_2\ 11\ \cdots\ 11\ C_n$$

(11 is used as a separator for transitions)

Additional Input.

- code for a TM, and additional string
- use 111 to separate TM and string input

Coding a TM onto tape – example



The transitions are

0	1	00	1	0	1	00	1	00
0	1	000	1	000	1	00	1	00
000	1	000	1	00	1	00	1	000

Recall: States, Symbols, Directions.

So the TM as a whole is encoded by the binary string

$$0100101001\mathbf{00}\ 11\ 0100010001001\mathbf{00}\ 11\ 0001000100100\mathbf{1000}$$