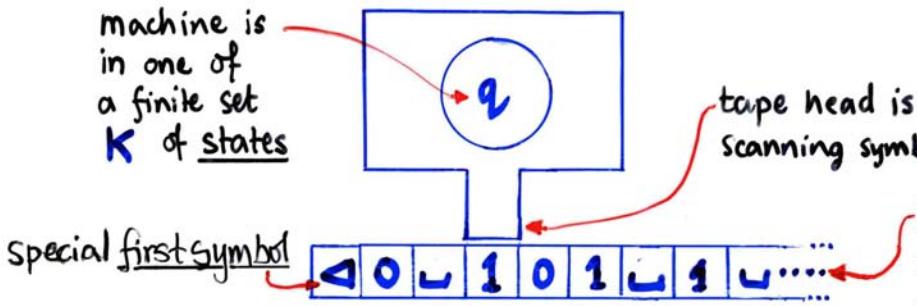
Turing machines and the Church-Turing Thesis

Register machine computation takes for granted that we have some concrete representation of the natural numbers and the elementary operations on them of increment, zero test and decrement.
Turing's original model of computation — now called a Turing machine — formalizes the intuitive notion of algorithm in the most concrete terms possible (Turing argued), where even numbers have to be represented explicitly in terms of a fixed, finite alphabet of symbols (eg unary notation, binary notation, etc.)
and the elementary operations have to be given explicitly in terms of elementary symbol-manipulating operations ...

Turing machines – informal description



- machine starts with tape head pointing to ◁
- machine computes in discrete steps, each of which depends only on current state & symbol being scanned by tape head.
- action at each step (if any) is:

  overwrite current tape cell with a symbol, move left or right one cell, or stay stationary, and change to another state.

DEFINITION: a **Turing machine** consists of:

- a finite set Σ of **tape symbols**, containing distinguished elements $\begin{cases} \text{␣} = \text{blank symbol} \\ \triangleleft = \text{first symbol} \end{cases}$
- a finite set K of **machine states** (disjoint from Σ) containing a special element s = **initial state**
- a function

  $$\delta \in \mathrm{Fun}(K \times \Sigma, (K \cup \{acc, rej\}) \times \Sigma \times \{L, R, S\})$$

  called the **transition function** of the machine, satisfying: for all q, q', a', D if $\delta(q, \triangleleft) = (q', a', D)$, then $a' = \triangleleft$ and $D = R$

(N.B. $acc, rej \notin K$ are special accepting/rejecting states.)

Example of a Turing Machine (p68)

$\Sigma = \{\triangleleft, \sqcup, 0, 1\}$

$K = \{s, q, q'\}$

δ given by:

	$\triangleleft$	$\sqcup$	0	1
s	$(s, \triangleleft, R)$	$(q, \sqcup, R)$	$(rej, 0, S)$	$(rej, 1, S)$
q	$(rej, \triangleleft, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(rej, \triangleleft, R)$	$(acc, \sqcup, S)$	$(rej, 0, S)$	$(q', 1, L)$

The transition function δ specifies how the Turing machine should act at each step. If

current state $= q \in K$

current tape symbol $= a \in \Sigma$

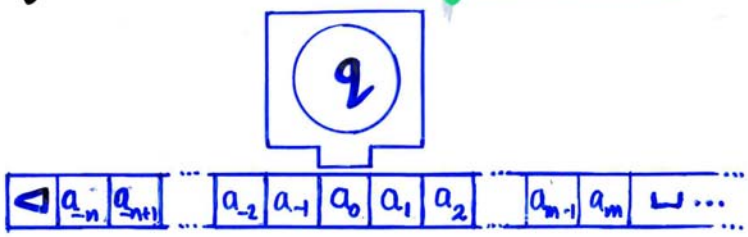
and $\delta(q, a) = (q', a', D)$

then:

- if $q' = acc$, the machine has reached accepting state (& stops after overwriting a with a')
- if $q' = rej$, the machine has reached rejecting state (& stops after overwriting a with a')
- if $q' \in K$, the machine
  - overwrites a with a',
  - changes to state q', and
  - moves one cell Left if $D = L$
    moves one cell Right if $D = R$
    stays Stationary if $D = S$.

(N.B. because of the condition on δ in the definition on p63, if $a = \triangleleft$, then machine never overwrites $\triangleleft$ with anything else and always moves to the right.)

A Turing machine configuration



can be specified by a triple

(q, l, r)

- q: current state $\in K \cup \{acc, rej\}$
- l: finite, non-nil list of elements of Σ
  $l = (a_0, a_{-1}, a_{-2}, \ldots, a_{-n}, \triangleleft)$ starting with current symbol, reading right-to-left, ending with $\triangleleft$
- r: finite (possibly nil) list of elements of Σ
  $r = (a_1, a_2, \ldots, a_m)$ starting with symbol to right of current one, reading left-to-right, ending at some point where there are only blanks to the right.

Transition relation $(q_1, l_1, r_1) \rightarrow (q_2, l_2, r_2)$ is defined to hold if & only if $q_1\ l_1\ r_1\ q_2\ l_2\ r_2$ match one of the following cases:

- $(q, cons(a, l), r) \rightarrow (q', l, cons(a', r))$
  where $\delta(q, a) = (q', a', L)$
- $(q, cons(a, l), nil) \rightarrow (q', cons(␣, cons(a', l)), nil)$
  where $\delta(q, a) = (q', a', R)$
- $(q, cons(a, l), cons(b, r)) \rightarrow (q', cons(b, cons(a', l)), r)$
  where $\delta(q, a) = (q', a', R)$
- $(q, cons(a, l), r) \rightarrow (q', cons(a', l), r)$
  where $\delta(q, a) = (q', a', S)$

A computation of the Turing machine consists of a finite or infinite sequence of transitions between configurations:

$$(s, \triangleleft, r) \to (q_1, l_1, r_1) \to (q_2, l_2, r_2) \to \cdots$$

↑ initial state; ↑ one-element list just containing $\triangleleft$

The computation <u>does not halt</u> if the sequence is infinite
" " <u>halts</u> " " " " finite, in which case the last configuration is of the form (acc, l, r) or (rej, l, r).

EXAMPLE

Consider the Turing machine with

$\Sigma = \{\triangleleft, \sqcup, 0, 1\}$, $K = \{s, q, q'\}$

and δ given by:

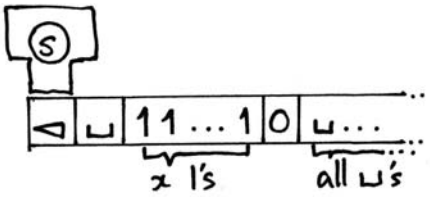
	$\triangleleft$	$\sqcup$	0	1
s	$(s, \triangleleft, R)$	$(q, \sqcup, R)$	$(rej, 0, S)$	$(rej, 1, S)$
q	$(rej, \triangleleft, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(rej, \triangleleft, R)$	$(acc, \sqcup, S)$	$(rej, 0, S)$	$(q', 1, L)$

Then the computation starting from configuration $(s, \triangleleft, \sqcup \overline{x0})$ halts in configuration $(acc, \sqcup\triangleleft, \overline{x+1\,0})$.

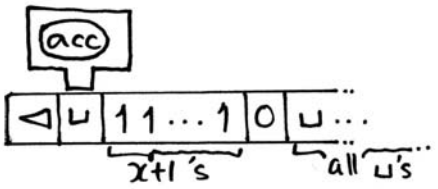
x 1's; list $(\sqcup, 1, \ldots, 1, 0)$

Example, continued.]

So the starting configuration looks like



the final configuration looks like



and the transitions between are

$$
\begin{aligned}
(s, \triangleleft, \sqcup\bar{x}0) &\to (s, \sqcup\triangleleft, \bar{x}0) \\
&\to (q, 1\sqcup\triangleleft, \overline{x-1}0) \\
&\vdots \\
&\to (q, \bar{x}\sqcup\triangleleft, 0) \\
&\to (q, 0\bar{x}\sqcup\triangleleft, \text{nil}) \\
&\to (q, \sqcup\overline{x+1}\sqcup\triangleleft, \text{nil}) \\
&\to (q', \overline{x+1}\sqcup\triangleleft, 0) \\
&\vdots \\
&\to (q', \sqcup\triangleleft, \overline{x+1}0) \\
&\to (\text{acc}, \sqcup\triangleleft, \overline{x+1}0)
\end{aligned}
$$

tape head moving right

tape head moving left

PROPOSITION :

The computation of a Turing machine can be implemented on a register machine.

Proof

First, represent tape and state symbols of the Turing machine by numbers, say:

$acc = 0$ $\quad\quad$ $\sqcup = 0$

$rej = 1$ $\quad\quad$ $\triangleleft = 1$

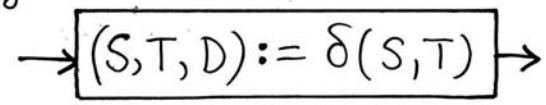
$s = 2$

$K = \{2,3,\ldots,n\}$ $\quad\quad$ $\Sigma = \{0,1,\ldots,m\}$

Then code Turing machine configurations (q,ℓ,r) as numbers $[q,[\ell],[r]]$ (using the coding of lists of numbers as numbers that we developed earlier). Since the transition function $\delta \in Fun(K\times\Sigma, (K\cup\{acc,rej\})\times\Sigma\times\{L,R,S\})$ is a finite set (of (argument, result)-pairs), one can construct a register machine program operating on registers

S (for state), T (for tape symbol), D (for direction, with $\begin{cases} L=0 \\ R=1 \\ S=2 \end{cases}$ say)

implementing



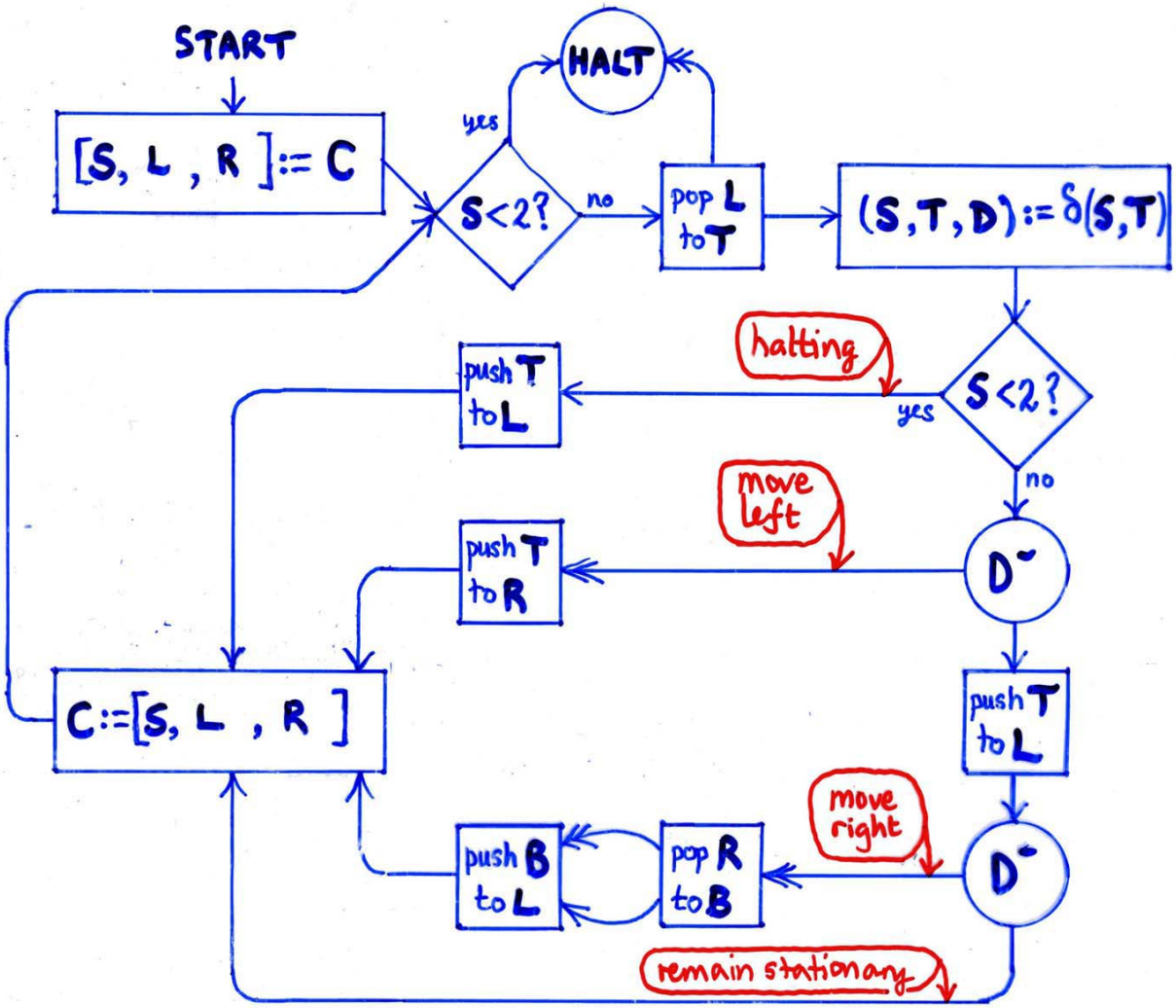
Then using registers

C to hold codes of Turing machine configurations

L to hold codes of tape symbol list at & to left of tape head

R " " " " " " " to right of " "

the computation of the Turing machine is carried out by the register machine specified on the next page — i.e. starting the register machine with C holding the code of the initial configuration (& all other registers zeroed), the register machine halts if & only if the corresponding Turing machine computation halts, and in that case C holds the code of the final configuration. □



Recall:

DEFINITION:

$f \in Pfn(\mathbb{N}^n, \mathbb{N})$ is (register machine) computable if & only if there is a register machine M with at least $n+1$ registers, $R0, R1, R2, \ldots, Rn$ say, (and maybe some other registers as well) with the property that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$

$f(x_1, \ldots, x_n) = y$ if & only if the computation of M starting with $R1 = x_1, \ldots, Rn = x_n$, and all other registers $= 0$, halts with $R0 = y$.

We have seen that Turing machine computation can be implemented by register machines. The converse also holds: the computation of a register machine can be implemented by a Turing machine.

To make sense of this statement, we first have to fix a tape representation of register contents, i.e. a tape representation of finite lists of numbers:

- we will use unary notation for individual numbers:
  number $x \leftrightarrow \underbrace{11\cdots1}_{x\ 1\text{'s}}$
- we will use 0 to mark the beginning and end of a list
- we will use ␣ (the blank symbol) to separate numbers in the list.

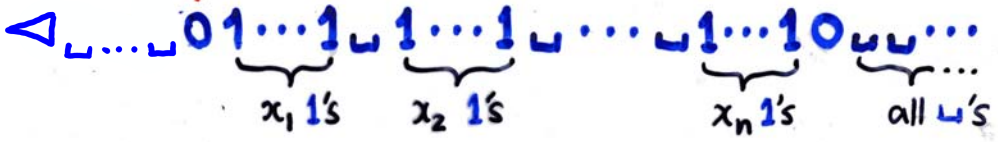
Thus we can take the alphabet of tape symbols to be $\Sigma = \{\triangleleft, ␣, 0, 1\}$ and then...

A tape over $\{\triangleleft, ␣, 0, 1\}$ codes a list of numbers if & only if

precisely two cells contain 0 and the only cells containing 1 occur between these

Such tapes look like:

we call this cell of such a tape the initial 0 cell



and the corresponding list of numbers is:

$\text{cons}(x_1, \text{cons}(x_2, \ldots, \text{cons}(x_n, \text{nil})\ldots))$

i.e. $(x_1, x_2, \ldots, x_n)$

DEFINITION:

$f \in Pfn(\mathbb{N}^n, \mathbb{N})$ is **Turing computable** if & only if there is a Turing machine T with the following property:

Starting T from its initial state with tape head on the first symbol $\triangleleft$ of a tape coding $(0, x_1, \ldots, x_n)$, T halts if & only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

THEOREM: A partial function is Turing computable if & only if it is register machine computable.

Proof

Since we can implement any Turing machine by a register machine, it follows that

Turing computable $\Rightarrow$ register machine computable.

To see that

" " $\Leftarrow$ " " "

one has to implement the computation of a register machine in terms of a Turing machine operating on a tape coding instantaneous register contents. To do this, one has to see how to carry out the action of each type of register machine instruction on the tape representation of register contents. It should be reasonably clear that this is possible in principle, even if the details (which we omit) are somewhat tedious. $\square$

CHURCH-TURING THESIS:

Every algorithm (in the intuitive sense)
can be realized as a Turing machine.

Or, equivalently, every algorithm can be realized as a register machine.

The Church-Turing Thesis is not a statement that can be proved formally – because it refers to the informal notion of "algorithm".

Turing gave a closely argued justification that his machines captured the fundamental elements of the notion of algorithm. Since his time much empirical evidence has accumulated to support the Church-Turing Thesis:

- Several extensions of the notion of Turing machine (and register machine) that have been proposed (e.g. extensions by non-deterministic features or by parallel computation) have all been shown to have equivalent computing power to the original formulation.
- A number of alternative formalizations of the intuitive notion of algorithm (some of which appear quite unconnected with the Turing/register machine formalism) have turned out to determine the same collection of computable functions:

Some approaches to computability

- Church (1936): (untyped) lambda calculus & λ-definable functions.
  [see: CST IB "Foundations of Functional Programming" course.]
- Turing (1936): Turing machines.
- Gödel-Kleene (1936): partial recursive functions
- Post (1943): canonical systems for generating theorems in a formal system.
- Markov (1951): deterministic version of Post's canonical systems.
- Lambek (1961), Minsky (1961), Shepherdson-Sturgis (1963): register machines.

All of the above approaches give rise to the same collection of partial functions from numbers to numbers.
The same is true for any "general purpose" programming language (indeed, one usually takes as a definition of "general purpose"* that the language can code any computable partial function).

We will look at one of the above, alternative approaches in some detail – namely the Gödel-Kleene characterization of computable functions as "partial recursive" functions...

---

* or "Turing powerful"

Church's (untyped) λ-calculus

[overview! not examinable]

λ-terms $M ::= x \mid \lambda x.M \mid M(M)$ (variables; function abstractions; function applications)

β-reduction $M \to M'$: smallest relation s.t.

$\lambda x.M\,(N) \to M[N/x]$ ← substitution

if $M \to M'$ then $N(M) \to N(M')$ & $M(N) \to M'(N)$ & $\lambda x.M \to \lambda x.M'$

n^{th} Church numeral : $\ulcorner n \urcorner = \lambda y.\lambda x.\underbrace{y(y(\cdots(y}_{n\text{ "y"s}}(x))\cdots)$

$f \in Pfn(\mathbb{N},\mathbb{N})$ is λ-definable iff $\exists$ λ-term F so that

$f(x)=y \iff F(\ulcorner x \urcorner) \to \cdots \to \ulcorner y \urcorner$

THEOREM : λ-definable = computable