CDM

Computability

Klaus Sutner

Carnegie Mellon University

Fall 2025

The Big Question

Do primitive recursive functions match our intuitive notion of computability perfectly?

There is no doubt that all p.r. functions are computable, but it is not so clear that every computable function is also p.r.

Functions that are computable in any practical sense are typically p.r., but we are trying to find a general definition, disregarding efficiency considerations.

Coding

Notation:

$\mathbb{N}^\star$	the set of all finite sequences of natural numbers
nil	the empty sequence.

We want to express any sequence $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}^\star$ as a code number (sequence number) $\langle a_0, a_1, \ldots, a_{n-1} \rangle$. We need a map

$$\langle . \rangle : \mathbb{N}^\star \to \mathbb{N}$$

that allows us to decode: from $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ we can recover n as well as all the a_i.

The Pieces

Suppose

$$s = \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$$

is some code number (0-indexing turns out to be easier to use).

We want a unary length function len : $\mathbb{N} \to \mathbb{N}$:

$$\mathsf{len}(s) = n$$

and a binary decoding function dec : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

$$\mathsf{dec}(s, i) = a_i$$

for all $i = 0, \dots, n-1$.

Traditionally, dec(b, i) is written $(b)_i$.

Computable Coding

Again, we need three functions:

$$\langle . \rangle : \mathbb{N}^\star \to \mathbb{N}$$

$$\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathsf{len} : \mathbb{N} \to \mathbb{N}$$

From a set theory perspective, this is trivial: $\mathbb{N}^\star$ is countable.

But we live in the computational universe: we want these functions to be easily computable, say, primitive recursive. More precisely, we want dec and len to be p.r.; the coding function $\langle . \rangle$ has the wrong domain, but we can still insist that the restriction $\langle . \rangle : \mathbb{N}^k \to \mathbb{N}$ is p.r. for all k.

Sequence Numbers

The set of sequence numbers is

$$\mathsf{Seq} = \{\, \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \mid a_i \in \mathbb{N}, n \in \mathbb{N} \,\}$$

Note that a priori len and dec need not be defined outside of Seq (in addition, for dec the index i needs to be in the right range).

One simply assumes that the functions return the default value 0 for meaningless arguments.

Prime Coding

Perhaps the most straightforward way to code sequences as numbers is to exploit the uniqueness of the prime decomposition.

$$\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle = p_0^{a_0+1} p_1^{a_1+1} \ldots p_{n-1}^{a_{n-1}+1}$$

Here (p_i) is the enumeration of the primes in increasing order.

For len and dec we can use the prime enumeration from last time.

Exercise

Figure out the details.

Gödel

Prime coding is perfectly fine, but it uses relatively complicated number-theoretic machinery.

In his work on the incompleteness theorem Gödel used a much more elegant and less sledge-hammerish approach to handle coding. There are also lots of coding functions based on iterated pairing.

See Coding for background.

Evaluation

Recall our simple programming language, terms τ that denote p.r. functions $[\![\tau]\!] : \mathbb{N}^k \to \mathbb{N}$.

$$\tau = \mathsf{Prec}[\mathsf{Prec}[\mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \circ (\mathsf{P}^3_2, \mathsf{P}^3_3), \mathsf{C}^{(1)}_0] \circ (\mathsf{S} \circ \mathsf{P}^2_1, \mathsf{P}^2_2), \mathsf{C}^{(0)}_1]$$

τ denotes the factorial function, $[\![\tau]\!](n) = n!$ for all n.

We saw in particular how to write an interpreter, a function eval that takes as input a term τ and a vector $\boldsymbol{a} = a_1, \ldots, a_n \in \mathbb{N}$ of the right length and returns the result of evaluating $[\![\tau]\!]$ on $\boldsymbol{a}$.

Exercises

Exercise

Write a compiler that given any string τ checks whether it is a well-formed expression denoting a primitive recursive function.

Exercise

Write an interpreter for primitive recursive functions (i.e., implement eval*) in your favorite programming language.*

Arithmetization

The type of eval is

$$\text{eval} : \text{pr-terms} \times \mathbb{N}^\star \longrightarrow \mathbb{N}$$

In his work on the incompleteness theorem Gödel figured out how to express strings as a natural numbers, the infamous Gödel numbers.

For computable functions, the corresponding Gödel number is usually called an index, written $e = \widehat{\tau}$. Similarly we can replace $\boldsymbol{a}$ by $\langle \boldsymbol{a} \rangle$.

So its safe to think of evaluation of unary p.r. functions as a map

$$\text{eval} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

If e is not an index, we may assume $\text{eval}(e, x) = 0$.

Coding PR

Here is one natural way of coding primitive recursive terms as naturals:

term	code
$\mathsf{C}_0^{(0)}$	$\langle 0, 0 \rangle$
S	$\langle 1, 1 \rangle$
$\mathsf{P}^{\mathsf{n}}_{\mathsf{i}}$	$\langle 2, n, i \rangle$
$\mathsf{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$
$\mathsf{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$

Thus for any index e, the first component $\mathsf{fst}(e)$ indicates the type of function, and $\mathsf{snd}(e)$ indicates the arity.

This type of coding makes it really easy to write an interpreter.

Hope Springs Eternally

Question: Maybe eval is primitive recursive?

Let's assume eval primitive recursive. Now define

$$f(x) := \mathsf{eval}(x, x) + 1$$

This may look weird, but certainly f is also p.r. and must have an index e. But then

$$f(e) = \mathsf{eval}(e, e) + 1 = f(e) + 1$$

and we have a contradiction, $0 = 1$.

Partial Functions

How do we avoid the problem with eval?

The only plausible solution appears to be to admit partial functions, functions that, like eval, are computable but may fail to be defined on some points in their domain. In this case, eval(e, e) is undefined.

Anyone who has ever written a sufficiently sophisticated program will have encountered divergence: on some inputs, the program simply fails to terminate. What may first seem like a mere programming error, is actually a fundamental feature of computable functions.

Incidentally, in the early days of recursion theory, partial functions were universally avoided.

General Computability

We presented the last argument in the context of primitive recursive functions, but note that the same reasoning also works for any clone of computable functions—as long as

- successor and eval both belong to the clone, and
- each function in the clone is represented by an index.

But then eval must already be partial, no matter what the details of our clone are.

A similar argument shows that an interpreter for, say, polynomial time computable functions cannot itself by polynomial time.

Fiat Halting

Since any general model of computation must deal with partial functions, it is entirely natural to ask whether a given function f is defined on some particular input x.

Another natural question would be to ask whether f is total.

So we automatically run into the Halting Problem, the first example of a perfectly well-defined question that turns out to be undecidable.

Notation Warning

We write

$$f : A \nrightarrow B$$

for a partial function from A to B. Terminology:

domain	$\operatorname{dom} f = A$
codomain	$\operatorname{cod} f = B$
support	$\operatorname{spt} f = \{\, a \in A \mid \exists\, b\, (f(a) = b) \,\}$

It is also convenient to write $f(x)\downarrow$ for $x \in \operatorname{spt} f$, and $f(x)\uparrow$ for $x \notin \operatorname{spt} f$ (converges/diverges).

Warning:
Some misguided authors call the support simply domain. Bad, bad idea.

Kleene's Notation

Since we cannot avoid partial functions, we should adjust notation a bit.

Given expressions α, β involving partial functions, we use Kleene equality rather than plain equality:

$$\alpha \simeq \beta$$

to indicate that either

- both α and β are defined (the computations involved all terminate) and have the same value, or
- both α and β are undefined (some computation diverges).

More Kleene

Given a clone of computable functions, such as the primitive recursive ones, and an index e for one of these functions, we write

$$\{e\}$$

for the eth function in the collection. Hence, $(\{e\})_{e\geq 0}$ is an enumeration of all the functions in the clone.

Since these functions are partial in general we have to be a bit careful and write

$$\{e\}(x) \simeq \{e'\}(x)$$

to indicate that functions $\{e\}$ and $\{e'\}$ agree on input x. We could add an annotation for arity, but we won't bother.

Ackermann's Function (1928)

Primitive recursion uses only a single variable. Maybe recursion over multiple variables could produce more complicated functions[†].

Here is a classical example: the Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by double recursion. We write x^+ instead of $x + 1$.

$$\begin{aligned} A(0, y) &= y^+ \\ A(x^+, 0) &= A(x, 1) \\ A(x^+, y^+) &= A(x, A(x^+, y)) \end{aligned}$$

On the surface, this looks more complicated than primitive recursion. We need to make sure that there really is no trick to rewrite this as a single recursion.

[†]It's not obvious, maybe one could use coding tricks to get everything down to just one variable. Try mutual recursion for example.

Family Perspective–Currying

It is useful to think of Ackermann's function as a family of unary functions $(A_x)_{x \geq 0}$ where $A_x(y) = A(x,y)$ ("level x of the Ackermann hierarchy").

The definition then looks like so:

$$A_0 = S \qquad\qquad \begin{aligned} A_{x^+}(0) &= A_x(1) \\ A_{x^+}(y^+) &= A_x(A_{x^+}(y)) \end{aligned}$$

From this it follows easily by induction that

Lemma

Each of the functions A_x is primitive recursive (and hence total).

The Bottom Hierarchy

$$A(0,y) = y^+$$

$$A(1,y) = y^{++}$$

$$A(2,y) = 2y+3$$

$$A(3,y) = 2^{y+3}-3$$

$$A(4,y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3$$

The first 4 levels of the Ackermann hierarchy are easy to understand, though A_4 starts causing problems: the stack of 2's in the exponentiation has height $y+3$.

Tetration

The basic operation behind A_4 is usually called super-exponentiation or tetration and often written ${}^n a$ or $a{\uparrow\uparrow}n$.

$$a{\uparrow\uparrow}n = \begin{cases} 1 & \text{if } n = 0, \\ a^{a{\uparrow\uparrow}(n-1)} & \text{otherwise.} \end{cases}$$

For example,

$$A(4,3) = 2{\uparrow\uparrow}6 - 3 = 2^{2^{65536}} - 3$$

an uncomfortably large number (we'll see much worse soon).

The Mystery of $A(6,6)$

Alas, if we continue just a few more levels, darkness befalls.

$$A(5,y) \approx \text{super-super-exponentiation}$$

$$A(6,y) \approx \text{an unspeakable horror}$$

$$A(7,y) \approx \text{speechlessness}$$

For level 5, one can get some vague understanding of iterated super-exponentiation, $A(5,y) = (\lambda z.z \uparrow\uparrow y+3)^{y+3}(1) - 3$ but things start to get quite murky at this point.

At level 6, we iterate over the already nebulous level 5 function, and things really start to fall apart.

At level 7, Wittgenstein comes to mind: “Whereof one cannot speak, thereof one must be silent.”*

*“Wovon man nicht sprechen kann, darüber muss man schweigen.” *Tractatus Logico-Philosophicus*

Ackermann vs. PR

Theorem

The Ackermann function dominates every primitive recursive function f in the sense that there is a k such that

$$f(\boldsymbol{x}) < A(k, \max \boldsymbol{x}).$$

Hence A is not primitive recursive.

Sketch of proof.

Since we are dealing with a rectype, we can argue by induction on the buildup of f.

The atomic functions are easy to deal with.

The interesting part is to show that the property is preserved during an application of composition and of primitive recursion. Alas, the details are rather tedious.

$\square$

Ackermann and Union/Find

One might think that the only purpose of the Ackermann function is to refute the claim that computable is the same as p.r. Surprisingly, the function pops up in the analysis of the Union/Find algorithm (with ranking and path compression).

The running time of Union/Find differs from linear only by a minuscule amount, which is something like the inverse of the Ackermann function.

But in general anything beyond level 3.5 of the Ackermann hierarchy is irrelevant for practical computation.

Exercise

Read an algorithms text that analyzes the run time of the Union/Find method.

Video with Tarjan.

But Is It Computable?

Here is an entirely heuristic argument: we can write a tiny bit of C code that implements the Ackermann function (assuming that we have infinite precision integers).

```
int acker(int x, int y)
{
  return( x ? (acker(x-1, y ? acker(x, y-1) : 1)) : y+1 );
}
```

All the work of organizing the nested recursion is easily handled by the compiler and the execution stack. So this provides overwhelming evidence that the Ackermann function is intuitively computable.

Dynamic Programming

Suppose we want to compute $A(a, b)$ bottom-up, dynamic programming style.

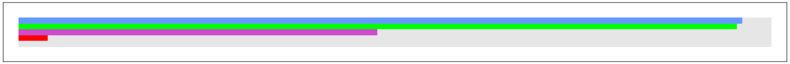
We need a 2-dim table T such that $T[i, j] = A(i, j)$. The table will be filled row by row, and each row from left to right.

Problem: How large should the table be?

To fill position $i, j > 0$, we look up $k = T[i, j - 1]$ and then $T[i - 1, k]$.

But k is huge, every time we go up one row we have to go much further to the right.

$A(3,4) = 125$



The effect becomes much more pronounced for larger arguments.

For $A(4,1) = 2^{16} - 3$ we get rows of lengths

$$2^{16} - 5 \qquad 2^{16} - 6 \qquad 2^{15} - 3 \qquad 13 \qquad 2$$

Good News

We can code the table T as a sequence number (the sequence number of the sequence numbers of the rows), say, $t = \langle T \rangle$.

Claim: There is a primitive recursive relation $\mathrm{table}(t, a, b)$ that checks that T is a correctly formed table for the computation of $A(a, b)$.

We have to check that all the entries are formed according to the Ackermann equations. This requires some helper functions, but is not hard.

E.g., there is a p.r. function $\mathrm{lookup}(t, i, j)$ that returns $T[i, j]$.

PR versus Computable

Obvious Question: how much do we have to add to primitive recursion to capture the Ackermann function?

As it turns out, we need just one modification: we have to allow unbounded search: a type of search where the property we are looking for is still primitive recursive, but we don't know ahead of time how far we have to go.

Unbounded Search vs. Ackermann

Proposition

There is a primitive recursive relation table *such that*

$$A(a,b) = \mathrm{lookup}\Big(\min\big(\, z \mid \mathrm{table}(z,a,b) = 1 \,\big), a, b\Big)$$

Sketch of proof.

We are searching for the sequence number that codes the right table for $A(a,b)$—which must exist, we just don't know how to bound the search.

Once we have the table, we just do a simple lookup.

□

Recursion Stack

In some cases, a recursion based computation unfolds in a very simple, predictable manner. One should try to figure how exactly things work.

With luck, this will make it possible to implement the operations directly on a list.

Alternatively, one can try to find a systematic approach to solving the system of equations, essentially by repeated instantiations and substitutions.

Again, with luck, a simple pattern will emerge that provides a computational shortcut.

Unfolding Ackermann

The computation of, say, $A(2,1)$ can be handled in a very systematic fashion: always unfold the rightmost subexpression.

$$A(2,1) = A(1, A(2,0)) = A(1, A(1,1)) = A(1, A(0, A(1,0))) = \ldots$$

Note that the A's and parens are just syntactic sugar, a better description would be

$$\begin{aligned} &2,1 \rightsquigarrow 1,2,0 \rightsquigarrow 1,1,1 \rightsquigarrow 1,0,1,0 \rightsquigarrow 1,0,0,1 \rightsquigarrow 1,0,2 \rightsquigarrow 1,3 \rightsquigarrow 0,1,2 \\ &\rightsquigarrow 0,0,1,1 \rightsquigarrow 0,0,0,1,0 \rightsquigarrow 0,0,0,0,1 \rightsquigarrow 0,0,0,2 \rightsquigarrow 0,0,3 \rightsquigarrow 0,4 \rightsquigarrow 5 \end{aligned}$$

We can model these steps by a list function Δ defined on sequences of naturals (or, we could use a stack).

List Operation

Here is an algorithm that works on integer lists: initially, the list is (a, b). The algorithm terminates when the list has length 1. A single step looks like so:

$$\mathit{\Delta}(\ldots, 0, y) = (\ldots, y^+)$$

$$\mathit{\Delta}(\ldots, x^+, 0) = (\ldots, x, 1)$$

$$\mathit{\Delta}(\ldots, x^+, y^+) = (\ldots, x, x^+, y)$$

If we encode integer lists as integers, the single-step operation $\mathit{\Delta}$ is primitive recursive. Using actual data structures, $\mathit{\Delta}$ is just about trivial.

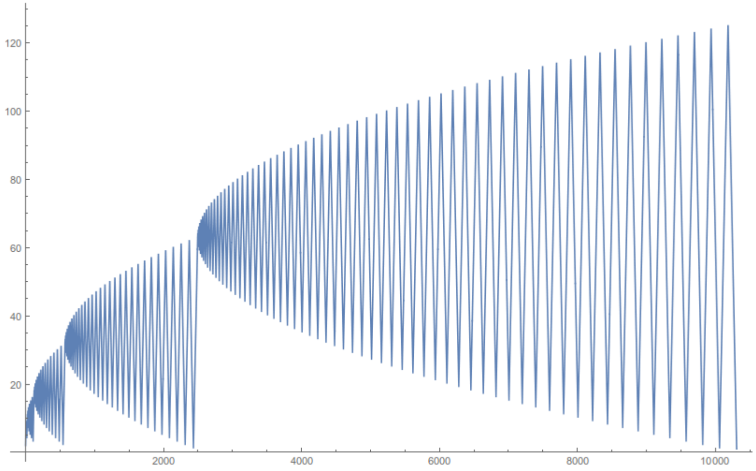
List Algorithm

```
int acker_list(int a, int b)
{
    L = (a,b);
    while( len(L) > 1 )
        L = Delta(L);
    return fst(L);
}
```

Everything is perfectly harmless, except that the loop runs for a long, long time (and the lists get horribly long).

BTW, this is another example of iterating a simple function until a fixed point occurs (just define $\Delta(L) = L$ when $|L| = 1$).

$A(3,4)$



The computation takes 10307 steps, the plot shows the lengths of the list.

Ackermann vs PR

The Ackermann function function $A(x, y)$ fails to be primitive recursive. But

Claim

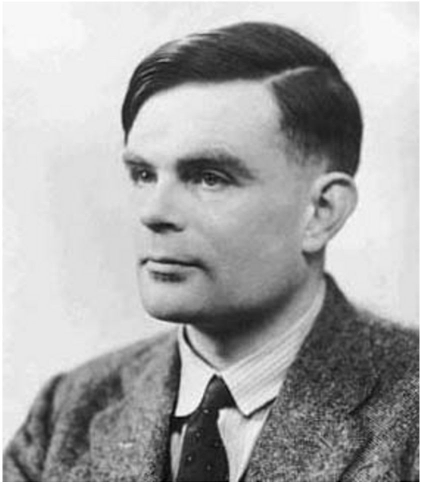
The predicate "$A(x, y) = z$" is primitive recursive.

Proof.

Exploit the monotonicity properties of A to bound the a suitable table.

$\square$

Alan Turing (1912–1954)



The Source

A. Turing

On Computable Numbers, with an Application to the Entscheidungsproblem

Proc. London Math.Soc., 2–42 (1936-7), pp. 230–265.

Turing called his now eponymous devices a-machines, a for automatic. These do not halt: their purpose is to write the infinite binary expansion of a real number on (part of) the tape.

Our description of a TM is the modern one due to Post, Kleene and Davis. The main justification for TMs as the standard model of computation is that they work very well for complexity theory.

Before Turing

There were two models of computation in existence before Turing's seminal 1936 paper, both developed by logicians, and based on elementary ideas in math: equations and functional composition.

- Herbrand-Gödel equations.

  Those describe recursive functions in the most general sense (recursion on multiple variables; by contrast primitive recursive functions allow recursion only on one variable).

- Church's λ-computable functions.

  The λ-calculus is the abstract theory of functional composition. Very elegant, very hard to use for any concrete purpose (there are no data structures).

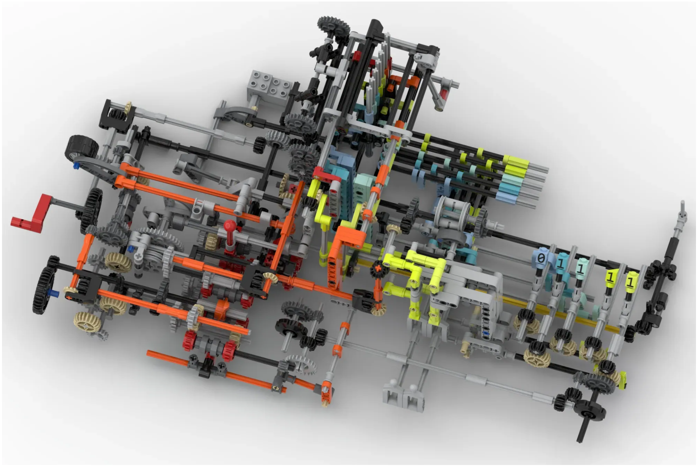
Turing's Machines

Brilliant Idea:
Observe a human computor, then abstract away all the merely biological stuff and formalize what is left.

Everyone agrees that mathematicians compute (among other things such as drinking coffee or proving theorems). So we could try to define an abstract machine that can perform any calculation whatsoever that could be performed in principle by a mathematician, and only those.

Note the hedge "in principle": we will ignore "merely physical" constraints such as the computor dying of old age and decrepitude, or running out of scratch paper after using up the whole universe.

A Concrete Turing Machine



video

No Joke

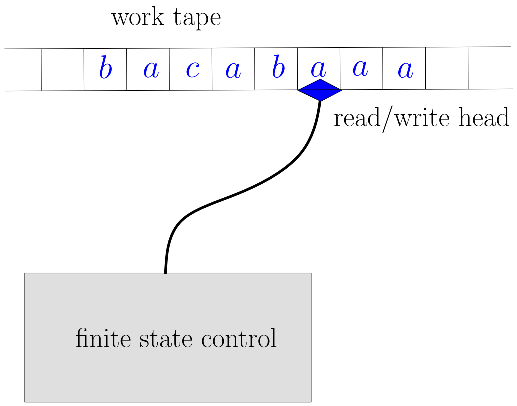
Obviously, whoever built the LEGO Turing machine has a lot of extra time on their hands.

But, the construction brings out a very important issue: Turing machines are clearly physically realizable. Our standard notion of computation is perfectly compatible with our physical universe.

This is not so clear for other mathematical tools: for example, do the reals[†] actually describe physical reality?

[†]If you think the answer is obviously Yes, note that a logician can easily come up with several versions of the reals. Which is the one that corresponds to actual physics?

An Abstract Turing Machine



The Pieces

- A tape: a bi-infinite strip of “paper,” subdivided into cells. Each cell contains a single letter; all but finitely many are just blanks. We refer to this assignment of letters as a tape inscription.
- A read/write head that is positioned at a particular cell. That head can move left and right.
- A finite state control that directs the head: symbols are read and written, the head moves and the internal state of the FSC changes.

Formalization

- alphabet Γ: finite set of symbols, special blank symbol ␣ in Γ
- state set Q: finite set of possible mind configurations
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$: transition function
- a special initial state $q^{\text{init}} \in Q$
- a special halting state $q^{\text{halt}} \in Q$.

Definition

A Turing machine[†] is a structure $\mathcal{M} = \langle Q, \Gamma, \delta; q^{\text{init}}, q^{\text{halt}} \rangle$.

[†] This is not Turing's original definition; he was interested in machines that do not halt and instead produce the infinite binary expansion of a real number. For us, halting is key.

Warning

The core of a Turing machine is the transition system $\langle Q, \Gamma, \delta \rangle$, it determines how the machine computes.

Some additional information like designated initial and halting states are convenient to organize the exact details regarding in particular input and output.

Specifying q^{init} and q^{halt} is one useful scenario, but there are others. E.g., it may be preferable to have to halting states q^{yes} and q^{no} that express acceptance or rejection. Or there may be no halting state at all (to generate recursively enumerable sets).

Turing Machines are Brilliant

Of all the standard models of computation, Turing machines are the most compelling when it comes to capturing the intuitive notion of computability: arguably they correspond to the abilities of a human computor[†].

In addition, TMs are fairly simple, certainly much more palatable than Herbrand-Gödel equations or Church's λ-calculus. Alas, they are not quite as nice as models that are closer to actual hardware such as register machines or random access machines, let alone programming languages.

One key advantage: they naturally work on strings and are ideally suited for complexity theory, unlike some of the other models.

[†] Gödel was completely convinced by Turing, not by his own model, nor by Church.

Wittgenstein

Turing's "Machines".

These machines are humans who calculate.

Turing Machines Suck

One substantial drawback of TMs is that it is hugely cumbersome to construct interesting examples. For example, try to actually write out the transition table for a machine that does the following:

- multiplication of numbers in binary
- depth-first search on a graph
- evaluation of primitive recursive functions
- universal Turing machine

The point here is not to wax poetically about whether this could be done, just do it. And verify that your machine is correct.

Lame Example: Successor Function

Tape alphabet $\Gamma = \{\sqcup, |\}$

States $Q = \{q^{\text{init}}, q_1, q_2, q^{\text{halt}}\}$

Initial state q^{init}, halting state q^{halt}.

The transition function δ is given be the following table:

p	σ	$\delta(p,\sigma)$		
q^{init}	$\sqcup$	q_1	$\sqcup$	$+1$
q_1	$\sqcup$	q_2	$\|$	-1
q_1	$\|$	q_1	$\|$	$+1$
q_2	$\sqcup$	q^{halt}	$\sqcup$	0
q_2	$\|$	q_2	$\|$	-1

As written, the behavior in state q^{init} and symbol $|$ is undefined; this does not matter since the situation never arises during an actual computation.

Input/Output Convention

The coding convention used is unary

$$\mathbb{N} \ni n \rightsquigarrow \underbrace{||\ldots|}_{n} \in \{|\}^{\star}$$

The machine starts in state q^{init} and the read head is immediately before in the inpute.

In the end, the machine is in state q^{halt} and the head has returned to its initial position. The output follows immediately.

And Diagrams?

Finite state machines can be nicely represented by diagrams that have labeled edges of the form

$$p \xrightarrow{a} q$$

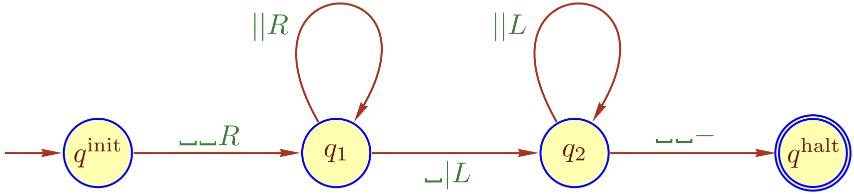
to indicate that $(p, a, q) \in \delta$.

It is still possible to draw diagrams for Turing machines, the transitions now take the form

$$p \xrightarrow{a:b:d} q$$

to indicate that $\delta(p, a) = (q, b, d)$. Alas, this produces a lot of visual clutter and is not as useful as for finite state machines.

Last Example



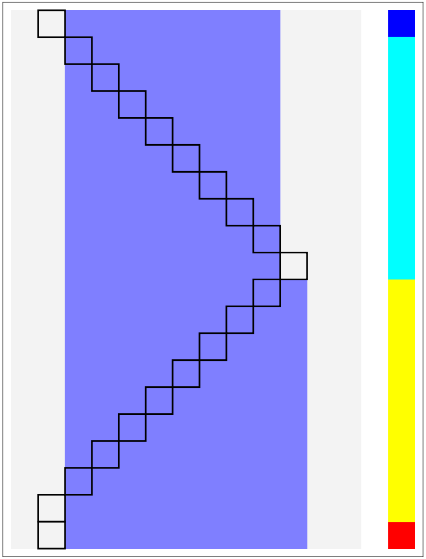
For clarity, we have written $R, -, L$ for the displacements.

Visualization

One extremely pleasant feature of Turing machines is that one can easily visualize computations.

Suprisingly, these pictures can help a lot to understand the nature of a given Turing machine. More on this in our discussion of Busy Beavers.

Sample Run



Addition

Just to make the point about coding conventions, this time we are using *fat unary* notation, n is represented by

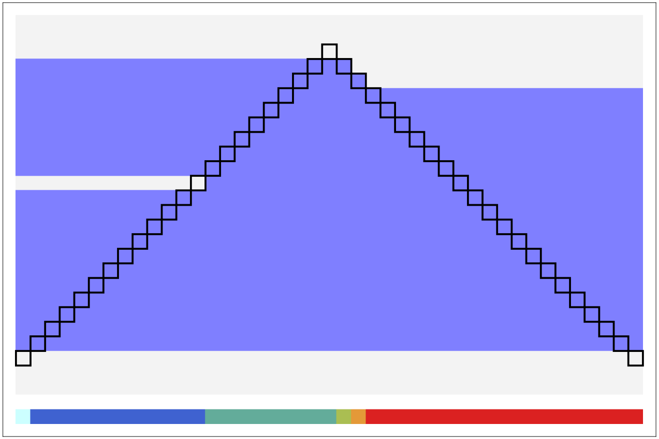
$$n \mapsto \underbrace{|||\ldots||}_{n+1}$$

Hence we have to erase two 1s at the end:

0	␣	1	␣	$+1$
1	␣	2	$\vert$	$+1$
1	$\vert$	1	$\vert$	$+1$
2	␣	3	␣	-1
2	$\vert$	2	$\vert$	$+1$
3	$\vert$	4	␣	-1
4	$\vert$	5	␣	-1
5	␣	6	␣	0
5	$\vert$	5	$\vert$	-1

0 is the initial state, 6 is the final state.

Sample Run



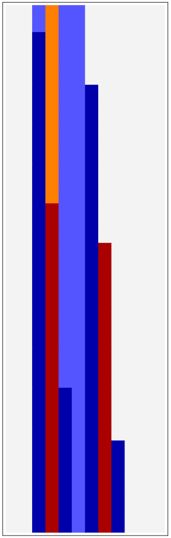
Copy

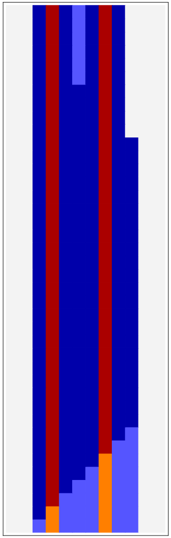
Here is a Turing machine[†] that copies its input: $x \mapsto xx$.

```
rules = {
    {0, 0} → {1, 0, 1},                              (* start *)
    {1, aa} → {1, aa, 1}, {1, bb} → {1, bb, 1},      (* look for unmarked letter *)
    {1, a} → {2, aa, 1}, {1, b} → {3, bb, 2},        (* change s to ss, remember *)
    {1, 0} → {6, 0, -1},
    {2, s_?Positive} → {2, s, 1},                    (* seek right blank *)
    {2, 0} → {4, aa, 1},                             (* change to ss *)
    {3, s_?Positive} → {3, s, 1},                    (* seek right blank *)
    {3, 0} → {4, bb, 1},                             (* change to ss *)
    {4, 0} → {5, 0, -1},
    {5, s_?Positive} → {5, s, -1},
    {5, 0} → {1, 0, 1},
    {6, s_?Positive} → {6, s, 1},                    (* seek right blank *)
    {6, 0} → {7, 0, -1},
    {7, 3} → {7, 1, -1},                             (* seek left blank, unmark *)
    {7, 4} → {7, 2, -1}                              (* halt  *)
  } /. {a -> 1, b -> 2, aa -> 3, bb -> 4}
```

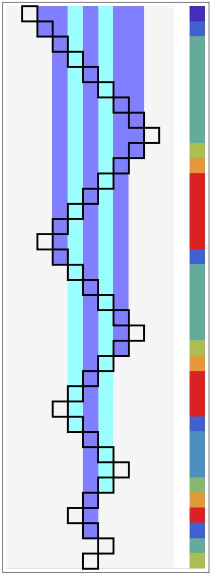
[†]Written in Mathematica, so it can actually be executed.

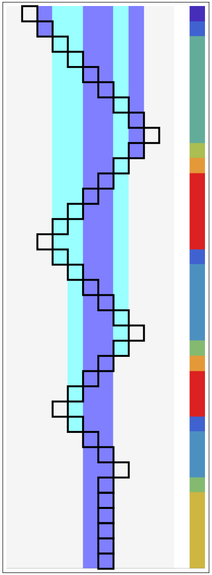
A Run





Palindromes





Correctness

Exercise

What would a complete correctness proof for the Turing machine that performs unary addition look like? What is difficult about the proof?

Exercise

Construct a Turing machine that performs addition when the input is given in binary. What would the execution pictures look like in this case? How hard is a correctness proof?

Exercise

Figure out how the palindrome TM works and prove that it is correct (you need a convention for accepting versus rejecting computations).

Exercise (Hard)

Show that any one-tape Turing machine requires quadratic time to recognize palindromes.