Recall that we are looking for a formal definition of computability. So far we have encountered primitive recursive functions. We have seen that primitive recursive functions provide a huge supply of intuitively computable functions. So could this be the final answer? Sadly, NO . . .

General Recursion There are intuitively computable functions, based on a general type of recursion, that fail to be primitive recursive.

Insane Growth Some computable functions have stupendous growth rates.

Evaluation Computability forces functions to be partial in general, we need to adjust our framework correspondingly.

Ackermann’s Function (1928)

We have seen that course-of-value recursion does not push us out of the realm of primitive recursive functions. Similarly, double recursion won’t hurt.

And yet, there is more to recursion than this. Define the Ackermann function $A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ by the following double recursion. We write $x^+$ instead of $x + 1$.

$A(0, y) = y^+$
$A(x^+, 0) = A(x, 1)$
$A(x^+, y^+) = A(x, A(x^+, y))$

On the surface, this looks more complicated than primitive recursion. Of course, it is not at all clear that Ackermann could not somehow be turned into a p.r. function.

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:

$A_0(y) = y^+$
$A_{x^+}(0) = A_x(1)$
$A_{x^+}(y^+) = A_x(A_x^+(y))$

From this it follows easily by induction that

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

The Bottom Hierarchy

$A(0, y) = y^+$
$A(1, y) = y^{++}$
$A(2, y) = 2y + 3$
$A(3, y) = 2^{y+3} - 3$
$A(4, y) = 2^{2^y} - 3$

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 $y + 3$.

This usually called super-exponentiation or tetration and often written $^yy$ or $x \uparrow \uparrow y$. 
The Mystery of $A(6, 6)$

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

$A(5, y) \approx$ super-super exponentiation
$A(6, y) \approx$ an unspeakable horror
$A(7, y) \approx$ speechless

For level 5, one can get some vague understanding of iterated super-exponentiation, but things start to get murky. 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: “Wovon man nicht sprechen kann, darüber muss man schweigen.”

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.

Ackermann vs. PR

Theorem

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

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

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.

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).

```c
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.

Memoizing

We could memoize the values that are computed during a call to $A(a, b)$.

It is not hard to see that whenever $A(x, y)$ is in the hash table, so is $A(x, z)$ for all $z < y$ (except for $x = 0$). Think of the table as having $x$ rows.

But note that the computation of $A(a, b)$ will require entries $A(a - 1, z)$ for $z > b$. So each row in the table gets longer and longer as $x$ gets smaller.

For example, the computation of $A(3, 3) = 61$ requires $A(2, 29)$ and $A(1, 59)$.

The size of this hash table spins out of control very quickly.

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.
Proposition
There is a primitive recursive relation $R$ such that
$$A(a,b) = (\min (z \mid R(a,b,z)))_0$$

Sketch of proof. Think of $z$ as a pair $\langle c,t \rangle$ where $t$ encodes a data structure that represents the whole computation of the Ackermann function on input $a$ and $b$, something like a huge memoization table. $c$ is the final result of this computation. $

Here is a much better description.

Ackermann and Lists

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

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

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

$$2, 1 \sim 1, 2 \sim 1, 1 \sim 1, 1 \sim 0, 1, 0 \sim 1, 0, 0, 1 \sim 1, 0, 2 \sim 1, 3 \sim 0, 1, 2 \sim 0, 0, 1 \sim 0, 0, 1, 0 \sim 0, 0, 0, 1 \sim 0, 0, 2 \sim 0, 0, 3 \sim 0, 3 \sim 4, 5$$

We can recover the original expression by applying $A$ like a right-associative binary operator to these argument lists.

List Operation

$$\Delta(..., 0,y) = (...,y+)$$
$$\Delta(...,x+,0) = (...,x,1)$$
$$\Delta(...,x+,y+) = (...,x,x+,y)$$

Since $A$ is total, we know that for any $a$ and $b$ there is some time $t$ such that
$$\Delta^t(a,b) = (c)$$

Clearly this condition is primitive recursive in $(a,b,c,t)$.

Insane Growth

Informally, the Ackermann function cannot be primitive recursive because it grows far too fast. On the other hand, it does not really have a particular purpose other than that.

We will give another example of mind-numbing growth based on actual counting problem. To this end, it is easier to use a slight variant of the Ackermann function.

$$B_1(x) = 2x$$
$$B_{k+}(x) = B_{k+1}(1)$$

$B_{k+1}(1)$ means: iterate $B_k$ $x$-times on 1. So $B_1$ is doubling, $B_2$ exponentiation, $B_3$ super-exponentiation and so on.

In general, $B_k$ is closely related to $A_{k+1}$. 

Subsequence Order

Recall the subsequence ordering on words where \( u = u_1 \ldots u_n \) precedes \( v = v_1v_2\ldots v_m \) if there exists a strictly increasing sequence \( 1 \leq i_1 < i_2 < \ldots < i_n \leq m \) of positions such that \( u = v_{i_1}v_{i_2}\ldots v_{i_n} \).

In symbols: \( u \sqsubseteq v \).

In other words, we can erase some letters in \( v \) to get \( u \).

Subsequence order is not total unless the alphabet has size 1.

Note that subsequence order is independent of any underlying order of the alphabet (unlike, say, lexicographic or length-lex order).

Higman’s Lemma

Theorem (Higman’s 1952)
Every antichain in the subsequence order is finite.

Proof. Here is the Nash-Williams proof (1963): assume there is an infinite antichain.

For each \( n \), let \( x_n \) be the length-lex minimal word such that \( x_0, x_1, \ldots, x_n \) starts such an antichain, producing a sequence \( x = (x_n) \).

Construct a new sequence \( y = (y_i) \) by choosing a letter \( \alpha \) that appears infinitely often as the first letter in \( (x_n) \) and copying the words up to the first occurrence of one of these \( \alpha \)-words. Follow by all the \( \alpha \)-words, but with the first letter removed.

One can check that the new sequence \( (y_i) \) is also an infinite antichain. But it violates the minimality constraint on \( (x_i) \), contradiction.

\[ \square \]

Note that this proof is highly non-constructive. A lot of work has gone into developing more constructive versions of the theorem, but things get a bit complicated.

See Seisenberger.

Proof, contd.

Friedman’s Self-Avoiding Words

For a finite or infinite word \( x \) write \( x[i] \) for the block \( x_i, x_{i+1}, \ldots, x_{2i} \).

We will always assume that \( i \leq |x|/2 \) when \( x \) is finite.

Bizarre Definition: A word is self-avoiding if for \( i < j \) the block \( x[i] \) is not a subsequence of \( x[j] \).

The following is an easy consequence of Higman’s theorem.

Theorem
Every self-avoiding word is finite.

How Long?

Write \( \Sigma_k \) for an alphabet of size \( k \).

By the last theorem and König’s lemma, the set \( S_k \) of all finite self-avoiding words over \( \Sigma_k \) must itself be finite.

But then we can define the following function:

\[ \alpha(k) = \max \{|x| \mid x \in S_k \} \]

So \( \alpha(k) \) is the length of the longest self-avoiding word over \( \Sigma_k \).

Note that \( \alpha \) is intuitively computable: we can build the tree of all self-avoiding words by brute-force.
Algorithmic Details

More precisely, we can build a trie $T$ of words over alphabet $\Sigma_k$. Originally, $T$ contains only $\varepsilon$.
At the next round, consider any branch in $T$ labeled $x$. If $x$ is not self-avoiding, freeze the corresponding leaf.
Otherwise extend the leaf by adding $k$ children.

Then, after finitely many steps, all leaves in $T$ will be frozen and we can measure the longest branch.

Exercise
Figure out the details.

How Big?

- Trivially, $\alpha(1) = 3$.
- A little work shows that $\alpha(2) = 11$, as witnessed by $abbbaaaaaaa$.
- But
  \[
  \alpha(3) > B_{7198}(158386),
  \]
  an incomprehensibly large number.

Smelling salts, anyone?
It is truly surprising that a function with as simple a definition as $\alpha$ should exhibit this kind of growth.

It’s a Feature

At this point one might wonder whether our definition of computability is perhaps a bit off – we did not intend to deal with monsters like $\alpha$.

Alas, as it turns out this is a feature: all reasonable definitions of computability admit things like $\alpha$, and worse.

It is a fundamental property of computable functions that some of them have absurd growth rates.

Evaluation

Very rapidly growing functions such as the Ackermann function are one reason primitive recursion is not strong enough to capture computability. Here is another obstruction: we really need to deal with partial functions.
Recall the evaluation operator for our PR terms:

\[
\text{eval}(\tau, x) = \text{value of } \tau^* \text{ on input } x
\]

It is clear that eval is intuitively computable (take a compilers course). In fact, it is not hard to implement in eval in any modern programming language.

Question: Could eval be primitive recursive?
A useless answer would be to say no, the types don’t match.

Indices

The first argument of eval is a term $\tau$ in our PR language, so our first step will be to replace $\tau$ by an index $\hat{\tau} \in \mathbb{N}$.

The index $\hat{\tau}$ will be constructed in a way that makes sure that all the operations we need on indices are clearly primitive recursive.

The argument vector $x \in \mathbb{N}^n$ will also be replaced by its sequence number $(x_1, \ldots, x_n)$. Hence we will be able to interpret eval as a function of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

and this function might potentially be primitive recursive.
Coding PR

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

\[
0 \langle 0, 0 \rangle \\
P^n_i \langle 1, n, i \rangle \\
\langle 2, 1 \rangle \\
\langle 3, n, \hat{h}, \hat{g} \rangle \\
\langle 4, m, \hat{h}, \hat{g}_1, \ldots, \hat{g}_n \rangle
\]

Thus for any index \( e \), the first component \((e)_0\) indicates the type of function, and \((e)_1\) indicates the arity.

There is nothing sacred about this particular way of coding PR terms, there are many other, equally natural ways.

Diagonalization

Now suppose eval is p.r., and define the following function

\[
f(x) := \text{eval}(x, x) + 1
\]

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

\[
f(e) = \text{eval}(e, e) + 1 = f(e) + 1
\]

and we have a contradiction.

So eval is another example of an intuitively computable function that fails to be primitive recursive.

This example may be less sexy than the Ackermann function, but it appears in similar form in other contexts.

Partial Functions

How do we avoid the problem with eval?

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

For a CS person, this is a fairly uncontroversial idea: everyone 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.

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 are part of 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.

General computability requires partial functions, basta.

Notation Warning

We write

\[ f: A \rightarrow B \]

for a partial function from \( A \) to \( B \).

Terminology:

- domain \( \text{dom } f = A \)
- codomain \( \text{cod } f = B \)
- support \( \text{spt } f = \{ a \in A \mid \exists b. f(a) = b \} \)

Warning: Some misguided authors mean support when they say domain.

Kleene’s Notation

We write

\[ \{e\} \]

for the \( e \)th computable function, \( e \geq 0 \). Here the index \( e \) is a sequence number, but it is helpful to think of it as a program (in some suitable language).

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

\[ \{e\}(x) \simeq y \]

to indicate that \( \{e\} \) with input \( x \) returns output \( y \) after finitely many steps.
More Convergence

To express convergence we also write
\[
\{e\}(x) \downarrow
\]
if \{e\} on input \(x\) terminates and produces some output, and
\[
\{e\}(x) \uparrow
\]
when the computation fails to terminate.

For example, Kleene equality \(\{e\}(x) \simeq \{e'\}(x)\) should be interpreted as:
- either \(\{e\}(x) \downarrow\) and \(\{e'\}(x) \downarrow\) and the output is the same; or
- \(\{e\}(x) \uparrow\) and \(\{e'\}(x) \uparrow\).

A Different Model

What now? Turn our problems into solutions: find a model of computation that, by design, can handle Ackermann, Friedman’s \(\alpha\) (and other perverse examples of computable functions) and partial evaluation.

We will do this by using a machine model, another critical method to define computability and complexity classes. There are many plausible approaches, we’ll start with a model that is slightly reminiscent of assembly language programming, only that our language is much, much simpler than real assembly languages.

Functions computed by these machines will turn out to be partial in general, so this might fix all our problems.

Instruction Set

Our instruction set is very, very primitive:
- \(\text{inc } r \ k\)
  increment register \(R_r\), goto \(k\).
- \(\text{dec } r \ k \ l\)
  if \(|R_r| > 0\) decrement register \(R_r\) and goto \(k\), otherwise goto \(l\).
- \(\text{halt}\)
  well . . .

The gotos refer to line numbers in the program; note that there is no indirect addressing. These machines are sometimes called counter machines.

Register Machine

Definition
A register machine (RM) consists of a finite number of registers and a control unit.

We write \(R_0, R_1, \ldots\) for the registers and \([R_i]\) for the content of the \(i\)th register: a single natural number.

Note: there is no bound on the size of the numbers stored in our registers, any number of bits is fine. This is where we break physics.

The control unit is capable of executing certain instructions that manipulate the register contents.

RM Programs

Definition
A register machine program (RMP) is a sequence of RM instructions \(P = I_0, I_1, \ldots, I_{n-1}\).

For example, the following program performs addition:

```plaintext
// addition R0 R1 --> R2
0: dec 0 1 2
1: inc 2 0
2: dec 1 3 4
3: inc 2 2
4: halt
```
Hardware versus Software

Since we have no intentions of actually building a physical version of aegister machine, this distinction between register machines and register
machines programs is slightly silly.

Still, it’s good mental hygiene: we can conceptually separate the physical
hardware that supports some kind of computation from the programs
that are executed on this hardware. For real digital computers this makes
perfect sense. A similar problem arises in the distinction between the
syntax and semantics of a programming language.

And, it leads to the juicy question: what is the relationship between
physics and computation? We’ll have more to say about this in a while.

Reasonable I/O Conventions

- Given input arguments \( a = (a_0, \ldots, a_{k-1}) \in \mathbb{N}^k \), set the input
  registers: \([R_i] = a_i\).
- All other registers in \( P \) are initialized to 0.
- Then run the program.
- If it terminates, read off the values of \( R_j \), \( j \in O \), producing the
  result \( b = (b_1, \ldots, b_\ell) = f(a) \).
- If \( P \) does not terminate, \( f(a) \) is undefined.

Run the Program?

To describe a computation of \( P \) we need to explain what a snapshot of a
computation is, and how get from one snapshot to the next. Clearly, for
RMPs we need two pieces of information:

- the current instruction, and
- the contents of all registers.

A configuration of \( P \) is a pair \( C = (p, x) \in \mathbb{N} \times \mathbb{N}^n \).

Steps in a Computation

Configuration \((p, x)\) evolves to configuration \((q, y)\) in one step under \( P \) if

- \( I_p = \text{inc } r k \):
  
  \( q = k \) and \( y = x[x_r \mapsto x_r + 1] \).

- \( I_p = \text{dec } r k 1 \):
  
  \( x_r > 0 \), \( q = k \) and \( y = x[x_r \mapsto x_r - 1] \) or
  
  \( x_r = 0 \), \( q = l \) and \( y = x \).

Notation: \((p, x) \xrightarrow{P} (q, y)\).

Note that if \((p, x)\) is halting (i.e. \( I_p = \text{halt} \)) there is no next
configuration. Ditto for \( p \geq n \).

Whole Computation

Define

\[
(p, x) \xrightarrow{P} (q, y) \iff (p, x) = (q, y)
\]

\[
(p, x) \xrightarrow{P} (q, y) \iff \exists q', y' \ (p, x) \xrightarrow{P}^{-1} (q', y') \xrightarrow{P} (q, y)
\]

\[
(p, x) \xrightarrow{P} (q, y) \iff \exists t \ (p, x) \xrightarrow{P} t (q, y)
\]

A computation (or a run) of \( P \) is a sequence of configurations \( C_0, C_1, C_2, \ldots \) where \( C_i \xrightarrow{P} C_{i+1} \).
Finite versus Infinite

Note that a computation may well be infinite:

0: inc 0 0

has no terminating computations at all. More generally, for some particular input a computation on a machine may be finite, and infinite for other inputs.

Also, computations may get stuck. The program

0: inc 0 1

cannot execute the first instruction since there is no goto label 1.

Cleaning Up

Note that we may safely assume that $P = I_0, I_1, \ldots, I_{n-1}$ uses only registers $R_i$, $i < n$, so all numbers in the instructions are bounded by $n$.

Furthermore, we may assume that all the goto targets $k$ lie in the range $0 \leq k < n$. Also, $I_{n-1}$ is a halt instruction, and there are no others.

It follows that these clean RMs cannot get stuck, every computation either ends in halting, or is infinite. From now on, we will always assume that our programs are syntactically correct in this sense.

Exercise

Write a program that transforms a RM program into an “equivalent” one that is syntactically correct.

Termination

Hence, we have two kinds of computations: finite ones (that necessarily end in a halt instruction), and infinite ones. We will write

$$(C_i)_{i<n} \quad \text{and} \quad (C_i)_{i<\omega}$$

for finite versus infinite computations.

$\omega$ is the first infinite ordinal, more later. If you don’t like ordinals, replace $\omega$ by some meaningless but pretty symbol like $\infty$.

Computing a Function

The initial configuration for input $a \in \mathbb{N}^k$ is $E_a = (0, (a, 0))$.

Definition

A RMP $P$ computes the partial function $f : \mathbb{N}^k \rightarrow \mathbb{N}^\ell$ if for all $a \in \mathbb{N}^k$ we have:

- If $a$ is in the domain of $f$, then the computation of $P$ on $C_0 = E_a$ terminates in configuration $C_t = (n-1, y)$ where $f(a) = (y_k, \ldots, y_{k+\ell-1})$ and $I_{n-1} = \text{halt}$.
- If $a$ is not in the support of $f$, then the computation of $P$ on $E_a$ fails to terminate.

A Subtlety

Recall that according to our convention, it is not admissible that an RM program could get stuck (because a goto uses a non-existing label). What if we allowed arbitrary RM programs instead of only clean ones?

The class of computable functions would not change one bit, our definitions are quite robust under (reasonable) modifications. This is a good sign, fragile definitions are usually of little interest.

Exercise

Modify the definition so “getting stuck” is allowed and show that we obtain exactly the same class of partial functions this way. Invent RMs without a halt instruction.

It’s A Clone

Clearly we can generalize the notion of a clone from total functions to partial ones.

Proposition

Register machines computable functions form a clone, containing the clone of primitive recursive functions.

Exercise

Figure out the details.
**Time Complexity**

The number of steps in a finite computation provides a measure of complexity, in this case time complexity.

Given a RM $P$ and some input $x$, let $(C_i)_{i < N}$, where $N \leq \omega$, be the computation of $P$ on $x$.

We write the time complexity of $P$ as

$$T_P(x) = \begin{cases} N & \text{if } N < \omega, \\ \omega & \text{otherwise.} \end{cases}$$

If you are worried about $\omega$ just read it as $\infty$. Alternatively, we could use $N - 1$ as our step-count.

This may sound trivial, but it’s one of the most important ideas in all of computer science. Period.

**Named Registers**

To make RMPs slightly easier to read we use names such as $X$, $Y$, $Z$ and so forth for the registers.

This is just a bit of syntactic sugar, if you like you can always replace $X$ by $R_0$, $Y$ by $R_1$ and so forth.

And we will be quite relaxed about distinguishing register $X$ from its content $[X]$.

---

**Digression: Notation**

There is actually something very important going on here: we are trying to produce notation that works well with the human cognitive system.

Humans are exceedingly bad at dealing with fully formalized systems; in fact, we really cannot read formal mathematics except in the most trivial (and useless) cases. Try reading Russell-Whitehead’s Principia Mathematica if you don’t believe me.

The current notation system in mathematics evolved over centuries and is very carefully fine-tuned to work for humans.

Computers need an entirely different presentation and it is very difficult to move between the two worlds.

**Example: Multiplication**

Here is a program that multiplies registers $X$ and $Y$, and places the product into $Z$. $U$ is auxiliary.

```
// multiplication X Y --> Z
0: dec X 1 6
1: dec Y 2 4
2: inc Z 3
3: inc U 1
4: dec U 5 0
5: inc Y 4
6: halt
```

**A Computation**

```
0 (2, 2, 0, 0) 1 (0, 2, 2, 0)
1 (1, 2, 0, 0) 2 (0, 1, 2, 0)
2 (1, 1, 0, 0) 3 (0, 1, 3, 0)
3 (1, 1, 1, 0) 1 (0, 1, 3, 1)
1 (1, 1, 1, 1) 2 (0, 0, 3, 1)
2 (1, 0, 1, 1) 3 (0, 0, 4, 1)
3 (1, 0, 2, 1) 1 (0, 0, 4, 2)
1 (1, 0, 2, 2) 4 (0, 0, 4, 2)
4 (1, 0, 2, 2) 5 (0, 0, 4, 1)
5 (1, 0, 2, 1) 4 (0, 1, 4, 1)
4 (1, 1, 2, 1) 5 (0, 1, 4, 0)
5 (1, 1, 2, 0) 4 (0, 2, 4, 0)
4 (1, 2, 2, 0) 0 (0, 2, 4, 0)
0 (1, 2, 2, 0) 6 (0, 2, 4, 0)
```

**Exercise**

Determine the time complexity of the multiplication RM.

---

**Time Complexity?**
Example: Binary Digit Sum

The following RMP computes the number of 1's in the binary expansion of \( X \), the so-called binary digit sum of \( x \).

```plaintext
// binary digitsum of X --> Z
0: dec X 1 4
1: dec X 2 3
2: inc Y 0
3: inc Z 4
4: dec Y 5 8
5: inc Y 6
6: dec Y 7 0
7: inc X 6
8: halt
```

Digit Sum

The (binary) digit sum is actually quite useful in some combinatorial arguments.

Exercises

1. **Exercise**
   Show that every primitive recursive function can be computed by a register machine. Implement a prec to RM compiler.

2. **Exercise**
   Suppose some register machine \( M \) computes a total function \( f \). Why can we not conclude that \( f \) is primitive recursive?

Prepend \( b \) to \( x \)
Recall: Coding

Recall the three coding functions from last time:

\[
\langle \cdot \rangle : \mathbb{N}^* \rightarrow \mathbb{N} \\
\text{dec} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
\text{len} : \mathbb{N} \rightarrow \mathbb{N}
\]

Note that \text{len} is just the binary digit-sum.

One can check that \text{dec} can be computed by a fairly small register machine. As usual, for \langle \cdot \rangle we would have to fix the number of arguments.

Self-Reference

As Gödel has shown devastatingly in his incompleteness theorem, self-reference is an amazingly powerful tool.

On occasion, it wreaks plain havoc: his famous incompleteness theorem takes a wrecking ball to first-order logic.

However, in the context of computation, self-reference turns into a genuine resource. We developed our coding machinery to show that standard discrete structures can be expressed as natural numbers and thus be used in an RPM. But an RPM is itself a discrete structure, so RPMs can compute with (representations of) RPMs.

This leads to the fundamental concept of **universality**.

Coding RMPs

Note that a single instruction of an RMP can also be coded as a number:

- \text{halt} \quad \langle 0 \rangle
- \text{inc} \ r \ k \quad \langle r, k \rangle
- \text{dec} \ r \ k \ l \quad \langle r, k, l \rangle

And a whole program can be coded as the sequence number of these numbers.

Example: Addition

For example, the simplified addition program

```
// addition  R0 + R1 --> R1
0:  dec 0 1 2
1:  inc 1 0
2:  halt
```

has code number

\[
\langle \langle 0,1,2 \rangle, \langle 1,0 \rangle, \langle 0 \rangle \rangle = 88098369175552.
\]

Note that this code number does not include I/O conventions, but it is not hard to tack these on if need be.
### Turing and Universality

This special property of digital computers, that they can mimic any discrete state machine, is described by saying that they are universal machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.

Alan Turing (1950)

### WTF?

Let’s define the state complexity of a RMP to be its length, the number of instructions used in the program.

An RMP of complexity 1 is pretty boring, 2 is slightly better, 3 better yet; a dozen already produces some useful functions. With 1000 states we can do even more, let alone with 1000000, and so on.

Except that the “so on” is plain wrong: there is some magic number $N$ such that every RMP can already be simulated by a RMP of state complexity just $N$: we can hide the complexity of the computation in one of the inputs. As far as state complexity is concerned, maximum power is already reached at $N$.

This is counterintuitive, to say the least.

### Simulating Random Access Memory

How does one construct a universal computer? According to the last section, we can code a RMP $P = I_0, I_1, \ldots, I_{n-1}$ as an index $e$, usually called an index for $P$ in this context.

Moreover, we can access the instructions in the program by performing a bit of arithmetic on the index. Note that we can do this non-destructively by making copies of the original values.

So, if index $e$ and some line number $p$ (for program counter) are stored in registers we can retrieve instruction $I_p$ and place it into register $I$.

### Simulating a RM

Suppose we are given a sequence number $e$ that is an index for some RMP $P$ requiring one input $x$.

We claim that there is a universal register machine (URM) $U$ that, on input $e$ and $x$, simulates program $P$ on $x$.

Alas, writing out $U$ as a pure RMP is too messy, we need to use a few “macros” that shorten the program.

Of course, one has to check that all the macros can be removed and replaced by corresponding RMPs, but that is not very hard.

### Macros

- **copy r s k**
  Non-destructively copy the contents of $R_r$ to $R_s$, goto $k$.
- **zero r k l**
  Test if the content of $R_r$ is 0; if so, goto $k$, otherwise goto $l$.
- **pop r s k**
  Interpret $R_r$ as a sequence number $a = (b, c)$; place $b$ into $R_s$ and $c$ into $R_r$, goto $k$. If $|R_r| = 0$ both registers will be set to 0.
- **read r t s k**
  Interpret $R_r$ as a sequence number and place the $|R_t|$th component into $R_s$, goto $k$. Halt if $|R_t|$ is out of bounds.
- **write r t s k**
  Interpret $R_r$ as a sequence number and replace the $|R_t|$th component by $|R_s|$, goto $k$. Halt if $|R_t|$ is out of bounds.

### Turing 1936

Computational universality was established by Turing in 1936 as a purely theoretical concept.

Surprisingly, within just a few years, practical universal computers (at least in principle) were actually built and used:

- **1941** Konrad Zuse, Z3
- **1943** Tommy Flowers, Colossus
- **1944** Howard Aiken, Mark I
- **1946** Prosper Eckert and John Mauchley, ENIAC
Here are the registers used in $U$:

- **$x$**: input for the simulated program $P$
- **$E$**: code number of $P$
- **$R$**: register that simulates the registers of $P$
- **$I$**: register for instructions of $P$
- **$p$**: program counter

**Hack**: $x$ is also used as an auxiliary variable to keep the whole program small.

```
0: copy E R 1    // R = E
1: write R p x 2  // R[0] = x
2: read E p I 3   // I = E[p]
3: pop I r 4      // r = pop(I)
4: zero I 13 5    // if( I == 0 ) halt
5: pop I p 6      // p = pop(I)
6: read R r x 7   // x = R[r]
7: zero I 8 9     // if( I != 0 ) goto 9
8: inc x 12       // x++; goto 12
9: zero x 10 11   // if( x != 0 ) goto 11
10: pop I p 2     // p = pop(I)
11: dec x 12      // x--
12: write R r x 2  // R[r] = x; goto 2
13: halt
```

**Size?**

Of course, the 13 lines in this universal machine are a bit fraudulent, we really should expand all the macros. Still, the resulting honest register machine would not be terribly large.

And there are lots of ways to optimize.

**Exercise**

Give a reasonable bound for the size of the register machine obtained by expanding all macros.

**Exercise**

Try to build a smaller universal register machine.

If we define computability in terms of RMs, it follows that the Halting Problem for RMs is undecidable: there is no RM that takes an index $e$ as input and determines whether the corresponding RM $P_e$ halts (on all-zero registers).

Since RMs are perfectly general computational devices, this means that there is no algorithm to determine whether RM $P_e$ halts; the Halting Problem is undecidable.

**A Universal Turing Machine**

**Exercise**

Figure out what this picture means.

**Exercise (Very Hard)**

Prove that this is really a universal Turing machine.