More Recursion

- Register Machines
- Universality
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 . . .

- There are clearly computable functions (based on a more general type of recursion) that fail to be primitive recursive. Some of these functions have stupendous growth rates.
- Computability forces functions to be partial in general, we need to adjust our framework correspondingly.
In primitive recursion one often encounters terms like $x + 1$.

This is fine, but one can remove a bit of visual clutter by writing $x^+$.

And, of course, $x^{++}$ stands for $x + 2$, and so on.
The key idea behind primitive recursive functions is to formalize a certain type of recursion (or, if you prefer, inductive definition). As we have seen, even some extension of this basic idea like course-of-value recursion remains within this framework.

Here is another failed example: simultaneous recursion. To simplify matters, let us only consider two simultaneously defined functions as in

\[
\begin{align*}
  f_i(0, y) &= g_i(y) \\
  f_i(x^+, y) &= h_i(x, f_1(x, y), f_2(x, y), y)
\end{align*}
\]

where \( i = 1, 2 \).
$f_1(0) = 1$

$f_1(x^+) = f_2(x)$

$f_2(0) = 0$

$f_2(x^+) = f_1(x)$

Then $f_1$ is the characteristic function of the even numbers. We’ll see better examples later when we talk about context free grammars.

**Lemma**

*Simultaneous recursion is admissible for primitive recursive functions.*
And yet, there is more to recursion that this. Define the Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by the following double recursion (this version is actually due to Rózsa Péter and simpler than Ackermann’s original function):

\[
\begin{align*}
A(0, y) &= y^+ \\
A(x^+, 0) &= A(x, 1) \\
A(x^+, y^+) &= A(x, A(x^+, y))
\end{align*}
\]

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.
Here is a 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 handled by the compiler and the execution stack. So this provides overwhelming evidence that the Ackermann function is intuitively computable.
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 $A(1, 59)$.

Note that the hash table provides a proof that $A(3, 3) = 61$. 

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 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 $^yx$ or $x^{^{^{^y2}^2}} - 3$.
Alas, if we continue just a few more levels, darkness befalls.

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

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

\[ A(7, y) \approx \text{speechless} \]

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.”
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.*
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.

One 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.
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.

\[ B_1(x) = 2x \]
\[ B_{k+}(x) = B_k^x(1) \]

\( B_k^x(1) \) 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 \( A_{k+1} \).
Recall the subsequence ordering on words where \( u = u_1 \ldots u_n \) precedes \( v = v_1 v_2 \ldots v_m \) 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: \( u \subseteq v \).

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).
An antichain in a partial order is a sequence \( x_0, x_1, \ldots, x_n, \ldots \) of elements such that \( i < j \) implies that \( x_i \) and \( x_j \) are incomparable.

**Example**

Consider the powerset of \([n] = \{1, 2, \ldots, n\}\) with the standard subset ordering. How does one construct a long antichain?

For example, \( x_0 = \{1\} \) is a bad idea, and \( x_0 = [n] \) is even worse.

What is the right way to get a long antichain?
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 $a$ that appears infinitely often as the first letter in $(x_n)$ and copying the words up to the first occurrence of one of these $a$-words. Follow by all the $a$-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.
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.*
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:

$$\alpha(k) = \max\left( |x| \mid x \in S_k \right)$$

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.
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.*
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.
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.
Functions like $A$, $B$ or $\alpha$ are all intuitively computable, but fail to be primitive recursive.

**Obvious Question:** how much do we have to add to primitive recursion to capture these functions?

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

$$A(a, b) = (\min(z \mid R(a, b, z)))_0$$

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.
A computation of $A(2, 1)$ might start like

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

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

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

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

Clearly, we can model these steps by a function $\Delta$ defined on sequences of naturals—or, via coding, just naturals.
\[
\Delta(\ldots, 0, y) = (\ldots, y^+) \\
\Delta(\ldots, x^+, 0) = (\ldots, x, 1) \\
\Delta(\ldots, x^+, y^+) = (\ldots, x, x^+, y)
\]

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 \((a, b, c, t)\).
$A(3, 4)$
Very rapidly growing functions are one reason primitive recursion is not strong enough to capture computability.

Here is another obstruction: recall the evaluation operator for our PR terms:

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

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.
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 $\langle x_1, \ldots, x_n \rangle$. Hence we will be able to interpret eval as a function of type

$$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

and this function might potentially be primitive recursive.
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 way of coding PR terms, there are many other, equally natural ways.
Now suppose eval is p.r., and define the following function

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

Then \( f \) 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.
More precisely, given a good formal definition of computability (one that matches our intuitive ideas), we would still expect to have indices for the functions in this class.

We also would expect to have something like an operation

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

that works like an interpreter for functions in this class.

Most importantly, we want eval to be itself computable.

In light of the last result, this may sound like mission impossible, but we will see that things work out fine as long as we are willing to deal with partial functions.
We write
\[ f : A \hookrightarrow B \]
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.
More Recursion

Register Machines

Universality
We could try to deal with the Ackermann function and evaluation by strengthening the machinery used to define primitive recursive functions. In particular a stronger recursion scheme seems promising.

Let’s delay this approach for the moment, and instead turn 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.
Definition

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

We write $R_0$, $R_1$, ... 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.
Our instruction set is very, very primitive:

- **inc r k**
  increment register $R_r$, goto $k$.

- **dec r k l**
  if $[R_r] > 0$ decrement register $R_r$ and goto $k$, otherwise goto $l$.

- **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**.
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:

```
// addition  R0 R1 --> R2
  0: dec 0 1 2
  1: inc 2 0
  2: dec 1 3 4
  3: inc 2 2
  4: halt
```
Since we have no intentions of actually building a physical version of a register 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.
Definition

A function is **RM-computable** if there is some RMP that implements the function.

This is a bit wishy-washy: we really need to fix

- a register machine program $P$,
- input registers $I$, and
- output registers $O$.

Then $(P, I, O)$ determines a partial function $f : \mathbb{N}^k \leftrightarrow \mathbb{N}^\ell$ where $k = |I|$ and $\ell = |O|$.
Given input arguments \( \mathbf{a} = (a_0, \ldots, a_{k-1}) \in \mathbb{N}^k \), 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 \( \mathbf{b} = (b_1, \ldots, b_\ell) = f(\mathbf{a}) \).

If \( P \) does not terminate, \( f(\mathbf{a}) \) is undefined.
Note the complication: we have to deal with the possibility that program $P$ does not produce any value, in which case $f(a)$ is undefined.

This causes a number of difficulties. For example, what should equality mean in

$$f(a) = g(a)$$

As the eval function shows, there is no way around this, we will have to deal with partial functions.
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.

**Definition**

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 \(y = x[x_r \mapsto x_r + 1]\).

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

Notation: \((p, x) \xrightarrow{\frac{1}{P}} (q, y)\).

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

\[(p, x) \xrightarrow{0}{\mathcal{P}} (q, y) \iff (p, x) = (q, y)\]

\[(p, x) \xrightarrow{t}{\mathcal{P}} (q, y) \iff \exists q', y' \ (p, x) \xrightarrow{t-1}{\mathcal{P}} (q', y') \xrightarrow{1}{\mathcal{P}} (q, y)\]

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

A computation (or a run) of \( P \) is a sequence of configurations \( C_0, C_1, C_2, \ldots \) where \( C_i \xrightarrow{1}{\mathcal{P}} C_{i+1} \).
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.
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.
Hence, we have two kinds of computations: finite ones (that necessarily end in a halt instruction), and infinite ones. We will write

\[(C_i)_{i<n}\]  and \[(C'_i)_{i<\omega}\]

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

**Definition**

A RMP $P$ computes the partial function $f : \mathbb{N}^k \hookrightarrow \mathbb{N}^\ell$ 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 domain of $f$, then the computation of $P$ on $E_a$ fails to terminate.
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.
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.
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 $[X]$. 
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.
Here is a program that multiplies registers $X$ and $Y$, and places the product into $Z$. $U$ is auxiliary.

```plaintext
// 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)

// 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
Exercise

*Determine the time complexity of the multiplication RM.*
Flowgraph for Multiplication

- X-
- Y-
- Z+
- U+
- H
- U-
- Y+

0 0 0

0 0 0
The following RMP computes the number of 1’s in the binary expansion of \( X \), the so-called binary digit sum of \( x \).

// 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
Flowgraph for DigitSum
The (binary) digit sum is actually quite useful in some combinatorial arguments.
**Exercise**

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

**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 the three coding functions from last time:

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

One can check that dec and len can be computed by fairly small register machines. As usual, for \( \langle . \rangle \) we would have to fix the number of arguments.
Flowgraph $\text{dec}(x, i)$
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.
Note that a single instruction of an RMP can also be coded as a number:

- halt \(\langle 0 \rangle\)
- inc \(r,k\) \(\langle r,k \rangle\)
- dec \(r,k,l\) \(\langle r,k,l \rangle\)

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

Register Machines

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)
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
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 by 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.
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 integer $e$, 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 $I$. 
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.
• **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 $a = \langle b, c \rangle$; 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.
Here are the registers used in $\mathcal{U}$:

- $x$ 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.
Universal RM

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 12  // x--
12: write R r x 2  // R[r] = x; goto 2
13: halt
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.
Exercise

Figure out what this picture means.

Exercise (Very Hard)

Prove that this is really a universal Turing machine.