Partial recursive functions

We saw above that
\[
\text{primitive recursive } \Rightarrow \text{ computable is total}
\]

Since not every computable partial function is total, it is certainly
not the case that every computable partial function is
primitive recursive.

One intuitively algorithmic method of calculation that can give
rise to non-total partial functions is that of searching for the
smallest value of a function's argument that produces a given
result (zero, say) - since no such value may exist. The
corresponding operation on functions is called minimization...
Minimization

Given \( f \in \text{Pfn}(\mathbb{N}^{n+1}, \mathbb{N}) \)
define \( \mu(f) \in \text{Pfn}(\mathbb{N}^n, \mathbb{N}) \) by

\[
\mu(f)(x_1, \ldots, x_n) = x \quad \text{def} \quad \text{there exist } y_0, y_1, \ldots, y_x \\
\text{such that } f(x_1, \ldots, x_n, i) = y_i \text{ for } i = 0, 1, \ldots, x \\
& y_i > 0 \text{ for } i = 0, 1, \ldots, x-1 \\
& y_x = 0
\]

Thus

\[
\mu(f)(x_1, \ldots, x_n) = \begin{cases} 
\text{the least } x \text{ such that} \\
f(x_1, \ldots, x_n, x) = 0 \text{ and} \\
f(x_1, \ldots, x_n, i) > 0 \text{ for } i < x \\
\uparrow \text{(in particular, } f(x_1, \ldots, x_n, i) \downarrow \text{ for } i < x) 
\end{cases}
\]

and in particular

\( \mu(f)(x_1, \ldots, x_n) \uparrow \) if no \( x \) exists satisfying these conditions.

PROPOSITION:

If \( f \) is computable, then so is \( \mu(f) \).

Proof:

Given a register machine program \( F \) computing \( f(x_1, \ldots, x_{n+1}) \) in RO starting with \( R_1, \ldots, R(n+1) \) set to \( x_1, \ldots, x_{n+1} \)
then the following diagram specifies a register machine program for computing \( \mu(f)(x_1, \ldots, x_n) \) in RO starting with \( R_1, \ldots, R_n \) set to \( x_1, \ldots, x_n \):
\[ \text{START} \rightarrow (x_1, \ldots, x_n) := (r_1, \ldots, r_n) \]

\[ (r_1, \ldots, r_n) := (x_1, \ldots, x_n); \]

\[ \text{C}^+ \]

\[ \text{F} \]

\[ R_0 := C \rightarrow \text{HALT} \]

---

\[ \text{START} \rightarrow (x_1, \ldots, x_n) := (r_1, \ldots, r_n) \]

\[ (r_1, \ldots, r_n) := (x_1, \ldots, x_n); \]

\[ \text{C}^+ \]

\[ \text{F} \]

\[ R_0 := C \rightarrow \text{HALT} \]

---

\[ X_1, \ldots, X_n, C \]

are some registers not mentioned in the program \( F \), which we assume only uses registers \( R_0, \ldots, R_N \) \( N > n \).
**Definition:**

A partial recursive function is a partial function that can be built up from the basic functions by repeated use of the operations of composition, primitive recursion and minimization.

In other words, the set $\mathcal{PR}$ of partial recursive functions is the smallest set of partial functions containing the basic functions and closed under the operations of composition, primitive recursion and minimization.

The members of $\mathcal{PR}$ that are total are called (total) recursive functions.

---

**Examples of minimization**

**Ex. 1** The everywhere undefined function is partial recursive. For if $f(x, y) \stackrel{\text{def}}{=} 1$, then $\mu(f)(x) \uparrow$, for all $x$; and $f = \text{suc} \circ \text{zero}^2$; so $\mu(f) = \mu(\text{suc} \circ \text{zero}^2)$ is partial recursive.

**Ex. 2**

\[ d(x, y) \stackrel{\text{def}}{=} \text{integer part of } x/y \text{ (undefined if } y = 0) \]
\[ m(x, y) \stackrel{\text{def}}{=} \text{remainder when } x \text{ is divided by } y \]

are partial recursive.

For note that

\[ d(x, y) \equiv \text{least } z \text{ such that } x < y(z+1) \text{ (thus } d(x, 0) \uparrow). \]

Now

\[ \text{ge}(x, y) = \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \geq y \end{cases} \]

is primitive recursive, since

\[ \text{ge}(x, y) = \text{ifzero}(y-x, 1, 0) \]

(and we saw above that $\text{ifzero} \& \text{ replicate are in PRIM}$).

So $f(x, y, z) \stackrel{\text{def}}{=} \text{ge}(x, y(z+1))$ is also in PRIM (since multiplication is).

Then $d = \mu(f)$ is partial recursive.

Finally, note that $m(x, y) = x \div y \cdot d(x, y)$, so $m$ is also in PR.
The function \( d \) in Ex. 2 is not total recursive because it is undefined when its second argument is 0.

Thus

\[
\text{div}(x, y) \overset{\text{def}}{=} \begin{cases} 
\text{integer part of } x/y & \text{if } y > 0 \\
0 & \text{if } y = 0
\end{cases}
\]

is (total) recursive: \( \text{div} = \text{ifzero} \circ (\text{proj}_2^2, \text{zero}^2, d) \).

In fact, \( \text{div} \) is primitive recursive (exercise: prove this).

Every \( f \in \text{PRIM} \) satisfies

\( f \in \text{PR} \) & \( f \) is total

But converse is false:

there are total recursive functions

which are not primitive recursive

Here is a sketch of the proof of this, making use of our next major result - the Theorem on page 114 ...
Proof (sketch)

First, $(eg \rho^2(\text{proj}_1, \text{suc} \circ \text{proj}_3) \text{ is a formal description for } \text{add}(x,y) \text{ def } x+y)$

code formal descriptions of primitive recursive functions as numbers so that

$$e(x,y) = \begin{cases} f_x(y) & \text{if } x \text{ is code of a formal description of } \\
0 & \text{if } x \text{ is not the code of a formal description of a unary prim. rec. function} \\
h \text{are computable functions.} \\
\end{cases}$$

Next, consider $e'(x) \text{ def } e(x,x)+1$

CLAIM: $e' \in \text{PR}, \text{ but } e' \notin \text{PRIM}$.

Next we make use of the Theorem to be proved below (p114), namely that PR coincides with the collection of computable functions. Since $e$ is computable, this Theorem implies that it is in PR — and hence so is $e'$ since $e' = \text{suc} \circ (e \circ (\text{proj}_1, \text{proj}_1))$.

To see that $e' \notin \text{PRIM}$, suppose the contrary and derive a contradiction: if $e' \in \text{PRIM}$, then it would have a formal description, and hence $e' = f_x$ for some code $x$. Then

$$f_x(x) = e'(x) \text{ since } e' = f_x$$
$$= e(x,x)+1 \text{ by definition of } e'$$
$$= f_x(x)+1 \text{ by definition of } e$$

Which is impossible! □

Here is a more explicit example of a non-primitive-recursive member of PR:
Ackermann's function

**FACT:** there is a total function

\[ \text{ack} \in \text{Fun}(\mathbb{N} \times \mathbb{N}, \mathbb{N}) \text{ satisfying} \]

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

\[ \text{ack} \text{ is recursive, but not primitive recursive.} \]

It is beyond the scope of this course to prove that \( \text{ack} \notin \text{PRIM} \).
(Roughly speaking, \( \text{ack} \notin \text{PRIM} \) because as \( x \) & \( y \) increase, 
\( \text{ack}(x,y) \) grows faster than any primitive recursive function possibly can grow.)

One way to see that \( \text{ack} \in \text{PR} \) is to design a register machine to compute \( \text{ack} \) (exercise), and then appeal to the
Theorem we are about to prove that states that \( \text{PR} \) coincides with the set of computable functions.

The proof that the register machine for \( \text{ack} \) always halts (i.e. that \( \text{ack} \) is total) is non-trivial. (It can be done by "well-founded induction" on pairs \((x,y) \in \mathbb{N}^2\) ordered lexicographically: \((x_1, y_1) < (x_2, y_2) \iff (x_1 < x_2 \text{ or } (x_1 = x_2 \text{ and } y_1 < y_2))\).)
Theorem:
A partial function is (register machine) computable if and only if it is partial recursive.

We have already proved that the collection of computable partial functions contains the basic functions and is closed under the operations of composition, primitive recursion and minimization, and hence contains all partial recursive functions.

So it remains to see that
\[ f \text{ computable } \Rightarrow f \text{ partial recursive} \]

Proof of \((f \text{ computable } \Rightarrow f \in \text{PR})\):
If \(f \in \text{Pfn}([N^m, N])\) is computable, there is a register machine \(M\) which when started with \(R_1, \ldots, R_n\) set to \(x_1, \ldots, x_n\) (and all other registers set to 0), halts if and only if \(f(x_1, \ldots, x_n) \downarrow\), and in that case \(R_0\) contains this value.

Suppose the registers of \(M\) are \(R_0, R_1, R_2, \ldots, R_n, R(n+1), \ldots, R_m\) (for some \(m \geq n\)).
Suppose \(M\)'s program has instructions labelled \(L_0, L_2, \ldots, L_I\) (some \(I \geq 0\)), and without loss of generality assume that the only HALT instruction is the last one \((L_I)\) and that there are no erroneous halts (i.e. the only labels referred to in increment/decrement instructions lie in the range \(L_0, \ldots, L_I\)).

The state of \(M\) at any stage in its computation can be specified by the code \([l, r_0, r_1, \ldots, r_m]\) of a list of length \(m+2\), where
- \(l = \text{current instruction}\) (so \(0 \leq l \leq I\))
- \(r_j = \text{current contents of} R_j\) (\(j = 0, 1, \ldots, m\)).

The proof that \(f\) is partial recursive depends upon the following lemmas:
**Lemma 1:**

There are primitive recursive functions $\text{lab}, \text{val}_0, \text{val}_1, \ldots, \text{val}_m \in \text{Fun}(\mathbb{N}, \mathbb{N})$ satisfying

\[
\begin{align*}
\text{lab}(\llbracket l, r_0, \ldots, r_m \rrbracket) &= l \\
\text{val}_j(\llbracket l, r_0, \ldots, r_m \rrbracket) &= r_j
\end{align*}
\]

(for all $j = 0, \ldots, m$ and)

Thus $\text{lab}$ gives the label of a state of $M$, whilst $\text{val}_j$ gives the value held in $R_j$.

**Lemma 2:**

There is a primitive recursive function $\text{next} \in \text{Fun}(\mathbb{N}, \mathbb{N})$ which gives the next state of $M$ in terms of the current one.

The proof of these lemmas uses the following property of our coding of lists of numbers as numbers...

**Proposition:**

The functions $\text{mklist}^n \in \text{Fun}(\mathbb{N}^n, \mathbb{N})$, $\text{hd}, \text{tl} \in \text{Fun}(\mathbb{N}, \mathbb{N})$ defined by

\[
\text{mklist}^n(x_1, \ldots, x_n) \overset{\text{def}}{=} \llbracket x_1, \ldots, x_n \rrbracket
\]

\[
\text{hd}(x) \overset{\text{def}}{=} \begin{cases} 
  x_1 & \text{if } x = \llbracket x_1, \ldots, x_n \rrbracket \text{ for some } n > 0 \\
  0 & \text{if } x = \llbracket \text{nil} \rrbracket = 0
\end{cases}
\]

\[
\text{tl}(x) \overset{\text{def}}{=} \begin{cases} 
  \llbracket x_2, \ldots, x_n \rrbracket & \text{if } x = \llbracket x_1, \ldots, x_n \rrbracket \text{ for some } n > 0 \text{ and } x_1, \ldots, x_n \\
  0 & \text{if } x = \llbracket \text{nil} \rrbracket = 0
\end{cases}
\]

are all primitive recursive.
First, note that

\[ \text{State}(x_1, \ldots, x_n, t) \text{ def } \begin{cases} \text{State of } M \text{ at } t^{\text{th}} \text{ step,} \\ \text{Start with } R1 = x_1, \ldots, Rn = x_n \\ \text{& all other registers } = 0 \end{cases} \]

is primitive recursive, because

\[ \begin{align*}
\text{state}(x_1, \ldots, x_n, 0) &= [0, 0, x_1, \ldots, x_n, 0, \ldots, 0] \\
\text{state}(x_1, \ldots, x_n, t+1) &= \text{next}(\text{state}(x_1, \ldots, x_n, t))
\end{align*} \]

so that

\[ \text{state} = \rho^n(\text{mklist}^{m^2} \circ (\text{zero}^n, \text{zero}^n, \text{proj}^n, \ldots, \text{proj}^n, \text{zero}^n, \ldots, \text{zero}^n), \\
\text{next} \circ \text{proj}^{m^2+2}) \]

with \( \text{mklist}^{m^2} \), next \( \in \text{PRIM} \).

Now:

\[ f(x_1, \ldots, x_n) \equiv \text{val}_o(\text{state}(x_1, \ldots, x_n, \text{halt}(x_1, \ldots, x_n))) \]

where

\[ \text{halt}(x_1, \ldots, x_n) \text{ def } \begin{cases} \text{number of steps taken to halt} \\ \text{& undefined if never halt} \end{cases} \]

\[ \equiv \text{least } t \text{ such that } \text{lab}(\text{state}(x_1, \ldots, x_n, t)) = 1 \]

\[ \equiv \mu(h)(x_1, \ldots, x_n) \]

where \( h(x_1, \ldots, x_n, t) \equiv 1 - \text{lab}(\text{state}(x_1, \ldots, x_n, t)) \).

Since \( \text{lab} \), \( \text{state} \), \( \text{halt} \) are in \( \text{PRIM} \), so is \( h \)

and hence \( \text{halt} = \mu(h) \) is in \( \text{PR} \).

Thus \( f = \text{val}_o \circ (\text{state} \circ (\text{proj}^n, \ldots, \text{proj}^n, \mu(h))) \)

is also in \( \text{PR} \). \( \square \)
To complete the proof of the Theorem, we have to prove the Proposition and Lemmas 1 & 2.

**Proof of the Proposition**

Since \( \text{mklist}^n(x_1, \ldots, x_n) = \langle x_1, \langle x_2, \ldots, \langle x_n, 0 \rangle \rangle \rangle \),

to see that \( \text{mklist}^n \in \text{PRIM} \), it suffices to show that \( \langle x, y \rangle = 2^y(2^{y+1}) \)
is primitive recursive—which follows from the fact that multiplication and exponentiation are in \( \text{PRIM} \).

The proof that \( \text{hd} \) and \( \text{tl} \) are in \( \text{PRIM} \) requires more effort.

We get to their primitive recursivity via that of a number of intermediate functions:

(i) \( \text{mod}_2(x) \) def \[ \begin{cases} 0 & \text{if } x \text{ even} \\ 1 & \text{if } x \text{ odd} \end{cases} \]
is primitive recursive, because it satisfies \( \begin{cases} \text{mod}_2(0) = 0 \\ \text{mod}_2(x+1) = \text{iszero}(\text{mod}_2(x), 1, 0) \end{cases} \).

(ii) \( \text{half}(x) \) def integer part of \( x/2 \) is primitive recursive by (i), since it satisfies \( \begin{cases} \text{half}(0) = 0 \\ \text{half}(x+1) = \text{iszero}(\text{mod}_2(x), \text{half}(x), \text{half}(x)+1) \end{cases} \).

(iii) \( f(x, y) \) def \[ \begin{cases} x/2^y & \text{if } x > 0 \text{ & } 2^y \text{ divides } x \\ 0 & \text{otherwise} \end{cases} \]
is primitive recursive by (i) & (ii), since it satisfies \( \begin{cases} f(x, 0) = x \\ f(x, y+1) = \text{iszero}(\text{mod}_2(f(x, y)), \text{half}(f(x, y)), 0) \end{cases} \).

Combining (ii), (iii), (iv), and the fact (cf. page 98) that bounded summations preserve the property of primitive recursiveness, we have that \( \text{hd} \) and \( \text{tl} \) are in \( \text{PRIM} \) because...
\[ \text{hd}(x) = \begin{cases} \text{largest } y \text{ such that } 2^y \text{ divides } x, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \end{cases} \]

\[ = \sum_{y < x} \text{ifzero}(f(x, y+1), 0, 1) \]

and

\[ \text{tl}(x) = \text{half}(f(x, \text{hd}(x))) \]

□ Proposition

Proof of Lemma 1
This follows immediately from the Proposition, because

\[ \text{lab} = \text{hd} \]

and

\[ \text{val} = \text{hd} \circ (\underbrace{\text{tl} \circ \cdots \circ \text{tl}}_{j+1}) . \]

□ Lemma 1

Proof of Lemma 2
By examining M's program, we can define four (I+1)-tuples of numbers

\((a_0, \ldots, a_I)\), \((b_0, \ldots, b_I)\), \((c_0, \ldots, c_I)\), and \((d_0, \ldots, d_I)\)

as follows:

\* for each \(i = 0, \ldots, I-1\)

if the \(i\)th instruction is an increment, say \(L_i : R_j^+ \rightarrow L_k\),
then define \(a_i = j\), \(b_i = 0\), \(c_i = k\), and \(d_i = k\);

else the instruction is a decrement, say \(L_i : R_j^- \rightarrow L_k, L_L\),
and define \(a_i = j\), \(b_i = 1\), \(c_i = k\), and \(d_i = \ell\);

\* define \(a_I = 0\), \(b_I = 0\), \(c_i = I\), and \(d_i = I\).
If \( i \)’th instruction is, then \( (a_i, b_i, c_i, d_i) \equiv (j, 0, k, k) \)

\( \text{Li} : R_j^+ \rightarrow L_k \)

\( \text{Li} : R_j^{-} \rightarrow L_k, \ell \)

\( \text{LI} : \text{HALT} \)

\( \text{next}_i^L(x) \overset{\text{def}}{=} \sum_{i=0}^{I} \text{ifzero}(\text{val}_{a_i}(x), d_i, c_i) \cdot \text{eq}(i, \text{lab}(x)) \)

\( \text{next}_j^L(x) \overset{\text{def}}{=} \sum_{i=0}^{I-1} f_{ij}(x) \cdot \text{eq}(i, \text{lab}(x)) + \text{val}_{j}(x) \cdot \text{eq}(I, \text{lab}(x)) \)

where

\[
f_{ij}(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \\ \end{cases}
\]

is primitive recursive, since \( \text{eq}(x, y) = \text{ifzero}(x \cdot y, \text{ifzero}(y \cdot x, 1, 0), 0) \).

Using it, we can define primitive recursive functions \( \text{next}_i^L \) and \( \text{next}_j^L \) as follows:

\( \text{next}_i^L(x) \overset{\text{def}}{=} \sum_{i=0}^{I} \text{ifzero}(\text{val}_{a_i}(x), d_i, c_i) \cdot \text{eq}(i, \text{lab}(x)) \)

\( \text{next}_j^L(x) \overset{\text{def}}{=} \sum_{i=0}^{I-1} f_{ij}(x) \cdot \text{eq}(i, \text{lab}(x)) + \text{val}_{j}(x) \cdot \text{eq}(I, \text{lab}(x)) \)

where

\[
f_{ij}(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x = y \\ b_i \cdot \text{pred}(\text{val}_{j}(x)) + (1-b_i) \cdot \text{suc}(\text{val}_{j}(x)) \cdot \text{eq}(a_i, j) + \text{val}_{j}(x) \cdot (1-\text{eq}(a_i, j)) \\ \end{cases}
\]

(and recall that \( \text{suc}(x) = x+1 \), \( \text{pred}(x) = x-1 \)).

By choice of the constants \( a_i, b_i, c_i, d_i \) (\( i = 0, \ldots, I \)), it follows that given a state \( [l, r_0, \ldots, r_m] \) of \( M \),

\( \text{next}_i^L([l, r_0, \ldots, r_m]) = \text{number of the instruction in the next state} \)

\( \text{next}_j^L([l, r_0, \ldots, r_m]) = \text{contents of } R_j \text{ in the next state} \)

(provided \( 0 \leq j \leq m \)).
Therefore the next-state function is given by
\[ \text{next}(x) = [\text{next}_l(x), \text{next}_0(x), \ldots, \text{next}_m(x)] \]

i.e.
\[ \text{next} = \text{mlist}^{m+2} \circ (\text{next}_l, \text{next}_0, \ldots, \text{next}_m) \]

and hence it is primitive recursive since next_l, next_0, and (by the proposition) \text{mlist}^{m+2} are all in PRIM.

\( \square \) Lemma 2