Definition. A partial function f is partial recursive ($f \in \mathbf{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.

In other words, the set $\mathbf{PR}$ of partial recursive functions is the smallest set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition, primitive recursion and minimization.

Computable = partial recursive

Theorem. Not only is every $f \in \mathbf{PR}$ computable, but conversely, every computable partial function is partial recursive.

Proof (sketch). Let f be computed by RM M. Recall how we coded instantaneous configurations $c = (\ell, r_0, \ldots, r_n)$ of M as numbers $\ulcorner [\ell, r_0, \ldots, r_n] \urcorner$. It is possible to construct primitive recursive functions $lab, val_0, next_M \in \mathbb{N} \to \mathbb{N}$ satisfying

$$
\begin{aligned}
lab(\ulcorner [\ell, r_0, \ldots, r_n] \urcorner) &= \ell \\
val_0(\ulcorner [\ell, r_0, \ldots, r_n] \urcorner) &= r_0 \\
next_M(\ulcorner [\ell, r_0, \ldots, r_n] \urcorner) &= \text{code of } M\text{'s next configuration}
\end{aligned}
$$

(Showing that $next_M \in \mathbf{PRIM}$ is tricky—proof omitted.)

Proof sketch, cont.

Let $\mathit{config}_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\vec{x}$. It's in $\mathbf{PRIM}$ because:

$$\begin{cases} \mathit{config}_M(\vec{x}, 0) & = \ulcorner [0, \vec{x}] \urcorner \\ \mathit{config}_M(\vec{x}, t+1) & = \mathit{next}_M(\mathit{config}_M(\vec{x}, t)) \end{cases}$$

Proof sketch, cont.

Let $config_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\vec{x}$. It's in $\mathbf{PRIM}$ because:

$$\begin{cases} config_M(\vec{x}, 0) & = \ulcorner [0, \vec{x}] \urcorner \\ config_M(\vec{x}, t+1) & = next_M(config_M(\vec{x}, t)) \end{cases}$$

Can assume M has a single `HALT` as last instruction, Ith say (and no erroneous halts). Let $halt_M(\vec{x})$ be the number of steps M takes to halt when started with initial register values $\vec{x}$ (undefined if M does not halt). It satisfies

$$halt_M(\vec{x}) \equiv \text{least } t \text{ such that } I - lab(config_M(\vec{x}, t)) = 0$$

and hence is in $\mathbf{PR}$ (because $lab, config_M, I - (\ ) \in \mathbf{PRIM}$).

Proof sketch, cont.

Let $config_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\vec{x}$. It's in $\mathbf{PRIM}$ because:

$$\begin{cases} config_M(\vec{x}, 0) & = \ulcorner [0, \vec{x}] \urcorner \\ config_M(\vec{x}, t+1) & = next_M(config_M(\vec{x}, t)) \end{cases}$$

Can assume M has a single `HALT` as last instruction, Ith say (and no erroneous halts). Let $halt_M(\vec{x})$ be the number of steps M takes to halt when started with initial register values $\vec{x}$ (undefined if M does not halt). It satisfies

$$halt_M(\vec{x}) \equiv \text{least } t \text{ such that } I - lab(config_M(\vec{x}, t)) = 0$$

and hence is in $\mathbf{PR}$ (because $lab, config_M, I - (\ ) \in \mathbf{PRIM}$).

So $f \in \mathbf{PR}$, because $f(\vec{x}) \equiv val_0(config_M(\vec{x}, halt_M(\vec{x})))$.

Definition. A partial function f is partial recursive ($f \in \mathbf{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.

The members of $\mathbf{PR}$ that are total are called recursive functions.

Fact: there are recursive functions that are not primitive recursive. For example. . .

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{aligned}
ack(0, x_2) &= x_2 + 1 \\
ack(x_1 + 1, 0) &= ack(x_1, 1) \\
ack(x_1 + 1, x_2 + 1) &= ack(x_1, ack(x_1 + 1, x_2))
\end{aligned}$$

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

- ack is computable, hence recursive [proof: exercise].

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

- ack is computable, hence recursive [proof: exercise].
- **Fact:** ack grows faster than any primitive recursive function $f \in \mathbb{N}^2 \to \mathbb{N}$:
  $\exists N_f \; \forall x_1, x_2 > N_f \; (f(x_1, x_2) < ack(x_1, x_2))$.
  Hence ack is not primitive recursive.

Lambda-Calculus

Notions of computability

- Church (1936): λ-calculus
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

λ-Terms, M

are built up from a given, countable collection of

- variables $x, y, z, \ldots$

by two operations for forming λ-terms:

- λ-abstraction: $(\lambda x.M)$
  (where x is a variable and M is a λ-term)
- application: $(M\, M')$
  (where M and M' are λ-terms).

λ-Terms, M

are built up from a given, countable collection of

- variables $x, y, z, \ldots$

by two operations for forming λ-terms:

- λ-abstraction: $(\lambda x.M)$
  (where x is a variable and M is a λ-term)
- application: $(M\,M')$
  (where M and M' are λ-terms).

Some random examples of λ-terms:

$$x \quad (\lambda x.x) \quad ((\lambda y.(x\,y))x) \quad (\lambda y.((\lambda y.(x\,y))x))$$

λ-Terms, M

Notational conventions:

- $(\lambda x_1\, x_2 \ldots x_n.M)$ means $(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M) \ldots))$
- $(M_1\, M_2 \ldots M_n)$ means $(\ldots (M_1\, M_2) \ldots M_n)$ (i.e. application is left-associative)
- drop outermost parentheses and those enclosing the body of a λ-abstraction. E.g. write $(\lambda x.(x(\lambda y.(y\, x))))$ as $\lambda x.x(\lambda y.y\, x)$.
- $x \# M$ means that the variable x does not occur anywhere in the λ-term M.

Free and bound variables

In $\lambda x.M$, we call x the bound variable and M the body of the λ-abstraction.

An occurrence of x in a λ-term M is called

- binding if in between λ and $.$
  (e.g. $(\lambda x.y\,x)\,x$)
- bound if in the body of a binding occurrence of x
  (e.g. $(\lambda x.y\,x)\,x$)
- free if neither binding nor bound
  (e.g. $(\lambda x.y\,x)x$).

Free and bound variables

Sets of free and bound variables:

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.M) &= FV(M) - \{x\} \\
FV(M\,N) &= FV(M) \cup FV(N)
\end{aligned}
$$

$$
\begin{aligned}
BV(x) &= \emptyset \\
BV(\lambda x.M) &= BV(M) \cup \{x\} \\
BV(M\,N) &= BV(M) \cup BV(N)
\end{aligned}
$$

If $FV(M) = \emptyset$, M is called a closed term, or combinator.

α-Equivalence $M =_\alpha M'$

$\lambda x.M$ is intended to represent the function f such that

$$f(x) = M \text{ for all } x.$$

So the name of the bound variable is immaterial: if $M' = M\{x'/x\}$ is the result of taking M and changing all occurrences of x to some variable $x' \# M$, then $\lambda x.M$ and $\lambda x'.M'$ both represent the same function.

For example, $\lambda x.x$ and $\lambda y.y$ represent the same function (the identity function).

α-Equivalence $M =_\alpha M'$

is the binary relation inductively generated by the rules:

$$\frac{}{x =_\alpha x} \qquad \frac{z \mathrel{\#} (M\,N) \qquad M\{z/x\} =_\alpha N\{z/y\}}{\lambda x.M =_\alpha \lambda y.N}$$

$$\frac{M =_\alpha M' \qquad N =_\alpha N'}{M\,N =_\alpha M'\,N'}$$

where $M\{z/x\}$ is M with all occurrences of x replaced by z.

α-Equivalence $M =_\alpha M'$

For example:

$$
\begin{array}{rc}
 & \lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x' \\
\text{because} & (\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x' \\
\text{because} & \lambda z\, x'.z =_\alpha \lambda x\, x'.x \text{ and } x' =_\alpha x' \\
\text{because} & \lambda x'.u =_\alpha \lambda x'.u \text{ and } x' =_\alpha x' \\
\text{because} & u =_\alpha u \text{ and } x' =_\alpha x'.
\end{array}
$$

α-Equivalence $M =_\alpha M'$

Fact: $=_\alpha$ is an equivalence relation (reflexive, symmetric and transitive).

We do not care about the particular names of bound variables, just about the distinctions between them. So α-equivalence classes of λ-terms are more important than λ-terms themselves.

- Textbooks (and these lectures) suppress any notation for α-equivalence classes and refer to an equivalence class via a representative λ-term (look for phrases like "we identify terms up to α-equivalence" or "we work up to α-equivalence").
- For implementations and computer-assisted reasoning, there are various devices for picking canonical representatives of α-equivalence classes (e.g. de Bruijn indexes, graphical representations, ...).