

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

In other words, the set **PRIM** of primitive 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 and primitive recursion.

FACT : every $f \in \mathbf{PRIM}$ is a total function

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. . .

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. . .

it's possible to construct a computable function $e : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ satisfying
 $e(n, x) = \text{value of } n^{\text{th}} \text{ PRIM fn. at } x$

A diagonalization argument shows $e \notin \mathbf{PRIM}$
(see CST 2017, p 6, 94)

Examples of recursive definitions

$$\left\{ \begin{array}{l} f_2(0) \equiv 0 \\ f_2(1) \equiv 1 \\ f_2(x+2) \equiv f_2(x) + f_2(x+1) \end{array} \right. \left. \vphantom{\begin{array}{l} f_2(0) \\ f_2(1) \\ f_2(x+2) \end{array}} \right\} f_2(x) = x\text{th Fibonacci number}$$

$f_2 \in \text{PRIM}$ even though this is not a primitive recursive definition

(see CST 2014, paper 6, question 4)

Ackermann's function

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

$$ack(0, x_2) = x_2 + 1$$

$$ack(x_1 + 1, 0) = ack(x_1, 1)$$

$$ack(x_1 + 1, x_2 + 1) = ack(x_1, ack(x_1 + 1, x_2))$$

Ackermann's function

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

$$ack(0, x_2) = x_2 + 1$$

$$ack(x_1 + 1, 0) = ack(x_1, 1)$$

$$ack(x_1 + 1, x_2 + 1) = ack(x_1, ack(x_1 + 1, x_2))$$

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

OCaml version 4.00.1

```
# let rec ack (x : int)(y : int) : int =
  match x ,y with
    0 , y -> y+1
  | x , 0 -> ack (x-1) 1
  | x ,y -> ack (x-1) (ack x (y-1));;
val ack : int -> int -> int = <fun>
# ack 0 0;;
- : int = 1
# ack 1 1;;
- : int = 3
# ack 2 2;;
- : int = 7
# ack 3 3;;
- : int = 61
# ack 4 4;;
Stack overflow during evaluation (looping recursion?).
#
```

Ackermann's function

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

$$ack(0, x_2) = x_2 + 1$$

$$ack(x_1 + 1, 0) = ack(x_1, 1)$$

$$ack(x_1 + 1, x_2 + 1) = ack(x_1, ack(x_1 + 1, x_2))$$

- **Fact:** ack grows faster than any primitive recursive function $f \in \mathbb{N}^2 \rightarrow \mathbb{N}$:

$$\exists N_f \forall x_1, x_2 > N_f (f(x_1, x_2) < ack(x_1, x_2)).$$

Hence ack is not primitive recursive.

Ackermann's function

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

$$ack(0, x_2) = x_2 + 1$$

$$ack(x_1 + 1, 0) = ack(x_1, 1)$$

$$ack(x_1 + 1, x_2 + 1) = ack(x_1, ack(x_1 + 1, x_2))$$

In fact, writing a_x for $ack(x, -) \in \mathbb{N} \rightarrow \mathbb{N}$, one has

$$a_{x+1}(y) = \underbrace{(a_x \circ \dots \circ a_x)}_{\text{compose } y \text{ times}}(1)$$

← this is an e.g. of
a prim. rec. definition
"of higher type"

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.

Notation for function definitions in mathematical discourse :

NAMED

" let f be the function $f(x) = x^2 + x + 1 \dots [f] \dots$ "

ANONYMOUS

" the function $x \mapsto x^2 + x + 1 \dots$ "

" the function $\lambda x . x^2 + x + 1 \dots$ "

↑
LAMBDA NOTATION

λ -Terms, M

are built up from a given, countable collection of

- ▶ variables x, y, z, \dots

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:

- ▶ $(\lambda x_1 x_2 \dots x_n. M)$ means $(\lambda x_1. (\lambda x_2 \dots (\lambda x_n. M) \dots))$
- ▶ $(M_1 M_2 \dots M_n)$ means $(\dots (M_1 M_2) \dots M_n)$
(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:

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

E.g.

$$\begin{aligned}FV((\lambda x.yx)x) &= \{x, y\} \\BV((\lambda x.yx)x) &= \{x\}\end{aligned}$$

Free and bound variables

Sets of **free** and **bound** variables:

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

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

$$\text{E.g. } FV(\lambda y. \lambda x. (\lambda x. y x) x) = \emptyset$$

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:

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

$$\frac{M =_{\alpha} M' \quad N =_{\alpha} N'}{MN =_{\alpha} M'N'}$$

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

α -Equivalence $M =_{\alpha} M'$

For example:

$$\lambda \underline{x}. (\lambda \underline{x x'} . \underline{x}) x' =_{\alpha} \lambda \underline{y}. (\lambda x x' . x) x'$$

because

α -Equivalence $M =_{\alpha} M'$

For example:

$$\lambda x. (\lambda x x'. x) x' =_{\alpha} \lambda y. (\lambda x x'. x) x'$$

because

$$(\lambda z x'. z) x' =_{\alpha} (\lambda x x'. x) x'$$

because

α -Equivalence $M =_{\alpha} M'$

For example:

$$\lambda x. (\lambda x x'. x) x' =_{\alpha} \lambda y. (\lambda x x'. x) x'$$

because

$$(\lambda z x'. z) x' =_{\alpha} (\lambda x x'. x) x'$$

because

$$\lambda \underline{z} x'. z =_{\alpha} \lambda \underline{x} x'. x \text{ and } x' =_{\alpha} x'$$

because

α -Equivalence $M =_{\alpha} M'$

For example:

$$\lambda x. (\lambda x x'. x) x' =_{\alpha} \lambda y. (\lambda x x'. x) x'$$

because

$$(\lambda z x'. z) x' =_{\alpha} (\lambda x x'. x) x'$$

because

$$\lambda z x'. z =_{\alpha} \lambda x x'. x \text{ and } x' =_{\alpha} x'$$

because

$$\lambda \underline{x}'. u =_{\alpha} \lambda \underline{x}'. u \text{ and } x' =_{\alpha} x'$$

because

α -Equivalence $M =_{\alpha} M'$

For example:

$$\lambda x. (\lambda x x'. x) x' =_{\alpha} \lambda y. (\lambda x x'. x) x'$$

because

$$(\lambda z x'. z) x' =_{\alpha} (\lambda x x'. x) x'$$

because

$$\lambda z x'. z =_{\alpha} \lambda x x'. x \text{ and } x' =_{\alpha} x'$$

because

$$\lambda x'. u =_{\alpha} \lambda x'. u \text{ and } x' =_{\alpha} x'$$

because

$$u =_{\alpha} u \text{ and } x' =_{\alpha} x'.$$

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, ...).