

Indian Association for the Cultivation of Science

(Deemed to be University under *de novo* Category)

Master's/Integrated Master's-PhD Program/ Integrated Bachelor's-Master's Program/PhD Course

Theory of Computation II: COM 5108

Adv-Recursive-and-Lambda-definable-Functions

Instructor: Goutam Biswas *Autumn Semester 2026*

1 Introduction

The foundational questions about the scope and limitations of symbolic computation were studied before the birth of a real programmable computer. It came from the works of mathematicians and logicians interested in the foundation of mathematics, questions related to the truth and proof of mathematical statements, characterization of effectively describable (comptable) functions on natural numbers, etc.

Different people approached in different ways to model the notion of effectively computable functions on natural numbers ($\mathbb{N}_0 = \{0, 1, 2, \cdots\}$. The following three such approaches are more famous.

1. *Turing machine* was proposed by Alan Turing in 1936 in his famous paper "On Computable Numbers, with an Application to the Entscheidungsproblem. The model was named as **a-machine** ('a' for automatic?).

2. *Primitive Recursive Functions* were first formalized by Kart Gödel in 1931 in his famous paper on incompleteness theorem. But the concept was much older. Richard Dedekind (1888) and Thoralf Skolem (1923) studied these functions. Rózsa Péter coined the term *primitive recursive*.

3. *μ-recursive* or *General recursive* functions were outlined by Kart Gödel in 1934, to overcome the limitation of *Primitive recursive functions*. But it was formally presented by Stephen Cole Kleene in 1936 in his famous paper, "General Recursive Functions of Natural Numbers".

4. *λ-Calculus* was proposed by Alonzo Church in 1930s.

The famous *Church-Turing* thesis Claims the equivalence of theses formalisims.

2 Primitive Recursive and General(μ) Recursive Functions

The *Primitive Recursive Functions (PR)* are functions on non-negative integers. But there are computable functions that cannot be described by the recipe of PR. The *General recursive* or *μ-recursive* functions were introduced to accommodate unbounded search (like `while`-loop is a programming language) for a value. The functions are no longer *total* but *partial*, undefined at certain arguments.

2.1 Primitive Recursive Functions

We define a class of numeric functions, $f : \mathbb{N}_0^k \to \mathbb{N}_0$ known as *primitive recursive functions* (**PR**).

Definition 1. The *initial* or *base* functions of **PR** are the following:

(a) *Constant function*: $C_m^k(n_1, \cdots, n_k) = m$, a k-variable function whose value is a constant m, where $k, m = 0, 1, \cdots$.

(b) *Projection function*: $\Pi_i^k(n_1, \cdots, n_k) = n_i$, where $1 \leq i \leq k$ and $k = 1, 2, \cdots$.

(c) *Successor function*: $S(n) = n + 1$.

Definition 2. The following are two function constructors.

(a) The class of functions **PR** is closed under *function composition*.
The *k-ary* function f, defined as follows belongs to **PR**, whenever the *l-ary* function h and *k-ary* functions, $g_1, \cdots, g_l$, are in **PR**.

$$f(n_1, \cdots, n_k) = h(g_1(n_1, \cdots, n_k), \cdots, g_l(n_1, \cdots, n_k)).$$

(b) The class **PR** is also closed under *primitive recursion*.
The *$k+1$-ary* function f defined as follows belongs to **PR**, whenever the *k-ary* function g and the $(k+2)$-it ary function h belongs to **PR**.

$$\begin{aligned} f(0, n_1, \cdots, n_k) &= g(n_1, \cdots, n_k), \\ f(m+1, n_1, \cdots, n_k) &= h(f(m, n_1, \cdots, n_k), m, n_1, \cdots, n_k), \end{aligned}$$

The class of *primitive recursive functions*, **PR**, is the smallest class that contains the *initial functions* and is *closed under, composition* and *primitive recursion*.

Definition 3. A *k-ary* predicate or relation $P(x_1, \cdots, x_k)$ on $\mathbb{N}_0^k$ is interpreted as $R_P \subseteq \mathbb{N}_0^k$. The predicate is *primitive recursive*, if its *characteristic function* $\chi_P : \mathbb{N}_0^k \to \{0, 1\}$ is primitive recursive.

$$\chi_P(n_1, \cdots, n_k) = \left\{ \begin{array}{ll} 1 & \text{if } (n_1, \cdots, n_k) \in R_P, \\ 0 & \text{if } (n_1, \cdots, n_k) \notin R_P. \end{array} \right\}.$$

Example 1. Following are a few examples of *primitive recursive functions*.

(a) *Predecessor function* (Pd):

$$\begin{aligned} Pd(0) &= C_0^0(), \\ Pd(m+1) &= \Pi_2^2(Pd(m), m)). \end{aligned}$$

(b) *Addition function* (Add):

$$\begin{aligned} Add(0,n) &= \Pi_1^1(n), \\ Add(m+1,n) &= h((Add(m,n), m, n)), \end{aligned}$$

where $h = S \circ \Pi_1^3$.

(c) *Multiplication* (Mult):

$$\begin{aligned} Mult(0,n) &= C_0^1(n), \\ Mult(m+1,n) &= Add(\Pi_1^3, \Pi_2^3)(Mult(m,n), m, n), \end{aligned}$$

(d) If $f(a,b)$ is in **PR**, then $f_R(b,a) = f(a,b)$ is in **PR**.
$f_R(b,a) = f(\Pi_2^2, \Pi_1^2)(b,a)$.

(e) *Power* (Pow):

$$\begin{aligned} Pow(0,n) &= C_1^1(n), \\ Pow(m+1,n) &= Mult(\Pi_1^3, \Pi_3^3)(Pow(m,n), m, n), \end{aligned}$$

(f)

$$\begin{aligned} Fact(0) &= C_0^1() \\ Fact(m+1) &= Mult(\pi_1^2, (S \circ \pi_2^2))(Fact(m), m). \end{aligned}$$

(g) *Subtraction functsin* (Monus): We define the reverse subtraction.

$$\begin{aligned} RSub(0,n) &= P_1^1(n),\ n-0 \\ RSub(m+1,n) &= (Pd \circ P_1^3)(Sub(m,n), m, n)) \\ SUB = RSub \circ (\Pi_2^2, \Pi_1^2) & \end{aligned}$$

We define a sequence of one variable functions and show how they are defined by primitive recursion.

$$f_0 = S,\ f_1(x) = 2 + x;\ f_2(x) = 2x;\ f_3(x) = 2^x.$$

So we have

$$\begin{aligned} f_1(0) &= 2, \\ f_1(m+1) &= f_0(f_1(m)). \end{aligned}$$

$$\begin{aligned} f_2(0) &= 0, \\ f_2(m+1) &= f_1(f_2(m)). \end{aligned}$$

$$\begin{aligned} f_3(0) &= 1, \\ f_3(m+1) &= f_2(f_3(m)) \end{aligned}$$

In this way we may define

$$\begin{aligned} f_4(0) &= 1, \\ f_4(m+1) &= f_3(f_4(m)) \end{aligned}$$

and subsequent functions.

$$f_4(0) = 1,\ f_4(1) = f_3(f_4(0)) = 2^{f_4(0)} = 2,\ f_4(2) = 2^2,\ f_4(3) = 2^{2^2}, \cdots, f_4(m) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{m-\text{times}}.$$

So the growth rate of f_n eventually exceeds the growth rate f_{n-1}. It is not difficult to see that the values are becoming very large. Finally we define f as follows:

$$f(n) = f_n(n).$$

So $f(0) = f_0(0) = 1$, $f(1) = f_1(1) = 2 + 1 = 3$, $f(2) = f_2(2) = 2 \times 2 = 4$, $f(3) = f_3(3) = 2^3 = 8$, $f(4) = f_4(4) = 2^{2^{2^2}} = 2^{16} = 65536$,

$$\begin{aligned} & f(5) \\ = \ & f_5(5) \\ = \ & f_4(f_5(4)) \\ = \ & f_4(f_4(f_5(3))) \\ = \ & f_4(f_4(f_4(f_5(2)))) \\ = \ & f_4(f_4(f_4(f_4(f_5(1))))) \\ = \ & f_4(f_4(f_4(f_4(f_4(f_5(0)))))) \\ = \ & f_4(f_4(f_4(f_4(f_4(1))))) \\ = \ & f_4(f_4(f_4(f_4(2)))) \\ = \ & f_4(f_4(f_4(2^2))) \\ = \ & f_4(f_4(2^{2^{2^2}})) \\ = \ & f_4(f_4(65536)) \\ = \ & f_4(\underbrace{2^{\cdot^{\cdot^{2}}}}_{2^{2^{2^2}}=65536-times}))) \cdots \end{aligned}$$

The claim is that the growth of f is more than any f_i and cannot be expressed as a primitive recursive function. A similar well-known function is called the *Ackermann function* defined as follows:

First we define a slightly different sequence of functions - $g_i,\ i = 0, 1, 2, \cdots$.

$$\begin{aligned} g_0 &= S, \\ g_{i+1}(0) &= g_i(1), \\ g_{i+1}(m+1) &= g_i(g_{i+1}(m)). \end{aligned}$$

Now instead of defining $g(m) = g_m(m)$ we define a two variable Ackermann's function $A(m, n) = g_m(n)$. The definition of Ackermann's function as follows:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)). \end{aligned}$$

A few elements of the Ackermann's functions are tabulated below.

$y \rightarrow$	0	1	2	3	4	5	$\cdots$
0	1	2	3	4	5	6	$\cdots$
1	2	3	4	5	6	7	$\cdots$
2	3	5	7	9	11	13	$\cdots$
3	5	13	29	61	125	253	$\cdots$
4	13	32765	$\cdots$				
$\cdots$							
$x \uparrow$							

2.2 General or μ-Recursive Functions

Let $P(n_1, \cdots, n_k, m)$ be a $(k+1)$-ary predicate. We define the *k-ary partial function* $f(n_1, \cdots, n_k) = \mu m[P(n_1, \cdots, n_k, m)]$, where the value of $f(n_1, \cdots, n_k)$ is the minimum value of m so that $P(n_1, \cdots, n_k, m)$ is *true*. Otherwise, $f(n_1, \cdots, n_k)$ is *undefined*.

A *k-ary* partial-function f (may not be defined at every point of N_0^k) can be defined using a $(k + 1)$-*ary* predicate $g(n_1, \cdots, n_k, m) = 0$ as follows:

$$f(n_1, \cdots, n_k) = \mu m[g(n_1, \cdots, n_k, m) = 0],$$

This is called a definition by minimalization.

Definition 4. The class of *(μ)-recursive functions*, **R**, is the smallest class that contains the *initial functions* and is *closed under*, *composition*, *primitive recursion* and *minimalization*.

There are two important theorems about Ackermann's function which we shall state without proof [FCH].

Proposition 1. Ackermann's function is not *primitive recursive*.

Proposition 2. Ackermann's function is *recursive*.

3 Lambda Definable Functions

The *untyped lambda calculus (λ-calculus)* was introduced by Alonzo Church around 1930 to formalize mathematics. The original system was inconsistent[1] But the theory gave another foundation of *computing*. Starting from Lisp, many programming languages were influenced by it. The study of the formal semantics of programming languages has its foundations in the semantics of λ-calculus. Subsequently, typed versions of λ-calculus also gave foundations to the type systems of modern programming languages.

The original λ-calculus is a *type free theory* that views a function as a *computation rule*[2] rather than a *graph* as viewed in set theoretic mathematics[3].

3.1 Lambda Terms

The set of λ-terms, Λ, is defined inductively as follows. Let x be a variable[4]

1. Every variable is a λ-term ($x \in \Lambda$).
2. If M is any λ-term ($M \in \Lambda$), then $(\lambda x M)$ is a λ-term ($(\lambda x M) \in \Lambda$). This is called a function abstraction. $(\lambda x M)$ may be viewed as a function with one argument x, where the body of the function is M.
3. If M, N are λ-terms ($M, N \in \Lambda$), then (MN) is a λ-term ($(MN) \in \Lambda$). This is called a function application, where M is a function and N is its argument.

We shall use following conventions to avoid cluttering of parenthesis.

1. The outer parenthesis will be dropped.
2. The function abstraction will go as far as possible to the right i.e. $\lambda x_1(\lambda x_2(\cdots(\lambda x_n M)))$ will be written as $\lambda x_1 \lambda x_2 \cdots \lambda x_n \cdot M$ or often as $\lambda x_1 x_2 \cdots x_n \cdot M$, or $\lambda \vec{x} \cdot M$, where $\vec{x} = x_1 \cdots x_n$.
3. The function application is *left associative* i.e. $(\cdots(((MN_1)N_2)\cdots N_n)$ is written as $MN_1N_2\cdots N_n$.

Following are a few examples of λ-terms.
Example 2. x; xx; $\lambda x \cdot x$, known as I; $\lambda xy \cdot x$, known as K; $\lambda x \cdot xx$; $\lambda f \cdot (\lambda x \cdot f(xx)(\lambda x \cdot f(xx))$, known as Y; etc.

3.2 Convertibility: An Equivalence Relation

We define equality in the collection of ordinary arithmetic or algebraic expressions. As an example the value of $3+2$ is same as that of 5. They are not same expressions (syntactically), but their values are same. So we write $3 + 2 = 5$. Similarly we write $a(b + c) = ab + ac$, where $a, b, c \in \mathbb{R}$. These *equalities* are

[1] Shown by S C Kleene and J B Rosser, two students of Church

[2] Think of a Python program to compute $n!$ as a specification of the dynamics of factorial computation.

[3] Viewing a function as a rule to compute the value from the argument is actually older than viewing it as a collection of *argument* and *value* pairs.

[4] There is a enumerable supply of variables: $V = \{v_0, v_1, \cdots\}$ and x is a meta-variable over V.

actually equivalent relations over the set of arithmetic expressions and the set of algebraic expressions.

In a similar way we wish to define an *equivalence relation* over Λ, the collection of λ-terms. Two λ terms are equivalent (we may write them equal) if they denote the same *value* (whatever that is). They belong to the same equivalence class.

Definition 5. Two λ-terms M, N are said to be equal and is written as $M = N$ under the following situations (first three are standard for any equivalence relation):

1. $M = M$,
2. if $M = N$, then $N = M$,
3. if $M = N$ and $N = L$, then $M = L$,
4. $(\lambda x \cdot M)N = M[x \leftarrow N]$. It means when a function abstraction is applied to its argument, its value is same as the λ-term obtained by replacing every x (corresponding to the '*argument variable* x') in the body of M, by N. This is known as *β-conversion*[5].

   There is some restriction about this replacement of x by the argument N, which we shall discuss after this.
5. $\lambda x \cdot M = \lambda y \cdot (M[x \leftarrow y])$. Again there is some restriction which we shall discuss. This is known as *α-conversion*i[6].
6. if $M = N$, then $\lambda x \cdot M = \lambda x \cdot N$,
7. if $M = N$, then $MP = NP$ and also $PM = PN$, where $P \in \Lambda$.

We have mentioned about some restriction in connection to our fourth and fifth conditions for equivalence. We formalize it by defining the set of *free variables* and the set of *bound variables* in a term M, $FV(M)$i and $BV(M)$ respectively.

1. If x is a variable, then $FV(x) = \{x\}$ and $BV(x) = \emptyset$,
2. $FV(\lambda x \cdot M) = FV(M) \setminus \{x\}$, the function abstraction binds a variable[7]. So, $BV(\lambda x \cdot M) = BV(M) \cup \{x\}$.
3. $FV(MN) = FV(M) \cup FV(N)$ and $BV(MN) = BV(M) \cup BV(N)$.

If a variable present in a λ-term M is not *free*, then it is bound. It is possible that a variable x is both *free* and *bound* in a λ-term.

Example 3. $FV(\lambda x \cdot y(\lambda y \cdot xy)) = \{y\}$ and $FV(\lambda x \cdot y(\lambda y \cdot xy)) = \{x, y\}$. Note that the first occurrence of y is free but the second occurrence of y is bound.

When we write $M[x \leftarrow N]$, we replace ever free occurrence of x in M by N. But there is one problem. There may be a free occurrence of some y in N.

[5] Compare it with *macro substitution* by the C pre-processor or parameter passing by *name*.

[6] This is renaming of formal parameter of a function. We can do that if there is no name clash.

[7] In $\int_1^2 y^2 x \, dx$, y is free but x is not. $\int_1^2 \cdots dx$ binds x.

After substitution, the free y of N may be bound by some λy in M. So it is necessary to do α-conversion of sub-terms of M while doing substitution[8].

Example 4. Let $M = x(\lambda y \cdot xy)$ and $N = \lambda x \cdot yx$. In N we have a free variable y. If we compute $M[x \leftarrow N]$ without care, we get $(\lambda x \cdot yx)(\lambda y \cdot (\lambda x \cdot yx)y$. Now the second y has become bound and the substitution gives incorrect result.

The correct approach is to rename y in the sub-term $\lambda y \cdot xy$ in M (α-conversion).

$$(x(\lambda y \cdot xy))[x \leftarrow \lambda x \cdot yx] =_\alpha (x(\lambda z \cdot xz))[x \leftarrow \lambda x \cdot yx] =_\beta (\lambda x \cdot yx)(\lambda z \cdot (\lambda x \cdot yx)z).$$

Definition 6. A λ-term M without a free variable is called a *combinator*. We have already given several examples of combinators.

$$\begin{aligned} I &\equiv \lambda x \cdot x, \\ K &\equiv \lambda xy \cdot x, \\ S &\equiv \lambda xyz \cdot xz(yz), \\ \Omega &\equiv (\lambda x \cdot xx)(\lambda x \cdot xx), \ \textit{etc.} \end{aligned}$$

A *context* $C[]$ is an incomplete term with a hole in it. We can plug a λ-term in the hole. **Example 5.** Let $C[] = \lambda x \cdot x(\lambda y \cdot [])$ and we plug $M = xy$ in it to get $C[M] = \lambda x \cdot x(\lambda y \cdot xy)$.

If two λ-terms M and N are equal, and $C[]$ is a context, then $C[M] = C[N]$. We accept it with out proof.

3.3 Reduction

In arithmetic $(59 + 7 \times 2 - 3) \div 7 = 10$. But there is an asymmetry as far as computation goes. Given $(59 + 7 \times 2 - 3) \div 7$ we reduce it to 10, but not the other way. So a *reduction* is an one way operation. The steps of reduction for this example are as follows:

$$(59 + 7 \times 2 - 3) \div 7 \to (59 + 14 - 3) \div 7 \to (73 - 3) \div 7 \to 70 \div 7 \to 10.$$

It is clear that terms in each step of reduction are in the same equivalence class. The final term 10 cannot be reduced further (in *reduced* or *normal* form), a representative of the equivalence class.

Definition 7. We define the binary relation β on the collection of λ-terms as follows:

$$\beta = \{((\lambda x \cdot M)N, M[x \leftarrow N]) : \ M, N \in \Lambda\}.$$

If $(M, N) \in \beta$, then M is called β-*redex* and N is called β-*contractum* of M.

Definition 8. The *compatible closure* of the binary relation β is the binary relation $\to_\beta$ defined as follows:

1. If $(M, N) \in \beta$, then $M \to_\beta N$.
2. If $M \to_\beta N$, then $PM \to_\beta PN$, $MP \to_\beta NP$, $\lambda x \cdot M \to_\beta \lambda \cdot N$.

[8]There are modified form of λ-expressions where this problem will not arise.

If $C[]$ is a *context* of λ-term and $M \to_\beta N$, then $C[M] \to_\beta C[N]$.

The *reflexive-transitive closure* of $\to_\beta$ is defined as usual and is denoted by $\to_\beta^*$. So $\to_\beta^*$ is the smallest binary relation such that $\to_\beta \subseteq \to_\beta^*$ and $\to_\beta^*$ is both reflexive and transitive.

A λ-term M is in *β-normal form* if no *sub-term*[9] of it is a β-redex.

The $\to_\beta^*$ induces the β-equivalence we have defined earlier.

Example 6. Following are a few examples of β-reductions.

1. $(\lambda x \cdot x)M \to_\beta M$, so $\lambda x \cdot x$ behaves like *identity function*.
2. $(\lambda xy \cdot x)MN \to_\beta (\lambda y \cdot M)N \to_\beta M$, as we know that there cannot be any free variable y in M.
3. $(\lambda x \cdot xx)(\lambda y \cdot y)M \to_\beta (\lambda y \cdot y)(\lambda y \cdot y)M \to_\beta (\lambda y \cdot y)M \to_\beta M$.
4. $(\lambda x \cdot xx)(\lambda x \cdot xx) \to_\beta (\lambda x \cdot xx)(\lambda x \cdot xx) \to_\beta \cdots$, a non terminating computation.
5. $(\lambda xy \cdot (\lambda xy \cdot xy)(\lambda x \cdot x)x)(\lambda xy \cdot y)$ can be β-reduced in more than one ways.
   (a) $\to_\beta (\lambda xy \cdot (\lambda y \cdot (\lambda x \cdot x)y)x)(\lambda xy \cdot y) \to_\beta (\lambda xy \cdot (\lambda y \cdot y)x)(\lambda xy \cdot y) \to_\beta (\lambda xy \cdot x)(\lambda xy \cdot y) \to_\beta (\lambda y \cdot (\lambda xy \cdot y)$. We can rename the first parameter y and get $\lambda zxy \cdot y$.
   (b) $\to_\beta \lambda y \cdot (\lambda xy \cdot xy)(\lambda x \cdot x)(\lambda xy \cdot y) \to_\beta \lambda y \cdot (\lambda y \cdot (\lambda x \cdot x)y)(\lambda xy \cdot y) \to_\beta \lambda y \cdot (\lambda x \cdot x)(\lambda xy \cdot y) \to_\beta \lambda y \cdot (\lambda xy \cdot y)$. Again by renaming we get $\lambda zxy \cdot y$.

Example 7. If we are not careful about substitution, we may prove almost anything. Let the λ-term be $K = \lambda x \lambda y \cdot x$. Take any λ-term M, N. $KMN = M$. Now take $M = y$ and we get $KyN = (\lambda y \cdot y)N = N$.

It is possible to compute $2 + 3 + 4$ in two different ways. The law of *associativity* ensures that the final values are same. Similarly, it is necessary to prove that different valid sequences of β-reductions should lead to same *β-normal form* (modulo *α-conversion*). But there is one problem - a reduction sequence may be non-terminating.

Example 8.

$$(\lambda xy \cdot y)((\lambda x \cdot xx)(\lambda x \cdot xx)) \to_\beta (\lambda xy \cdot y)((\lambda x \cdot xx)(\lambda x \cdot xx)) \cdots,$$

is non-terminating if we evaluate $(\lambda x \cdot xx)(\lambda x \cdot xx)$ first. But

$$(\lambda xy \cdot y)((\lambda x \cdot xx)(\lambda x \cdot xx)) \to_\beta (\lambda y \cdot y),$$

is terminating.

We assume two very important properties of β-reduction without proof. The first one states that whatever be the path of reduction, two terminating reductions will give equivalent β-normal form (same value) modulo renaming (α-conversion). In fact for every λ-term has at most one β-normal form.

This is known as *Church-Rosser* property of term rewriting. Let M be a λ-term which can be β-reduced to M_1 and M_2 in two different ways, then there is a λ-term M such that both M_1 and M_2 can be reduced to in finite number of steps.

[9] We have not formally defined *sub-terms* of a λ-term M. Informally it is a portion of M that is a valid λ-term. As an example sub terms of $x\lambda x \cdot xy$ are x, $\lambda x \cdot xy$, xy, x, y and the whole term.

3.4 Fixed Point Theorem and Combinators

We start with the following proposition:

Proposition 3. For all λ-term F, there is a λ-term X so that $FX = X$.

Proof: Given F, we define $W = \lambda x \cdot F(xx)$ and $X = WW$. We see that X β-reduces to FX!

$$\begin{aligned} & X = WW = (\lambda x \cdot F(xx))W, \\ \rightarrow_\beta \quad & F(WW) = FX. \end{aligned}$$

QED.

Definition 9. A *fixed point combinator* is a closed λ-term M so that for all λ-term F we have $MF = F(MF)$. So MF is a fixed point of F.

Example 9. $Y = \lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx))$ is a fixed point combinator.

$\Theta = (\lambda x \cdot \lambda f \cdot f(xxf))(\lambda x \cdot \lambda f \cdot f(xxf))$ is Turing fixed point combinator.

Example 10. Assume that *conditional, test for zero, subtraction* and *multiplication* are λ-definable (soon we shall do that). The factorial function F is defined as follows:

$$Fn \equiv (\lambda n \cdot \text{ if } n = 0 \text{ then } 1 \text{ else } M\ n\ (F\ (P\ n)))n,$$

where Mxy gives $x * y$. Pn is the predecessor function, gives $n - 1$ when $n > 0$. But in $\lambda - calculus$ F cannot be defined using F (no recursion).

Let

$$H \equiv \lambda f \lambda n \cdot (\text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (f\ (P\ n))).$$

Our factorial function is a fixed point of $H = (\lambda f \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (f\ (P\ n)))$ i.e

$$HF = \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (F\ (P\ n)) \equiv F.$$

Y combinator computes the fixed point: $YH = H(YH)$.

The computation goes as follows - we call the λ-term $\lambda f \lambda n \cdot (\text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (F\ (P\ n)))$ as F.

$$\begin{aligned} & (YH)\ 5, \\ = \quad & Y(\lambda f \lambda n \cdot (\text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (f\ (P\ n))))\ 5, \\ \equiv \quad & (\lambda f \lambda n \cdot (\text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ (f\ (P\ n))))\ (YH)\ 5, \\ \equiv \quad & (\lambda n \cdot (\text{if } n = 0 \text{ then } 1 \text{ else } M\ n\ ((YH)\ (P\ n))))\ 5, \\ \equiv \quad & (\text{if } 5 = 0 \text{ then } 1 \text{ else } M\ 5\ ((YH)\ (P\ 5)))), \\ \equiv \quad & M\ 5\ ((YH)\ 4), \end{aligned}$$

And the computation continues.

3.5 Lambda Definable

The λ-calculus (λ-terms and β-reduction etc.) may be used to represent the class of μ-*recursive functions* on natural numbers[10]. Following are representations of different mathematical objects as λ-terms.

Conditional and Truth Values - A *conditional* is a λ-term of the form BMN, where B corresponds to truth values *true* or *false*. If B is *true*, it evaluates to M, if B is *false*, it evaluates to N. So the truth values are *true* $\equiv\ \lambda xy \cdot x$ and *false* $\equiv\ \lambda xy \cdot y$.

[10] This was first proved by S C Kleene, a student of Alonzo Church.

n-tuples An ordered pair defined by Church keeping projection function in mind. $(M, N) \equiv \lambda z \cdot zMN$. So

$$(M, N)T = (\lambda z \cdot zMN)(\lambda xy \cdot x) \to_\beta (\lambda xy \cdot x)MN \to_\beta M.$$

Similarly,

$$(M, N)F = (\lambda z \cdot zMN)(\lambda xy \cdot x) \to_\beta (\lambda xy \cdot y)MN \to_\beta N.$$

An ordered n-tuple $(M_1, M_2, \cdots, M_n) \equiv (M_1, (M_2, \cdots, M_n)\cdots)$, where $n > 1$ and $[M] \equiv M$[11] The important function are the projection functions. There are n projection functions, π_n^i that takes an n-tuple and projects its i^{th} component. We define

$$\pi_n^i = \lambda x \cdot x \overbrace{FF\cdots F}^{(i-1)} T,\ 1 \le i < n, \text{ and } \pi_n^n = \lambda x \cdot x \overbrace{FF\cdots F}^{n}.$$

Example 11. Consider a 3-tuple $(P, (Q, R))$. We apply $\pi_3^1, \pi_3^2, \pi_3^3$.

$$\begin{aligned} & \pi_3^1(P, (Q, R)) = (\lambda x \cdot xT)(P, (Q, R)), \\ \to_\beta\ & (P, (Q, R))T = (\lambda z \cdot zP(Q, R))T, \\ \to_\beta\ & TP(Q, R), \\ \to_\beta\ & P. \end{aligned}$$

$$\begin{aligned} & \pi_3^2(P, (Q, R)) = (\lambda x \cdot xFT)(P, (Q, R)), \\ \to_\beta\ & (P, (Q, R))FT = (\lambda z \cdot zP(Q, R))FT, \\ \to_\beta\ & FP(Q, R)T, \\ \to_\beta\ & (Q, R)T = (\lambda z \cdot zQR)T, \\ \to_\beta\ & Q. \end{aligned}$$

Note that $\to_\beta$ may be one or more steps of β-reductions.

Numerals can be defined using λ-terms in different ways. The requirement is that if $i, j \in \mathbb{N}_0$ and $i \neq j$, then the λ-numerals corresponding to i and j should not belong to the same equivalence class. On top of this the numerals should be such that the basic operations can be easily encoded as λ-terms. Following is a convenient λ-numeral system.

Definition 10. If $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$,

$$\lambda(n) \begin{cases} I = \lambda x \cdot x & \text{if } n = 0, \\ (F, \lambda(n-1)), & \text{if } n > 0. \end{cases}$$

So we have

$$\lambda(1) = (F, \lambda(0)) = \lambda z \cdot zFI;\ \lambda(2) = (F, \lambda(1)) = \lambda z \cdot zF(F, \lambda(0)),\ \text{etc.}$$

We need to define the following three *primitive* functions:

[11] There may be other definitions as well.

- Test for zero:
$$Z\lambda(n) = \begin{cases} T & \text{if } n = 0, \\ F & \text{if } n > 0. \end{cases}$$
- Successor function: $S\lambda(n) = \lambda(n+1)$.
- Predecessor function:
$$P\lambda(n) = \begin{cases} I & \text{if } n = 0, \\ \lambda(n-1) & \text{if } n > 0. \end{cases}$$

We define

- $Z = \lambda x \cdot xT$. So,
$$ZI = (\lambda x \cdot xT)I = IT = T,$$
and
$$Z\ \lambda(n{+}1) = Z\ (F, \lambda(n)) = (\lambda x{\cdot}xT)(\lambda z{\cdot}zF\lambda(n)) = (\lambda z{\cdot}zF\lambda(n))T = TF\lambda(n) = F.$$
- $S = \lambda x \cdot (F, x)$. It is clear that it works.
- $P = \lambda x \cdot (Zx)I(xF)$. It is clear that
$$PI = (\lambda x \cdot (Zx)I(xF))I = (ZI)I(IF) = TIF = I.$$
and
$$\begin{aligned} P\ \lambda(n+1) &= (\lambda x \cdot (Z\ x)\ I\ (xF))\ \lambda(n+1) \\ &= (Z\ \lambda(n+1))\ I\ (\lambda(n+1)\ F) \\ &= F\ I\ (\lambda(n+1)\ F) \\ &= (\lambda(n+1)\ F) \\ &= (\lambda z \cdot z\ F\lambda(n))\ F \\ &= F\ F\ \lambda(n) \\ &= \lambda(n). \end{aligned}$$

Our next question is how do we λ-define functions from $\mathbb{N}_0^k \to \mathbb{N}_0$. A typical example is $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. But in λ-calculus we take one argument at a time[12]. So we take our addition as Plus : $\mathbb{N}_0 \to [\mathbb{N}_0 \to \mathbb{N}_0]$[13], and define it as follows:
$$Plus\ m\ n = \begin{cases} n & \text{if } m = 0, \\ S\ (Plus\ (P\ m)\ n) & \text{if } m > 0. \end{cases}$$
So our '*Plus*' satisfies the following equation and by the fixed point theorem it has a solution.
$$Plus\ m\ n = (Z\ m)\ n\ (S\ (Plus\ (P\ m)\ n)).$$
Consider the λ-term
$$F = \lambda fmn \cdot (Z\ m)\ n\ (S\ (f\ (P\ m)\ n)).$$

[12] We may us *pair* as the single argument and use projection etc.

[13] *Curried* version of '+'. One can show that $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ is isomorphic to $\mathbb{N}_0 \to [\mathbb{N} \to \mathbb{N}_0]$.

F has a *fixed point* and that is a solution of the equation, our *Plus*. We already know that $Y\ F$ is a fixed point of F.
Example 12.

$$\begin{array}{ll} & (YF)\ \lambda(2)\ \lambda(3) \\ \rightarrow_\beta^* & F(YF)\ \lambda(2)\ \lambda(3) \\ \rightarrow_\beta & (\lambda mn \cdot (Z\ m)\ n\ (S\ ((YF)\ (P\ m)\ n))\ \lambda(2)\ \lambda(3) \\ \rightarrow_\beta^2 & (Z\ \lambda(2))\ n\ (S\ ((YF)\ (P\ \lambda(2) i)\ \lambda(3)) \\ \rightarrow_\beta^* & (S((YF)\ (P\ \lambda(2))\ \lambda(3))) \\ \rightarrow_\beta^* & (S((YF)\ \lambda(1)\ \lambda(3))) \\ \rightarrow_\beta^* & (S(S((YF)\ \lambda(0)\ \lambda(3)))) \\ \rightarrow_\beta^* & (S(S\lambda(3))) \\ \rightarrow_\beta^* & \lambda(5) \end{array}$$

Definition 11. A function $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ is λ-definable if there is a λ-term F so that for each $(n_1, \cdots, n_k) \in \mathbb{N}_0^k$,

$$F\ \lambda(n_1)\ \cdots\ \lambda(n_k) \rightarrow_\beta^* \lambda(f(n_1, \cdots, n_k).$$

Addition, multiplication, exponentiation, factorial etc. can be shown to be λ-definable

3.6 Lambda Definable and Recursive Functions

We shall show that any numeric function that is λ-definable is also *recursive*[14].

Proposition 4. All initial functions are λ-definable.
Proof:

1. *Constant function*: $\lambda(C_m^k) = \lambda x_1 \cdots x_k \cdot m$.
2. *Projection function*: $\lambda(\Pi_i^k) = \lambda x_1 \cdots x_k \cdot x_i$.
3. *Successor function:* $\lambda(S) = \lambda x \cdot (F, x)$.

QED.

Proposition 5. λ-definable functions are closed under composition.
Proof: Let the *k-ary* functions $g_1, \cdots, g_l$ and the *l-ary* function h be λ-definable by $G_1, \cdots, G_k$ and H. So, $f(x_1, \cdots, x_k) = h(g_1(x_1, \cdots, x_k), \cdots, g_l(x_1, \cdots, x_k))$ is λ-defined by

$$\lambda(f) = \lambda x_1 \cdots x_k \cdot H\ (G_1 x_1 \cdots x_k) \cdots (G_l x_1 \cdots x_k).$$

QED.

Proposition 6. λ-definable functions are closed under primitive recursion.

[14]In fact any function that is recursive is λ-definable, Turing computable and vice versa.

Proof: Let the *k-ary* function g and the $(k+2)$-*ary* function h be λ-definable and a $(k+1)$-*ary* function is defined as follows:

$$\begin{aligned} f(0, n_1, \cdots, n_k) &= g(n_1, \cdots, n_k), \\ f(m+1, n_1, \cdots, n_k) &= h(f(m, n_1, \cdots, n_k), m, n_1, \cdots, n_k), \end{aligned}$$

We define

$$F m\ n_1\ \cdots\ n_k = (Z\ m)(G\ n_1\ \cdots\ n_k)(H\ (F(P\ m)n_1\ \cdots\ n_k)n_1\ \cdots\ n_k)$$

From our discussion of *fixed point theorem* we know that there is a solution of this equation e.g.

$$Y(\lambda f \lambda m \lambda n_1\ \cdots\ n_k \cdot (Z\ m)(G\ n_1\ \cdots\ n_k)(H\ (f(P\ m)n_1\ \cdots\ n_k)n_1\ \cdots\ n_k))$$

QED.

Finally we wish to prove that λ-definable functions are closed under *minimalization*.

Let P be a λ-term. We define $H_P = \Theta\ F_P$, where $\Theta = (\lambda xy \cdot y(xxy))(\lambda xy \cdot y(xxy))$ is the *Turing's fixed point combinator* and $F_P = \lambda hz \cdot (P\ z)\ z\ (h\ (S\ z))$, where S is the λ-term corresponding to the *successor* function.

We also define

$$\mu P = H_P\ \lambda(0) = (P\ \lambda(0))\ \lambda(0)\ ((\Theta\ F_P)\ (S\ \lambda(0)).$$

That is, "*if* $(P\ \lambda(0))$, then $\lambda(0)$, *else* $((\Theta\ F_P)\ (S\ \lambda(0))$".

Proposition 7. Let P be a λ-term so that for all $n \in \mathbb{N}_0$, $P\ \lambda(n)$ is either *true* $(\lambda xy \cdot x)$ or *false* $(\lambda xy \cdot y)$, then

1. $H_P\ \lambda(n) \rightarrow_\beta^*$ *if* $(P\ \lambda(n))$, then $\lambda(n)$, *else* $((\Theta\ F_P)\ (S\ \lambda(n))$".
2. If there is an $n \in \mathbb{N}_0$, so that $P\ \lambda(n)$ is *true* and m is the smallest of such n, then μP defined as $H_P\ \lambda(0)$, is $\lambda(m)$.

Proof:

1. This comes from the property of *fixed point combinator*.
2. The second part is also from the property of the *fixed point combinator*. We know that for all k, $0 \leq k < m$, $P\ \lambda(k) =$ *false*, and $P\ \lambda(m) =$ *true*. So we have

$$H_p\lambda(0) = H_p\lambda(1) = \cdots = H_p\lambda(m) = \lambda(m).$$

QED.

Proposition 8. λ-definable functions are closed under *minimalization*.

Proof: Let the *k-ary* function f be defined by *minimalization* where

$$f(n_1, \cdots, n_k) = \mu m[g(n_1, \cdots, n_k, m) = 0],$$

where g, the $(k+1)$-*ary* function λ-definable by G.

Our P is $\lambda m \cdot Z\ (G\ \lambda(n_1) \cdots \lambda(n_k)\ \lambda(m))$, where Z tests for zero and f is defined as

$$F = \lambda n_1 \cdots \lambda n_k \cdot \mu[\lambda m \cdot Z\ (G\ n_1 \cdots n_k\ m)].$$

So,

$$F\ \lambda(n_1)\ \cdots\ \lambda(n_k) = \mu[\lambda m \cdot Z\ (G\ \lambda(n_1)\ \cdots\ \lambda(n_k)\ m].$$

The right hand side will extract the smallest value of m, if there is one, so that

$$Z\ (G\ \lambda(n_1)\ \cdots\ \lambda(n_k)\ \lambda(m)) \rightarrow_\beta^*\ true(\lambda xy \cdot x).$$

Otherwise the reduction will not terminate. QED.

Proposition 9. All recursive functions are λ-definable and vice versa.

Definition 12. *Church numerals* are $\lambda(n) = \lambda fx \cdot f^n(x)$, where $n \in \mathbb{N}_0$. It applies the first argument f on the second argument x, n number of times.

So, we have $c_0 = \lambda fx \cdot x = true$, $c_1 = \lambda fx \cdot fx$, $c_1 = \lambda fx \cdot f(fx)$ etc.

The successor function for this numeral is $S_c = \lambda nfx \cdot f(nfx)$. The predecessor function is more complicated.

$$P_c = \lambda nfx \cdot n\ (\lambda ab \cdot b\ (af))\ (\lambda c \cdot x)\ (\lambda d \cdot d).$$

We apply $P_c c_2$ and get

$$\begin{aligned}
& (\lambda nfx \cdot n\ (\lambda ab \cdot b\ (af))\ (\lambda c \cdot x)\ (\lambda d \cdot d))\ (\lambda fx \cdot f\ (f\ x)), \\
= \ & \lambda fx \cdot (\lambda fx \cdot f\ (f\ x))\ (\lambda ab \cdot b\ (af))\ (\lambda c \cdot x)\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda x \cdot (\lambda ab \cdot b\ (af))\ ((\lambda ab \cdot b\ (af))\ x))\ (\lambda c \cdot x)\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda x \cdot (\lambda ab \cdot b\ (af))\ (\lambda b \cdot b\ (xf)))\ (\lambda c \cdot x)\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda x \cdot (\lambda b \cdot b\ ((\lambda b \cdot b\ (xf))\ f)))\ (\lambda c \cdot x)\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda x \cdot (\lambda b \cdot b\ (f\ (xf))))\ (\lambda c \cdot x)\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda b \cdot b\ (f\ ((\lambda c \cdot x)f)))\ (\lambda d \cdot d)), \\
= \ & \lambda fx \cdot (\lambda b \cdot b\ f\ x)\ (\lambda d \cdot d), \\
= \ & \lambda fx \cdot (\lambda d \cdot d)\ f\ x, \\
= \ & \lambda fx \cdot f\ x = c_1.
\end{aligned}$$

References

[HPB] *The Lambda Calculus, Its Syntax and Semantics* by H P Barendregt, Series: Studies in Logic and The Foundations of mathematics, vol. 103, Pub. North-Holland, 1984, ISBN 0-444-87508-5.

[AS] `www.dsi.unive.it/~salibra/pdf15sep2003-chapter01.pdf` by Antonino Salibra, chapter 1, Lambda Calculus.

[FCH] *Introduction to Computability* by F C Hennie, Pub. Addison-Wesley, 1977, ISBN 0201028485.