Introduction to Lambda Calculus

The Lambda calculus, developed by Alonzo Church, is – besides the Turing and the Markov machines – an elegant model of what is meant by effective computation. Lambda calculus works with anonymous unary functions and the core action is the function application. In particular, function application is performed by a purely substitutive process: the (free) occurrences , within the function body, of the formal parameter of the function are replaced by the actual parameter. The process of function application has no state and, therefore, there is no return from a function. The full computation process is applicative and unwinds on a single direction, eventually terminating with a result that cannot be reduced further.

Lambda calculus stands as a basis for an interesting class of programming languages, called functional or applicative, such as ML, Haskell, Miranda, Clean etc. The following discussion is on the basic elements of (untyped) Lambda calculus and suggests the way the pure mathematical formalism of the calculus can be turned into a weak-typed functional programming language based on normal-order evaluation.

Lambda expressions

The basic element in Lambda calculus is the λ-*expression*, which can be a variable, a function or a function application.

Definition 1. Let **V** be a set of symbols, called variables, such that **V** does not contain the reserved symbols λ, **.**, **(** and **)**. We also consider as symbols from **V** sequences built from alphanumeric characters different from white spaces and the reserved symbols. A λ-expression is constructed according to the following rules:

- Variable: any variable from **V** is a λ-expression.
- Abstraction: if **x**∈**V** and **E** is a λ-expression, then λ**x.E** is a λ-expression that stands for a function with the formal parameter **x** and the body **E**.
- Application: if **F** and **A** are λ-expressions, then **(F A)** is a λ-expression that represents the application of the expression **F**, standing for a **1**-ary function, on the parameter **A**.

We note Λ, the set of all λ-expressions. The following formulae are correct λ-expressions (values from Λ).

x	is a λ-expression in the simplest form: a variable;
λ**x.x**	is the identity function;
λ**x.**λ**y.x**	is a selection function which selects the first value from a pair of values;
(λ**x.x y)**	stands for the application of λ**x.x** on the actual parameter **y**.

Definition 2. Let **E** be a λ-expression and **x** a variable from **E**. An occurrence x_n of **x** within **E** is *bound* in **E** if **E** has the form λ**x.F** or contains a sub-expression of the form λ**x.F** such that: (1) x_n is **F** or (2) x_n is contained in **F** or (3) x_n occurs immediately after the symbol λ. An unbound occurrence of **x** in **E** is *free* in **E**.

Consider the expression **E** $\equiv_{def}$ **(**λ**x.x x)** and let's use subscripts to highlight the occurrences of the variable **x**. The expression can be written $(\lambda x_1.x_2\ x_3)$, where the occurrences x_1 and x_2 of the variable **x** are bound, whereas the occurrence x_3 is free in **E**.

Definition 3. A variable is bound in a λ-expression **E** if all its occurrences in **E** are bound. If at least an occurrence of the variable is free in **E**, then the variable is free in **E**. An expression that does not contain free variables is *closed*.

In the expression **(λx.λz.(z x) (z y))** the variable **x** is bound, whereas **y** and **z** are free (although **z** is bound within **λz.(z x)**). The variable **x** from a λ-expression **λx.F** is a *binding variable* and stands for the formal parameter of the function **λx.F**. The free variables of a λ-expression may be considered constants with a predefined meaning.

For example, the λ-expression **λx.((+ x) 1)** corresponds to a function with a single parameter **x** and the body **((+ x) 1)**. Provided that the free variable **+** represents the integer addition, and the free variable **1** corresponds to the natural number **1**, the expression describes the computation of the successor of **x**. The resemblance of the function **λx.((+ x) 1)** with the Haskell function **\x -> x + 1** or with the Scheme function **(lambda (x) (+ x 1))** is obvious.

The application **(λx.F A)** is computed by replacing all *free occurrences* of the binding variable **x** in **F** by **A**. We note $F_{[A/x]}$ the result of this substitution. For instance, the application **(λx.x y)** yields the result **y**. Here the single free occurrence of **x** is the body of the function. If **F** does not contain free occurrences of the binding variable **x** then the application **(λx.F A)** produces **F**.

Definition 4. The transformation of λ-expression **(λx.F A)** to $F_{[A/x]}$ is noted $(\lambda x.F\ A)\to_\beta F_{[A/x]}$ and is called β-*reduction*. The expression **(λx.F A)** is called a β-*redex* (redex, for short). If **F** does not contain free occurrences of the variable **x**, then the transformation $\lambda x.(F\ x)\to_\eta F$ is called η-*reduction*.

Convention. We also write $E \to_\beta E'$ if the λ-expression **E** can be converted to **E'** by reducing a β-redex nested at any depth within **E**. The same convention applies for the η-reduction. Therefore, if $E\to_\beta E'$, and **H** is an arbitrary λ-expression, we will write:

$$(H\ E) \to_\beta (H\ E'),\quad (E\ H) \to_\beta (E'\ H),\quad \lambda x.E \to_\beta \lambda x.E'$$

For any λ-expression **F** (without free occurrences of the variable **x**) and any expression **A** the following property is observed:

$$(\lambda x.(F\ x)\ A)\to_\beta (F\ A) \text{ and } (\lambda x.(F\ x)\ A)\to_\eta (F\ A).$$

Therefore, from the pure computational view-point, the β-*reduction* does suffice for computing λ-expressions. When we are using the β-*reduction* what is important is the result of the computation. Nevertheless, the η-*reduction* is useful when we are interested in the equivalence of λ-expressions. For example, although the expressions **λx.λy.(x y)** and **λx.x** are structurally different and cannot be β-reduced to a common form, from the computational point of view (behaviorally) they designate the same function:

$$((\lambda x.\lambda y.(x\ y)\ a)\ b) \to_\beta (\lambda y.(a\ y)\ b)\to_\beta (a\ b)$$
$$((\lambda x.x\ a)\ b) \to_\beta (a\ b)$$

While the β-reduction is a computation rule, the η-reduction is a rule that can be used to investigate the equivalence of λ-expressions. Indeed, $\lambda x.\underline{\lambda y.(x\ y)} \to_\eta \lambda x.x$.

The mechanical substitution $F_{[A/x]}$ may lead to clashes between the bound variables and the free variables from within an expression.

For instance, consider the expression:

$$(\lambda x.\underbrace{\lambda y.(x\ y)}_{F}\ \underbrace{y}_{A})$$

The substitution $F_{[A/x]}$ yields $\lambda y.(y\ y)$. This result is in error since the variable **y**, which is free in **A**, becomes bound in $\lambda y.(y\ y)$. The free variable **y** changes its meaning. In order to avoid errors of this kind, due to free variables from **A** that coincide with bound variables from **F**, the expression **F** must be re-labeled by renaming all the bound variables that coincide with the free variables from **A**. In the case above, the variable **y** from $\lambda y.(x\ y)$ must be renamed, say **z**: $(\lambda x.\lambda z.(x\ z)\ y)$. The right result of the application is $\lambda z.(y\ z)$. The variable **y** stays free thus preserving its meaning.

Definition 5. The systematic re-labeling of bound variables from a function, usually variables that coincide with the free variables from the actual parameter onto which the function is applied, is called α-*conversion* and is noted $\rightarrow_\alpha$. The basic transformation performed by α-*conversion* is $\lambda x.F \rightarrow_\alpha \lambda y.F_{[y/x]}$, where **y** does not have free occurrences within **F**.

By convention, if a λ-expression **E** is transformed to an expression **E'** by α-converting an expression nested at any depth within **E**, we also write $E \rightarrow_\alpha E'$. For example, consider the following α-*conversions*:

$\lambda x.(x\ y) \rightarrow_\alpha \lambda z.(z\ y)$ is right since **z** is not free in $(x\ y)$
$\lambda x.\lambda x.(x\ y) \rightarrow_\alpha \lambda y.\lambda x.(x\ y)$ is wrong since **y** is free in $\lambda x.(x\ y)$
$\lambda y.\lambda x.(x\ x) \rightarrow_\alpha \lambda x.\lambda x.(x\ x)$ is right since **x** is not free in $\lambda x.(x\ x)$[1]

The process of α-conversion does not modify the meaning of the λ-expression it applies to. The result is equivalent to the initial expression in the same way two structurally equivalent functions may differ only in the names of their parameters.

Notice that if the α-conversion of a β-redex $(\lambda x.F\ A)$ labels the bound variables from the redex such that $\forall x \in BV(\lambda x.F) \bullet x \notin FV(A)$, where $BV(\lambda x.F)$ is set of the bound variables from $\lambda x.F$ and $FV(A)$ is the set of the free variables from **A**, and, moreover, the binding variables from $\lambda x.F$ are distinct, then the β-reduction $(\lambda x.F\ A) \rightarrow_\beta F_{[A/x]}$ can be performed by mechanically substituting of *all occurrences* of the binding variable **x** within **F** by the argument **A**. For instance,

$$\begin{aligned}(\lambda x.(\lambda x.(x\ x)\ x)\ x) &\rightarrow_\alpha (\lambda y.(\lambda x.(x\ x)\ y)\ x)\\ &\rightarrow_\alpha (\lambda y.(\lambda z.(z\ z)\ y)\ x)\\ &\rightarrow_\beta (\lambda z.(z\ z)\ x)\\ &\rightarrow_\beta (x\ x)\end{aligned}$$

Reduction

Definition 6. Let **E** be a λ-expression and **E'** a β-*redex* from **E** such that $E' \rightarrow_\alpha E_i \rightarrow_\beta E''$. A *reduction step* of the expression **E** is to substitute **E'** in **E** by **E"**. A sequence of

[1] Usually, it is clearer to work with bound variables that have different names. The renaming of **y** to **x** is performed here just for the sake of the example itself.

reductions steps of a λ-expression **`E`** is a reduction sequence of **`E`**. We note with $\longrightarrow$ a reduction step and with $\longrightarrow^*$ a reduction sequence.

A reduction step **`E`**$\longrightarrow$**`E'`** consists of a $\rightarrow_\beta$ reduction performed on a redex **`(λx.F A)`** from within **`E`**, redex that is α-converted such that $\forall$**`x∈BV(λx.F) • x∉FV(A)`**. Therefore, a reduction step can be viewed as the composition $\rightarrow_\beta \circ \rightarrow_\alpha$.

Formally, the $\longrightarrow^*$ relation is the reflexive, transitive closure of the one-step reduction $\longrightarrow$, as follows :

- **`if E`** $\longrightarrow$ **`E' then E`** $\longrightarrow^*$ **`E'`**
- **`E`** $\longrightarrow^*$ **`E`**
- **`if E`** $\longrightarrow^*$ **`E' and E'`** $\longrightarrow^*$ **`E" then E`** $\longrightarrow^*$ **`E"`**

For example,

`((λx.λy.(y x) y) λx.x)` $\longrightarrow$ **`(λu.(u y) λx.x)`** $\longrightarrow$ **`(λx.x y)`** $\longrightarrow$ **`y`**
`((λx.λy.(y x) y) λx.x)` $\longrightarrow^*$ **`y`**

Normal forms

Definition 7. A reduction sequence of an expression terminates when the expression does not contain β-*redexes* (therefore, there is no application left to be computed). An expression that does not contain β-*redexes* is in *normal form*.

Apart from the normal form specified by definition 7, there are additional normal forms of practical value. In each such particular case, it is considered that the reduction process terminates when the expression reaches a conventional format, even if it contains β-*redexes.* From the programming point of view, relevant are the *head normal form* and the *weak head normal form*.

Definition 8. A λ-expression is in *head normal form* if it has the format $\lambda x_1.\ \lambda x_2.\ldots.\lambda x_n.(\ldots((y\ E_1)\ldots E_m)$, where **`n≥0`**, **`m≥0`**, **`y`** is a variable, and E_k, **`k=1`**,**`m`**, are λ-expressions. An expression is in *weak head normal form* if it is either a functional abstraction **`λx.F`** or has the format $(\ldots((y\ E_1)\ldots E_m)$ where **`m≥0`**, **`y`** is a variable, and E_k, **`k=1`**,**`m`**, are λ-expressions.

The normal weak head form **`λx.F`** corresponds to a functional abstraction. Stopping the reduction (execution) of the body of a function before function application is ordinary practice in programming. For instance the reduction sequence:

`(λx.λy.(x y) λx.x)` $\longrightarrow$ **`λy.(λx.x y)`** $\longrightarrow$ **`λy.y`**

may stop with the weak head normal form **`λy.(λx.x y)`** or with the normal form **`λy.y`** depending on the convention we follow. When viewing Lambda calculus as a conventional programming language we may stop with the weak head normal form **`λy.(λx.x y)`**. When investigating theoretically the value of the expression we consider the normal form **`λy.y`**. Moreover, the (weak) head normal form **`(y E)`** may correspond to the application of a function designated by the free variable **`y`**. The entire expression may eventually reduce to a normal form even if **`E`** does not have a normal form. For instance, if **`y`** is **`λx.z`** then the application **`(λx.z Ω)`** reduces to **`z`**, where Ω is the λ-expression **`(λx.(x x) λx.(x x)))`**.

The three normal forms above are inclusive. The weak head normal form includes the head normal form which, in turn, includes the normal form. In what follows, we drop the prefix *weak head* and say that a functional abstraction **λx.E** is in *functional* normal form or, simply, in normal form.

A λ-expression may have several different reduction sequences. Some may terminate, others may not be finite. For example, the expression **(λx.y (λx.(x x) λx.(x x)))** has an infinity of reduction sequences. If we note by $\xrightarrow{1}$ a reduction step of the β-redex **(λx.y Ω)**, and by $\xrightarrow{2}$ a reduction step of Ω, then two possible reduction sequences of **(λx.y Ω)** are:

```
(λx.y  (λx.(x x) λx.(x x)))  —1→  y
(λx.y  (λx.(x x) λx.(x x)))  —2→  (λx.y  (λx.(x x)  λx.(x x)))  —2→  …
```

There are two types of reductions sequences: $\xrightarrow{2^n\ 1}{}^*$ with $n \geq 0$ and $\xrightarrow{2^\infty}{}^*$. Although the λ-expression **(λx.y Ω)** has a reduction sequence which does not terminate, it has the normal form **y**. Moreover, the length of the reduction sequences which terminate is not bounded.

Definition 9. If a λ-expression **E** has at least a terminating reduction sequence (which leads to a normal form) then **E** is *reducible*. If all the reduction sequences of **E** do not terminate then **E** is *irreducible*.

The expression **(λx.y Ω)** is reducible, whereas Ω is irreducible. Nevertheless, if an expression accepts several terminating reduction sequences is the normal form unique? In other words, the order of applying the terminating reduction steps is irrelevant and leads to a single result or it really matters and may possibly lead to different results.

Theorem 1. (Church-Rosser or the diamond theorem)

$$\forall E,E_1,E_2\in\Lambda \bullet (E \longrightarrow^* E_1 \wedge E \longrightarrow^* E_2) \Rightarrow (\exists E_3\in\Lambda \bullet E_1 \longrightarrow^* E_3 \wedge E_2 \longrightarrow^* E_3)$$

$$\begin{tikzcd}
 & E_1 \arrow[dr, "*"] & \\
E \arrow[ur, "*"] \arrow[dr, "*"'] & & E_3 \\
 & E_2 \arrow[ur, "*"'] &
\end{tikzcd}$$

If **E**, E_1 and E_2 are λ-expressions such that $E \longrightarrow^* E_1$ and $E \longrightarrow^* E_2$ then there is an expression E_3 such that $E_1 \longrightarrow^* E_3$ and $E_2 \longrightarrow^* E_3$.

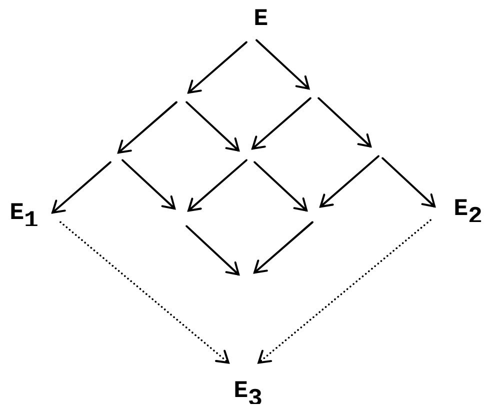
1. Consider **E'** a λ-expression such that there are two different redexes within **E'**: r_1 and r_2. Then, we can perform two reduction steps on **E'** in two different ways, ending with the same result (it can be proved easily according to the structure of the expression **E'**):

$$E' \xrightarrow{\text{reduce } r_1} E_a \xrightarrow{\text{reduce } r_2} E''$$
$$E' \xrightarrow{\text{reduce } r_2} E_b \xrightarrow{\text{reduce } r_1} E''$$

$$\begin{tikzcd}
 & E_a \arrow[dr] & \\
E' \arrow[ur] \arrow[dr] & & E'' \\
 & E_b \arrow[ur] &
\end{tikzcd}$$

Therefore, the diamond property of the one step reduction holds for any λ-expression **E'**.

2. If there are two different reduction sequences $E \longrightarrow^* E_1$ and $E \longrightarrow^* E_2$, ending with different results, then we can use repeatedly the diamond property of the one step reduction to pull together the intermediate results of the reduction sequences, as shown in the diagram. By



induction, it follows, that the results E_1 and E_2 can be pulled together in the same way.
■

As a corollary, if a λ-expression is reducible it has a unique normal form corresponding to a class of λ-expressions equivalent under systematic re-labeling. The *value* of **E** is represented by a conventionally labeled member from the class of the normal forms of **E**.

For instance, the λ-expression $E \equiv_{def}$ **(λx.λy.(x y) (λx.x y))** can be reduced according to two different reduction sequences.

1. **(λx.λy.(x y) (λx.x y))** $\longrightarrow$ **λz.((λx.x y) z)** $\longrightarrow$ **λz.(y z)** $\to_\alpha$ **λa.(y a)**
2. **(λx.λy.(x y) (λx.x y))** $\longrightarrow$ **(λx.λy.(x y) y)** $\longrightarrow$ **λw.(y w)** $\to_\alpha$ **λa.(y a)**

The normal forms **λz.(y z)** and **λw.(y w)** are equivalent under the re-labeling **{a/z,a/w}**. The value of the expression **E** is **λa.(y a)**. Moreover, the expressions **λz.((λx.x y) z)** and **(λx.λy.(x y) y)** are structurally equivalent since they lead to the same normal form **λa.(y a)**.

The *structural equivalence* (or β-equivalence) $E_1 = E_2$ of two λ-expressions, eventually irreducible, can be interpreted in the following way: there are at least two reduction sequences $E_1 \longrightarrow^* E'$ and $E_2 \longrightarrow^* E''$, and $E' \to_\alpha E''$. Notice that λ-expressions may not be equivalent in the sense above, although computationally they may behave in the same way. We have seen that the λ-expressions **λy.λx.(y x)** and **λx.x** do not have the same normal form, although they are computationally equivalent:

((λy.λx .(y x) a) b) $\longrightarrow^*$ **(a b)**
((λx .x a) b)) $\longrightarrow^*$ **(a b)**

The structural equivalence, based on the β-reduction and α-conversion (re-labeling), can be extended to the βη-*equivalence* by using the β-reduction, η-reduction and α-conversion of λ-expressions. From this point of view the expressions **λy.λx.(y x)** and **λx.x** are βη-equivalent. Generally speaking, if $E_1 \to^*_{\alpha\beta\eta} E_2$ or $E_2 \to^*_{\alpha\beta\eta} E_1$, where $\to^*_{\alpha\beta\eta}$ is a sequence of α, β and η reductions, then $E_1 =_{\beta\eta} E_2$.

Notice that equivalence does not imply the reducibility of λ-expressions, i.e. two expressions may be equivalent even if they are not reducible. For instance, consider the following expressions:

$E_1 \equiv_{def}$ **(λx.(f (x x)) λx.(f (x x)))**
$E_2 \equiv_{def}$ **(f (λx.(f (x x)) λx.(f (x x))))**

We have,

E_1 = **(λx.(f (x x)) λx.(f (x x)))** $\longrightarrow$ **(f (λx.(f (x x)) λx.(f (x x))))**
= E_2

Therefore, E_1 is structurally equivalent to E_2, since there is a reduction sequence that transforms E_1 into E_2. For a more detailed discussion on equivalence see [Thompson 1991].

Parameter evaluation

The possibility of reducing the same λ-expression in several different ways rises an additional question. If a λ-expression is reducible, how must the expression be reduced in order to obtain its value? In other words, how to compute safely the result of the expression.

Definition 10. Let **`E`** be a λ-expression and **`E'`** the outermost, leftmost β-*redex* of **`E`**. A reduction step focused on **`E'`** is a *left-to-right reduction step* of **`E`**. The *left-to-right reduction sequence* of **`E`** contains only left-to-right reduction steps. Obviously, each λ-expression has a unique *left-to-right reduction sequence*.

For the expression **`((λx.x λx.y) (λx.(x x) λx.(x x)))`** the *left-to-right reduction sequence* is:

$$(\underbrace{(\lambda x.x\ \ \lambda x.y)}_{1}\ (\lambda x.(x\ x)\ \lambda x.(x\ x))) \xrightarrow{1} \underbrace{(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))}_{2} \xrightarrow{2} y$$

Theorem 2. (The normalization theorem). The *left-to-right reduction sequence* of any reducible λ-expression terminates. The proof of the theorem can be found in [Barendregt 1984].

As a corollary to the theorem above, the computation of a λ-expression **`E`** must be performed by a *left-to-right reduction sequence*. If the expression is reducible, the reduction sequence will terminate. If the expression is irreducible the reduction will not terminate and also, any other reduction sequence will not terminate as well.

The *left-to-right reduction* corresponds to the *normal-order* evaluation and, considering how it is done, it is somewhat similar to the *call by name* parameter transfer (i.e. the unevaluated actual parameters replace all the free occurrences of the corresponding formal parameters within the function body). An occurrence of an actual parameter of a function is evaluated only when its value is needed during the execution of the function body. Therefore, the same actual parameter is evaluated each time its value is needed. Functions that do not evaluate their parameters when applied are *non-strict functions.* Examples are the **`if...else`** selection function and the **`&&,||`** operators from C.

Theoretically, the *left-to-right reduction* is a safe choice, although when used in conventional programming as a parameter transfer mode it may prove inefficient. Another, more "efficient" reduction alternative exists: the *right-to-left* reduction.

Definition 11. Let **`E`** be a λ-expression and **`E'`** the innermost, rightmost β-*redex* of **`E`**. A reduction step focused on **`E'`** is a *right-to-left reduction step* of **`E`**. The *right-to-left reduction sequence* of **`E`** contains only right-to-left reduction steps.

If for each β-*redex* **`(λx.F A)`** from a λ-expression, the expression **`A`** is reduced prior to the substitution $F_{[A/x]}$, and therefore the substitution is $F_{[(\text{value } A)/x]}$, then the reduction corresponds to the *applicative-order* evaluation. The parameter transfer *by value* and transfer *by sharing* are possible implementations of the *applicative-order* evaluation. Functions that evaluate their parameters when applied are *strict functions*.

Although the call by value is efficient, it has practical utility only when it is known that the reduction process (the evaluation) of the actual parameters always terminates. This is the case of conventional programming where functions are meant to produce a result when applied on sensible actual parameters, and where the responsibility to check whether parameters are right is delegated to the user. In conventional programming we do not usually work with parameters that stand for infinite objects or for non-terminating computations. If we'd like to work with apparently such weird objects we would have to use or simulate non-strict functions, as we will soon do in programming languages such as Scheme and Haskell and ML.

Notice also an important particularity of the Lambda calculus: a function application may produce another function. For example, **(λx.λy.x a)** $\longrightarrow$ **λy.a**. In this way **n**-ary functions are represented by $\lambda x_1.\lambda x_2.\ldots\lambda x_n.F$ and can be applied partially, on a number $m \le n$ of actual parameters.

$$(\ldots((\lambda x_1.\lambda x_2.\ldots\lambda x_n.F\ a_1)\ a_2)\ldots a_m) \longrightarrow^{*} \lambda x_{m+1}.\lambda x_{m+2}.\ldots\lambda x_n.F[a_1/x_1,\ldots,a_m/x_m]$$

Functions of this kind are called *curried*[2], in opposition to *uncurried* functions that must be applied on all of their actual parameters at once. From the programming perspective, curried functions imply that functions are treated as first-class values in the language and therefore can be computed and manipulated as any other values.

An axiomatic definition of Lambda calculus

The lax and textually scattered description of the Lambda calculus given in the previous sections can be pulled together to a formal specification. Besides coherence and precision the specification can be taken as the base for the implementation of a Lambda machine, able to compute with λ-expressions. In what follows an operational definition [Pierce 2002] of Lambda calculus is given.

An operational definition considers the calculus as a programming language, with a given syntax and semantics. The syntax assigns generic names to the basic constructs of the language and shows the ways these constructs are built. In the specification below the basic constructs of the Lambda calculus are variables, expressions[3] and values. The expressions are similar to the instructions of an expressional programming language, meaning that they compute values. In the specification, values are restricted to functional abstractions.

The semantics is given using reduction rules (or evaluation rules) and complete descriptions of the operations used in the rules, such as the substitution $e_{[e'/x]}$, where **e** and **e'** are expressions and **x** is a variable. A reduction rule is written

$$\frac{precondition_1,\ precondition_2,\ldots\ precondition_n}{e \longrightarrow e'}(rule\ id)$$

The rule reads: the expression **e** can be legally rewritten **e'** (or evaluates to **e'**) provided the preconditions $precondition_i$, **i=1,n**, are fulfilled. Usually, the preconditions specify themselves rewriting of expressions. For instance, considering that **e**, **e'** and **e"** are expressions, the rule

$$\frac{e \longrightarrow e'}{(e\ e") \longrightarrow (e'\ e")}(Eval_1)$$

reads: provided the expression **e** evaluates (reduces or can be rewritten) to **e'**, the application **(e e")** evaluates to **(e' e")**. The rules with no preconditions are axioms, such as $(\lambda x.e\ e') \longrightarrow e_{[e'/x]}$ **(Reduce)** that stands for the rule

[2] The name "curry" comes from Haskell Curry, a famous logician. There is also a functional programming language called Haskell.

[3] Some texts use the name **term** for **expression**.

$$\dfrac{}{(\lambda x.e\ e') \longrightarrow e_{[e'/x]}}\ \textbf{(Reduce)}$$

Specification 1: Lambda calculus with random-order evaluation.
(small step operational semantics)

Syntax	**Semantics**
Variable **Var::=** any symbol different than λ,**.**,**(**,**)** **Expression** **Expr ::=** **Var** **| λVar.Expr** **| (Expr Expr)** **Value** **Val ::= λVar.Expr**	**e,e',e"∈Expr; x,y∈Var** **Evaluation** $\dfrac{e \longrightarrow e'}{(e\ e'') \longrightarrow (e'\ e'')}$ **(Eval$_1$)** $\dfrac{e \longrightarrow e'}{(e''\ e) \longrightarrow (e''\ e')}$ **(Eval$_2$)** $\dfrac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'}$ **(Eval$_3$)** $(\lambda x.e\ e') \longrightarrow e_{[e'/x]}$ **(Reduce)**
	Substitution $x_{[e/x]} = e$ $y_{[e/x]} = y,\ x \neq y$ $(\lambda x.e)_{[e'/x]} = \lambda x.e$ $(\lambda y.e)_{[e'/x]} = \lambda y.e_{[e'/x]},\ x \neq y \wedge y \notin FV(e')$ $(e'\ e'')_{[e/x]} = (e'_{[e/x]}\ e''_{[e/x]})$
	Free variables $FV(x) = \{x\}$ $FV(\lambda x.e) = FV(e) \setminus \{x\}$ $FV(e'\ e'') = FV(e') \cup FV(e'')$

Specification 2: Lambda calculus with normal-order evaluation.
(small step operational semantics)

Syntax	**Semantics**
Variable **Var::=** any symbol different than λ,**.**,**(**,**)** **Expression** **Expr ::=** **Var** **| λVar.Expr** **| (Expr Expr)**	**e,e',e"∈Expr; x,y∈Var** **Evaluation** $\dfrac{e \longrightarrow e'}{(e\ e'') \longrightarrow (e'\ e'')}$ **(Eval)** $(\lambda x.e\ e') \longrightarrow e_{[e'/x]}$ **(Reduce)**
	Substitution **as in specification 1**

Value	Free variables
Val ::= λVar.Expr	**as in specification 1**

Specification 3: Lambda calculus with applicative-order evaluation.
(small step operational semantics)

Syntax	**Semantics**
Variable **Var::=** any symbol different than λ,.,(,) **Expression** **Expr ::=** **Var** **| λVar.Expr** **| (Expr Expr)** **Value** **Val ::= λVar.Expr**	**e,e',e"∈Expr; x,y∈Var; v∈Val** **Evaluation** $\dfrac{e \longrightarrow e'}{(e\ e") \longrightarrow (e'\ e")}$ **(Eval$_1$)** $\dfrac{e \longrightarrow e'}{(v\ e) \longrightarrow (v\ e')}$ **(Eval$_2$)** $(\lambda x.e\ v) \longrightarrow e_{[v/x]}$ **(Reduce)**
	Substitution **as in specification 1**
	Free variables **as in specification 1**

The operational semantics used in the specifications is "small step" and describe the individual steps used by a machine to evaluate an expression. For instance, consider the evaluation of the expression: **((λx.λy.y a) b)**, where **a,b∈Expr**, according to the specification (2):

$$\dfrac{(\lambda x.\lambda y.y\ a) \longrightarrow \lambda y.y\ \textbf{(Reduce)}\ \ldots\ldots\ldots\ldots\ \textbf{step 1}}{((\lambda x.\lambda y.y\ a)\ b) \longrightarrow (\lambda y.y\ b)}\textbf{(Eval}_2\textbf{)}\ \ldots\ldots\ \textbf{step 2}$$

$$(\lambda y.y\ b) \longrightarrow b\ \textbf{(Reduce)}\ \ldots\ldots\ldots\ldots\ldots\ldots\ \textbf{step 3}$$

Equivalently, we can describe the evaluation process linearly, as shown below, where the symbol o stands for rule composition:

$$((\lambda x.\lambda y.y\ a)\ b) \xrightarrow{\textbf{Eval}_2\ \circ\ \textbf{Reduce}} (\lambda y.y\ b) \xrightarrow{\textbf{Reduce}} b$$

The specifications above consider the Lambda Calculus as a programming language, where the aim is to compute meaningful values by applying meaningful operators (functions) on meaningful values. In the specifications a value is a function. The specification 1 is the closest to the Lambda Calculus, except for the restricted values: a value is a function. Specification 2, corresponds to a language where the parameters of functions are transferred by name and where the body of a function cannot be reduced prior of the function application[4], regardless whether the body contains β-redexes. Specification 3 is alike to specification 2, but the parameter transfer is by value.

[4] This kind of reduction is called weak reduction (or weak evaluation).

As far as the meaning of a value or of an operation is concerned, i.e. the type of the value or the signature of the function, it is not explicitly stated or enforced by any implicit typing mechanisms in the language. The expressions and values are typeless. The meaningfulness of values and operations is delegated entirely to the programmer.

For instance, it may happen that the same value stands for different objects of different types (e.g. the value **`λx.λx.λy.x`** could represent both the empty list and the natural number **`0`**). It is the responsibility of the programmer to process such a value using appropriate operators according to the meaning of the value in the current context of the computation. The meaning of the value and of the operation is in the mind of the programmer. In the language any operation can be applied on any value, regardless the meaning of both the operation and the value. Indeed, the specifications of the three "Lambda languages" make it possible to write expressions that do not reduce to a value (a function). Such expressions are "stuck" and signal programming errors. For instance, the expression **`(x λx.x)`** is not a value and cannot be reduced.

In spite of the mentioned flaws, the languages above may stand as the model of a class of functional programming languages with a lax typing discipline. These languages work with predefined types that are implicitly associated to values and the type checking is performed during the program execution. The application dependent types cannot be explicitly declared and, therefore, the checking of the program soundness is limited to the checking of the atomic, predefined, operations. The good point is that the primitive typing system of these languages enable for a convenient and rapid programming that suit applications where a data structure can have a meaning that is context dependent.

An example of such a language is Scheme. In Scheme a binary tree can be represented as a list **`(l k r)`**, where **`r`** is the left subtree, **`r`** is the right subtree, and **`k`** is the key of the root node of the tree. A set $\{k_1\ k_2...k_n\}$ can also be represented as the list $(k_1\ k_2...k_n)$. Therefore, in a program, the meaning of a list depends on the current computation context: it may stand for a tree, for a set, or for some other type of value. From the Scheme point of view, all lists are simply lists and can be processed by predefined list operators. Keeping the processing meaningful is entirely the task of the programmer. Nevertheless, for some applications, representing uniformly different values with different meanings as lists of symbols prove very convenient and highly simplify the programming. The price paid for the programming flexibility is the lower program stability and the higher difficulty of program validation.

References

Barendregt H.P., *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984
Thompson S., *Type Theory And Functional Programming*, Addison-Wesley, 1991
Pierce C.Benjamin. Types and Programming Languages, The MIT Press 2002