Lambda Calculus and Functional Programming

Anahid Bassiri[1] Mohammad Reza. Malek[2]
Pouria Amirian[3]

GJRE Classification (FOR) 080299, 010199, 010203, 010109

***Abstract*-The lambda calculus can be thought of as an idealized, minimalistic programming language. It is capable of expressing any algorithm, and it is this fact that makes the model of functional programming an important one. This paper is focused on introducing lambda calculus and its application. As an application dikjestra algorithm is implemented using lambda calculus. As program shows algorithm is more understandable using lambda calculus in comparison with other imperative languages.**

I. INTRODUCTION

Lambda calculus (λ-calculus) is a useful device to make the theories realizable. Lambda calculus, introduced by Alonzo Church and Stephen Cole Kleene in the 1930s is a formal system designed to investigate function definition, function application and recursion in mathematical logic and computer science. It has emerged as a useful tool in the investigation of problems in computability or recursion theory, and forms the basis of a paradigm of computer programming called functional programming.

As lambda calculus is capable of expressing any algorithm the lambda calculus can be thought of as an idealized, minimalistic programming language. Based on these capabilities lambda calculus became an important model of functional programming. Functional programs are stateless and deal exclusively with functions that accept and return data (including other functions), but they produce no side effects in 'state' and thus make no alterations to incoming data. Modern functional languages, building on the lambda calculus, include Erlang, Haskell, Lisp, ML, Scheme and Microsoft has in the past couple years has turned its attention towards functional programming with introduction of .NET based functional programming language called F#.(ref1)

The lambda calculus continues to play an important role in mathematical foundations, through the Curry-Howard correspondence.(ref1)

Church (1936) invented a formal system called the lambda calculus and defined the notion of computable function via this system. Turing (1936, 1937) invented a class of machines (later to be called Turing machines) and defined the notion of computable function via these machines. In 1936 Turing proved that both models are equally strong in the sense that they define the same class of computable functions.

*About*GIS Department, Faculty Of Geodesy And Geomatic Eng., K.N.Toosi University Of Technology, Mirdamad Cross, Valiasr St., Tehran, IRAN*
About- (e-mail;[1]Anahid_bassiri1984@yahoo.com)
About- (e-mail;[2]mrmalek@kntu.ac.ir)
About-(e-mail;[3]Pouria.amirian@gmail.com)

Basis concept of a Turing machine is the present day Von Neumann computers. Conceptually these are Turing machines with random access registers. Imperative programming languages such as FORTRAN, Pascal etcetera as well as all the assembler languages are based on the way a Turing machine is instructed by a sequence of statements.

In addition functional programming languages, like Miranda, ML etcetera, are based on the lambda calculus. Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state.

Lambda calculus provides a theoretical framework for describing functions and their evaluation. Though it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today. Modern functional languages can be viewed as embellishments to the lambda calculus. (ref2)

In the next section first we introduce functional programming and after that functional and non-functional programming are compared.

II. FUNCTIONAL PROGRAMMING

Functional programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development. However, notable functional programming languages used in industry and commercial applications include Erlang, OCaml, Haskell, Scheme (since 1986) and domain-specific programming languages like R (statistics), Mathematica (symbolic math), J and K (financial analysis), and XSLT (XML).

Many non-functional programming languages such as C, C++ and C# can be made to exhibit functional behaviors using function pointers, the <functional> library and lambda functions respectively.

A functional program consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules.

Reduction consists of replacing a part P of E by another expression P' according to the given rewrite rules. In schematic notation

$$E[P] \rightarrow E[P']$$

Provided that P -> P' is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called normal form E* of the expression E consists of the output of the given functional program.

Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. Recursion may require maintaining a stack, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme programming language standard requires implementations to recognize and optimize tail recursion.

Functional languages can be categorized by whether they use strict or non-strict evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated.

In brief, strict evaluation always fully evaluates function arguments before invoking the function. Non-strict evaluation is free to do otherwise

To illustrate, consider the following two functions f and g:

f(x) := x^2 + x + 1

g(x, y) := x + y

Under strict evaluation, we would have to evaluate function arguments first, for example:

f(g(1, 4))
= f(1 + 4)
= f(5)
= 5^2 + 5 + 1
= 31

By contrast, non-strict evaluation need not fully evaluate the arguments; in particular it may send the arguments unevaluated to the function, perhaps evaluating them later. For example, one non-strict strategy (call-by-name) might work as follows:

f(g(1, 4))
= g(1, 4)^2 + g(1, 4) + 1
= (1 + 4)^2 + (1 + 4) + 1
= 5^2 + 5 + 1
= 31

A key property of strict evaluation is that when an argument expression fails to terminate, the whole expression fails to terminate. With non-strict evaluation, this need not be the case, since argument expressions need not be evaluated at all.

Advantages of strict-evaluation can be categorized into two categories as it denoted in below:

Parameters are usually passed around as (simple) atomic units, rather than as (rich) expressions. (For example, the integer 5 can be passed on a register, whereas the expression 1+4 will require several memory locations). This has a direct implementation on standard hardware.

The order of evaluation is quite clear to the programmer: every argument must be evaluated before the function body is invoked.

Advantages of non-strict-evaluation can be categorized into three categories as it denoted in below:

Lambda calculus provides a stronger theoretic foundation for languages that employ non-strict evaluation.

A non-strict evaluator may recognize that a sub-expression does not need to be evaluated. For example, given the definitions

Multiply (0, x) = 0;
Multiply (n, x) = x + multiply (n-1, x);
F (0) = 1;
F (n) = n * f (n-1);

Multiply (0, f (1000000)) a strict evaluator would (strictly speaking) need to take (on the order of) 1,000,000 steps to find the value of f (1000000). A non-strict evaluator may use the definition of multiply first, reducing the whole expression to 0 before even trying to compute f (1000000).

- Non-strict evaluation can use the above to allow "infinite" data structures.

III. Comparison of Functional and Imperative Programming

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming disallows side effects completely. Disallowing side effects provides for referential transparency, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks. This means that pure functions have several useful properties, many of which can be used to optimize the code:

- If the result of a pure expression is not used, it can be removed without affecting other expressions.
- If a pure function is called with parameters that cause no side-effects, the result is constant with respect to that parameter list (sometimes called referential transparency), i.e. if the pure function is again called with the same parameters, the same result will be returned (this can enable caching optimizations).

If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe).

- If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program (for example, using lazy evaluation).

While most compilers for imperative programming languages detect pure functions, and perform common-subexpression elimination for pure function calls, they cannot always do this for pre-compiled libraries, which generally do not expose this information, thus preventing optimizations that involve those external functions.

Higher order functions are rarely used in older imperative programming. Where a traditional imperative program might use a loop to traverse a list, a functional style would often use a higher-order function, map, that takes as

arguments a function and a list, applies the function to each element of the list, and returns a list of the results.

IV. LAMBDA CALCULUS

The **λ**-calculus can be called the smallest universal programming language of the world. The **λ**-calculus consists of a single transformation rule (variable substitution) and a single function definition scheme. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. The **λ**-calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines. However, the **λ**-calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

V. FORMAL LAMBDA CALCULUS

The central concept in **λ** -calculus is the "expression". A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters a; b; c,…. An expression is defined recursively as follows:

```
<expression>  :=  <name> | <function> | <application>
  <function>  :=  λ <name>.<expression>
<application> :=  <expression><expression>
```

An expression can be surrounded with parenthesis for clarity, that is, if E is an expression, (E) is the same expression. The only keywords used in the language are **λ** and the dot. In order to avoid cluttering expressions with parenthesis, we adopt the convention that function application associates from the left, that is, the expression
$E_1E_2E_3 \ldots E_n$
is evaluated applying the expressions as follows:
$(\ldots ((E_1E_2)\ E3)\ldots E_n)$
As can be seen from the definition of **λ** expressions given above, a single identifier is a **λ** expression. An example of a function is the following:
λ x.x
For instance, the "add-two" function *f* such that $f(x) = x + 2$ would be expressed in lambda calculus as $\lambda\ x.\ x + 2$ (or equivalently as $\lambda\ y.\ y + 2$; the name of the formal parameter is immaterial) and the application of the function $f(3)$ would be written as $(\lambda\ x.\ x + 2)\ 3$. Note that part of what makes this description "informal" is that the expression $x + 2$ (or even the number 2) is not part of lambda calculus; an explanation of how numbers and arithmetic can be represented in lambda calculus is below. Function application is left associative: f x y = (f x) y. Consider the function which takes a function as an argument and applies it to the number 3 as follows: $\lambda\ f.\ f\ 3$. This latter function could be applied to our earlier "add-two" function as follows: $(\lambda\ f.\ f\ 3)\ (\lambda\ x.\ x + 2)$. The three expressions:

$(\lambda\ f.\ f\ 3)\ (\lambda\ x.\ x + 2)$
$(\lambda\ x.\ x + 2)\ 3$
$3 + 2$

are equivalent.
A function of two variables is expressed in lambda calculus as a function of one argument which returns a function of one argument. For instance, the function f(x, y) = x - y would be written as λ x. λ y. x - y. A common convention is to abbreviate curried functions as, in this example, λ x y. x - y. While it is not part of the formal definition of the language,
λ x1 x2 … xn. Expression
Is used as an abbreviation for
λ x1. λ x2. … λ xn. Expression
Not every lambda expression can be reduced to a definite value like the ones above; consider for instance
(λ x. x x) (λ x. x x)
or
(λ x. x x x) (λ x. x x x)
and try to visualize what happens when you start to apply the first function to its argument. (λ x. x x) is also known as the ω combinator; ((λ x. x x) (λ x. x x)) is known as Ω, ((λ x. x x x) (λ x. x x x)) as Ω2, etc.
Lambda calculus expressions may contain free variables, i.e. variables not bound by any λ. For example, the variable y is free in the expression (λ x. y), representing a function which always produces the result y. occasionally, this necessitates the renaming of formal arguments. For example, in the formula below, the letter y is used first as a formal parameter, then as a free variable:
(λ x y. y x) (λ x. y).
To reduce the expression, we rename the first identifier z so that the reduction does not mix up the names:
(λ x z. z x) (λ x. y)
the reduction is then
λ z. z (λ x. y).
If one only formalizes the notion of function application and replaces the use of lambda expressions by the use of combinators, one obtains combinatory logic.

A. Definition

Lambda expressions are composed of

- Variables $v_1, v_2, \ldots v_n$
- The abstraction symbols λ
- Parentheses ( )

The set of lambda expressions, Λ, can be defined recursively:

1. If x is a variable, then x ∈ Λ
2. If x is a variable and M ∈ Λ, then (λ x . M) ∈ Λ
3. If M, N ∈ Λ, then (M N) ∈ Λ

Instances of 2 are known as abstractions and instances of 3, applications.

B. Notation

To keep the notation of lambda expressions uncluttered, the following conventions are usually applied.
Outermost parentheses are dropped: M N instead of (M N).
Applications are assumed to be left associative: M N P means (M N) P.
The body of an abstraction extends as far right as possible: λ x. M N means (λ x.M N) and not (λ x. M) N

A sequence of abstractions are contracted: λ x λ y λ z. N is abbreviated as λ x y z. N

C. Free and bound variables

The abstraction operator, λ, is said to bind its variable wherever it occurs in the body of the abstraction. Variables that fall within the scope of a lambda are said to be bound. All other variables are called free. For example in the following expression y is a bound variable and x is free:
λ y . xxy
Also note that a variable binds to its "nearest" lambda. In the following expression one single occurrence of x is bound by the second lambda:
λ x . y (λ x . z x)
The set of *free variables* of a lambda expression, M, is denoted as FV (M) and is defined by recursion on the structure of the terms, as follows:
FV(x) = {x}, where x is a variable
FV (λ x . M) = FV (M) - {x}

FV (M N) = FV (M) FV (N)
An expression which contains no free variables is said to be closed. Closed lambda expressions are also known as combinators and are equivalent to terms in combinatory logic.

VI. Reduction

A-conversion

Alpha conversion allows bound variable names to be changed. For example, an alpha conversion of λx.x would be λy.y . Frequently in uses of lambda calculus, terms that differ only by alpha conversion are considered to be equivalent.
The precise rules for alpha conversion are not completely trivial. First, when alpha-converting abstractions, the only variable occurrences that are renamed are those that are bound to the same abstraction. For example, an alpha conversion of λx.λx.x could result in λy.λx.x , but it could *not* result in λy.λx.y . The latter has a different meaning from the original.
Second, alpha conversion is not possible if it would result in a variable getting captured by a different abstraction. For example, if we replace x with y in λx.λy.x, we get λy.λy.y, which is not at all the same.

A. Substitution

Substitution, written E[V := E'], corresponds to the replacement of a variable V by expression E' every place it is free within E. The precise definition must be careful in order to avoid accidental variable capture. For example, it is not correct for (λ x.y)[y := x] to result in (λ x.x), because the substituted x was supposed to be free but ended up being bound. The correct substitution in this case is (λ z.x), up-to α-equivalence.
Substitution on terms of the λ-calculus is defined by recursion on the structure of terms, as follows.
x[x := N] ≡ N
y[x := N] ≡ y, if x ≠ y
$(M_1\ M_2)[x := N] \equiv (M_1[x := N])\ (M_2[x := N])$
(λ y. M)[x := N] ≡ λ y. (M[x := N]), if x ≠ y and y∉fv(N)
Notice that substitution is defined uniquely up-to α-equivalence.
β-reduction
Beta reduction expresses the idea of function application. The beta reduction of ((λ V. E) E') is simply E[V := E'] .

H-conversion

Eta conversion expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Eta-conversion converts between λ x. f x and f whenever x does not appear free in f.
This conversion is not always appropriate when lambda expressions are interpreted as programs. Evaluation of λ x. f x can terminate even when evaluation of *f* does not.

VII. Arithmetic in Lambda Calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:
0:= λ f x. x
1:= λ f x. f x
2:= λ f x. f (f x)
3:= λ f x. f (f (f x))
And so on.
A Church numeral is a higher-order function—it takes a single-argument function f, and returns another single-argument function. The Church numeral n is a function that takes a function f as argument and returns the n-th composition of f, i.e. the function f composed with itself n times. This is denoted f(n) and is in fact the n-th power of f (considered as an operator); f(0) is defined to be the identity function. Such repeated compositions (of a single function f) obey the laws of exponents, which is why these numerals can be used for arithmetic. Note that 1 returns f itself, i.e. it is essentially the identity function, and 0 returns the identity function. (Also note that in Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of 0 impossible.)
We can define a successor function, which takes a number n and returns $n + 1$ by adding an additional application of f:
SUCC: = λ n f x. f (n f x)
Because the m-th composition of f composed with the n-th composition of f gives the $m+n$-th composition of f, addition can be defined as follows:
PLUS: = λ m n f x. n f (m f x)
PLUS can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that
PLUS 2 3 and 5

Are equivalent lambda expressions. Since adding m to a number, n can be accomplished by adding 1 m times, an equivalent definition is:
PLUS: = λ n m. m SUCC n
Similarly, multiplication can be defined as
MULT: = λ m n f . m (n f)
Alternatively
MULT: = λ m n. m (PLUS n) 0,
Since multiplying m and n is the same as repeating the "add n" function m times and then applying it to zero. The predecessor function defined by PRED n = n - 1 for a positive integer n and PRED 0 = 0 is considerably more difficult. The formula
`PRED: = λ n f x. n (λ g h. h (g f)) (λ u. x) (λ u. u)`
Can be validated by showing inductively that if `T` denotes `(λ g h. h (g f))`, then $T^{(n)}(\lambda u.\ x) = (\lambda h.\ h(f^{(n-1)}(x))\ )$ for n > 0. Two other definitions of PRED are given below, one using conditionals and the other using pairs. With the predecessor function, subtraction is straightforward. Defining
`SUB: = λ m n. n PRED m`,
`SUB m n` yields $m - n$ when $m > n$ and `0` otherwise.

VIII. Logic and Predicates

By convention, the following two definitions (known as Church booleans) are used for the boolean values `TRUE` and `FALSE`:
`TRUE: = λ x y. x`
`FALSE: = λ x y. y`
(Note that `FALSE` is equivalent to the Church numeral zero defined above)
Then, with these two λ-terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):
`AND: = λ p q. p q p`
`OR: = λ p q. p p q`
`NOT: = λ p. λ a b. p b a`
`IFTHENELSE: = λ p a b. p a b`
We are now able to compute some logic functions, for example:
`AND TRUE FALSE`
≡ `(λ p q. p q p) TRUE FALSE` $\rightarrow_\beta$ `TRUE FALSE TRUE`
≡ `(λ x y. x) FALSE TRUE` $\rightarrow_\beta$ `FALSE`
and we see that `AND TRUE FALSE` is equivalent to `FALSE`.
A *predicate* is a function which returns a boolean value. The most fundamental predicate is `ISZERO` which returns `TRUE` if its argument is the Church numeral `0`, and `FALSE` if its argument is any other Church numeral:
`ISZERO: = λ n. n (λ x. FALSE) TRUE`
The following predicate tests whether the first argument is less-than-or-equal-to the second:
`LEQ:= λ m n. ISZERO (SUB m n)`,
and since $m = n$ iff `LEQ m n` and `LEQ n m`, it is straightforward to build a predicate for numerical equality.

The availability of predicates and the above definition of `TRUE` and `FALSE` make it convenient to write "if-then-else" expressions in lambda calculus. For example, the predecessor function can be defined as' '
`PRED: = λ n. n (λ g k. ISZERO (g 1) k (PLUS (g k) 1)) (λ v. 0) 0`
Which can be verified by showing inductively that `n (λ g k. ISZERO (g 1) k (PLUS (g k) 1)) (λ v. 0)` is the "add n - 1" function for $n > 0$.

IX. Pairs

A pair (2-tuple) can be defined in terms of `TRUE` and `FALSE`, by using the Church encoding for pairs. For example, `PAIR` encapsulates the pair (x,y), `FIRST` returns the first element of the pair, and `SECOND` returns the second.
PAIR:= λ x y f. f x y
FIRST: = λ p. p TRUE
SECOND: = λ p. p FALSE
NIL: = λ x. TRUE
NULL: = λp. p (λx y.FALSE)
A linked list can be defined as either NIL for the empty list, or the `PAIR` of an element and a smaller list. The predicate `NULL` tests for the value `NIL`.
As an example of the use of pairs, the shift-and-increment function that maps (m, n) to $(n, n+1)$ can be defined as
`Φ := λ x. PAIR (SECOND x) (SUCC (SECOND x))`
which allows us to give perhaps the most transparent version of the predecessor function:
`PRED := λ n. FIRST (n Φ (PAIR 0 0))`

X. Recursion

Recursion is the definition of a function using the function itself; on the face of it, lambda calculus does not allow this. However, this impression is misleading. Consider for instance the factorial function $f(n)$ recursively defined by
$f(n) = 1$, if $n = 0$; and $n \cdot f(n-1)$, if $n>0$.
In lambda calculus, one cannot define a function which includes itself. To get around this, one may start by defining a function, here called g, which takes a function f as an argument and returns another function that takes n as an argument:
`g := λ f n. (1, if n = 0; and n·f(n-1), if n>0).`
The function that g returns is either the constant `1`, or n times the application of the function f to $n-1$. Using the `ISZERO` predicate, and boolean and algebraic definitions described above, the function g can be defined in lambda calculus.
However, g by itself is still not recursive; in order to use g to create the recursive factorial function, the function passed to g as f must have specific properties. Namely, the function passed as f must expand to the function g called with one argument -- and that argument must be the function that was passed as f again!

In other words, `f` must expand to `g(f)`. This call to `g` will then expand to the above factorial function and calculate down to another level of recursion. In that expansion the function `f` will appear again, and will again expand to `g(f)` and continue the recursion. This kind of function, where `f = g(f)`, is called a *fixed-point* of `g`, and it turns out that it can be implemented in the lambda calculus using what is known as the *paradoxical operator* or *fixed-point operator* and is represented as `Y` -- the Y combinator:

`Y = λ g. (λ x. g (x x)) (λ x. g (x x))`

In the lambda calculus, `Y g` is a fixed-point of `g`, as it expands to `g (Y g)`. Now, to complete our recursive call to the factorial function, we would simply call `g (Y g) n`, where *n* is the number we are calculating the factorial of.

Given *n* = 5, for example, this expands to:
(λ n.(1, if n = 0; and n·((Y g)(n-1)), if n>0)) 5
1, if 5 = 0; and 5·(g(Y g)(5-1)), if 5>0
5·(g(Y g) 4)
5·(λ n. (1, if n = 0; and n·((Y g)(n-1)), if n>0) 4)
5·(1, if 4 = 0; and 4·(g(Y g)(4-1)), if 4>0)
5·(4·(g(Y g) 3))
5·(4·(λ n. (1, if n = 0; and n·((Y g)(n-1)), if n>0) 3))
5·(4·(1, if 3 = 0; and 3·(g(Y g)(3-1)), if 3>0))
5·(4·(3·(g(Y g) 2)))

And so on, evaluating the structure of the algorithm recursively. Every recursively defined function can be seen as a fixed point of some other suitable function, and therefore, using `Y`, every recursively defined function can be expressed as a lambda expression. In particular, we can now clearly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

XI. Implementation and Application

The strength of the lambda-calculus is that it is easily used as "glue" on top of a richer world of primitives. Its advantages as glue are that it has a natural correspondence with the way that people program, and natural compilation techniques yield high-performance code.

There are software engineering advantages to a language glued together with lambda-calculus. Lambda expressions can be understood locally - their dependence on their environment is entirely through their free variables. Lambda expressions tend to have fewer free variables and more bound variables than comparable imperative code, since they do not rely as heavily on assignment to express the computation. An imperative program proceeds by altering some globally-accessible store of values. By contrast, a functional program proceeds by function application and the return of values. This eliminates large classes of errors associated with maintaining a global store of values.

Based on these advantages we are interested in implementing some algorithm by lambda calculus as a programming language. In this regards, djikstra as an algorithm to find the shortest path between two nodes in a graph is implemented. Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non negative edge path costs, outputting a shortest path tree. This algorithm is often used in routing.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols.

Let's call the node we are starting with an initial node. Let a distance of a node X be the distance from the initial node to it. Our algorithm will assign some initial distance values and will try to improve them step-by-step.

1. Assign every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes as unvisited. Set initial node as current.
3. For current node, consider all its unvisited neighbors and calculate their distance (from the initial node) in case they are reached through the current node. For example, if current node (A) has distance of 6, and an edge connecting it with another node (B) is 2, the distance to B through A will be 6+2=8. If this distance is less than the previously recorded distance (infinity in the beginning, zero for the initial node), overwrite the distance.
4. When we are done visiting all neighbors of the current node, mark it as visited. A visited node will not be checked ever again, its distance recorded now is final and minimal.
5. Set the nearest unvisited neighbor of the current node as the next "current node" and continue from step 3.
6. When all nodes are visited, algorithm ends.

The program using lambda calculus languages became as illustrated in below:

λ n. n (λ x. λ x y. y) λ x y. x
Weight Node. λ n. n (λ x. λ x y. y) λ x y. x Weight Node
λ n f x. f (n f x)
λ m n. λ n. n (λ x. λ x y. y) λ x y. x (λ m n. n λ n. n (λ g k. λ n. n (λ x. λ x y. y)
λ x y. x (g λ f x. f x) k (λ m n f x. n f (m f x) (g k) λ f x. f x))
(λ v. λ f x. x) λ f x. x m n) Weight Node. m n. λ n. n (λ x. λ x y. y) λ x y. x (λ m n. n λ n. n (λ g k. λ n. n (λ x. λ x y. y)
λ x y. x (g λ f x. f x) k (λ m n f x. n f (m f x) (g k) λ f x. f x)) (λ v. λ f x. x) λ f x. x m n) Weight Node y f. f λ m n. λ n. n (λ x. λ x y. y)
λ x y. x (λ m n. n λ n. n (λ g k. λ n. n (λ x. λ x y. y) λ x y. x (g 1) k (λ m n f x. n f (m f x) (g k) λ f x. f x)) (λ v. λ f x. x) λ f x. x

m n) Weight Node. m n. λ n. n (λ x. λ x y. y) λ x y. x (λ m n. n λ n. n (λ g k. λ n. n (λ x. λ x y. y)
λ x y. x (g 1) k (λ m n f x. n f (m f x) (g k) λ f x. f x)) (λ v. λ f x. x) λ f x. x m n) Weight Node y

In comparison with the other programming languages, as an example we wrote some of them in below, Lambda expressions tend to have fewer free variables and more bound variables than comparable imperative code. In addition it is easier to understand the algorithm in lambda-based program because our understandings are not limited to the variable definition.

XII. Python Implementation

```
Import heapq
From collections import defaultdict

class Edge(object):
  def __init__(self, u, v, weight):
    self.start, self.end, self.weight = u, v, weight

  # For heapq.
  def __cmp__(self, other): return cmp(self.weight, other.weight)

class Graph(object):
  def __init__(self):
    # The adjacency list.
    self.adj = defaultdict(list)

  def add_e(self, u, v, weight = 0):
    self.adj[u].append (Edge (u, v, weight))

  def s_path(self, src):
    """
    Returns the distance to every vertex from the source and the
    array representing, at index i, the node visited before
    visiting node i. This is in the form (dist, previous).
    """
    dist, visited, previous, queue = {src: 0}, {}, {}, []
    heapq.heappush(queue, src)
    while len(queue) > 0:
      current = heapq.heappop(queue)
      if current in visited:
        Continue
      visited[current] = True

      for edge in self.adj[current]:
        relaxed = dist[current] + edge.weight
        v = edge.end
        if v not in dist or relaxed < dist[v]:
          previous[v], dist[v] = current, relaxed
          heapq.heappush(queue, v)
    return dist, previous

g = Graph()
g.add_e(1,2,4)
g.add_e(1,4,1)
g.add_e(2,1,74)
g.add_e(2,3,2)
g.add_e(2,5,12)
g.add_e(3,2,12)
g.add_e(3,10,12)
g.add_e(3,6,74)
g.add_e(4,7,22)
g.add_e(4,5,32)
g.add_e(5,8,33)
g.add_e(5,4,66)
g.add_e(5,6,76)
g.add_e(6,10,21)
g.add_e(6,9,11)
g.add_e(7,3,12)
g.add_e(7,8,10)
g.add_e(8,7,2)
g.add_e(8,9,72)
g.add_e(9,10,7)
g.add_e(9,6,31)
g.add_e(9,8,18)
g.add_e(10,6,8)

# Find a shortest path from vertex 'a' (1) to 'j' (10).
Dist, prev = g.s_path(1)
# Trace the path back using the prev array.
Path, current, end = [], 10, 10
While current in prev:
  path.insert(0, prev[current])
  current = prev[current]

print path
print dist[end]
```

as it is clear, the program written first is much easier to develop because the developer is not supposed to know syntaxes. Although lambda calculus is easy, it is not a user friendly language which should be like human's language.

XIII. CONCLUSION

The strength of the lambda-calculus is that it is easy to use and in order to implement, you are not supposed to learn a huge amount of syntaxes. In comparison to other, although lambda calculus is easy, it is not a user friendly language which should be like human's language.