USER MANUAL

LAMBDA-CALCULUS INTERPRETER

AUTHOR: Víctor Nathanael Badillo Aldama

Introduction:

This user manual purpose is to ilustrate the user with some examples showing the new developed functionalities and a brief explanation of the necessary changes for developing the final code.

Table of Contents:

1.1 Multi-Line Expression Recognition

To allow the user to input multiline inputs, there are mainly two solutions: manipulating the string typed by the user or introducing new tokens in the lexical analyzer and new grammatical rules in the syntactic analyzer. In this implementation, the second option was chosen.

First, a new token DOUBLE_SEMICOLON was created to represent double semicolons. Then, in the syntactic analyzer, this token was added at the end of each rule of the axiom so that every input must end with double semicolons. In the main function, a new function was created to continuously read the input until the line contains a semicolon (;). This is sufficient as a check since semicolons are not used for any other type of input in this interpreter, so their presence indicates the end of a line. If the user accidentally types just a semicolon, a syntax error will be displayed, as explained earlier.

This solution prevents situations like one expression ended up by double semicolons and a following expression without fishing token which could cause problems in other kind of implementations.

The implementation also allows the user to enter ";;" without any relevant information. If this happens, the input loop is called again.

Example:

```
let
  a = 11
in
  pred a
;;
```

1.2 Pretty-printer.

The pretty-printer allows for a much more readable output. First, depending on the command to be executed, a corresponding output is generated for each command, and instead of storing the output in a string, it is printed directly to the screen.

The implementation was done by creating a cascade of functions that call each other depending on the term to be printed. For terms, the surrounding parentheses have been removed, keeping only those that enclose non-atomic terms. Besides achieving the main goal of removing unnecessary parentheses, the output has been organized with good indentation to make the code as readable as possible.

Regarding the code, the changes required for implementation are straightforward. To begin with, the string_of_ty and string_of_term functions are no longerneeded in the main module, so their signatures have been removed from the .mli file. This pretty-printer function is called for evaluations and bindings (this new functionalities will be discused later in a few sections).It divides terms into atomic terms, applications, and others, and applies appropriate indentation based on the type of term being printed. This indentation is achieved using the boxes from OCaml's Format module.

In addition to removing unnecessary parentheses from terms, parentheses have also been removed from types.

Examples:

lambda x : Nat.x;;

- : Nat -> Nat = lambda x : Nat. x

letrec sum : Nat -> Nat -> Nat =

lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in

sum ;;

- : Nat -> Nat -> Nat =

lambda n : Nat.

```
  lambda m : Nat.
  if iszero n then m
  else (succ (fix (lambda sum : Nat -> Nat -> Nat.
            lambda n : Nat.
            lambda m : Nat.
            if iszero n then m else (succ (sum (pred n) m)))
        (pred n) m))
```

2.1 Fixed-Point Combinator

Previously, writing functions with recursion required an inconvenient syntax that was not suitable for this interpreter due to type specifications.

With this new functionality, we can move from this:

```
let fix = lambda f.(lambda x. f (lambda y. x x y)) (lambda x. f (lambda y. x x y)) in
let sumaux =
  lambda f. (lambda n. (lambda m. if (iszero n) then m else succ (f (pred n) m))) in
let sum = fix sumaux in
sum 55 45
```

to a more convenient syntax:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in
sum 55 45
```

As can be observed in the new syntax, a new token, letrec, has been introduced, which is used to define recursive functions. This new functionality requires changes in all the main files (lexer.mll, parser.mly, lambda.ml, and lambda.mli). Simply defining letrec is not enough; the fix operator has also been implemented to achieve the goal of self-referential functions. The outputs provided by this input should also be valid outputs. An example of an output used as input that returns the corresponding value of executing the operation:

```
(lambda n : Nat.
  lambda m : Nat.
  if iszero n then m
  else (succ (fix (lambda sum : Nat -> Nat -> Nat.
          lambda n : Nat.
          lambda m : Nat.
          if iszero n then m else (succ (sum (pred n) m)))
       (pred n) m))) 55 45;;
```

The necessary changes in parser.mly involve capturing letrec, but it internally works by using fix, and includes fix to recognize and enable examples like the one above to work. The following required changes pertain to lambda.ml, modifying the functions typeof, pretty_printer, free_vars, subst, and eval1.

Examples:

Multiplication:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  letrec prod : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero m then 0 else sum n (prod n (pred m))
```

```
  in
    prod 10 50
;;
```

Fibonacci:

```
letrec sum : Nat -> Nat -> Nat =
    lambda n: Nat. lambda m : Nat. if iszero n then m
        else succ (sum (pred n) m) in
        letrec fib: Nat -> Nat =
            lambda n : Nat. if iszero n then 0 else if iszero (pred n) then 1
                else sum(fib (pred (pred n))) (fib (pred n)) in
        fib 7
;;
```

Factorial:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m
    else succ (sum (pred n) m) in
    letrec prod : Nat -> Nat -> Nat =
      lambda n: Nat. lambda m : Nat. if iszero n then 0
        else sum (prod (pred n) m) m in
        letrec fac: Nat -> Nat =
          lambda n : Nat. if iszero n then 1
            else prod n (fac (pred n))
        in fac 5
;;
```

2.2 Global Definitions Context

In this section, the code has been modified to allow free variable names to be associated with values or terms. These can later be used in subsequent lambda expressions.

The developed syntax is as follows:

identifier = term

Example:

x = 9

id = lambda x : Nat. succ x;;

id x is a valid expression and returns 10.

In addition to associating free variables with terms, the possibility of creating type definitions or aliases has been included, following this syntax:

Identifier = types

Example:

N = Bool;;

After doing this, the expression lambda x: N. x must be valid, and it should have the type Bool -> Bool. It is important to note that it returns Bool -> Bool and not N -> N. This will be explained later.

This global context of definitions can be created using either an imperative context or a functional context. For a lambda calculus interpreter, it is more appropriate to implement it using a functional context.

Therefore, it has been implemented following this approach.

Previously, we discussed command types in the pretty-printer. This refers to the first change made to handle the global definitions context.In the main.ml, a new function, execute, is called to process the command entered by the user within the current context. The context has been updated from being a pair of string * ty to string * binding, where binding is a new type created to specify whether it is a type or a type with a term.

There are four general command types: evaluation, value or term binding, type binding, and quit, which is used to exit. Corresponding functions have been created to add and retrieve identifier values from the context. These functions are split between those dedicated to variables and those for type variables since there is a single context storing everything. This approach is both more convenient and simpler.

Type identifiers start with an uppercase letter, so we need to be able to recognize these identifiers. To achieve this, a new token, IDT, was created in lexer.mll specifically for this type of identifier.

In parser.mly, apart from including the new IDT token, which returns a string, it is important to note that the returned type is no longer a Lambda.term but a Lambda.command.Naturally, the previously mentioned syntax must be included, so the rule s now incorporates the four types of commands.

In lambda.mli, the new signatures and types mentioned are included, and their implementations are provided in lambda.ml. TyVar is used in this global context for the type of identifiers. In code sections that handle variables, such as letin, the new identifiers being defined are added to the context. Additionally, the case of a variable term is included in the evaluation process to retrieve its value from the context.

apply_ctx is another neccessary function for this section, performing a substitution on the term tm for all free variables using the values associated with those variables in the context ctx when there are not more evaluation rules for one term.

For simplicity, it was decided to store the basic type of type identifiers to make tasks like type checking easier. Additionally, the basic types are printed instead of the identifiers. To obtain the base type, the typeofTy function is used. This function works recursively to determine the base type from the given type.

Examples:

x = 5;;

x : Nat = 5

pred x;;

- : Nat = 4

f = lambda y : Nat. x;;

f : Nat -> Nat = lambda y : Nat. 5

f 3;;

- : Nat = 5

x = 7;;

x : Nat = 7

f 3;;

- : Nat = 5

```
x;;
- : Nat = 7

sum =
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum
;;

sum x 3;;
- : Nat = 10

M = Nat;;
type M = Nat

NinN = M -> M;;
type NinN = Nat -> Nat

N3 = M -> NinN;;
type N3 = Nat -> Nat -> Nat


letrec sum : N3 =
  lambda n : M. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum 21 34
```

;;

- : Nat = 55

2.3 String Type

In this section, a new type String has been defined to support character strings. A string is enclosed in double quotes. To recognize this, the STRINGV token was added in lexer.mll. Note that the double quotes are not stored as part of the string.

This new type has been incorporated into lambda.mli, lambda.ml, and parser.mly to ensure proper recognition, with the usual modifications to the functions handling newly added terms. It is worth mentioning that in isval, TmString returns true as strings are always considered values.

In addition to including the String type, the concat operator has been implemented to handle string concatenation. To recognize this operator, its corresponding token was added to lexer.mll, and the relevant functions in lambda.ml were updated accordingly.

Examples:

"under";;

- : String = "under"

concat "under" "water";;

- : String = "underwater";;

s = "tart";;

s : String = "tart"

concat "apple" s;;

- : String = "appletart"

concat (concat "apple" s) "time";;

- : String = "appletarttime"

2.4 Tuples

Tuples with any number of elements have been included, along with projection operations based on the position of the elements.

To begin, we need to recognize tuples, so curly brackets and commas have been included to separate the values within the tuple. As expected, a TyTuple type has been created for tuples, which stores the type of each element in a list of types, and a TmTuple has been created to represent the term as a list of terms.

These two have been included in lambda.mli, lambda.ml, and the rules for recognizing them are implemented in parser.mly.

For projection, the term TmProj was created and included in lambda.mli and lambda.ml. This TmProj is a term * string, with the goal of later using this term to project values for records. In this way, when including TmProj, the type of projection is first recognized—whether for tuples or for records. In this case, the string is converted to an integer. Projections within other projections have also been taken into account.

indexTerm is made for managing projections, including a posible projection or an atomicTerm at parser.mly.

Examples:

{ 10, true, "messi"};;

```
- : {Nat, Bool, String} = {10, true, "messi"}

{ 10, true, "messi"}.1;;
- : Nat = 10

{ 10, true, "messi"}.2;;
- : Bool = true

{ 10, true, "messi"}.3;;
- : String = "messi"

{ 10, true, "messi"}.7;;
type error: label 7 not found

{ 10, true, "messi"}.w;;
type error: label w not found

{ 10, true, {"messi", "cr7", 9}};;
- : {Nat, Bool, {String, String, Nat}} = {10, true, {"messi", "cr7", 9}}

{ 10, true, {"messi", "cr7", 9}}.3;;
- : {String, String, Nat} = {"messi", "cr7", 9}

{ 10, true, {"messi", "cr7", 9}}.3.2;;
- : String = "cr7"

t = { 10, true, {"messi", "cr7", 9}};;
t : {Nat, Bool, {String, String, Nat}} = {10, true, {"messi", "cr7", 9}}
```

t.3.3;;

- : Nat = 9

2.5 Records

Records use the same tokens as tuples, so nothing new was added to lexer.mll.They are very similar to tuples, so again, a type for them, TyRecord, was created. Unlike tuples, in records, each element has an identifier and an associated value. In this way, the type is stored as a list of string * ty, and the term is stored as a list of string * ty.

Due to this difference, new rules were created in parser.mly for records, which store these values in the corresponding list.The corresponding projection was included in indexTerm for records.

Examples:

{x=3, y=6, z=9};;

- : {x : Nat, y : Nat, z : Nat} = {x = 3, y = 6, z = 9}

{x=3, y=6, z=9}.x;;

- : Nat = 3

{x=3, y=6, z=9}.y;;

- : Nat = 6

{x=3, y=6, z="hi"}.z;;

- : String = "hi"

```
t = {x=3, y=6, z=9};;
t : {x : Nat, y : Nat, z : Nat} = {x = 3, y = 6, z = 9}

t.z;;
- : Nat = 9

w = {day={"hi", "tomas"}, gp = 3};;
w : {day : {String, String}, gp : Nat} = {day = {"hi", "tomas"}, gp = 3}

w.day;;
- : {String, String} = {"hi", "tomas"}

w.gp;;
- : Nat = 3

w.1;;
type error: label 1 not found

w.hello;;
type error: label hello not found
```

2.6 Variants

To include variants in this lambda calculus interpreter, specific tokens for variants were added. These tokens are <, >, |, as, =>, case, and of. Once the variant tokens are recognized, specific rules can be created for them. These rules, included in the parser, add the variant tagging case and the case expression to the term definition. For the

case expression, an additional rule, variantCases, has been created. This rule operates with VariantCases, which is responsible for storing each variant case as a tuple of three elements: the tag, the variable, and the body, in a list.

Of course, the new type for variants has been included, registering the types in the same way as for records, using a list of pairs with an identifier and a type.

At this point, the new types and terms were added to lambda.mli and lambda.ml. In lambda.ml, the necessary functions were modified to accommodate the new type (such as typeof, typeofTy, eval1, etc.). It is worth noting that in this interpreter, when associating a variable with a variant value, the value is not automatically associated with the variant's type. Therefore, we must explicitly specify the variant's type using as followed by the type.

Since there is no wildcard, all values of the cases must be explicitly named. For the same reason, it is not possible to create cases where one returns a value for a specific case and uses the wildcard for the rest of the cases.

Another clarification is that in the bodies of each case, to ensure proper syntactic analysis, the body is parenthesized if it contains more than just a simple value.

The following examples are provided, highlighting the add function of type Int -> Int -> Int, which implements integer addition for this data type. This add function was included in the examples.txt file.

Examples:

Int = <pos:Nat, zero:Bool, neg:Nat>;;

type Int = <pos : Nat, zero : Bool, neg : Nat>

p3 = <pos=3> as Int;;

p3 : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>

```
z0 = <zero=true> as Int;;
z0 : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>

n5 = <neg=5> as Int;;
n5 : <pos : Nat, zero : Bool, neg : Nat> = <neg = 5>

is_zero = L i : Int.
   case i of
      <pos=p> => false
   | <zero=z> => true
   | <neg=n> => false
;;

is_zero p3;;
- : Bool = false

is_zero z0;;
- : Bool = true

is_zero n5;;
- : Bool = false

abs = L i : Int.
   case i of
      <pos=p> => (<pos=p> as Int)
   | <zero=z> => (<zero=true> as Int)
   | <neg=n> => (<pos=n> as Int)
```

```
;;

abs p3;;
- : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>

abs z0;;
- : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>

abs n5;;
- : <pos : Nat, zero : Bool, neg : Nat> = <pos = 5>

sum =
  letrec sum : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
  in
    sum
;;

add =
  letrec add : Int -> Int -> Int =
  lambda i1 : Int. lambda i2 : Int.
  case i1 of
  <zero=z1> => i2

  | <pos=p1> =>
    (case i2 of
      <zero=z2> => i1
    | <pos=p2> => (<pos=sum p1 p2> as Int)
```

```
    | <neg=n2> =>
      (
      if iszero p1 then
      if iszero n2 then
        <zero=true> as Int
      else
        <neg=n2> as Int
      else
      if iszero n2 then
        <pos=p1> as Int
      else
        add (<pos=pred p1> as Int) (<neg=pred n2> as Int)
      )
    )

| <neg=n1> =>
    (case i2 of
      <zero=z2> => i1
    | <neg=n2> => (<neg=sum n1 n2> as Int)
    | <pos=p2> =>
    (
      if iszero n1 then
      if iszero p2 then
        <zero=true> as Int
      else
        <pos=p2> as Int
      else
      if iszero p2 then
```

```
            <neg=n1> as Int
          else
            add (<neg=pred n1> as Int) (<pos=pred p2> as Int)
            )
      )
      in add
;;

add (<neg=4> as Int) (<neg=3> as Int);;
- : <pos : Nat, zero : Bool, neg : Nat> = <neg = 7>

add (<pos=3> as Int) (<neg=3> as Int);;
- : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>

add (<pos=5> as Int) (<neg=2> as Int);;
- : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>

add (<zero=true> as Int) (<neg=3> as Int);;
- : <pos : Nat, zero : Bool, neg : Nat> = <neg = 3>

add (<neg=7> as Int) (<pos=10> as Int);;
- : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>
```

2.7 Lists

In this section, lists containing elements of the same type have been incorporated. Additionally, basic operations have been included to get the head, the tail, and check if the list is empty.

The new tokens to incorporate this type are square brackets, cons for the constructor, nil, isnil, head, tail, and List to represent this type in the output. Each has its corresponding token, which is subsequently handled in parser.mly to recognize them in the input and process them appropriately.In this list representation, the type of each element must be specified within square brackets, followed by the term, except for cons, which must have two associated terms.

In lambda.mli and lambda.ml, the new type and terms are included. For these types and terms, the necessary changes are made to accommodate the new type, including updates to: subst, free_vars, type_of, eval1, isval, pretty-printer, string_of_ty, typeofTy.

Three functions involving lists have been added to examples.txt:

length: Calculates the length of a given list recursively based on summation.

append: Concatenates two lists.

map: Computes the resulting list by applying a given function to each element of the input list.

These functions, along with additional examples, are included in the file.

As has been done so far, the base type is applied to lists. This can also be observed in the following examples through the types they have.

Examples:

nil[Nat];;

- : List[Nat] = nil[Nat]

cons[Nat] 1 nil[Nat];;

```
- : List[Nat] = cons[Nat] 1 nil[Nat]

cons[Nat] 1 (cons[Nat] 2 nil[Nat]);;
- : List[Nat] = cons[Nat] 1 (cons[Nat] 2 nil[Nat])

l = cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3 nil[Nat]));;
l : List[Nat] = cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3 nil[Nat]))

isnil[Nat] l;;
- : Bool = false

head[Nat] l;;
- : Nat = 1

tail[Nat] l;;
- : List[Nat] = cons[Nat] 2 (cons[Nat] 3 nil[Nat])

l2 = cons[Nat] 4 (cons[Nat] 5 nil[Nat]);;
l2 : List[Nat] = cons[Nat] 4 (cons[Nat] 5 nil[Nat])

f = L x:Nat. pred x;;

sum =
  letrec sum : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
  in
    sum
;;
```

```
prod =
  letrec prod : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero m then 0 else sum n (prod n (pred m))
  in
    prod
;;

length =
  letrec length : List[Nat] -> Nat =
    lambda l : List[Nat]. if isnil[Nat] l then 0 else sum 1 (length (tail[Nat] l))
  in
    length
;;

N3 = Nat -> Nat -> Nat;;
nil[N3];;
- : List[Nat -> Nat -> Nat] = nil[N3]

cons[N3] sum nil[N3];;

cons[N3] prod (cons[N3] sum nil[N3]);;

length (cons[N3] prod (cons[N3] sum nil[N3]));;
type error: parameter type mismatch

length l;;
- : Nat = 3
```

```
(head[N3] (cons[N3] prod (cons[N3] sum nil[N3]))) 9 9;;
- : Nat = 81

letrec append : List[Nat] -> List[Nat] -> List[Nat] =
  lambda l1 : List[Nat]. lambda l2 : List[Nat].
  if isnil[Nat] l1 then l2
  else cons[Nat] (head[Nat] l1) (append (tail[Nat] l1) l2)
  in
  append l l2
;;

letrec map : (Nat -> Nat) -> List[Nat] -> List[Nat] =
  lambda f : (Nat -> Nat). lambda l1 : List[Nat].
  if isnil[Nat] l1 then nil[Nat]
  else cons[Nat] (f (head[Nat] l1)) (map f (tail[Nat] l1))
  in
  map f l
;;
```

2.8 Subtyping

In this final section, subtyping has been incorporated by creating a function that implements subtype polymorphism for records and functions. To achieve this, the general type-checking function has been reimplemented to use this form of polymorphism wherever applicable.

This new function for verifying subtyping has been named subtypeof and has been included in lambda.ml.It determines if one type (tm1) is a subtype of another (tm2) by recursively analyzing their structure.

For record types, it ensures all fields in tm1 exist in tm2 with compatible types. For function types, it enforces contravariance on input types and covariance on output types. In all other cases, it checks for type equality. This function is crucial for supporting flexible and type-safe operations, especially when working with complex type structures like records or functions.

Additionally, three lambda expressions that involve subtyping operations were included in examples.txt.Those three examples and some other examples are provided to show some results.

Examples:

```
{};;
- : {} = {}

let
  idr = lambda r : {}. r
in
  idr {i=5, j=10}
;;
- : {} = {i = 5, j = 10}

(lambda r : {i : Nat}. r.i) {i=0, j=true};;
- : Nat = 0

(lambda r : {x : Nat}. r.x) {x=10, y=true, z="underwater"};;
```

```
- : Nat = 10

(lambda r : {x : Nat, y : Bool}. {r.x, r.y}) {x=10, y=true, z="extra"};;
- : {Nat, Bool} = {10, true}

(lambda r : {x : Nat, y : Bool}. r.x) {y=true, x=10, z="extra"};;
- : Nat = 10

let
  f = lambda r : {x : Nat, y : Bool}. {x=r.x}
in
  let
  apply = lambda g : ({x : Nat} -> {x : Nat}). g {x=10, y=true}
in
  apply f
;;
- : {x : Nat} = {x = 10}

(lambda r : {x : Nat, y : Bool}. r.x) {x=10};;
type error: parameter type mismatch

let
  apply_to_x = lambda f : {x : Nat} -> Nat.
    f {x=7, y=true}
in
  apply_to_x (lambda r : {x : Nat, y : Bool}. r.x)
;;
- : Nat = 7
```