Lambda Calculus Interpreter

This is the first lab practice (P1_DLP_Q7_2022_23.pdf) for the college course of DLP, which consists of the implementation of a Lambda Calculus interpreter, following the rules here (summary_of_rules.pdf) explained.
By Sofía Abal Freire and Alejandro Rivera García

Technical manual

In this paragraph we will show how to use all the functions implemented for each exercise. All the examples shown next (and more), can be found in the *examples.txt* file.

Multi-line expressions

For this exercise, only the file *main.ml* was modified, because we decided to directly manipulate the string the user writes. For this, we created a new function called *read_multi_lines*, which retrieves every line the user writes on the interpreter, trims and processes it, so if reads a non empty line that doesn't finish with ";" (no need for ";;", it will stop with just one semicolon), it will concat it to a new instruction string. Then, we call this new function from the *top_level_loop* function. This way, a multi-line expression is allowed.

Internal fixed point combiner

```
>> 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 21 34;;
```

prod, fib and fact examples

PROD:
Product of two natural numbers

```
>> prod = letrec prod : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then 0 else if iszero m then 0 else
sum n (prod (pred m) n) in
prod
;;
>> prod 21 34;;
```

FIB:
n-term of the Fibonacci series

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

>> fib 10;;
```

FACT:
Factorial

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

>> fib 10;;
```

Global definitions context

```
>> x = true;;
>> id = lambda x : Bool. x;;
>> id x;;
```

String type

String and concatenation(Union of 2 Strings), not valid for characters ; and "

```
>> "Cad3n4";
>> "cad3" ^ "n4";
>> "" ^ ((Lx:String. x^"by") "ru");;
```

Pairs type

A pair consists of two elements between curly brackets, and we can access both those elements thanks to having added two new tokens "first" and "second" that go before the pair. We decided to follow this notation and not the one with the tokens ".1" and ".2" after the pair, because we also wanted to implement tuples, so this way we can easily differenciate each function.

```
>> p = {2, true};;
>> p2 = {false, p};;
>> pair = {p, p2};;
>> first pair;;
>> second pair;;
```

Tuples type and Records type

We write both this sections together, because we followed the same logic to implement them, with the main difference that for tuples we separate each element by commas, and we access each element looking at its index, and for records we separate each element by commas, and each element consists of a tag, followed by the equals token, and followed by the element's value for that tag, and we access each element by looking at its tag.

Tuples type

```
>> tuple = {pair, 2, 3};;
>> tuple.1;;
>> first (tuple.1);;
```

Records type

As a side note, we can only have a record that it's empty, meaning that if we write two curly brackets together, with no elements, the interpreter will understand that as a record.

```
>> record = {a = {a = 1, b = 2}, b = tuple};;
>> record.a;;
>> (record.a).b;;
>> record.b;;
>> {};;
```

List type

Our interpreter can make use of the list type, the lists have the form -> cons[Type] value (Set of the rest of cons(Type) ... (nil[Type])). The lists end with a null element (nil[Type] in the previous example), and the practice has the typical operations indicated in the statement. "isNil[Type] list1", checks if it is a null list. "head[Type] list1" returns the first element of list1. "tail[Nat] list 1" returns the tail of list1.

```
>> nil[Nat];
>> cons[Nat] 10 (nil[Nat]);;
>> empty_list = nil[Nat];;
>> list1 = cons[Nat] 11 (cons[Nat] 77 (cons[Nat] 6 (nil[Nat])));;
>> isnil[Nat] empty_list;;
>> isnil[Nat] list1;;
>> head[Nat] list1;;
>> tail[Nat] list1;;
```

length, append and map examples

LENGTH:
Returns the number of elements in a list.

```
>> list1 = cons[Nat] 11 (cons[Nat] 77 (cons[Nat] 6 (nil[Nat])));;
>> len = letrec len : (Nat list) -> Nat = lambda l : Nat list. if (isnil[Nat]
l) then 0 else (succ (len (tail[Nat] l)))
in len;;
>> len list1;;
```

APPEND:
Concatenates two lists.

```
>> list1 = cons[Nat] 11 (cons[Nat] 77 (cons[Nat] 6 (nil[Nat])));;
>> list2 = cons[Nat] 32 (cons[Nat] 9 (cons[Nat] 16 (cons[Nat] 5 (nil[Nat]))));;

>> append = letrec append : (Nat list) -> (Nat list) -> Nat list = lambda ap1:
Nat list. lambda ap2: Nat list.
    if (isnil[Nat] ap1) then
        ap2
    else
        if (isnil[Nat] (tail[Nat] ap1)) then
            cons[Nat] (head[Nat] ap1) ap2
        else
            cons[Nat] (head[Nat] ap1) (append (tail[Nat] ap1) ap2)
in append;;

>> append list1 list2;;
```

MAP:
Applies a function to each item in a list.

```
>> list_fib = cons[Nat] 4 (cons[Nat] 3 (cons[Nat] 10 (cons[Nat] 5
(nil[Nat]))));;

>> func = lambda x:Nat. fib x;;

>> map = letrec map : (Nat list) -> (Nat -> Nat) -> Nat list =
lambda lst: Nat list. lambda f: (Nat -> Nat).
        if (isnil[Nat] (tail[Nat] lst)) then
                cons[Nat] (f (head[Nat] lst)) (nil[Nat])
        else
                cons[Nat] (f (head[Nat] lst)) (map (tail[Nat] lst) f)
in map;;
>> map list_fib func;;
```

Unit Type

Type indicating the absence of a specific value.

```
>> ();;
>> unit;;
```

Exit program

We don't have and expecific command for closing our interpreter, but we can use the default exit shortcut *CTRL + D*. This way, it'll show a goodbye message before closing the program.

User manual

We use Makefile to build our set up for the program.
To compile:

```
make all
```

To run the interpreter alone:

```
./top
```

To to run it alongside a file:

```
./top < examples.txt
```

To clean up after execution, keeping the executable top:

```
make clean
```

To clean up everything:

```
make cleanall
```