LAMBDA CALCULUS INTERPRETER

Authors:

Daniel Ferreiro Villamor - d.ferreiro@udc.es

Eva Maria Arce Ale - eva.arce@udc.es

Introduction

This will be a user guide where will explain in detail how we implemented the lambda interpreter based in Ocaml language, how the code works and the different modules where we implemented the different functionalities, as well as show some self-explanatory examples with code ready to execute and try out.

In the following sections we will be showing all the code improvements made in order to implement the before-mentioned functionalities.

Functionalities

1. Easier writing and introduction to lambda expressions.

   1.2 Multilinear expression recognition.

   In order to implement this section the changes made were on the main.ml document, where we implemented the function **read_command** which processes the lines while checking for the last 2 characters of a line being semicolons, ie: *succ 3;;* , and if so, then ending the acceptance of lines and evaluating.

```
if iszero 3
  then 0
else 1;;
```

2. Extensions on the lambda-calculus language.

   2.1 Addition of a **Fixed-point combinator**, making possible the definition of recursive functions.

   Firstly we implemented the recognition of a new token by the lexer.mll file, this being **LETREC**, posteriorly adding those tokens onto the parser.mly file and adding in the **term :** section support for the let rec term (**TmFix**) and finally adding functionality for this new term in the lambda files, this meaning making the corresponding addition to the **string_of_term, free_vars** & **subst** functions of the TmFix term, which is a pretty straight-forward one.

   For the **eval1** function, where we evaluate the term until a value is found, or in case an abstraction is found, then this one is substituted.

   For the **typeof** function, we need to check that the type of the term is the expected.

   Product

```
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 12 5;;
```

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 n)) (fib (pred (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 m then 0 else sum n (prod n (pred m))
in
letrec fac : Nat -> Nat =
    lambda n : Nat. if iszero n then 1 else prod n (fac (pred n))
in fac 3;;
```

2.2 Implementation of a global definition context that allows binding free-variable names to a value or term, so that it permits the future usage of them.

To implement this functionality a lot of changes were made, we will be explaining them file by file:

Starting with the parser file in which in the **s :** section we change it to Eval when it is a term followed by the **EOF** and to Bind when **IDV EQ** term **EOF** .

In the lambda files is where the majority of the changes where made:

- We added a **contextv** type of (string * term) list and the supporting functions for it **getdef**, **adddef** and **emptydef** .
- Changes were made to the **eval** and **eval1** function to take the context (ctx) as parameter whenever it is called.
- Added a **give_ctx** function which takes the context and a term and substitutes the name associated with a value for the value itself when evaluating terms.
- Finally we implemented an **execute** function which decides if and Eval or Bind is needed and also now handles part of the printing to be done.

At last in the main file we changed the loop, now both contexts are passed to the loop, same for the empty contexts. Also now in the try part we now loop the **execute** function mentioned before.

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

```
fib = 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 n)) (fib (pred (pred n)))
in fib;;
fib 7;;
```

2.3 Addition of the String type in order to support character chains as well as a concatenation function.

As with almost every new type to be implemented the first part is to add tokenization for that character, in this case: **STRINGV, CONCAT** & **STRING,** last one for the type, posteriorly in the parser adding the corresponding terms and type, this being **TmConcat** in the **appTerm :** section, **TmString** in the **atomicTerm :** section and **TyString** in of course the **atomicTy :** section.

Finally in the lambda files we added all the support for these new terms and type in the following functions:

string_of_term, free_vars, subst, string_of_ty, isval, typeof & **eval1**

With the only one being relevant enough to mention being the implementation of concat in the eval1 function which receives 2 terms as parameter, evaluating both of them to be Strings and concatenating them with the ^ operator.

```
"";;
"abc";;
concat "para" "sol";;
concat (concat "para" "sol") "es";;
lambda s : String. s;;
(lambda s : String. s) "abc" ;;
letrec replicate : String -> Nat -> String =
    lambda s : String. lambda n : Nat.
        if iszero n then "" else concat s (replicate s (pred n))
in replicate "abc" 3;;
```