DEVELOPMENT OF A LAMBDA CALCULUS INTERPRETER

Javier Fernández Rozas
Tomás Villalba Ferreiro

Practice group: 3.2

Design of Programming Languages
University of A Coruña
Academic year 2023-2024

Contents

1 Introduction **1**

2 Implementation of the exercises **1**

2.1 Improvements in entering and writing lambda expressions 1

2.1.1 Multi-line expression recognition 1

2.1.2 Implementation of a more complete “pretty-printer” 2

2.2 Extensions of the lambda-calculus language 3

2.2.1 Incorporation of an internal fixed point combiner (recursion) 3

2.2.2 Incorporation of a context of global definitions 6

2.2.3 Incorporation of the String type 8

2.2.4 Incorporation of tuples 9

2.2.5 Incorporation of records 11

2.2.6 Incorporation of variants 12

1 Introduction

The aim of this practice is the understanding and improvement of a lambda-calculus interpreter.

The main modifications are explained within the practise code, and in this memory we are giving a brief overview of the functionalities added, and providing examples

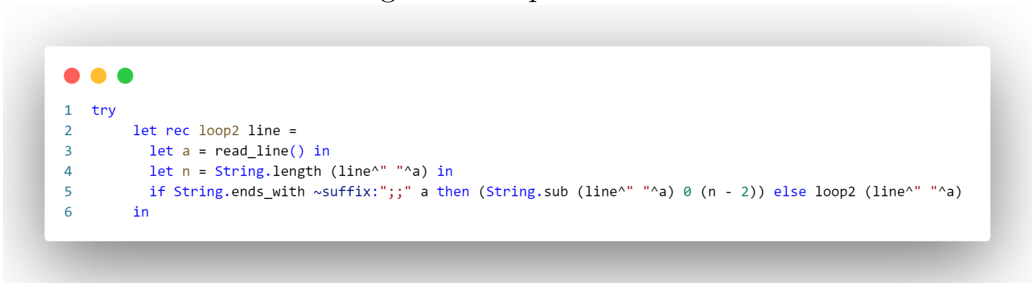
2 Implementation of the exercises

2.1 Improvements in entering and writing lambda expressions

2.1.1 Multi-line expression recognition

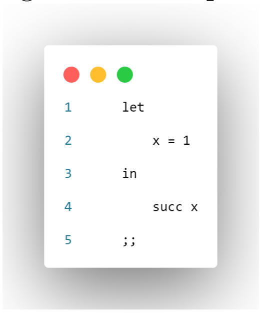
This new functionality adds the ability to recognise multi-line expressions, which is, expressions that can be extended for more than one line. To solve this exercise, our approach was to introduce a new function in the main.ml, loop2 (figure 20), that keeps reading lines, until it reaches a predefined string used for stopping, in our case, ";;".

Figure 1: loop2 function



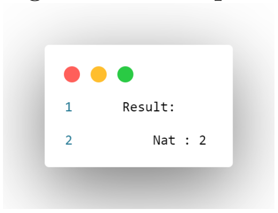
```
try
      let rec loop2 line =
        let a = read_line() in
        let n = String.length (line^" "^a) in
        if String.ends_with ~suffix:";;" a then (String.sub (line^" "^a) 0 (n - 2)) else loop2 (line^" "^a)
      in
```

Figure 2: Example 1



```
    let

        x = 1

    in

        succ x

    ;;
```

Figure 3: Example 2



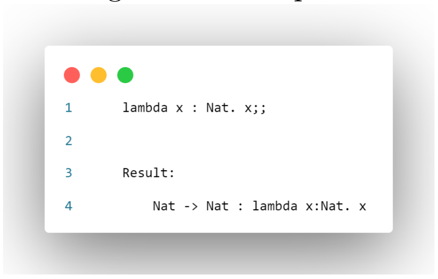
```
Result:
    Nat : 2
```

2.1.2 Implementation of a more complete "pretty-printer"

This functionality consists in reducing the number of parenthesis in total in the outputs.

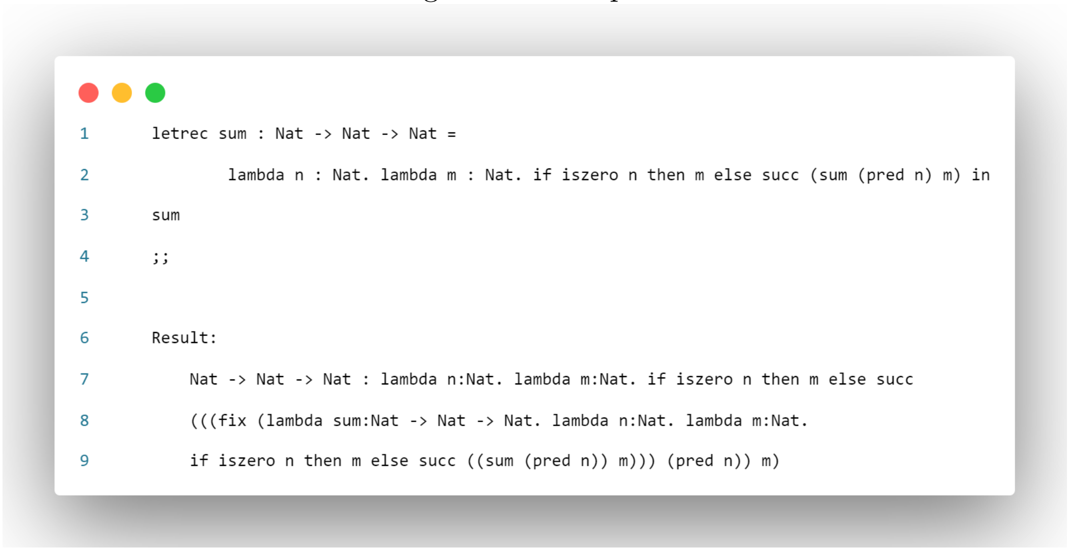
To solve this exercise, we modified the pre-existing string-of-term function to match every term that can be received, and display a print accordingly, thus reducing the total number of parenthesis.

Figure 4: Example 1



```
lambda x : Nat. x;;

Result:
    Nat -> Nat : lambda x:Nat. x
```

Figure 5: Example 2



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

Result:
    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)
```

2.2 Extensions of the lambda-calculus language

2.2.1 Incorporation of an internal fixed point combiner (recursion)

This functionality adds the ability to make recursive functions, by using the letrec in operator.

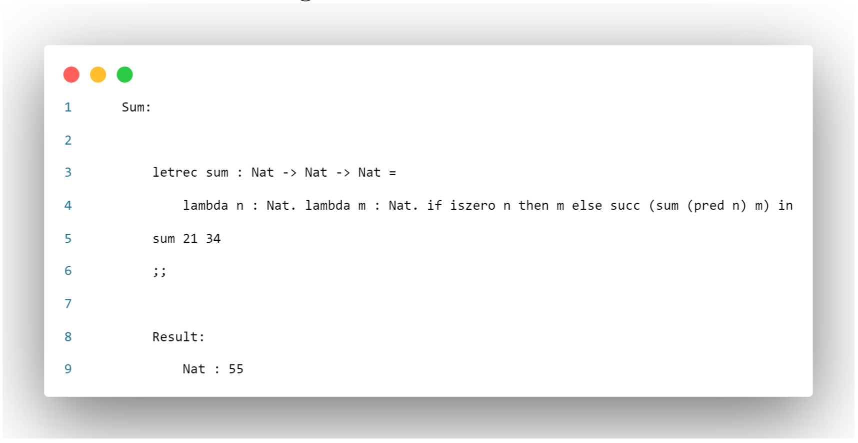
Firstly, to solve this exercise, we modified the files lexer.mll, parser,mly and main.ml to be able to recognize the "let rec in" sentence. This was achieved by internally adding the new definitions of the term fix (In the parser.mly, lambda.mli, lambda.ml and lexer.mll) and the term letrec (In the lexer.mll and parser.mli).

The term letrec will only be used to cath that term, because internally, the letrec will work with fix.

To make the fix work, we had to add changes to some pre-existing functions (Basically, every function which depended on adding/matching a new term). This changes were driven by the rules defined in the summary-of-rules.pdf that was given on class. This changes were applied to a number of funcions:

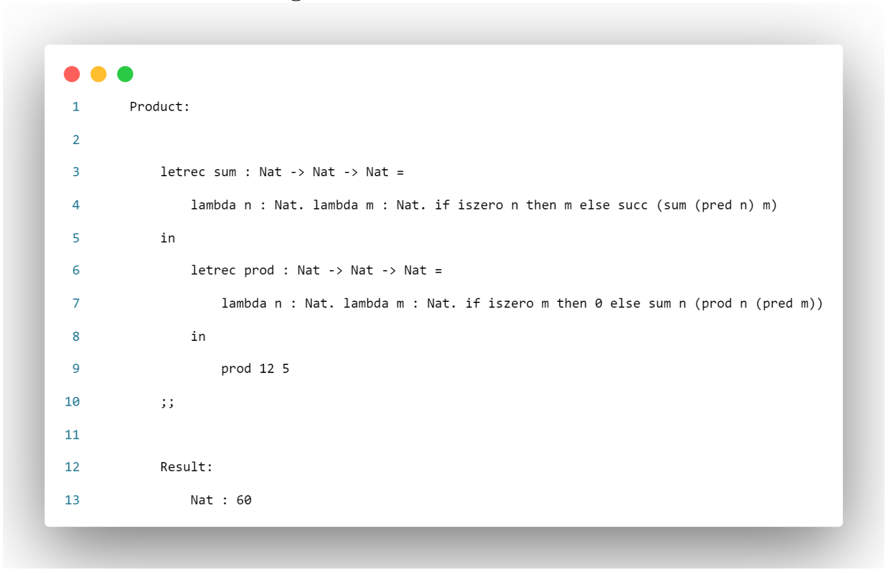
-Typeof, string-of-term, free-vars, subst and eval1.

Figure 6: Sum function



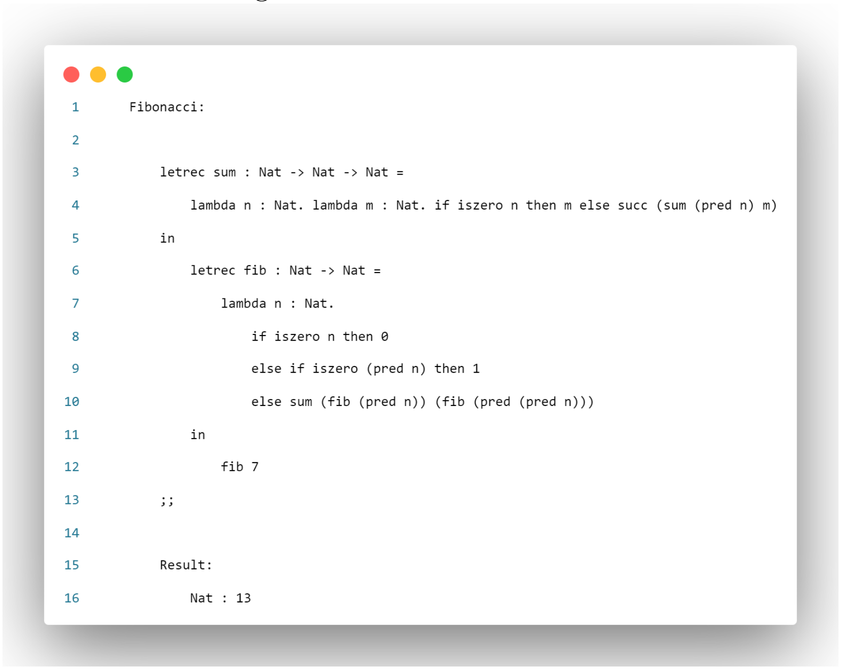
```
Sum:


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


    Result:
        Nat : 55
```

Figure 7: Product function



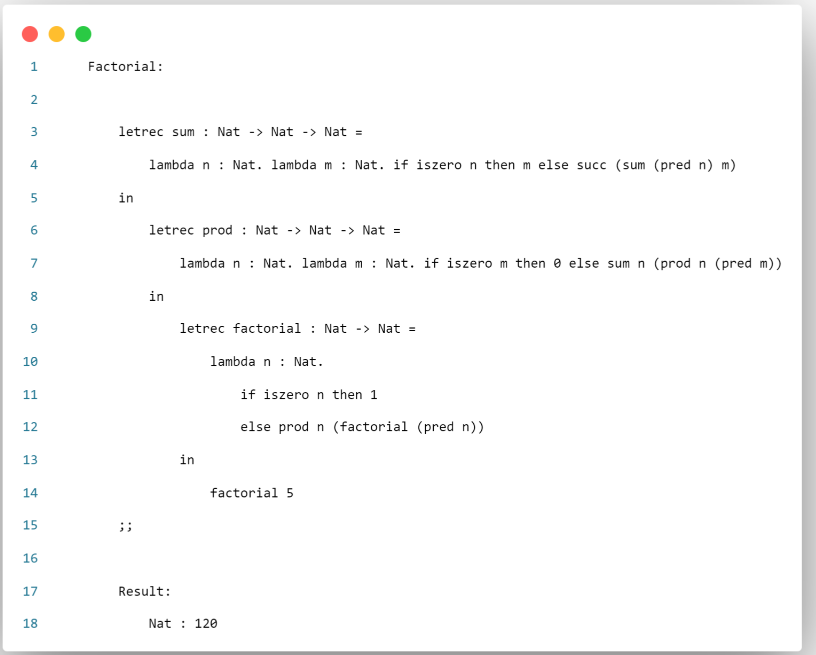
```
Product:

    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
    ;;

    Result:
        Nat : 60
```

Figure 8: Fibonacci function



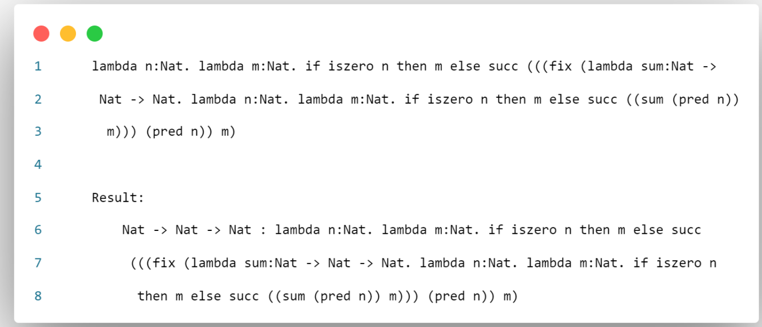
```
Fibonacci:

    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
    ;;

    Result:
        Nat : 13
```

Figure 9: Factorial function



```
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 factorial : Nat -> Nat =
                lambda n : Nat.
                    if iszero n then 1
                    else prod n (factorial (pred n))
            in
                factorial 5
    ;;

    Result:
        Nat : 120
```

Figure 10: Fix check

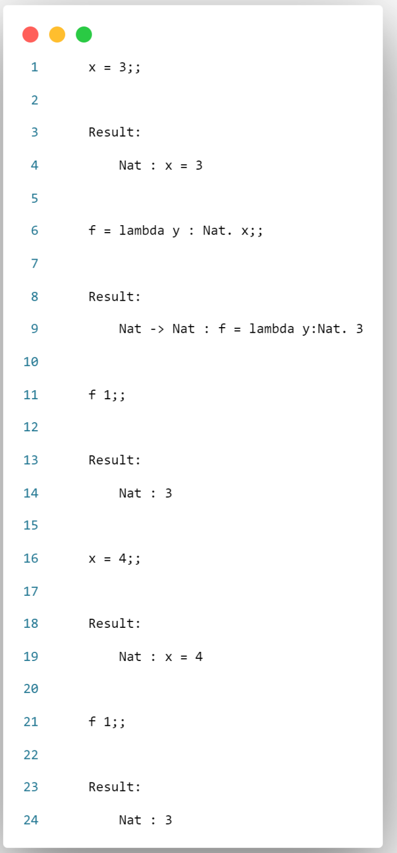


```
lambda n:Nat. lambda m:Nat. if iszero n then m else succ (((fix (lambda sum:Nat ->
 Nat -> Nat. lambda n:Nat. lambda m:Nat. if iszero n then m else succ ((sum (pred n))
  m))) (pred n)) m)

Result:
    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)
```

2.2.2 Incorporation of a context of global definitions

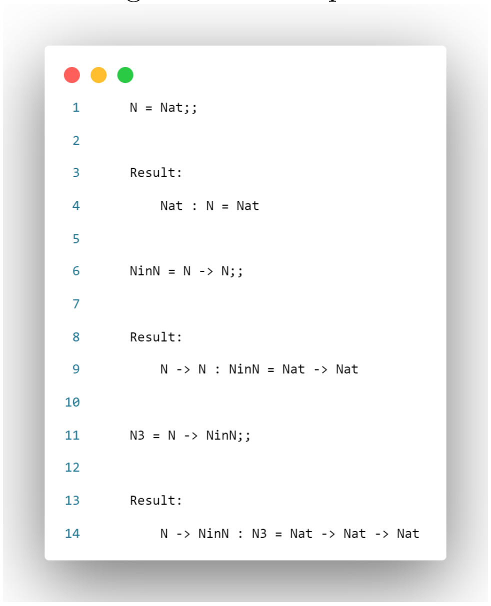
This new functionality adds the ability of associating names of variables with values, terms and types. The way we did this was: we use two global contexts (lists of string * type and string * term), one for types and the other for terms.

Figure 11: Example 1



```
x = 3;;

Result:
    Nat : x = 3

f = lambda y : Nat. x;;

Result:
    Nat -> Nat : f = lambda y:Nat. 3

f 1;;

Result:
    Nat : 3

x = 4;;

Result:
    Nat : x = 4

f 1;;

Result:
    Nat : 3
```

Figure 12: Example 2

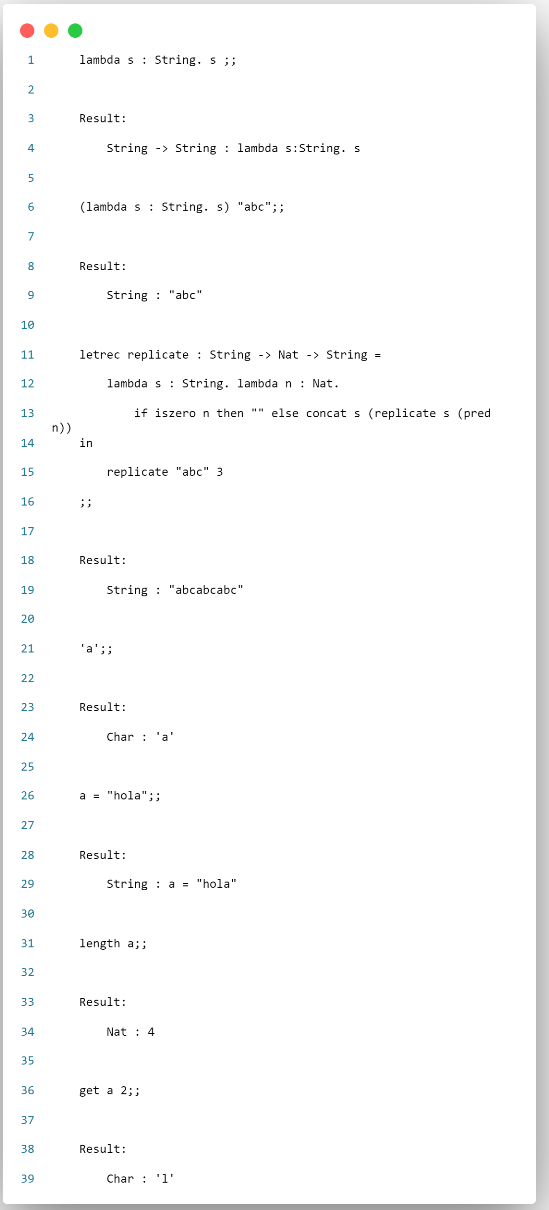


```
N = Nat;;

Result:
    Nat : N = Nat

NinN = N -> N;;

Result:
    N -> N : NinN = Nat -> Nat

N3 = N -> NinN;;

Result:
    N -> NinN : N3 = Nat -> Nat -> Nat
```

2.2.3 Incorporation of the String type

This functionality adds the new type String, besides some new functions such as concat,length and get. Concat gives us the possibility to concatenate two strings, length gives us the length of a string and get gives us the char at a given position of a string.

Figure 13: Example 1

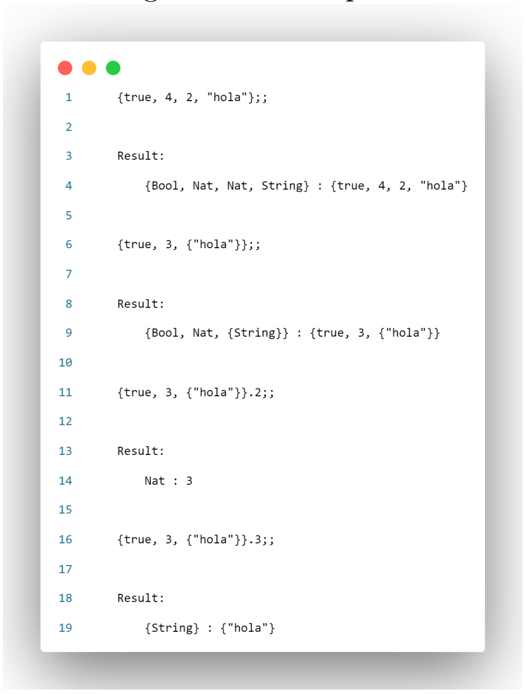


```
lambda s : String. s ;;

Result:
    String -> String : lambda s:String. s

(lambda s : String. s) "abc";;

Result:
    String : "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
;;

Result:
    String : "abcabcabc"

'a';;

Result:
    Char : 'a'

a = "hola";;

Result:
    String : a = "hola"

length a;;

Result:
    Nat : 4

get a 2;;

Result:
    Char : 'l'
```

2.2.4 Incorporation of tuples

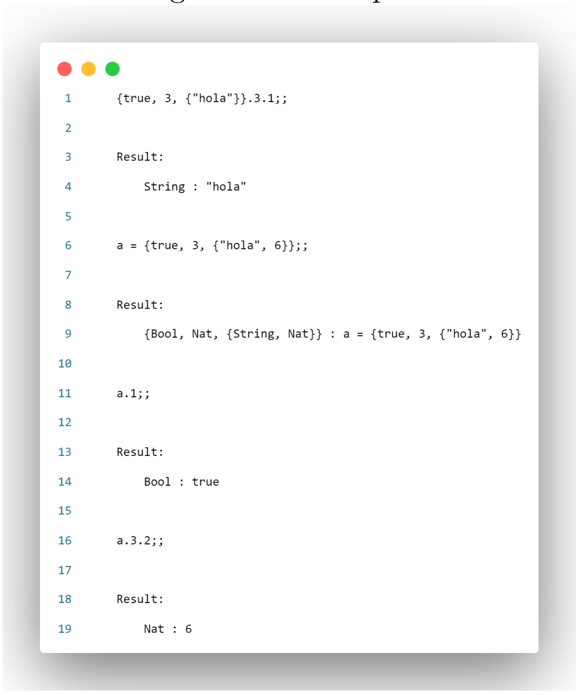
This functionality adds the new type tuples (set of any number of terms, enclosed in square brackets and separated by commas) , and the operations of projection based on the position of said elements. We add its tokens to the parser and lexer. Then, we save the tuple as a tmTuple, and if there is more than one term, they are saved with ands , and if not, as a term.

Figure 14: Example 1



```
{true, 4, 2, "hola"};;

Result:
    {Bool, Nat, Nat, String} : {true, 4, 2, "hola"}

{true, 3, {"hola"}};;

Result:
    {Bool, Nat, {String}} : {true, 3, {"hola"}}

{true, 3, {"hola"}}.2;;

Result:
    Nat : 3

{true, 3, {"hola"}}.3;;

Result:
    {String} : {"hola"}
```

Figure 15: Example 2

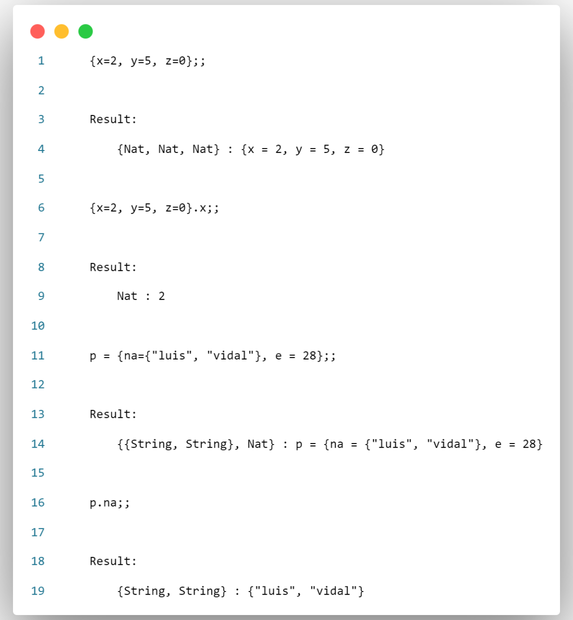


```
{true, 3, {"hola"}}.3.1;;

Result:
    String : "hola"

a = {true, 3, {"hola", 6}};;

Result:
    {Bool, Nat, {String, Nat}} : a = {true, 3, {"hola", 6}}

a.1;;

Result:
    Bool : true

a.3.2;;

Result:
    Nat : 6
```

2.2.5 Incorporation of records

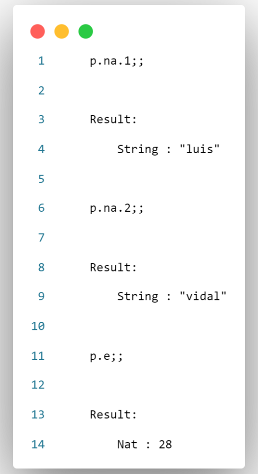
This functionality adds the new type records (set of any number of assigns (pair of label - value) , enclosed in square brackets and separated by commas), and the operations of projection based on the position of said elements. We add its tokens to the parser and lexer. Then, we save the register as a tmRegister, and if there is more than one assign, they are saved with ands , and if not, as a single TmAssign.

Figure 16: Example 1



```
{x=2, y=5, z=0};;

Result:
    {Nat, Nat, Nat} : {x = 2, y = 5, z = 0}

{x=2, y=5, z=0}.x;;

Result:
    Nat : 2

p = {na={"luis", "vidal"}, e = 28};;

Result:
    {{String, String}, Nat} : p = {na = {"luis", "vidal"}, e = 28}

p.na;;

Result:
    {String, String} : {"luis", "vidal"}
```

Figure 17: Example 2

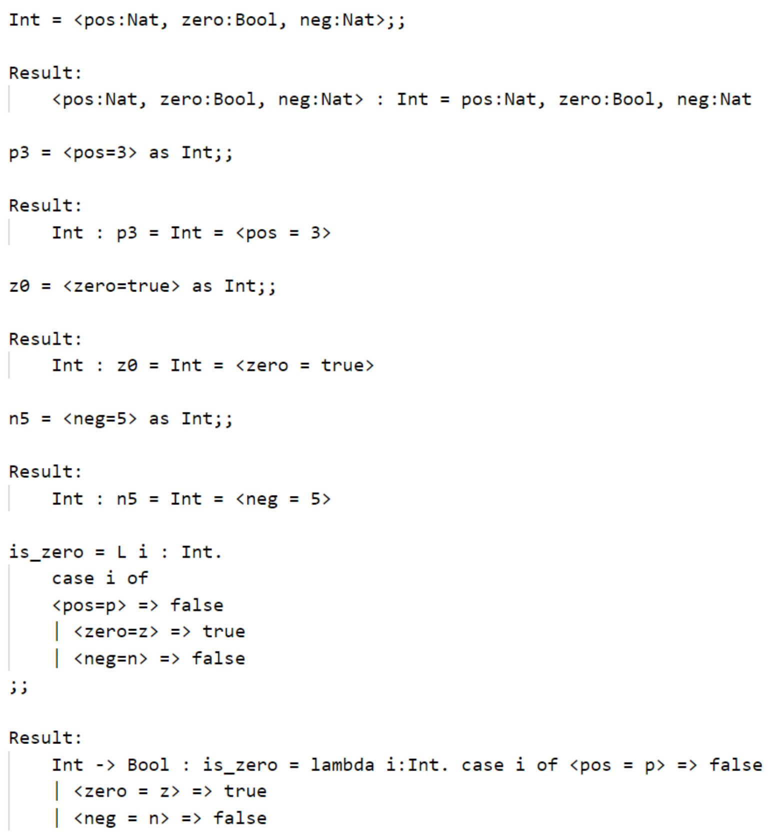


```
p.na.1;;

Result:
    String : "luis"

p.na.2;;

Result:
    String : "vidal"

p.e;;

Result:
    Nat : 28
```

2.2.6 Incorporation of variants

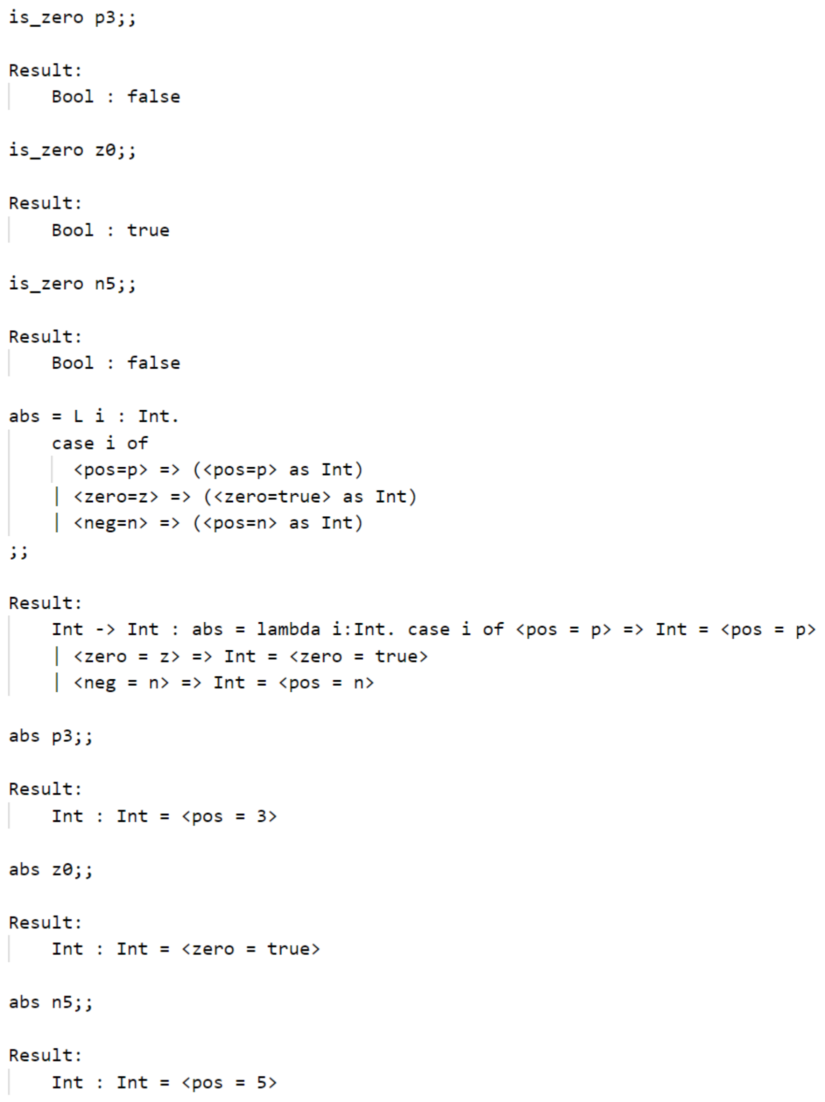
This functionality adds the new type variants (labeled generalization of the binary sum types) and the case of, that allows us to do pattern-matching on TmVariante. Explained in the code with an example.

Figure 18: Example 1



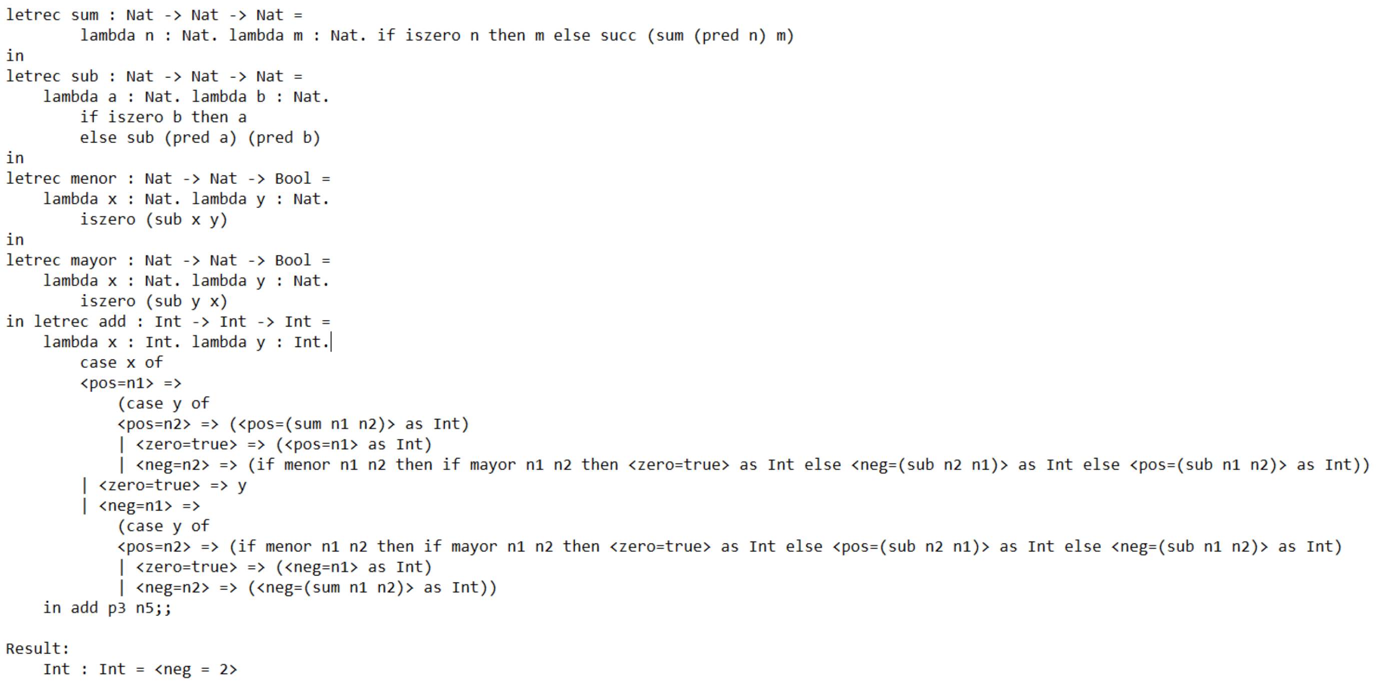
```
Int = <pos:Nat, zero:Bool, neg:Nat>;;

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

p3 = <pos=3> as Int;;

Result:
    Int : p3 = Int = <pos = 3>

z0 = <zero=true> as Int;;

Result:
    Int : z0 = Int = <zero = true>

n5 = <neg=5> as Int;;

Result:
    Int : n5 = Int = <neg = 5>

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

Result:
    Int -> Bool : is_zero = lambda i:Int. case i of <pos = p> => false
    | <zero = z> => true
    | <neg = n> => false
```

Figure 19: Example 2



```
is_zero p3;;

Result:
    Bool : false

is_zero z0;;

Result:
    Bool : true

is_zero n5;;

Result:
    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)
;;

Result:
    Int -> Int : abs = lambda i:Int. case i of <pos = p> => Int = <pos = p>
    | <zero = z> => Int = <zero = true>
    | <neg = n> => Int = <pos = n>

abs p3;;

Result:
    Int : Int = <pos = 3>

abs z0;;

Result:
    Int : Int = <zero = true>

abs n5;;

Result:
    Int : Int = <pos = 5>
```

Figure 20: Example 3



```
letrec sum : Nat -> Nat -> Nat =
        lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
letrec sub : Nat -> Nat -> Nat =
    lambda a : Nat. lambda b : Nat.
        if iszero b then a
        else sub (pred a) (pred b)
in
letrec menor : Nat -> Nat -> Bool =
    lambda x : Nat. lambda y : Nat.
        iszero (sub x y)
in
letrec mayor : Nat -> Nat -> Bool =
    lambda x : Nat. lambda y : Nat.
        iszero (sub y x)
in letrec add : Int -> Int -> Int =
    lambda x : Int. lambda y : Int.
        case x of
        <pos=n1> =>
            (case y of
            <pos=n2> => (<pos=(sum n1 n2)> as Int)
            | <zero=true> => (<pos=n1> as Int)
            | <neg=n2> => (if menor n1 n2 then if mayor n1 n2 then <zero=true> as Int else <neg=(sub n2 n1)> as Int else <pos=(sub n1 n2)> as Int))
        | <zero=true> => y
        | <neg=n1> =>
            (case y of
            <pos=n2> => (if menor n1 n2 then if mayor n1 n2 then <zero=true> as Int else <pos=(sub n2 n1)> as Int else <neg=(sub n1 n2)> as Int)
            | <zero=true> => (<neg=n1> as Int)
            | <neg=n2> => (<neg=(sum n1 n2)> as Int))
    in add p3 n5;;

Result:
    Int : Int = <neg = 2>
```