Before explaning any implementation details we need to add that this assignment has been done by:

Xavier Rodríguez Muñoz - xavier.rodriguez.munoz@udc.es

Anxo Tajuelo Ferrer - anxo.tajuelo@udc.es

This document serves the purpose of clarifying any changes done to the initial code given to start the assignment, in here we will discuss every point that we implemented with it's corresponding explanation of what and why we added or changed certain lines of code.

First of all, we'll start with the first point of all, **section 1.1**, where we were told to implement multi-line recognition for our interpreter, we approached the problem from the processing perspective so that we didn't have to make any changes to the lexer and parser. We replaced the "read_line" function with a new function called "read_line_semicolon" in which we receive the result of the "read_line" function and check to see if there's any semicolon on it, if there's no semicolon on the string then we add it as it is to our "command" string variable which represents the final line that is going to be given as a result. If there's a semicolon then we parse the line to get the string previous to the first semicolon encountered on the line, if there's more content starting from that semicolon onwards we just simply ignore it, as it is not relevant to us.

The second point we need to discuss is **section 3.1**, it is requested to implement the fix operator to add the recursion to our language, first we need to add the right rule to the grammar on the parser, on which we return a LetIn but the second argument will be a TmFix, a new term defined for the implementation of this section which represents the function that is going to be subject to the recursion. Once we finish making the necessary changes on the parser we'll have to look into the "typeof" and "eval" function to which we need to add the appropriate rules to. That would mean that on "eval" we need to evaluate the term until we get a value and if the term is an abstraction then we need to do the proper substitution of the parameter given for that abstraction. On the "typeof" function we first need to check the type of the term, if the type is not an arrow then we display the corresponding error message, if it is an arrow we check to see if the type of the body is compatible with the domain. Also on "free_vars" we'll return the free variables of the term term. For the rest of the functions like "string_of_term" we'll do the proper changes to add the TmFix possibility.

As stated on the assigment we needed to implement the multiplication and the fibonacci sequence, these functions were added to the "examples.txt" file.

The next point we need to address would the variable context stated on **section 3.2**, for it we need to redefine the structure of our interpreter a little bit, we decided to preserve the variable that represented the type context but instead of being a "string -> ty", now we would have "string -> ty -> term option" to try to minimize the changes we would need to make on the code. This change will allow us to insert a variable with it's type, name reference and original term, we decided to make it an option also allow the possibility of still inserting a reference and it's type, like we were doing up until this moment. For this to work we will need two new operations similar to addbinding and getbinding that were already implemented, to help with the getbinding on both situations we implement a new function called "assoc" (description of its use already included in the code). Also it is needed a small change on both "eval" and "eval1" that now also receive a context as a parameter and a new rule was added to allow extracting the value of a variable (E-TmVar). With all of this said we need to address probably the biggest change on the code, wich the addition of a new function

called “execute” that now distinguishes between “Eval t” that we use to indicate that we want to evaluate a term, allows us to preserve the same functionality that we had up until this point, and Bind(s, t) that indicates that we want to add a new variable to our context. In both cases we will first type and then evaluate the terms, and store those results accordingly with the aformentioned functions. Of course, we need to change the parser with the same criteria, this can be seen on the “parser.ml” file.

In the “main” of our interpreter we now need to consider the new context and pass it as a paremeter to the main loop function so it keeps up to date with every execution.

Next up in the list we have **section 3.3** regarding the type “String” and the “Concat” operation used to append strings together. The implementation in general is really simple we need to create a new type called “TyString” and a new term called “TmTerm”, add the pattern matching to all the necessary functions like “typeof” and add a new rule to our parser and lexer with the appropriate definitions in order to recognize “Strings”. For the “Concat” operations we need to do something similar to the “String” case but with a sligth difference, we now need to add some new rules to our “eval” function, two of them to represent the cases in which one of the parameters is a value and one of them to represent the case in which both of them are values. It’s important to note that the order of the definition of the rules matters so we decided to go with the following order:

- Both of them are values
- Second one is a value
- First one is a value

Last but not least we have to talk about **section 3.4** which includes the implementation of the “Pairs” type and it’s projections, first of all we need to define in the parser how we are going to recognize both of these cases, we decided to do it with the following scheme:

- LBRACKET term COMMA term RBRACKET (For the TmPair)
- LBRACKET term COMMA term RBRACKET DOT INTV (For the TmProj)

Now on the “typeof” and “eval” functions we need to add the rules accordingly, as well as defining the new types and terms “TyPair” and “TyProj”, we have a similar situation to the “String”/”Concat” case in which the order of definition might change the final result so in order solve this problem we decided to take the following approach of definition:

- E-Pair2
- E-Pair1

For the evaluation “Pair” rules.

- E-Proj
- E-Proj1/2

For the evaluation “Projection” rules for this “Pairs”.

As done with previous sections we also need to update any function that involves working this terms of types