UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Master degree in Computer Science

Course of "Formal Methods in Computer Science"

PImp

Parser and Interpreter for an Imperative Language

Davide De Simone
id: 742235

Accademic year 2020/21

Table of contents

1. Introduction

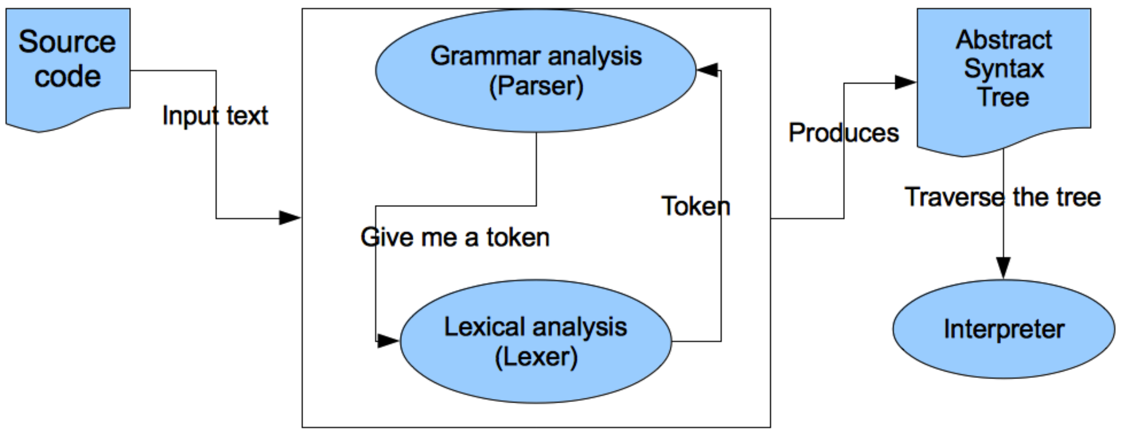
This is the documentation of the project developed for the lab part of the exam "Formal Methods in Computer Science". The aim of this project is developing a parser and an interpreter using Haskell, a pure functional language.

Let's start with some definitions:

- A parser is a software component that takes input data and builds a hierarchical data structure -often some kind of tree- giving a structural representation of the input while checking for correct syntax. A parser is often preceded by a lexical analyzer, which tokenize the input [1].

- An interpreter is a software component that directly executes instructions written in a programming language, without requiring them previously to have been compiled into a machine language program [2]. An interpreter generally uses one of the following strategies for program execution:
    - Parse the source code and perform its behavior directly; (e.g. Lisp)
    - Translate source code into some efficient intermediate representation and immediately execute that. (e.g. Python, Ruby)
    - Explicitly execute stored precompiled bytecode made by a compiler and matched with the interpreter Virtual Machine. (e.g. Java)

An overall picture:



The interpreter implemented in this project is of the first type of those listed in the interpreter definition above i.e. it will parse the source code and perform its behavior directly, rather that producing an intermediate representation to execute.

The language which the parser and the interpreter can handle is a simple imperative language composed by the following constructs:

- ***Assignment***: assign a value to a variable. If the variable already exists overwrite the previous value. Otherwise create a new a variable and assign to it the value.
- ***Skip***: does nothing, the program move forward to the execution of the next command (if there is one).
- ***If-then-else***: a selection structure which based on the truthiness of a condition executes one or another sequence of commands. If the condition is true, the *then* branch will be executed otherwise the *else* branch (which is optional).
- **Switch**: a selection structure that executes one between various sequence of commands depending on the truthiness of their condition.
- ***While***: an iterative structure that keeps executing a certain sequence of instructions until a condition is verified.
- ***For***: executes a certain sequence of instructions for a fixed number of times specified through a counter variable.

The language support variables of Numeric (integer and float), Array or Matrix type. The classical arithmetical operations can be done on the Numeric variables (sum +, difference -, product *, division /, power ^, module %), also Boolean comparisons can be performed (==, !=, > , <, >=, <=). For the Booleans expressions, the three logical operators are provided (and, or, not). The language also provides the array and matrices data structures, which can only contain Numeric values.

Even if Haskell uses a *Lazy Evaluation* strategy (aka Call-by-need), which delays the evaluation of an expression until its value is needed, our interpreter will use an *Eager Evaluation* strategy since is the most common for imperative languages. Eager Evaluation, also known as *Call-by-value or Strict Evaluation*, evaluates expressions in the exact moment in which they are assigned to a variable or when they are arguments of functions or other expressions [3].

2. PImp grammar

The language grammar in BNF is shown below.

```
1. <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2. <lower> ::= a-z
3. <upper> ::= A-Z
4. <nat> ::= <digit> <nat> | <digit>
5. <integer> ::= [-] <nat>
6. <float> ::= <integer> ‘.’ <nat>
7. <alphanum> ::= <upper> <alphanum> | <lower> <alphanum> |
    <nat> <alphanum> | <upper> | <lower> | <nat>
8. <identifier> ::= <lower> | <lower> <alphanum>
9. <afactor> ::= ‘(‘ <aexp> ‘)’ | <identifier> ‘[‘ <aexp> ‘]’ |
   <identifier> ‘[‘ <aexp> ‘]’ ‘[‘ <aexp> ‘]’ | <identifier> | <integer>|
        <float>
10.  <aterm> ::= <afactor> ‘*’ <aterm> | <afactor> ‘/ ’ <aterm> |
     <afactor> ‘^’ <aterm> | <afactor> ‘%’ <aterm> | <afactor>
11.  <aexp> ::= <aterm> ‘+’ <aexp> | <aterm> ‘-’ <aexp> |<aterm>
12.  <bcomparison> ::= <aexp> ‘<’ <aexp> | <aexp> ‘<=’ <aexp> |
     <aexp> ‘>’ <aexp> | <aexp> ‘>=’ <aexp> |
     <aexp> “==” <aexp>” | <aexp> “!=” <aexp>”
13.  <bfactor> ::= ‘true’ | ‘false’ | ‘!’ <bfactor> | ‘(’ <bexp> ‘)’ |
     <bcomparison>
14.  <bterm> ::= <bfactor> ‘AND’ <bterm> | <bfactor>
15.  <bexp> ::= <bterm> ‘OR’ <bexp> | <bterm>
16.  <arrayContent := <aexp> ‘ , ’ <arrayContent | <aexp>
17.  <array> := ‘[‘ <arrayContent ‘]’
18.  <matrixContent> := <array> ‘ , ’ <matrixContent> | <array>
19.  <matrix> := ‘[‘ <matrixContent> ‘]’
20. <assignment> ::= <identifier> = <aexp>;| <identifier>[<aexp>] =
    <aexp>; | <identifier> = <array> ; |<identifier> = <matrix>;
21.  <if-Else> ::= ‘if’ <bexp> ‘{’ <program> ‘}’ | ‘if’ <bexp> ‘{’
     <program> ‘}’ ‘else’ ‘{‘ <program> ‘}’
22.  <while> ::=  ‘while’ <bexp> ‘{’ <program> ‘}’
23.  <for> ::= ‘for’  ‘(‘ <assignment> ‘;’ <bexp> ‘;’ <assignment> ‘)’  ‘{‘
     <program> ‘}’
24.  <switch> := ‘switch’ ‘(‘ aexp ‘)’ ‘{’<case_stmt>’}’
25.  <case_stmt> :=  ‘case’ <integer> ‘:’ <program> <case_stmt> |
     ‘default’ ‘:’ <program>

26.  <command> ::= <assignment> | ‘skip’ | <if-Else> | <while>
       | <for> | <switch>
27.  <program> ::= <command> | <command> <program>
```

3. Environment

To keep track of the variables and their values used during the program execution, we introduce a new datatype, called *Variable*, which store info about a variable. *Variable* datatype is implemented as a structure composed by three fields: the variable name, the type, and the value. As we can see, the value of a variable is a list of lists, so to be able to handle also array and metrices values.

```
data Variable = Variable {

    name :: String,
    type :: String,
    value :: [[Numeric]] }
    deriving Show
```

A program could contain more variables, so we need a data structure to store them. Therefore, we create the type *Environment* which represent a list of variables.

```
type Env = [Variable]
```

The environment can be seen as a memory, which state is modified by the assignment instruction. The operations implemented to manipulate the Environment are:

- Search a variable, array, or matrix by name.
- Read a value from a variable, array, or matrix.
- Write a new variable in the environment.
- Overwrite the value of an existing variable or of an array element.

In the *Environment_management.hs* module there is the code regarding the environment. It is also reported here for convenience:

```
-- Return the value of a variable given the name (and given the indices
in case of array or matrix)

readVariable :: String -> Parser Numeric

readVariable name = P (\env input -> case searchVariable env name of

        [[]] -> []

        [[value]] -> [(env, value, input)])


-- needed to modify an array element

readWholeArray :: String -> Parser [Numeric]

readWholeArray name = P (\env input -> case searchVariable env name of
        [[]] -> []
        [x:xs] -> [(env, x:xs, input)])


-- read single array elements eg.x[i]
readArrayVariable :: String -> Int -> Parser Numeric
readArrayVariable name j = P (\env input -> case searchArrayVariable env
name j of
        [[]] -> []
        [[value]] -> [(env, value, input)])


-- read single matrix elements eg.x[i][j]
readMatrixVariable :: String -> Int -> Int -> Parser Numeric
readMatrixVariable name j k = P (\env input -> case searchMatrixVariable
env name j k of
        [[]] -> []
        [[value]] -> [(env, value, input)])
-- Search the value of a variable given the name (and indices in case of
array or matrix)
-- if the env list is empty there is no variable, if it isn't empty check
the head and if it isn't the searched variable check recursively the tail
searchVariable :: Env -> String -> [[Numeric]]
searchVariable [] queryname = []
searchVariable (x:xs) queryname|(name x == queryname) && (vtype x ==
"Array") = [(value x) !! 0]          --get: [[Numeric]], [Numeric]
|name x == queryname = [[((value x) !! 0) !! 0]]    --get: [[Numeric]],
                                                        [Numeric], Numeric
|otherwise = searchVariable xs queryname


-- search single array elements eg.x[i]
searchArrayVariable :: Env -> String -> Int -> [[Numeric]]
searchArrayVariable [] queryname j = [[]]
searchArrayVariable (x:xs) queryname j = if ((name x) == queryname) then
[[((value x) !! 0) !! j]]   --takes the element j of the only list in
[[int]]
else searchArrayVariable xs queryname j
```

```
-- search single matrix elements eg.x[i][j]
searchMatrixVariable :: Env -> String -> Int -> Int -> [[Numeric]]

searchMatrixVariable [] queryname j k = [[]]

searchMatrixVariable (x:xs) queryname j k = if ((name x) == queryname)

then [[((value x) !! j) !! k]]  --takes the element at row j and column k
(takes list j in
-                                 --[[int]] and then its element k)

else searchMatrixVariable xs queryname j k


-- replace an element in an array with a new one

replace :: Int -> a -> [a] -> [a]

replace pos newVal list | length list >= pos = take pos list ++ newVal :
drop (pos+1) list

                        | otherwise = list


-- Update the environment with a variable

updateEnv :: Variable -> Parser String

updateEnv var = P (\env input -> case input of

        xs -> [(modifyEnv env var,"",xs)])


-- If the variable is new, it is added to the environment, if the
variable already exist, its value will be overwritten

modifyEnv :: Env -> Variable -> Env

modifyEnv [] var = [var]

modifyEnv (x:xs) newVar = if name x == name newVar then newVar : xs else
x : modifyEnv xs newVar


--print the environment content without type information

showMemoryState :: Env -> String

showMemoryState [] = []

showMemoryState (x:xs) | vtype x == "Numeric" = name x ++ "=" ++
numericToString (value x !! 0 !! 0) ++ " " ++ showMemoryState xs

| vtype x == "Array" = name x ++ "=" ++ "[" ++ arrayNumericToString
(value x !! 0) ++ " " ++ showMemoryState xs     --(value x !! 0) take the
                                                 first list in [[Numeric]]
```

```
| otherwise = name x ++ "=" ++ "[" ++ matrixNumericToString (value x) ++
"] " ++ " " ++ showMemoryState xs  --vtype is a matrix so (value x) take
                                            all [[Numeric]]
--print raw environment content
getMemory :: [(Env, String, String)] -> String
getMemory [] = "Invalid input\n"
getMemory [(x:xs, parsedString, "")] = vtype x ++ "  " ++ name x ++ " = "
++ show (value x) ++ "\n" ++ getMemory [(xs,parsedString,"")]
getMemory [(env, parsedString, unparsedString)] = case unparsedString of
    "" -> ""
    _  -> "Error (unused input '" ++ unparsedString ++ "')\n" ++
getMemory [(env,parsedString, "")]
```

4. Parsing and execution

As suggested in [4], a parser can be seen as a function that takes a string as input and produces a result of a generic type as output. In general, a parser could not always consume its entire input, for this reason, we generalize the parser type to also return any unconsumed part of the input. Moreover, during the program execution, the environment could change, then the parser should also provide the final state of the environment. Given those facts we can represent our parser in Haskell in the following way:

```
newtype Parser a = P (Env -> String -> [(Env, a, String)])
```

So, we have a new parametric type *Parser a* which uses a dummy constructor P which take in input a function taking the actual environment, the program string to parse, and returns a triple consisting of an (eventually) changed environment, the parsing output and the unparsed input.

4.1 Parser as functor, applicative and monad

With *newtype,* through the dummy constructor P, we can make our Parser an instance of the Functor, Applicative, Monad and Alternative classes, so to use some useful operators.

- A Functor allows to apply a function to a wrapped value. We use it to apply a function to the result of a Parser if it succeeds, otherwise propagating the failure.
- An Applicative allows to apply a wrapped function to a wrapped value. We use it to apply a parser that returns a function to a parser that returns an argument, so to have a parser that returns the result of the application of the function to the argument.
- A Monad allows to apply a function, that return a wrapped value, to a wrapped value. We use it to apply a function *f* (that return a parser) to the output *v* of the parser *p* and then have a new parser *f v*
- With the Alternative we can use of the operator <|> to simulate choices between parsers; more specifically, returns the output of the first parser if it succeeds, otherwise we apply the same input to the second parser.
- we can have a more readable code using the **do** notation instead of cascading **bind** (**>>=**) symbols.

The following code, in *Core.hs* module, make our Parser an instance of the Functor, Applicative, Monad and Alternative classes:

```haskell
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap g p = P (\env inp -> case parse p env inp of
        [] -> []
        [(env, v, out)] -> [(env, g v, out)])

instance Applicative Parser where
    -- pure :: a -> Parser a
    pure v = P (\env inp -> [(env, v,inp)])

    -- <*> :: Parser (a -> b) -> Parser a -> Parser b
    pg <*> px = P (\env inp -> case parse pg env inp of
        [] -> []
        [(env, g, out)] -> parse (fmap g px) env out)
```

```haskell
instance Monad Parser where
-- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = P (\env inp -> case parse p env inp of
        [] -> []
        [(env, v, out)] -> parse (f v) env out)
   return a = P (\env cs -> [(env, a, cs)])

instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\env inp -> [])
    -- (<|>) :: Parser a -> Parser a -> Parser a
    p <|> q = P (\env inp -> case parse p env inp of
        [] -> parse q  env inp
        [(envout, v,out)] -> [(envout, v,out)])
```

4.2 Utility parsers

In the *Core.hs* module we can find useful basic parsers to parse certain input strings, for example parsers for single digits, letters, spaces, alphanumeric identifiers, natural, integer and floating point numbers.

```haskell

digit :: Parser Char
digit = satisfy isDigit


letter :: Parser Char
letter = satisfy isAlpha


space :: Parser ()             --es. parse space " abc" --> [((),"abc")]
space = do {
        Core.many (satisfy isSpace);
        return ();
        }


ident :: Parser String            --es. parse ident "abc def" produces
[("abc"," def")]
ident = do {
        x <- lower;
        xs <- Core.many alphanum;
        return (x:xs);
        }

nat :: Parser Int             --es. parse nat "123 abc" --> [(123,"
abc")]
nat = do {
        xs <- Core.some digit;
        return (read xs);
        }

int :: Parser Int          --es. parse int "-123 abc" --> [(-123," abc")]
int = do{
          char '-';
          n <- nat;     --nat return the number if the parse is ok
          return (-n);
        }
        Core.<|>
         do{
           char '+';
           nat         --same as 'return nat'
         }
    Core.<|> nat

-- parser for floating point numbers eg. 13.9
numberFloat :: Parser Float
numberFloat = do{
                numbersBeforeComma <- Core.many digit;
                char '.';
                numbersAfterComma <- Core.many digit;
                return (read
(numbersBeforeComma++"."++numbersAfterComma))
                }

--parser for positive and negative floating point numbers eg.+13.9, -2.34
numberFloatWithSign :: Parser Float
numberFloatWithSign = do {
  symbol "+";
  numberFloat;
 } Core.<|> do {
  symbol "-";
  num <- numberFloat;
  return (-num);
 } Core.<|> numberFloat;
```

Moreover, are implemented some utility functions for the Numeric type management, arrays and matrices printing:

```
numericToInt:: Numeric ->  Int
numericToInt (I x) = x
numericToInt (F x) = round x

numericToFloat:: Numeric -> Float
numericToFloat (F x) =  x
numericToFloat (I x) =  fromIntegral x
```

```
--check if a float is an integer e.g. 17.0
checkInt :: Float -> Bool
checkInt n = floor n == ceiling n

--print int and float. Float ending with .0 will be printed as int
numericToString :: Numeric -> String
numericToString (F x) = if checkInt (numericToFloat (F x)) then show
(numericToInt (F x)) else show (numericToFloat (F x))
numericToString (I x) = show (numericToInt (I x))

--format an array to print it
arrayNumericToString:: [Numeric] ->  String
arrayNumericToString [] = "]"    --if it is the last recursion step

arrayNumericToString (x:xs) | checkInt (numericToFloat x) =  if null xs
then numericToString x ++ "" ++ arrayNumericToString xs else
numericToString x ++ ", " ++ arrayNumericToString xs
                            | otherwise = if null xs then numericToString
x ++ "" ++ arrayNumericToString xs else numericToString x ++ ", " ++
arrayNumericToString xs

--format a matrix to print it
matrixNumericToString:: [[Numeric]] ->  String
matrixNumericToString [] = []
matrixNumericToString (x:xs) = "[" ++ if null xs then
arrayNumericToString x ++ "" ++ matrixNumericToString xs else
arrayNumericToString x ++ "," ++ matrixNumericToString xs
```

4.3 Arrays and matrices

Our interpreter can also manage arrays and matrices of Numeric type, as described in the language grammar definition in Chapter 2. The arrays are implemented as lists of numbers, while matrices as list of lists of numbers.

In particular, it's possible to:

- Assign a non-empty array to a variable e.g. x=[1.2, 5, 9];
- Assign a non-empty matrix to a variable e.g. k=[[1.2, 5], [6, 7]];
- Access arrays and matrix elements e.g. y=x[1]; z= k[0][1];
- Modify arrays single elements   e.g. x[1]= 2;

The definition of arrays and matrices parsers in the *Parse_and_execute.hs* module:

```
-- <array> := [ <arrayContent> ]
--parse the array brackets and (eventual)spaces after them
array :: Parser [Numeric]
array = do {
```

```
            char '[';                   --parse the [
            space;                      --parse spaces
            a <- arrayContent;          --parse the array content
            space;
            char ']';
            return a;
        }

-- <arrayContent> := <aexp> , <arrayContent> | <aexp>
--parse the array elements, spaces and commas after them
arrayContent :: Parser [Numeric]
arrayContent = do {
                a0 <- aexp;       --parse the first element of the array
                space;
                char ',';
                a1 <- arrayContent;    --parse the recursively the other
                                        elements
                return (a0 : a1);      -- return the parsed array content
        }
        <|> do{                  --in case of an array with only one element
                space;
                a0 <- aexp;
                space;
                return [a0];
        }

-- <matrix> := [ <matrixContent> ]
--parse the matrix external brackets and (eventual)spaces after them
matrix :: Parser [[Numeric]]
matrix = do {
                char '[';
                space;
                a <- matrixContent;
                space;
                char ']';
                return a;
        }

-- <matrixContent> := <array> , <matrixContent> | <array>
--parse all the arrays which composes the matrix
matrixContent :: Parser [[Numeric]]
matrixContent = do {
                space;
                a0 <- array;      --parse the first list(row) of the
                                    'list of list'(matrix)
                space;
                char ',';
                a1 <- matrixContent;   --parse the other rows,recursively
                return (a0 : a1);
        }
        <|> do{                       --if the matrix has only one row
                space;
                a <- array;
                space;
                return [a];
        }


--to parse 0 or more spaces
tokenArray :: Parser [Numeric]
tokenArray = token array

tokenMatrix :: Parser [[Numeric]]
tokenMatrix = token matrix
```

The operations performable with arrays and matrices are implemented in the following way:

```

-- <arrayAssignment> := <identifier> '=' <tokenArray> ';'

arrayAssignment :: Parser String

arrayAssignment = do{
                     x <- identifier;
                     symbol "=";
                     v <- tokenArray;
                     symbol ";";
                     updateEnv Variable{name=x,
                     vtype="Array", value=[v]}; --v=[Numeric]
                    }



-- <arrayValueAssignment> := <identifier>[i] '=' <aexp> ';'
arrayValueAssignment :: Parser String
arrayValueAssignment = do{
                            x <- identifier;
                            symbol "[";
                            i <- aexp;
                            symbol "]";
                            symbol "=";
                            v <- aexp;
                            symbol ";";
                            arr <- readWholeArray x;
  updateEnv Variable{name=x, vtype="Array", value= [replace (numericToInt
i) v arr]}; --replace the old array with the one containing the new value
                        }
```

```
-- <matrixAssignment> := <identifier> '=' <tokenMatrix> ';'
matrixAssignment :: Parser String
matrixAssignment = do{
                        x <- identifier;
                        symbol "=";
                        v <- tokenMatrix;
                        symbol ";";
    updateEnv Variable{name=x, vtype="Matrix", value= v}; --v=[[Numeric]]
                    }
```

The above parsers parse and execute the input strings, so we can also consider them interpreters. Parsers that only parse/consume the input strings are also needed (e.g in the conditional or iterative statements) and have been implemented in the *Only_parse.hs* module. The code is not showed here since it's similar to that of the parsers/interpreters, the only difference is that the parser simply returns the string it has read.

4.4 Arithmetic expressions

As we can see in the language grammar, the arithmetic expressions are composed by terms and factors, this allow to impose a priority ordering in the evaluation of the operators *, / , ^ , % with respect to + and - .

So, for the arithmetic expression parsing we have 3 parsers:

- *aexp* which manages additions, subtractions, or the expansions of a term.
- *aterm* manages *, / , ^ , % between a factor and another term.
- *afactor* manages the recursion on aexp surrounded by parentheses, the reading of variables, arrays and matrix elements or parsing an integer or float number.

```
--parser for arithmetic expressions, also evaluate them
-- aexp := <aterm> + <aexp> | <aterm> - <aexp> | <aterm>
Aexp :: Parser Numeric
aexp = do {
        do {
            t <- aterm;     --store the output of the 'aterm' parser in t
            symbol "+";      --parse the +
            a <- aexp;       --recursively parse other aexp
            return (F (numericToFloat t + numericToFloat a))
        }
        <|>
        do {
            t <- aterm;
            symbol "-";
            a <- aexp;
            return (F (numericToFloat t - numericToFloat a))
        }
        <|> aterm
      }

--aterm:= <afactor> * <aterm> | <afactor> / <aterm> | <afactor> ^ <aterm>
|<afactor> % <aterm> |<afactor>
aterm :: Parser Numeric
aterm = do {
            do {
                f <- afactor;
                symbol "*";
                t <- aterm;
                return (F (numericToFloat f * numericToFloat t))
            }
        <|>
            do{
                f <- afactor;
                symbol "/";
                t <- aterm;
                return (F (numericToFloat f / numericToFloat t))
            }
        <|>
            do{
                f <- afactor;
                symbol "^";
                t <- aterm;
                if checkInt (numericToFloat f) then
                   return (I (numericToInt f ^ numericToInt t))
                else
                   return (F(numericToFloat f ** numericToFloat))
            }
        <|>
            do{
                f <- afactor;
                symbol "%";
                t <- aterm;
                return (I (numericToInt f `mod` numericToInt t))
            }
        <|> afactor
        }

-- afactor := (<aexp>) | <identifier>[<aexp>][<aexp>] |
<identifier>[<aexp>] | <identifier> | <integer> | <float>
afactor :: Parser Numeric
afactor = do {
                do{
                    symbol "(";
                    a <- aexp;
```

```
                     symbol ")";
                     return a
                 }
                 <|>
                 do{
                     i <- identifier;
                     symbol "[";
                     j <- aexp;
                     symbol "]";
                     symbol "[";
                     k <- aexp;
                     symbol "]";
    readMatrixVariable i (numericToInt j) (numericToInt k); --read i[j][k]
                 }
                 <|>
                 do{
                     i <- identifier;
                     symbol "[";
                     j <- aexp;
                     symbol "]";
                     readArrayVariable i (numericToInt j);
                 }
                 <|>
                 do{
                     i <- identifier;
                     readVariable i;
                 }
                 <|> fmap F float --F is a function, float a parser
                 <|> fmap I integer
             }
```

As we can see the arithmetic operations (except the modulus %) are performed using floating point numbers and then the result is wrapped in the Numeric datatype using the F and I constructors. In this way the interpreter can perform both integer and float operations and also handle float results, as in the case of the division operation between to integers.

4.5 Boolean expressions

Boolean expression management is performed in a similar way. We have four parsers/interpreters: bexp, bterm, bfactor and bcomparison. To remove ambiguity, AND operator gets precedence on OR operator. In bcomparison, aexp parser is used to evaluate arithmetic expressions if nested in the Boolean ones. Float numbers are rounded to Integer before comparison.

```
--parser for boolean expressions, also evaluate them

--bexp := <bterm> OR <bexp> | <bterm>
bexp :: Parser Bool
bexp = do{
            do{
                b0 <- bterm;
                symbol "OR";
                b1 <- bexp;
                return (b0 || b1);
            }
            <|> bterm
        }

-- bterm := <bfactor> AND <bterm> | <bfactor>
bterm :: Parser Bool
bterm = do{
            do{
                f0 <- bfactor;
                symbol "AND";
                f1 <- bterm;
                return (f0 && f1);
            }
            <|> bfactor
        }

-- bfactor := true | false | !<bfactor> | (bexp) | <bcomparison>
bfactor:: Parser Bool
bfactor = do{
                do{
                    symbol "true";
                    return True;
                }
                <|>
                do {
                    symbol "false";
                    return False;
                    }
                <|>
                do{
                    symbol "!";
                    fmap not bfactor;  --apply the not function to the
                                       -- bool wrapped in bfactor
                }
                <|>
                do{
                    symbol "(";
                    b <- bexp;
                    symbol ")";
                    return b;
                }
                <|> bcomparison
            }

-- bcomparison := <aexp> == <aexp> | <aexp> != <aexp> |<aexp> < <aexp> |
<aexp> <= <aexp> | <aexp> > <aexp> | <aexp> >= <aexp>
bcomparison:: Parser Bool
bcomparison = do {
                    do{
                        a0 <- aexp;
                        symbol "==";
                        a1 <- aexp;
                        return ( numericToInt a0 == numericToInt a1);
                    }
                    <|>
                    do{
                        a0 <- aexp;
                        symbol "!=";
                        a1 <- aexp;
                        return ( numericToInt a0 /= numericToInt a1);
                    }
                    <|>
                    do{
                        a0 <- aexp;
                        symbol "<";
                        a1 <- aexp;
                        return ( numericToInt a0 < numericToInt a1);
                    }
                    <|>
                    do{
                        a0 <- aexp;
                        symbol "<=";
                        a1 <- aexp;
                        return (numericToInt a0 <= numericToInt a1);
                    }
                    <|>
                    do{
                        a0 <- aexp;
                        symbol ">";
                        a1 <- aexp;
                        return ( numericToInt a0 > numericToInt a1);
                    }
                    <|>
                    do{
                        a0 <- aexp;
                        symbol ">=";
                        a1 <- aexp;
                        return ( numericToInt a0 >= numericToInt a1);
                    }
                }
```

4.6 Program and commands

In this section, we will introduce the modalities for parsing and evaluating programs as sequence of commands. The program parser/interpreter, parse and evaluate sequences of commands separated by a ‘;’

```
--Commands parsing and evaluation

-- program := <command> <program> | <command>
program :: Parser String
program = do {
               do {
                   command;
                   program;
               }
               <|> command
           }
```

The commands that our interpreter can manage are those listed below:

```
--command:= <assignment> | <arrayAssignment> | <arrayValueAssignment> |
<matrixAssignment> | <ifThenElse> | <switch> | <while> | <for> | skip ;
command :: Parser String
command = do{
               do assignment;
               <|>
               do arrayAssignment;
               <|>
               do arrayValueAssignment;
               <|>
               do matrixAssignment;
               <|>
               do ifThenElse;
               <|>
               do switch;
               <|>
               do while;
               <|>
               do for;
               <|>
               do{
                  symbol "skip";
                  symbol ";"
               }
           }
```

4.6.1 Assignments

We can assign to a variable: an arithmetic expression, a not-empty array or matrix, or reassign an array value. The syntax uses '= ' as notation for assigning the value, and a semicolon ';' to end the statement. Variables cannot be declared but just assigned. Trying to declare a variable or read a never assigned variable will throw an error.

```
-- assignment := <identifier> '=' <aexp>;
assignment :: Parser String
assignment = do{
                x <- identifier;
                symbol "=";
                v <- aexp;
                symbol ";";
                updateEnv Variable{name=x,vtype="Numeric", value= [[v]]};
            }
```

The code for the arrays and matrices assignment is the one showed above in the section 4.3 Array and matrices.

4.6.2 If-then-else

Conditional statements (if-then-else) in our language don't need parentheses around the Boolean condition, and the 'else' is not mandatory. At least one instruction must be present in a block. If a block is expected to be empty, we can put the 'skip;' command inside it.
If the condition is met, the block inside the first curly braces will be parsed and executed, and the block in the 'else' condition will be only parsed. If the condition is not met, the first block will be only parsed, while the 'else' block (if present) will be parsed and executed.

```
--ifThenElse:=if(<bexp>){<program>}|if(<bexp>){<program>} else{<program>}
ifThenElse :: Parser String
ifThenElse = do {
                symbol "if";
                b <- bexp;
                symbol "{";
                if b then do{   --parse and evaluate the program in the
                                'if' branch and only parse(consume) the
                                         program in the 'else' branch
                                program;
                                symbol "}";
                                do{
                                    symbol "else";
                                    symbol "{";
                                    consumeProgram;
                                    symbol "}";
                                    return "";
                                }
                                <|> return ""  --in case there is no
                                                 'else' branch
                            }
                else do{    --only parse(consume) the program in the 'if'
                               branch, parse and evaluate the program in
                               the 'else' branch(if it is present)
                            consumeProgram;
                            symbol "}";
                            do{
                                symbol "else";
                                symbol "{";
                                program;
                                symbol "}";
                                return "";
                            }
                            <|> return ""     --in case there is no
                                                     'else' branch
                        }
            }
```

4.6.3 Switch

The Switch works as a cascade of if-then-else but it offers a more readable syntax. In PImp, the Switch condition is an aexp which result will be compared with the case's labels. If the aexp result is a float it will be rounded to the closest integer and then compared with the case's labels (which can only be integer). If there is a match with a case's label then its commands will be executed and the rest of the Switch will only be parsed. If no case's label matches the aexp result, will be executed the commands in the *default* branch, which is mandatory. In the Switch block, can be defined an arbitrary number of 'case' but just one 'default' branch.

```
-- <switch>:= switch ( <aexp> ) { <case_stmt> }
switch :: Parser String
switch = do {
            symbol "switch";
            a <- aexp;          -- aexp already parse the ( )
            symbol "{";
            case_stmt a;    -- the Numeric 'a' will be compared with the
                               'case' condition
            symbol "}";
            }
```

```
--<case_stmt>:= case <integer>:<program> <case_stmt>|default:<program>
case_stmt :: Numeric -> Parser String
case_stmt a = do {
                symbol "case";
                i <- integer;
                symbol ":";
                if i == numericToInt a then   --if the 'case' condition
                                                match the aexp then its
                                              code is executed and the
                                            rest of the switch only parsed
                        do {
                            program;
                            consumeCaseStmt;
                            }
                else do {     --if the 'case' condition is not matched by
                                the aexp then its code is parsed and are
                                tried the next case
                        consumeProgram;
                        case_stmt a;
                        }
                }
            <|>
            do {     --if no 'case' condition is matched then is executed
                          the default branch
                symbol "default:";
                program;
            }
```

4.6.4 While loop

Is an iterative statement. If the Boolean condition is met, the parser parse and evaluates the block, adds the block back to unparsed string, and it calls another 'while' parser to evaluate the expression. When the Boolean condition is not met anymore, it only parses the block, and it returns a successful parser.

```
-- while := while (<bexp>) {<program>}
while :: Parser String
while = do {
            w <- consumeWhile;  -- parse the 'while'
            repeatBlock w;     --adds the block back to unparsed string
            symbol "while";
            b <- bexp;
            symbol "{";
            if b then do{
                        program;    --parse and evaluate the 'while' body
                        symbol "}";
                        repeatBlock w;
                        while;
                        }
            else do {
                    consumeProgram;     -- only parse the 'while' body
                    symbol "}";
                    return "";
                    }
        }


repeatBlock :: String -> Parser String
repeatBlock c = P(\env input -> [(env, "", c ++ input)])
```

4.6.5 For loop

Executes a loop for a fixed number of times, usually through a counter variable. There are in sequence an assignment, a Boolean expression, another assignment (executed after every loop) and the program block.
The "for" and "forLoop" share the same structure and consume strategy. The only difference is that the latter don't run the first assignment, which must be executed just once in the "for" command.

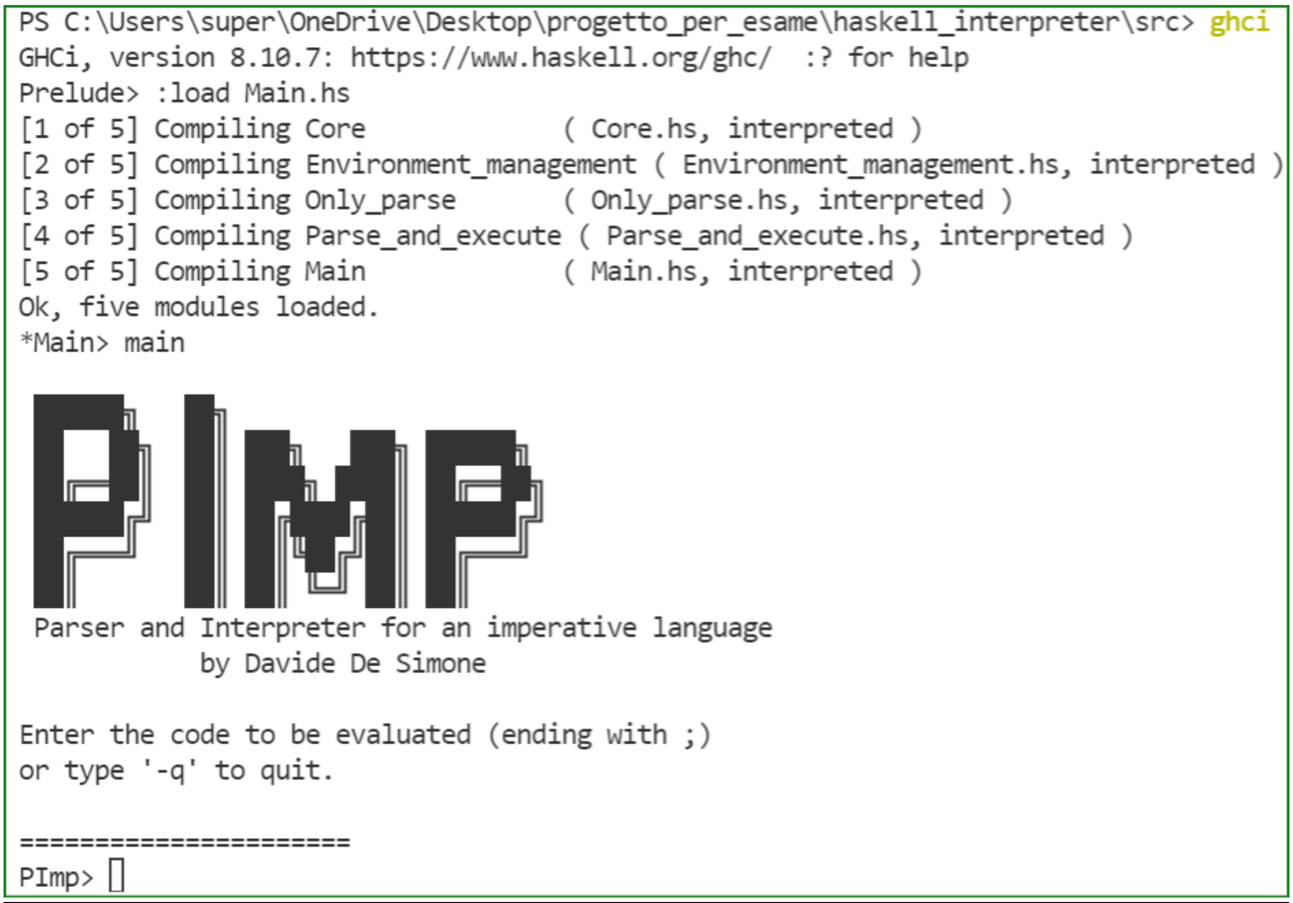
```
-- <for> := <consumeFor> for(<assignment> <bexp> ; <consumeAssignment>) {
<program> <assignment> } <forLoop> | <consumeFor> for (<assignment>
<bexp> ; <consumeAssignment>) { }
for :: Parser String
for = do{
        w <- consumeFor;        --only parse
        repeatBlock w;          --adds the block back to unparsed string
        symbol "for";
        symbol "(";
        assignment;       --already parse ';'
        b <- bexp;
        symbol ";";
        a <- consumeAssignment;
        symbol ")";
        symbol "{";
        if b then do{
                    program;         --parse and execute
                    repeatBlock a;
                    assignment;
                    symbol "}";
                    repeatBlock w;
                    forLoop;      --doesn't execute the first assignment,
                                    only parse it
                  }
        else do {
                consumeProgram;
                symbol "}";
```

```
               return "";
               }
    }

-- <forLoop> := <consumeFor> for(<consumeAssignment> <bexp> ;
<consumeAssignment> ) {<program> <assignment>} <forLoop> | <consumeFor>
for( <consumeAssignment> <bexp> ; <consumeAssignment> ) {<parseProgram>}
forLoop:: Parser String
forLoop = do{
               w <- consumeFor;
               repeatBlock w;
               symbol "for";
               symbol "(";
               consumeAssignment;
               b <- bexp;
               symbol ";";
               a <- consumeAssignment;
               symbol ")";
               symbol "{";
               if b then do{
                           program;
                           repeatBlock a;
                           assignment;
                           symbol "}";
                           repeatBlock w;
                           forLoop;
                           }
               else do{
                       consumeProgram;
                       symbol "}";
                       return "";
                       }
           }
```

5. Usage examples

To run the interpreter:

- start **ghci** in the **src** folder
- :load Main.hs
- call the **main** function


```
PS C:\Users\super\OneDrive\Desktop\progetto_per_esame\haskell_interpreter\src> ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> :load Main.hs
[1 of 5] Compiling Core                 ( Core.hs, interpreted )
[2 of 5] Compiling Environment_management ( Environment_management.hs, interpreted )
[3 of 5] Compiling Only_parse           ( Only_parse.hs, interpreted )
[4 of 5] Compiling Parse_and_execute ( Parse_and_execute.hs, interpreted )
[5 of 5] Compiling Main                 ( Main.hs, interpreted )
Ok, five modules loaded.
*Main> main
```

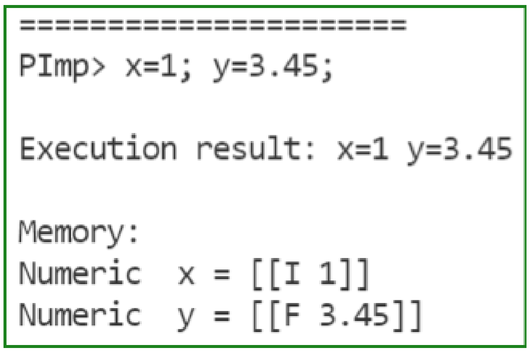
```
 Parser and Interpreter for an imperative language
             by Davide De Simone

Enter the code to be evaluated (ending with ;)
or type '-q' to quit.

======================
PImp> 
```

Some input examples for the interpreter can be found in the **test_interprete.txt** file. For convenience, we show the results in the following.

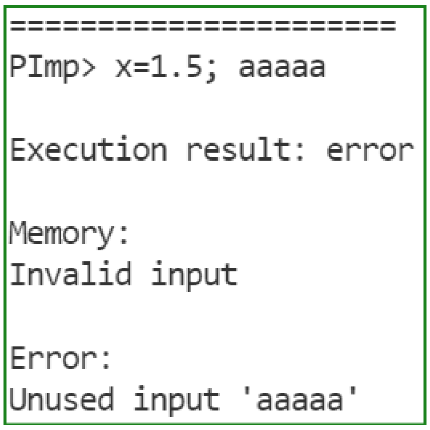
It is worth noting that Variables are not kept between program runs since the environment is discarded. The output of the interpreter is the pretty printed environment without type information ('Execution result') and a raw printed environment ('Memory') with type info.

➢ Variable assignment

We can see how the variables values are represented internally as list of lists of Numeric type, with **I** for Int and **F** for Float.


```
======================
PImp> x=1; y=3.45;

Execution result: x=1 y=3.45

Memory:
Numeric  x = [[I 1]]
Numeric  y = [[F 3.45]]
```

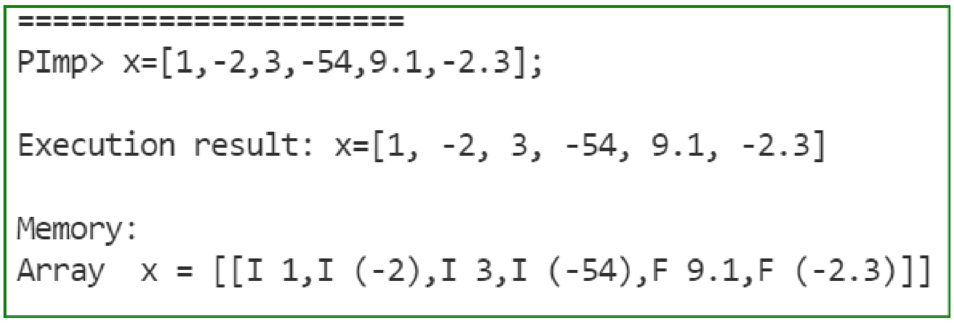
➢ Wrong input



```
======================
PImp> x=1.5; aaaaa

Execution result: error

Memory:
Invalid input

Error:
Unused input 'aaaaa'
```

➢ Array assignment

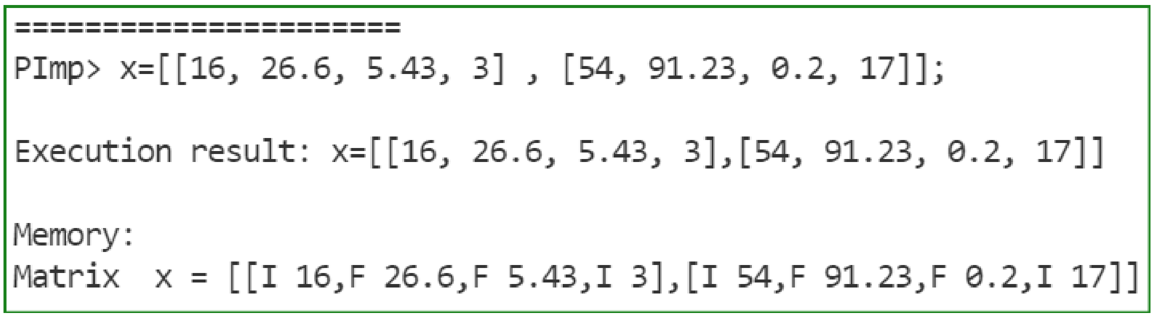
Pimp can mange arrays of Float and Int.



```
======================
PImp> x=[1,-2,3,-54,9.1,-2.3];

Execution result: x=[1, -2, 3, -54, 9.1, -2.3]

Memory:
Array  x = [[I 1,I (-2),I 3,I (-54),F 9.1,F (-2.3)]]
```

➢ Matrix assignment

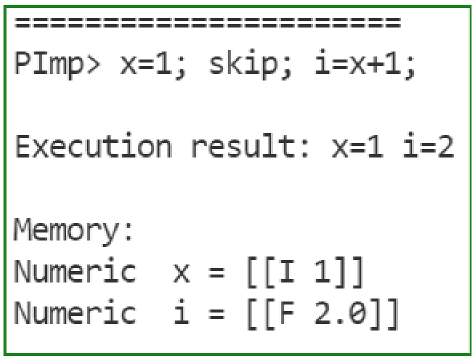
Matrices are defined as lists of lists.



```
======================
PImp> x=[[16, 26.6, 5.43, 3] , [54, 91.23, 0.2, 17]];

Execution result: x=[[16, 26.6, 5.43, 3],[54, 91.23, 0.2, 17]]

Memory:
Matrix  x = [[I 16,F 26.6,F 5.43,I 3],[I 54,F 91.23,F 0.2,I 17]]
```

➢ Variable access

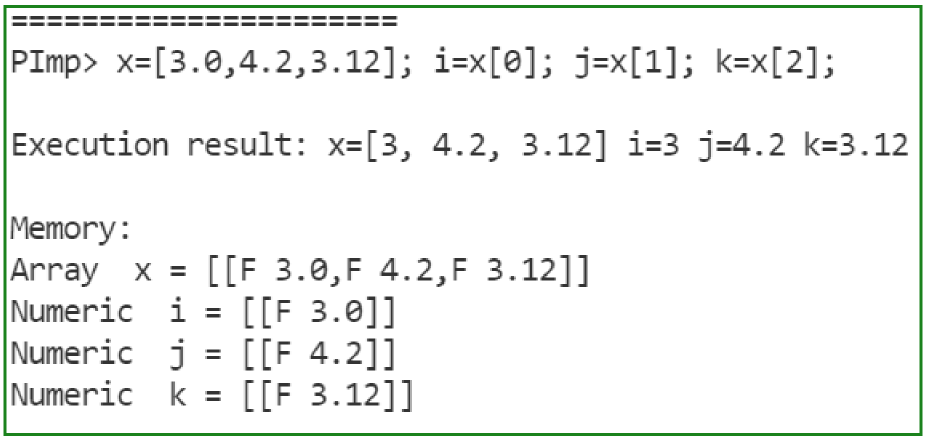
The skip does nothing, it’s only parsed.



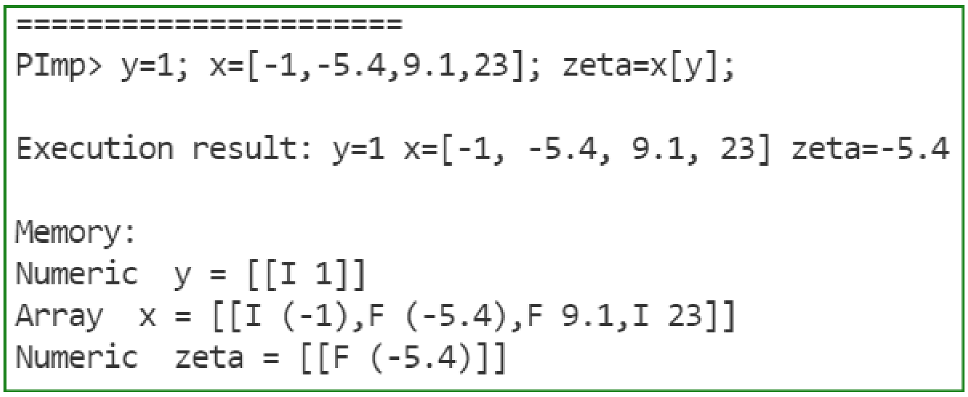
```
======================
PImp> x=1; skip; i=x+1;

Execution result: x=1 i=2

Memory:
Numeric  x = [[I 1]]
Numeric  i = [[F 2.0]]
```

➢ Array access

Array indices start from 0.



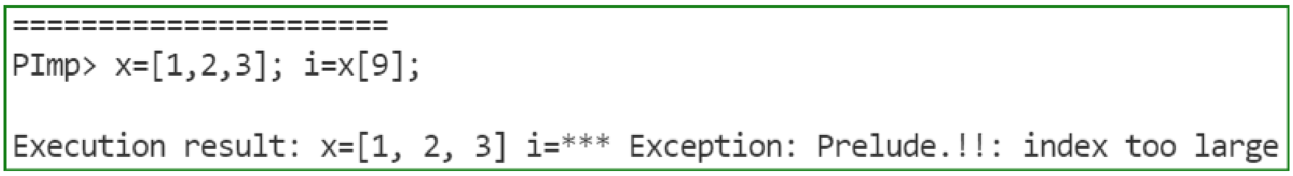
```
======================
PImp> x=[3.0,4.2,3.12]; i=x[0]; j=x[1]; k=x[2];

Execution result: x=[3, 4.2, 3.12] i=3 j=4.2 k=3.12

Memory:
Array  x = [[F 3.0,F 4.2,F 3.12]]
Numeric  i = [[F 3.0]]
Numeric  j = [[F 4.2]]
Numeric  k = [[F 3.12]]
```



```
======================
PImp> y=1; x=[-1,-5.4,9.1,23]; zeta=x[y];

Execution result: y=1 x=[-1, -5.4, 9.1, 23] zeta=-5.4

Memory:
Numeric  y = [[I 1]]
Array  x = [[I (-1),F (-5.4),F 9.1,I 23]]
Numeric  zeta = [[F (-5.4)]]
```

➢ Index too large

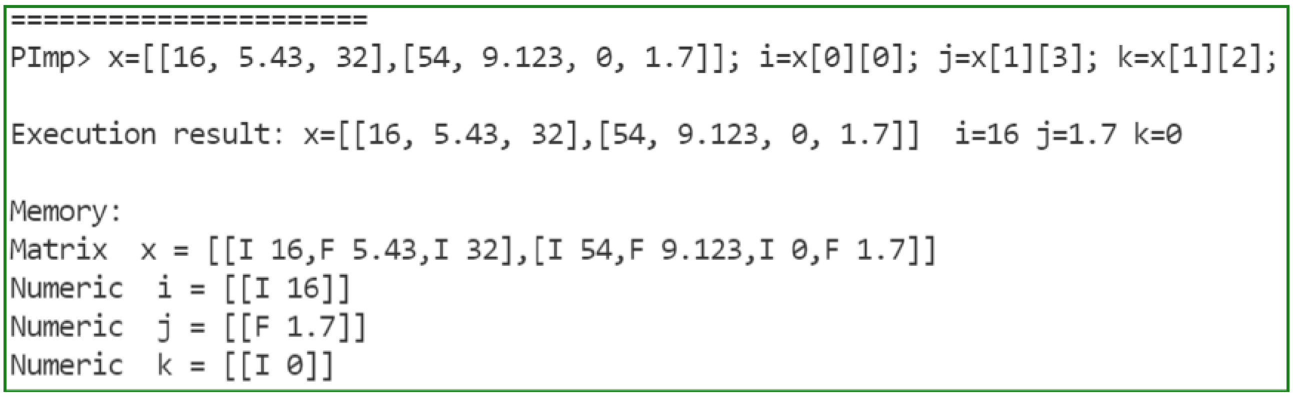
Trying to access a too large array index will result in an error, in this case we have to restart the interpreter calling the main function.



```
=======================
PImp> x=[1,2,3]; i=x[9];

Execution result: x=[1, 2, 3] i=*** Exception: Prelude.!!: index too large
```

➢ Matrix access

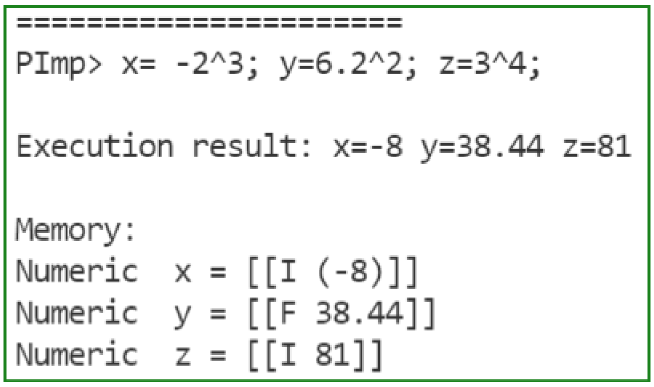
Matrices indices start from 0.



```
=======================
PImp> x=[[16, 5.43, 32],[54, 9.123, 0, 1.7]]; i=x[0][0]; j=x[1][3]; k=x[1][2];

Execution result: x=[[16, 5.43, 32],[54, 9.123, 0, 1.7]]  i=16 j=1.7 k=0

Memory:
Matrix  x = [[I 16,F 5.43,I 32],[I 54,F 9.123,I 0,F 1.7]]
Numeric  i = [[I 16]]
Numeric  j = [[F 1.7]]
Numeric  k = [[I 0]]
```

➢ Arithmetic expressions

We can perform addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), modulus (%). Internally all operations (except % and ^ between Int) are carried out using Floats.



```
=======================
PImp> x= -2^3; y=6.2^2; z=3^4;

Execution result: x=-8 y=38.44 z=81

Memory:
Numeric  x = [[I (-8)]]
Numeric  y = [[F 38.44]]
Numeric  z = [[I 81]]
```

```
======================
PImp> x= 7 % 9 + 4 % 2;

Execution result: x=7

Memory:
Numeric  x = [[F 7.0]]

======================
PImp> x= 1 + 2*20 /5 -9;

Execution result: x=0

Memory:
Numeric  x = [[F 0.0]]

======================
PImp> x= 7+4+1/5-20/100;

Execution result: x=11

Memory:
Numeric  x = [[F 11.0]]

======================
PImp> x= 21/(47+21)-3;

Execution result: x=-2.6911764

Memory:
Numeric  x = [[F (-2.6911764)]]
```

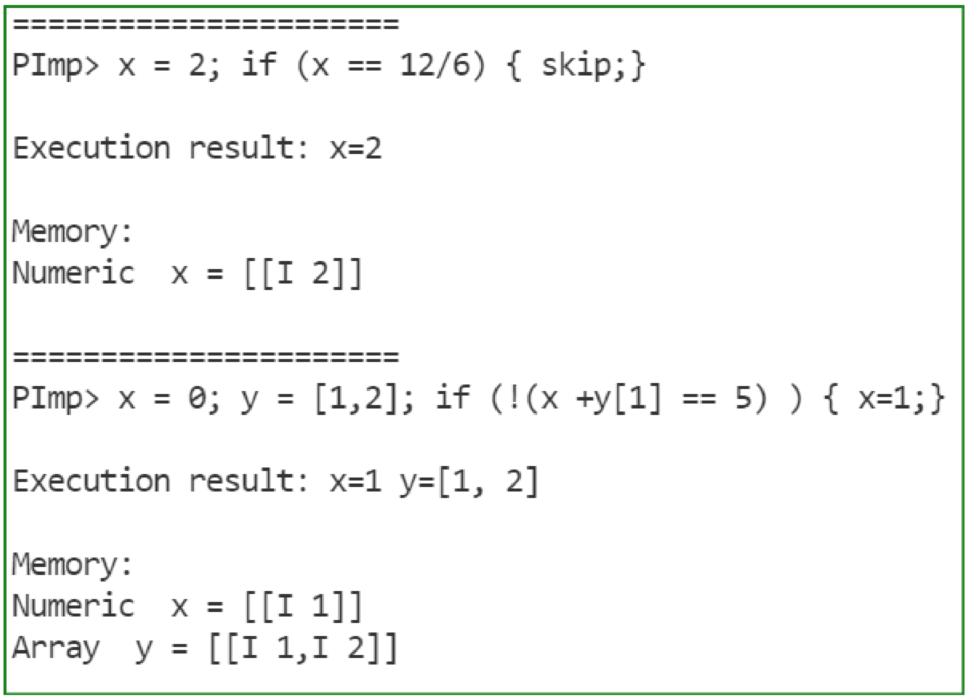
```
======================
PImp> x= 6-8.9;

Execution result: x=-2.8999996

Memory:
Numeric  x = [[F (-2.8999996)]]

======================
PImp> x= 2/3;

Execution result: x=0.6666667

Memory:
Numeric  x = [[F 0.6666667]]
```

We can also use array or matrix elements as operands:

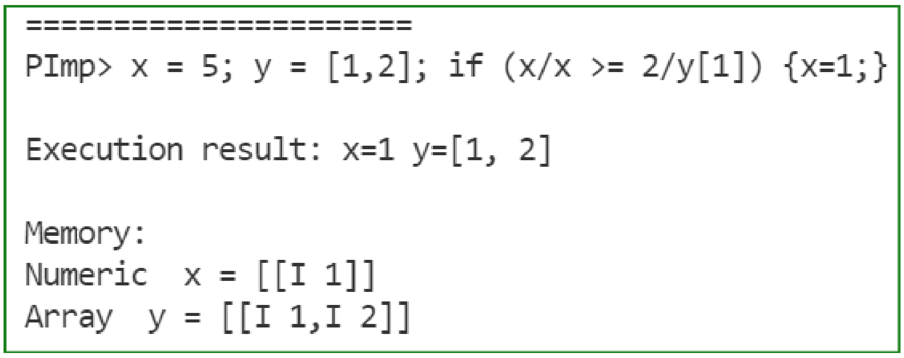
```
======================
PImp> x=[[-1,2,-3.2],[6,-5.2,4]]; y=[7,8,9]; i=x[0][2] * y[1] + x[1][2] -12 * x[0][0];

Execution result: x=[[-1, 2, -3.2],[6, -5.2, 4]]  y=[7, 8, 9] i=-9.6

Memory:
Matrix  x = [[I (-1),I 2,F (-3.2)],[I 6,F (-5.2),I 4]]
Array  y = [[I 7,I 8,I 9]]
Numeric  i = [[F (-9.6)]]
```

➢ Boolean expressions

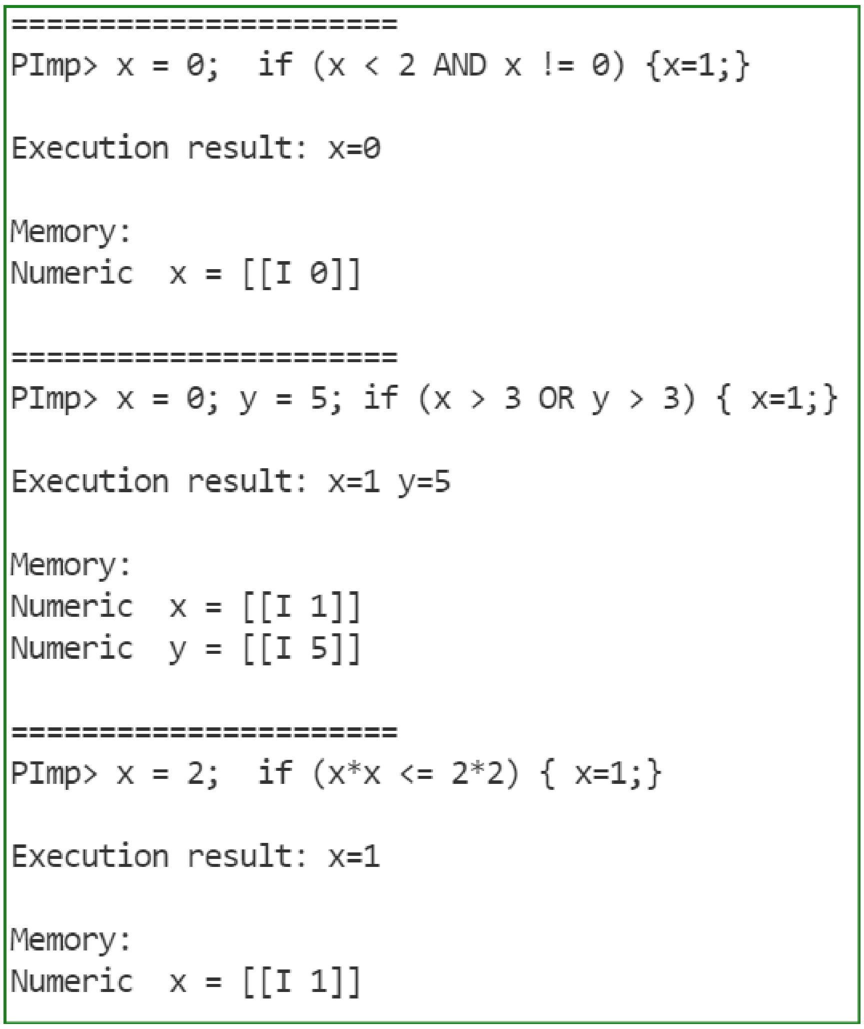
We cannot assign bexp to variables, but we can use them inside commands condition e.g. if, switch for, while. In PImp, bexp can include AND, OR, not (!), equality (==), inequality (!=), less (<), greater (>), less or equal (<=), greater or equal (>=).



```
======================
PImp> x = 2; if (x == 12/6) { skip;}

Execution result: x=2

Memory:
Numeric  x = [[I 2]]

======================
PImp> x = 0; y = [1,2]; if (!(x +y[1] == 5) ) { x=1;}

Execution result: x=1 y=[1, 2]

Memory:
Numeric  x = [[I 1]]
Array  y = [[I 1,I 2]]
```

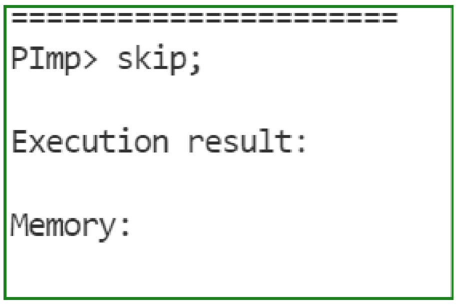


```
======================
PImp> x = 5; y = [1,2]; if (x/x >= 2/y[1]) {x=1;}

Execution result: x=1 y=[1, 2]

Memory:
Numeric  x = [[I 1]]
Array  y = [[I 1,I 2]]
```



```
======================
PImp> x = 0;  if (x < 2 AND x != 0) {x=1;}

Execution result: x=0

Memory:
Numeric  x = [[I 0]]

======================
PImp> x = 0; y = 5; if (x > 3 OR y > 3) { x=1;}

Execution result: x=1 y=5

Memory:
Numeric  x = [[I 1]]
Numeric  y = [[I 5]]

======================
PImp> x = 2;  if (x*x <= 2*2) { x=1;}

Execution result: x=1

Memory:
Numeric  x = [[I 1]]
```

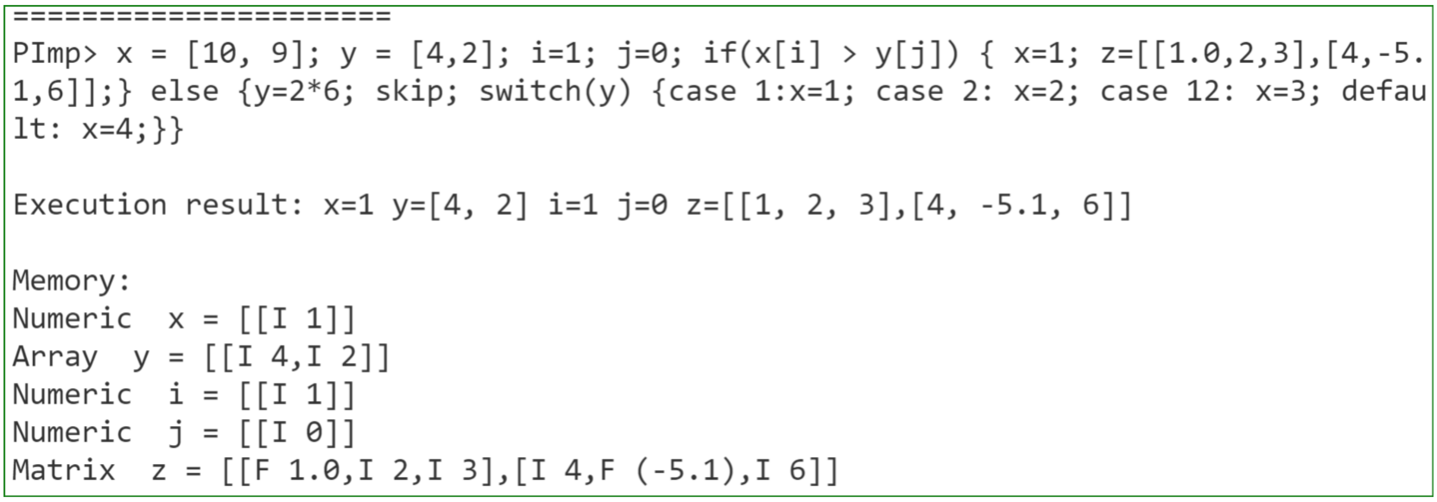
➢ **skip**



```
======================
PImp> skip;

Execution result:

Memory:
```

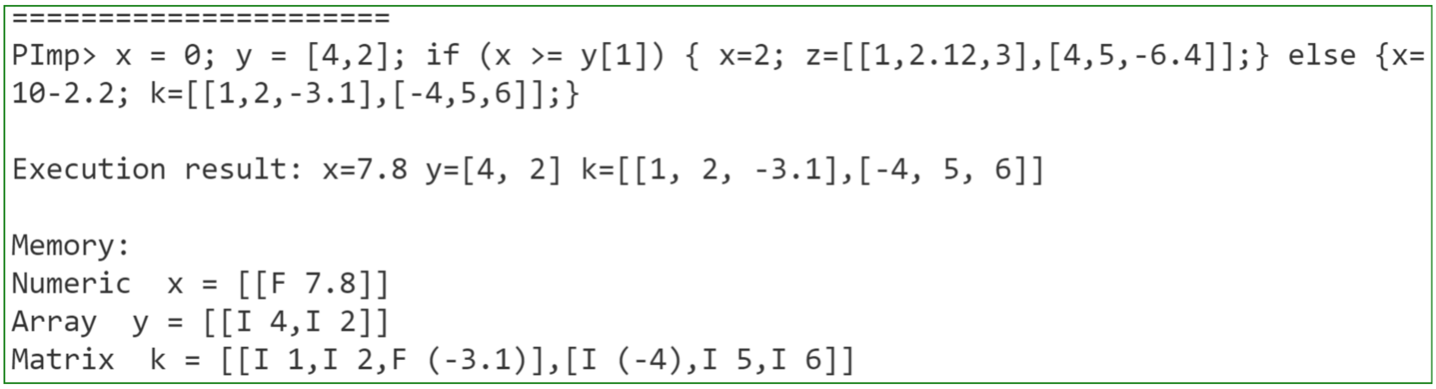
➢ If-else

In this case the *if* branch is executed and the *else* branch only parsed.



```
=======================
PImp> x = [10, 9]; y = [4,2]; i=1; j=0; if(x[i] > y[j]) { x=1; z=[[1.0,2,3],[4,-5.
1,6]];} else {y=2*6; skip; switch(y) {case 1:x=1; case 2: x=2; case 12: x=3; defau
lt: x=4;}}

Execution result: x=1 y=[4, 2] i=1 j=0 z=[[1, 2, 3],[4, -5.1, 6]]

Memory:
Numeric  x = [[I 1]]
Array  y = [[I 4,I 2]]
Numeric  i = [[I 1]]
Numeric  j = [[I 0]]
Matrix  z = [[F 1.0,I 2,I 3],[I 4,F (-5.1),I 6]]
```

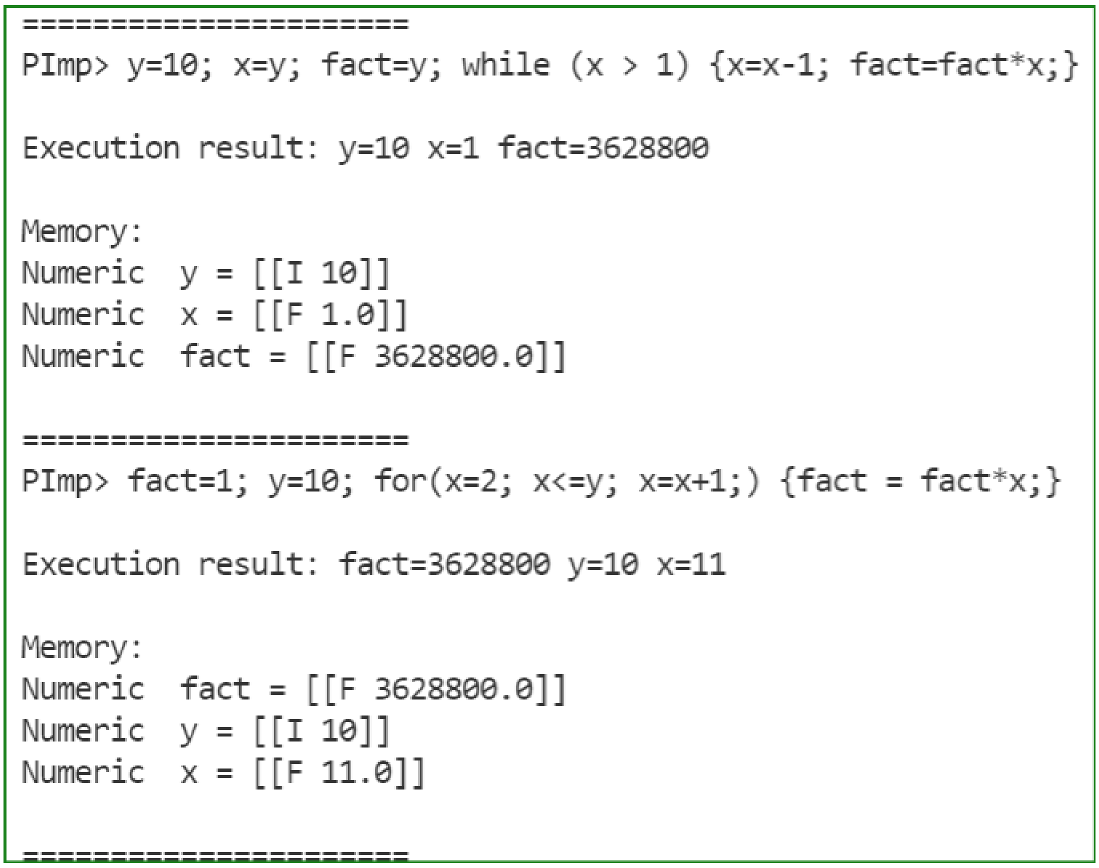
In this case the *if* branch is only parsed and the *else* branch executed.



```
=======================
PImp> x = 0; y = [4,2]; if (x >= y[1]) { x=2; z=[[1,2.12,3],[4,5,-6.4]];} else {x=
10-2.2; k=[[1,2,-3.1],[-4,5,6]];}

Execution result: x=7.8 y=[4, 2] k=[[1, 2, -3.1],[-4, 5, 6]]

Memory:
Numeric  x = [[F 7.8]]
Array  y = [[I 4,I 2]]
Matrix  k = [[I 1,I 2,F (-3.1)],[I (-4),I 5,I 6]]
```

➢ While and for

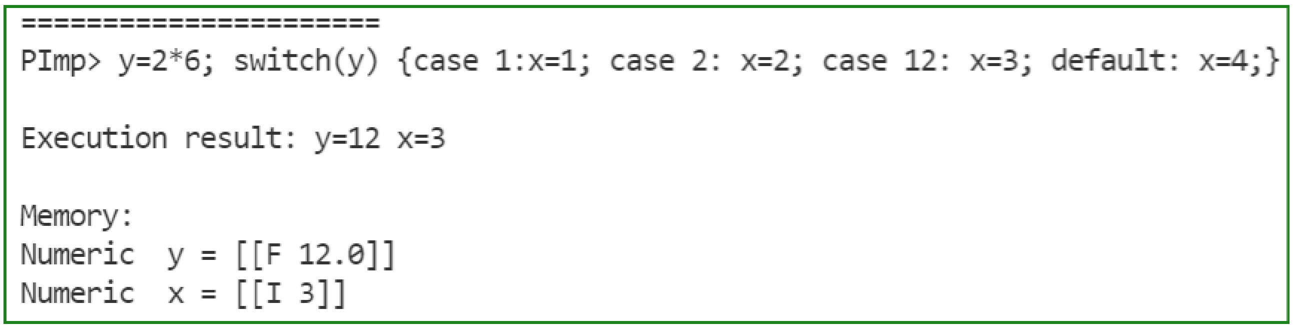
Calculating the factorial of 10 using the *while* and *for* loop:



```
======================
PImp> y=10; x=y; fact=y; while (x > 1) {x=x-1; fact=fact*x;}

Execution result: y=10 x=1 fact=3628800

Memory:
Numeric  y = [[I 10]]
Numeric  x = [[F 1.0]]
Numeric  fact = [[F 3628800.0]]

======================
PImp> fact=1; y=10; for(x=2; x<=y; x=x+1;) {fact = fact*x;}

Execution result: fact=3628800 y=10 x=11

Memory:
Numeric  fact = [[F 3628800.0]]
Numeric  y = [[I 10]]
Numeric  x = [[F 11.0]]

======================
```

➢ Switch

Simple assignments in the case branches:



```
======================
PImp> y=2*6; switch(y) {case 1:x=1; case 2: x=2; case 12: x=3; default: x=4;}

Execution result: y=12 x=3

Memory:
Numeric  y = [[F 12.0]]
Numeric  x = [[I 3]]
```

Aexp in 'case' branches:

```
=======================
PImp> y=21/7; switch(y-1) {case 1: x=1 +4 -6.7; z=[2.1,41]; case 2: x=2 * 4 % 3; c
ase 3: x=3/9; k=[[-2,3],[6.43,-1.2]]; default: x=4%2;}

Execution result: y=3 x=2

Memory:
Numeric  y = [[F 3.0]]
Numeric  x = [[F 2.0]]
```

Complex constructs in 'case' branches:

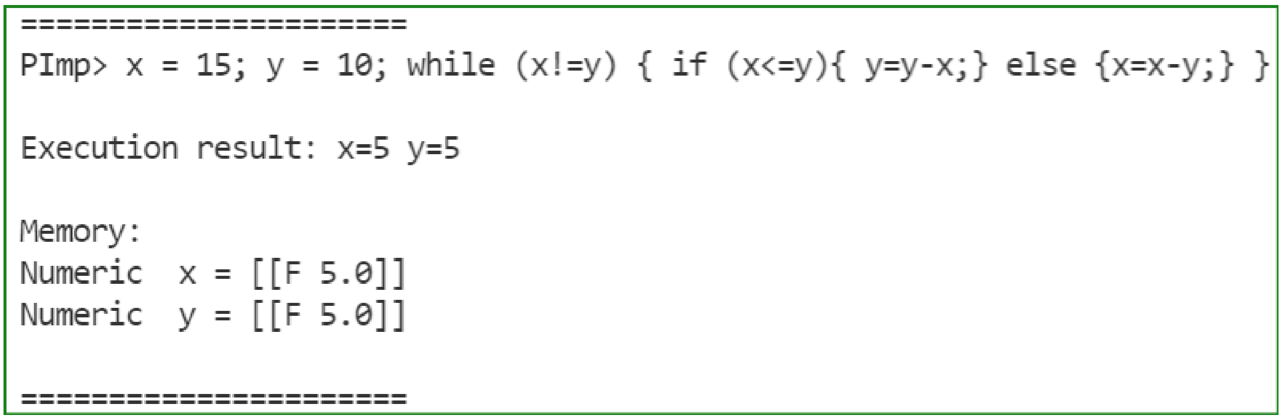
```
=======================
PImp> y=3.8; switch(y) {case 1: while(y<8){y=y+1;} case 2:if(y>29){x=1;} else{k=5;
} default:fact=1; y=10; for(x=1; x<=y; x=x+1;) {fact=fact*x;}}

Execution result: y=10 fact=3628800 x=11

Memory:
Numeric  y = [[I 10]]
Numeric  fact = [[F 3628800.0]]
Numeric  x = [[F 11.0]]
```

➢ Array sorting using the for statement

```
=======================
PImp> x=[4,8,1]; temp=0; for(i=0; i<2; i=i+1;) {for(j=i+1; j<3; j=j+1;) {if(x[i] >
 x[j]) {temp=x[i]; x[i]=x[j]; x[j]=temp;}}}

Execution result: x=[1, 4, 8] temp=8 i=2 j=3

Memory:
Array  x = [[I 1,I 4,I 8]]
Numeric  temp = [[I 8]]
Numeric  i = [[F 2.0]]
Numeric  j = [[F 3.0]]
```

➢ GCD

Euclidean algorithm for finding the greatest common divisor of two integers.



```
======================
PImp> x = 15; y = 10; while (x!=y) { if (x<=y){ y=y-x;} else {x=x-y;} }

Execution result: x=5 y=5

Memory:
Numeric  x = [[F 5.0]]
Numeric  y = [[F 5.0]]

======================
```

6. Conclusions

This project was useful to understand how a functional language like Haskell works and how it can be used to build a parser and an interpreter for a language defined by us.

The monadic implementation of the parser gave us the possibility of concatenating multiple parsers of sub-grammars. It's also possible to extend the grammar of the language to include other statements.

Future works may include the introduction of other variable types or adding useful information about the parsing errors, i.e. missing tokens and the location in the source code where the error occurred.

7. References

[1] En.wikipedia.org. 2022. *Parsing - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Parsing#Parser> [Accessed 2 March 2022].

[2] En.wikipedia.org. 2022. *Interpreter (computing) - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Interpreter_(computing)> [Accessed 2 March 2022].

[3] En.wikipedia.org. 2022. Evaluation strategy - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Evaluation_strategy> [Accessed 3 March 2022].

[4] HUTTON, G., 2016. Programming in Haskell. 2nd ed. Cambridge: Cambridge University Press.

[ ] Material provided in the course "Formal Methods in Computer Science"