IMPterpreter

Table of contents

1. Introduction

An interpreter is a program that directly analyse and execute instructions written in a certain programming language. In this project was built a simple interpreter in Haskell. IMPterpreter, as the name suggests, interprets a simple imperative programming language called IMP.

The basic constructs of IMP are:

- Skip: does nothing
- Assignment: evaluate an expression and assign it to a variable
- If then else: Performs the conditional selection between two paths
- While: Repeat a sequence of instructions until a boolean condition is satisfied

The followed strategy is based on the one given in the book *Programming in Haskell – Second Edition" by Graham Hutton*.
IMPterpreter uses an eager evaluation (call-by-value), so in functions calls like g(f(x)), is evaluated first f(x), then the ground result is given to g.

2. Grammar

W.t.r. to the original grammar of IMP, it has been modified to allow the representation of real numbers and arrays. The produced grammar is the following:

```
Program := [<Command>]*
Command ::= <Skip> | <Assignment> | <IfThenElse> | <While>
Skip ::= "skip" <Semicolon>
Assignment ::= <Variable> "=" <Exp> <Semicolon>
IfThenElse ::= "if" <S> <BExp> <S> "then" <S> (<Program> | <Program> "else" <S>
<Program>) "end" <Semicolon>
While ::= "while" <S> <BExp> <S> "do" <S> <Program> <S> "end" <Semicolon>
Exp ::= AExp | BExp | "array" "(" <AExp> ")" | "[" <Exp> ["," <Exp> ]* "]"

BExp ::= <BTerm> ["||" <BTerm>]*
BTerm ::= <Fact> ["&&" <BFact>]*
BFact ::= <AExp> <ComparisonOp> <AExp> | <Bool> | <Variable> | "!" <BExp> | "("
<bexp> ")"

AExp ::= <ATerm> [{"+" | "-"} <ATerm>]*
ATerm = <AFactor> [{"*" | "/"} <AFactor>]*
AFactor = <PositiveNumber> | <Variable> | "(" <AExp> ")" | "-" <AExp>
PositiveNumber ::= [0-9]+ | [0-9]+ "." [0-9]+

Identifier ::= [a-zA-z]+ \ Keyword
Variable ::= Identifier ( "[" AExp "]" )
ComparisonOp ::= "<" | ">" | "=" | "<=" | ">=" | "!="
Semicolon ::= ";" | ";" <S>
S ::= " "
```

In this language there are no declarations, or we can say that declarations coincides with the assignments, so we have **dynamic typing**.

3. Architecture and implementation

The interpreter is made out by two modules:

- **The parser** : Given an input program processes it returning a tree representation of the program
- **The evaluator** : Given the output of the parser and an initial environment, it executes the program producing a final environment

3.1 The parser

The parser can be seen as a function that given a String, produces a representation of the program in a **tree structure**, in this is case we see the tree structure as a generic type "a". Sometimes a part of the string or the entire can be not consumed, so it can return the unconsumed part or can return Nothing if the parsing completely fails; so the definition is:

```
newtype Parser a = P (String -> Maybe (a, String))
```

The output of the Parser is a syntax tree, that according to the grammar, is defined by the following constructs:

```
type Program = [Command]

data Command
    = Skip
    | Assignment Variable Exp
    | IfThenElse BExp Program Program
    | While BExp Program
    deriving (Show, Eq)

data Exp
    = Var Variable
    | BExp BExp
    | AExp AExp
    | ArrIntensional AExp
    | ArrExtensional [Exp]
    deriving (Show, Eq)

data AExp
    = Number Double
    | AVar Variable
    | AExpOp AExp AOp AExp
    | Negation AExp
    deriving (Show, Eq)

data AOp
    = Add
    | Sub
    | Mul
    | Div
    deriving (Show, Eq)

data BExp
    = Boolean Bool
    | Not BExp
```

```
    | Comparison AExp ComparisonOp AExp
    | BVar Variable
    | BExpOp BExp BOp BExp
    deriving (Show, Eq)

data BOp
    = Or
    | And
    deriving (Show, Eq)

data ComparisonOp
  = Lt
  | Le
  | Gt
  | Ge
  | Eq
  | Neq
  deriving (Show, Eq)

data Variable
    = Identifier [Char]
    | ArrayLocation Variable AExp
    deriving (Show, Eq)
```

3.1.2 Functor, Applicative, Monad, Alternative

The following four instances are the core of the parser, because allow multiple parsers to operate as alternatives and in sequence.

Functor

The Functor class provides the function *fmap*, in order to give the possibility to apply a function g to a wrapped value, in our case a function to a value wrapped in a Parser.

```
instance Functor Parser where
    fmap g (P p) =
        P
        ( \input -> case p input of
            Nothing -> Nothing
            Just (v, out) -> Just (g v, out)
        )
```

Applicative

The Applicative class provides the function `pure` that simply wraps the given input, and the function `<*>` used to apply a wrapped function to a wrapped value, also the result will be wrapped. An Applicative is also a Functor, therefore we can use the fmap function.

```haskell
instance Applicative Parser where
    pure v = P (\input -> Just (v, input))
    (P pg) <*> px =
        P
        ( \input -> case pg input of
            Nothing -> Nothing
            Just (g, out) -> case fmap g px of
                (P p) -> p out
        )
```

Monad

The Monad class is a natural extension of Applicative. It provides the function *bind* (>>=), that takes a wrapped value `m a`, a function `a -> m b` and returns `m b`.
Given an instance to Monad, we can use the `do` notation to combine parsers in sequence.

```haskell
instance Monad Parser where
    (P p) >>= f =
        P
        ( \input -> case p input of
            Nothing -> Nothing
            Just (v, out) -> case f v of
                (P p) -> p out
        )
```

Alternative

As the name says, Alternative can be used to choose between multiple alternatives or to allow parallel computing. In this case given two Parsers p and q, if the first fails, we use q, otherwise we return the results of p. `empty` represents an applicative computation with zero results, in this case we return the failure `Nothing`.

```haskell
instance Alternative Parser where
    empty = P (const Nothing)
    (P p) <|> (P q) =
        P
        ( \input -> case p input of
            Nothing -> q input
            Just (v, out) -> Just (v, out)
        )
```

Also, with alternative we can define the function `many`, that applies the parser many times as possible, and `some`, that has the difference that at least one parser have to succeeds.

```haskell
class Monad f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
    many :: f a -> f [a]
    some :: f a -> f [a]
    many x = some x <|> pure []
    some x = ((:) <$> x) <*> many x
```

3.1.3 Token parsing

in order to parse the tokens of the language we need to start from parsing a single char, thus the Parser `item` do this, it fails with the empty string otherwise consumes the first character.

```
item :: Parser Char
item =
  P
    ( \input -> case input of
        [] -> Nothing
        (x : xs) -> Just (x, xs)
    )
```

The other basic element is `sat`, it succeeds if the given char satisfies the predicate `p` .

```
sat :: (Char -> Bool) -> Parser Char
sat p =
    do
        x <- item;
        if p x then return x else empty;
```

Now is possible to define the following parsers:

- digit: parses a digit
- lower: parses a lowercase letter
- upper: parses an uppercase letter
- letter: parses a letter
- alphanum: parses an alphanumeric char
- space: parses a single space or newline
- char x: parses a specific char x
- string s: parses a specific string s
- token p: parses a specific value p and ignores spaces
- symbol s: parses a specific symbol token s

3.1.4 Variable parsing

In our language a variable is either an identifier or a location of an array. An identifier is an alphanumeric string starting with a letter; a location of an array is an identifier plus an arithmetic expression surrounded by square parenthesis. The `arrayindexes` parser gives us the possibility to use an arbitrary number of square bracket operators in our variables.
For example:
`x[i][j][k]`

```
identifier ::  Parser Variable
identifier =
    do
        l <- letter
        ls <- many alphanum
        if  isIdentifier (l:ls) then return (Identifier (l:ls)) else empty

variable :: Parser Variable
variable =
    do
        id <- identifier
        do
```

```
            arrayindexes id
            <|>
            return id

arrayindexes :: Variable -> Parser Variable
arrayindexes id =
    do
        symbol "["
        a <- aexp
        symbol "]"
        do
            v <- arrayindexes id
            return (ArrayLocation v a)
            <|>
            return (ArrayLocation id a)
```

3.1.5 Arithmetic Expression Parsing

First we need to parse numbers (IMPterpreter works with doubles).

```
nat :: Parser Double
nat =
    do
        ds <- some digit
        return (read ds)

positive :: Parser Double
positive =
    do
        int <- some digit
        dot <- char '.'
        decimal <- some digit
        return (read (int ++ [dot] ++ decimal))

number :: Parser Double
number =
    positive
    <|>
    nat
    <|>
    (do
        char '-'
        num <- number
        return (- num)
    )
```

Then we can write the `aexp` parser. It builds a derivation tree in a left-associative way, so `4+5*4 = (4+5)*4`. Also the primary operations "+" and "-" have lower priority on "*" and "/". It uses the subparsers `aterm` and `afactor`.

```
afactor :: Parser AExp
afactor =
    (do Number <$> number)
```

```
    <|>
    (do AVar <$> variable)
    <|>
    (do
        symbol "("
        a <- aexp
        symbol ")"
        return a
    )
    <|>
    (do
        symbol "-"
        Negation <$> aexp
    )
```

```
aterm :: Parser AExp
aterm =
    do
        lf <- afactor
        rest lf
    where rest lf = (do op <- asecondaryop; rf <- afactor; rest (AExpOp lf op
rf)) <|> return lf
```

```
aexp :: Parser AExp
aexp =
    do
        lt <- aterm
        rest lt
    where rest lt = (do op <- aprimaryop; rt <- aterm; rest (AExpOp lt op rt))
<|> return lt
```

3.1.6 Boolean expression parsing

The followed strategy is similar to the `aexp` parser:

```
bool :: Parser BExp
bool =
    (do
        s <- symbol "True"
        return (Boolean True)
        )
    <|>
    (do
        s <- symbol "False"
        return (Boolean False)
        )
```

```
bfactor :: Parser BExp
bfactor =
    bool
    <|>
    (do
        a1 <- aexp
        c <- comparison
        Comparison a1 c <$> aexp
```

```
    )
    <|>
    (do BVar <$> variable)
    <|>
    (do
        symbol "("
        bx <- bexp
        symbol ")"
        return bx
    )
    <|>
    (do
        symbol "!"
        Not <$> bexp
    )
```

```
bterm :: Parser BExp
bterm =
    do
        lf <- bfactor
        rest lf
    where rest lf = (do symbol "&&"; rf <- bfactor; rest (BExpOp lf Or rf)) <|>
return lf
```

```
bexp :: Parser BExp
bexp =
    do
        lt <- bterm
        rest lt
    where rest lt = (do symbol "||"; rt <- bterm; rest (BExpOp lt And rt)) <|>
return lt
```

3.1.7 Command parsing

The `program` parsers see a program as a list of commands. A comand is one of the basic construct of IMP.

```
program :: Parser Program
program =
    do
        c <- command
        do
            p <- program
            return (c:p)
            <|>
            return [c]

command :: Parser Command
command =
    skip <|>
    assignment <|>
    ifthenelse <|>
    while
```

`skip` parser:

```
skip :: Parser Command
skip =
    do
        symbol "skip"
        symbol ";"
        return Skip
```

The `assignment` parser consumes a variable followed by "=", and then we have 4 alternatives:

- aexp
- bexp
- variable
- arraydeclaration

The "variable" can seem a bit redundant, as aexp and bexp contain it, but at the parsing level in an assignment like `x = y` the parser can not infer the type of y so we need this extra case.

```
assignment :: Parser Command
assignment =
    do
        v <- variable
        symbol "="
        Assignment v <$> rightValue

rightValue :: Parser Exp
rightValue =
    do
        x <- variable
        symbol ";"
        return (Var x)
    <|>
    do
        e <- (do AExp <$> aexp)
        symbol ";"
        return e
    <|>
        do
        e <- (do BExp <$> bexp)
        symbol ";"
        return e
    <|>
    do
        e <- arraydeclaration
        symbol ";"
        return e
```

IMPterpreter allow an extensional declaration of arrays, for example: `v = [1, 2, 3, [4, False], 5];`. As we can see arrays can have elements of different type.
The parser `arrayelements` parses the all the expressions contained in the extensional declaration. It can be noticed that this parser allow arrays with elements of different type.
Moreover we can declare an array giving the number of elements: `v = array(n);`

```
arraydeclaration :: Parser Exp
arraydeclaration =
    do
```

```
        symbol "array"
        symbol "("
        n <- aexp
        symbol ")"
        return (ArrIntensional n)
    <|>
    do
        symbol "["
        exps <- arrayelements
        symbol "]"
        return (ArrExtensional exps)

arrayelements :: Parser [Exp]
arrayelements =
    do
        v <- (do Var <$> variable)
        do
            symbol ","
            t <- arrayelements
            return (v:t)
            <|>
            return [v]
    <|>
    do
        b <- (do BExp <$> bexp)
        do
            symbol ","
            t <- arrayelements
            return (b:t)
            <|>
            return [b]
    <|>
    do
        a <- (do AExp <$> aexp)
        do
            symbol ","
            t <- arrayelements
            return (a:t)
            <|>
            return [a]
    <|>
    do
        a <- arraydeclaration
        do
            symbol ","
            t <- arrayelements
            return (a:t)
            <|>
            return [a]
```

The `ifthenelse` parser recognize the keyword **if**, subsequently it parses a boolean expression, and look for the **then** keyword. In the then-block as in the else-block is used the `program` parser. The IMP IfThenElse can omit the else-block and it ends with the **end** keyword.

```
ifthenelse :: Parser Command
ifthenelse =
    do
```

```
        symbol "if"
        b <- bexp
        symbol "then"
        pthen <- program
        do
            symbol "else"
            pelse <- program
            symbol "end"
            return (IfThenElse b pthen pelse)
            <|> do
                symbol "end"
                return (IfThenElse b pthen [Skip])
```

The last parser is `while`, it look for the **while** keyword, then parse a boolean expression, subsequently it recognize the **do** keyword, parses a `program` and look for the **end** keyword.

```
while :: Parser Command
while =
    do
        symbol "while"
        b <- bexp
        symbol "do"
        p <- program
        symbol "end"
        return (while b p)
```

3.1.8 Parse function

All the parsers are encapsulated by the function `parse` that returns a list of commands and the not consumed part of the input string. If the parsing fails is returned the empty list.

```
parse :: String -> (Program, String)
parse s = case p s of
  Nothing -> ([], s)
  Just (c, s) -> (c, s)
  where
    (P p) = program
```

Examples:

Successful parsing:

```
*IMPterpreter.Parser> parse "x = 10.7 / 5;"
([Assignment (Identifier "x") (AExp (AExpOp (Number 10.7) Div (Number 5.0)))],"")
```

Failing parsing:

```
*IMPterpreter.Parser> parse "x = 10"
([],"x = 10")
```

Partial successful parsing:

```
*IMPterpreter.Parser> parse "x = True; y = 11"
([Assignment (Identifier "x") (BExp (Boolean True))],"y = 11")
```

3.2 The evaluator

The module Evaluator provides the functions to evaluate a program in the *internal tree representation* given an environment, and to produce a new environment.

3.2.1 Environment

An **environment** is a list of variables:

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

A **variable** is composed by a name and a value:

```
data Variable = Variable {
    name :: String,
    value :: VType
} deriving Show
```

The three admitted **VTypes** are:

```
data VType = TDouble Double | TBool Bool | TArray [VType]
```

The environment is managed by a *reading* and a *writing* function.

getVarValue returns value of the variable wrapped by **Either** monad given its name. If the name is not present returns a `VariableNotDefined` exception.

```
-- Search the value of a variable store in the Env given its name
getVarValue :: Env -> String -> Either VType Exception
getVarValue [] v = Right (VariableNotDefined (show v))
getVarValue (x:xs) qName =
    if name x == qName
        then Left (value x)
    else getVarValue xs qName
```

insertVar insert a new variable in the environment or update an existing one. In the update the value can be arbitrary changed, modifying also its VType.

```
-- insert or update a new variable
insertVar :: Env -> Variable -> Env
insertVar [] var  = [var]
insertVar (x:xs) newVar =
    if name x == name newVar
        then newVar: xs
    else x : insertVar xs newVar
```

3.2.2 Variable evaluation

evalVar given a variable check if it exists in the environment and then returns its value, otherwise raise an error. If the variable is an array location it also evaluate is **index expressions** and retrieve the array element.

```
-- Get a variable from the Env and return its value
evalVar :: Env -> IMPterpreter.Tree.Variable -> Either VType Exception
evalVar env (Identifier var) = getVarValue env var
evalVar env (ArrayLocation var i) =
    case unfoldArrayIndexes env (ArrayLocation var i) of
        (v, Left indexes) ->
            case evalVar env (Identifier v) of
                Left value -> getArrayElement value indexes
                Right exc -> Right exc
        (v, Right exc) -> Right exc
```

safeEvalVar check also if the variable is of a given VType.

```
-- Get a variable value from the Env and check the type matching
safeEvalVar :: Env -> IMPterpreter.Tree.Variable -> [Char] -> Either VType
Exception
safeEvalVar env var varType =
    case (evalVar env var, varType) of
        (Left (TDouble d), "TDouble") -> Left(TDouble d)
        (Left (TBool b), "TBool") -> Left (TBool b)
        (Left (TArray a), "TArray") -> Left (TArray a)
        (Right exc, _) ->  Right exc
        (Left v, t) -> Right (TypeMismatchValue (show t) (show v))
```

3.2.3 Arithmetic expression evaluation

evalAExpr evaluates an AExp and returns *Left Double* if the evaluations succeeds, otherwise *Right Exception*

```
-- Evaluates arithmetic expressions
evalAExpr :: Env -> AExp -> Either Double Exception
evalAExpr _ (Number x) = Left x
evalAExpr env (AVar (Identifier v)) = getDouble (safeEvalVar env (Identifier v)
"TDouble")
evalAExpr env (AVar (ArrayLocation v a)) = getDouble (safeEvalVar env
(ArrayLocation v a) "TDouble")
evalAExpr env (AExpOp al op ar) =
    case (evalAExpr env al, evalAExpr env ar) of
        (Left l, Left r) -> Left (opADict op l r)
        (_, Right exc) -> Right exc
        (Right exc, _) -> Right exc
evalAExpr env (Negation a) = checkException (evalAExpr env a) (\x -> -x)
```

3.2.4 Boolean expression evaluation

evalBExpr evaluates a BExp and returns *Left Bool* if the evaluations succeeds, otherwise *Right Exception*

```
-- Evaluates boolean expressions
evalBExpr :: Env -> BExp -> Either Bool Exception
evalBExpr _ (Boolean b) = Left b
evalBExpr env (BVar (Identifier v)) = getBool (safeEvalVar env (Identifier v)
"TBool")
evalBExpr env (BVar (ArrayLocation v a)) = getBool (safeEvalVar env
(ArrayLocation v a) "TBool")
```

```
evalBExpr env (BExpOp bl op br) =
    case (evalBExpr env bl, evalBExpr env br) of
        (Left l, Left r) -> Left (opBDict op l r)
        (_, Right exc) -> Right exc
        (Right exc, _) -> Right exc
evalBExpr env (Not b) = checkException (evalBExpr env b) not
evalBExpr env (Comparison al op ar) =
    case (evalAExpr env al, evalAExpr env ar) of
        (Left l, Left r) -> Left (opCDict op l r)
        (_, Right exc) -> Right exc
        (Right exc, _) -> Right exc
```

3.2.5 Command evaluation

execExpr is a function used in the command evaluation that evaluates an arbitrary expression and if it returns a *Left x*, it applies a given *operation* on it. Otherwise it propagates the Exception. This function is useful because it generalize a repeating pattern in the statement evaluation.

```
-- Evaluate an expression and applicate a function "operation" on it if return
Just x
execExpr :: Env -> Exp -> (VType -> Either a Exception) -> Either a Exception
execExpr env e operation =
    case e of
        AExp a ->
            case evalAExpr env a of
                Left x -> operation (TDouble x)
                Right exc ->  Right exc
        BExp b ->
            case evalBExpr env b of
                Left x -> operation (TBool x)
                Right exc ->  Right exc
        Var v ->
            case evalVar env v of
                Left x -> operation x
                Right exc ->  Right exc
        ArrIntensional n ->
            case execExpr env (AExp n) (Left . makeArrayIntensional) of
                Left x -> operation x
                Right exc ->  Right exc
        ArrExtensional exps ->
            case makeArrayExtensional env exps of
                Left arr -> operation arr
                Right exc ->  Right exc
```

execStatement evaluates the commands of the program. Assignments are made out using **execExpr**, so evaluating expressions and then inserting the resulting value in the environment. The ifthenelse is executed evaluating the condition and then choosing the corresponding path. Lastly for the while loop we evaluate the conditions if is false we return the unchanged environment, otherwise we execute the commands in the do-block of the while, and then recursively call execStatement.
In any statement, one of the sub-evaluators could fail, in that case it is returned an Exception.

```
-- Execute a statement of the program and apply its effects to the enviroment
execStatement :: Env -> Command -> Either Env Exception
execStatement env Skip = Left env
```

```
execStatement env (Assignment (Identifier v) e) =  execExpr env e (Left .
insertVar env . IMPterpreter.Evaluator.Variable v)
execStatement env (Assignment (ArrayLocation var i) e) =
    case unfoldArrayIndexes env (ArrayLocation var i) of
        (id, Left idx) -> execExpr env e (arrayElementAssignment env id idx)
        (id, Right exc) -> Right exc

execStatement env (IfThenElse b pthen pelse) = execExpr env (BExp b)
    (\bvalue ->
        case getBool (Left bvalue) of
            Left True -> exec env pthen
            Left False -> exec env pelse
            Right exc -> Right exc
    )
execStatement env (While b p) = execExpr env (BExp b)
    (\bvalue ->
        case getBool (Left bvalue) of
            Left True ->
                case exec env p of
                    Left e -> exec e [While b p]
                    Right exc -> Right exc
            Left False -> Left env
            Right exc -> Right exc
    )
```

exec executes a program and returns an Env if the computation succeeds, otherwise an Exception

```
-- Execute a program
exec :: Env -> Program -> Either Env Exception
exec env (c:p) =
    case execStatement env c of
        Left e -> exec e p
        Right s -> Right s
exec env [] = Left env
```

3.2.5 Array management

Arrays are stored in the environment as lists of VType, so they can have different VTypes for each element, even arrays.

The handling of elements is performed by the functions **getArrayElement** and **setArrayElement**. This functions can raise an `IndexOutOBounds` error if the index is higher or equal the array length, or an `DimensionOutOfBounds` exception if the number of indexes is greater than the dimension of the array.

```
-- Set an element of an array given an index and a value
setArrayElement :: VType -> [Double] -> VType -> Either VType Exception
setArrayElement (TArray []) i v =  Right (IndexOutOfBounds (floor <$> i))
setArrayElement (TArray l) [i] v =
    case splitAt (floor i) l of
        (f, a:as) -> Left (TArray (f ++ (v:as)))
        (f, []) -> Right (IndexOutOfBounds [floor i])
setArrayElement (TArray l) (i:is) v =
    case setArrayElement a is v of
        Left (TArray arr) -> Left (TArray (f ++ [TArray arr] ++ as))
```

```
        Right exc -> Right exc
    where (f, a:as) = splitAt (floor i) l
setArrayElement _ i _ = Right (DimensionOutOfBounds (floor <$> i))

-- Get an element of the list, if the index is correct, otherwise returns an
exception
safeListAccess :: [a] -> Int -> Either a Exception
safeListAccess l i =
    if length l > i && i >= 0 then Left (l !! i) else Right (IndexOutOfBounds
[i])

-- Get an element of an array given an index
getArrayElement :: VType -> [Double] -> Either VType Exception
getArrayElement (TArray l) [i] = safeListAccess l (floor i)
getArrayElement (TArray l) (i:is) = getArrayElement (l !! floor i) is
getArrayElement _ idx  = Right (DimensionOutOfBounds (floor <$> idx))
```

The intensional declaration is made out by the **makeArrayIntensional** function, that given a number `d`, builds an array of d elements. The default values of arrays are `TDouble 0`.

```
-- Construct an array given the number of elements
makeArrayIntensional :: VType -> VType
makeArrayIntensional (TDouble d)
    | d <= 0 = TArray []
    | d > 0  =  TArray (TDouble 0: t)
        where TArray t = makeArrayIntensional (TDouble (d-1))
```

makeArrayExtensional builds an array from a list of expressions, evaluating each one.

```
-- Construct an array given the list of expression
makeArrayExtensional :: Env -> [Exp] -> Either VType Exception
makeArrayExtensional env [exp] = execExpr env exp (\x -> Left (TArray [x]))
makeArrayExtensional env (e:ex) =
    case makeArrayExtensional env ex of
        Left (TArray t) -> execExpr env e (\x -> Left (TArray (x:t)))
        Right s -> Right s
```

4. Usage

Load the **Main.hs** module from the ghci shell, and then type `main` to launch IMPterpreter, this should be shown:

Instructions can be directly typed into the shell or can be loaded from file with the **:l** command. All commands available are:

- **:l filename** loads and executes instructions contained in the given file
- **:env** prints all the variables in the environment
- **:p v** search the variable "v" in the environment: if is found prints its value, otherwise prints *VariableNotDefined "v"*
- **:cl** empties the environment
- **:h** shows the help message
- **:q** quits from the interactive shell

4.1 Code examples

Assigning variables

```
x = 6.7;
y = False;
v = [1, False, 4.5];
b = True && False || 3 < 4;
```

Using selection and loop

```
b = True;
if b then
    x = 1;
else
    x = -1;
end
```

```
n = 0;
i = 0;
while i < n do
    i = i + 1;
end
```

Factorial of 6

```
f = 1;
n = 6;
i = 1;
while i < n + 1 do
    f = f * i;
    i = i + 1;
end
```

Average value of an array

```
v = [2, 3, 6, 2, 3, 4];
n = 6;
i = 0;
s = 0;
while  i < n do
    s = s + v[i];
    i = i + 1;
end
m = s / n;
```

Other examples can be found in the *examples* folder.

```
:l examples/matrixProd.imp
:env
```

```
:l examples/pow.imp
:env
```