# IMPterpreter

## by Francesco Greco

an Haskell Interpreter of the “IMP” language

Course in Formal Methods in Computer Science

Prof. Giovanni Pani

University of Bari “Aldo Moro” - Computer Science

# Index

# 1 Introduction

This document is meant to describe the project “IMPterpreter”, an interpeter for the imperative language *IMP,* written in Haskell by Francesco Greco. The basic language has been extended to manage some features that will be descripted later on.

The interpreter works on a unique input string which represent the program to be interpreted; therefore, it can be used via command line interface.

## 1.1 The IMP language

“Imp” is a simple imperative language that is defined by several rules which formally describe what the language can do. It is called “imperative” since program execution involves executing a series of explicit commands to change state. Since the purpose is to have an idea of what IMP is capable of in an operational way, so that we would be able to write an interpeter for it, we will write and consider the rules in operational semantics.

First of all, IMP can be divided in three different sets of rules, grouped by a common semantics of their nature.

### 1.1.1 Aexp

Aexp represents the set of arithmetic expressions of the language and is formally describe as:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

*Figure 1: Aexp in IMP*

where n is a natural number and X is a variable.

So in IMP we have that an arithmetic expression Aexp can be:

1. just a natural **number** *n*;
2. a **variable** (or a memory location) X;

3. the **addition** operation between any two arithmetic expressions $a_0$ $a_1$;
4. the **subtraction** operation between any two arithmetic expressions $a_0$ $a_1$;
5. the **multiplication** operation between any two arithmetic expressions $a_0$ $a_1$.

### 1.1.2 Bexp

Bexp represents the set of boolean expressions, defined as following:

$$b ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

*Figure 2: Bexp in IMP*

where $a_0$ and $a_1$ belong to Aexp.

So in IMP we can have:

1. a boolean value of ***true***;
2. a boolean value of ***false***;
3. the **equality** comparison operation between any two arithmetic expressions $a_0$ $a_1$, which is equal to the value*true* if and only if the two arithmetic expressions are equal, and *false* otherwise;
4. the comparison operation of "**less or equal**" between any two arithmetic expressions $a_0$ $a_1$, which is equal to the value *true* if and only if the first arithmetic expression $a_0$ is less or equal in value to $a_1$, and *false* otherwise;
5. the logical **negation** of a boolean expression, which commutes a boolean value *true* to *false* and vice-versa;
6. the logical **conjunction** between two boolean expressions $b_0$ $b_1$, which returns *true* if and only if $b_0$=$b_1$=*true*, and *false* otherwise;
7. the logical **disjunction** between two boolean expressions $b_0$ $b_1$, which returns *false* if and only if $b_0$=$b_1$=*false*, and *true* otherwise;

### 1.1.3 Com

Com represents the set of commands that can be executed in IMP:

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

*Figure 3: Com in IMP*

where a belongs to Aexp and b belongs to Bexp.

In IMP the following commands are defined:

1. the **skip** instruction, which does nothing at all;
2. the **assignment** operation, which substitutes the value in the memory location *X* with the value of the arithmetic expression *a*;
3. the **sequence** of commands, which makes possible to execute any couple of commands defined in Com in sequence, executing first $c_0$ and then $c_1$;
4. the **selection** of commands with respect to a boolean expression *b*, which executes $c_0$ if b=*true* and $c_0$ otherwise;
5. the **iteration** of commands, which executes the command *c* in loop until the boolean expression *b* is *true*. If at the very first iteration *b* is equal to *false*, then the command *c* is never executed.

## 1.2 Added features in IMPterpreter

In addition to the core of operations and expressions in the IMP language, IMPterpreter is able to manage several other data types.

Starting from Aexp, we have no more the constraint of *n* being a natural number, but it can assume any **integer** value, so it can be even a negative number. This was also possible in IMP, just by considering the subtraction between 0 and n, with n being a natural (thus non-negative) number. Also the **integer division** has been implemented.

In Bexp we have **other comparison operators**, in order to complete the set already found in IMP. All of these are just derivable from the (almost) minimal set of instruction found in IMP (e.g. the operator "greater than" can be constructed as "not – less or equal"). Anyway IMPterpreter understands also: "less than", "greater than", "greater or equal", "different from" operators.

Other than arithmetic and boolean expressions, in IMPterpreter we have also the **string** data type, which can handle sequences of 0 or more characters, delimited by a single quote (') symbol.
Also the **concatenator** operator (++) is defined for strings, so that the operation $s_0$ *++* $s_1$ (with $s_0$ $s_1$ string expressions) results in a new string composed as the juxtaposition of the two strings (first $s_0$ and then $s_1$).

It has made possible to have memory locations (i.e. variables) that can store not only integers, but also booleans and strings values, as well as arrays of integers, booleans and strings, and 2-dimensional arrays of the same types.

As we already suspect, that's right, **one and two dimensional arrays of integers, booleans and strings** have been implemented in the laAnguage! Note that we avoided calling the two dimensional arrays straight up "matrices": that's because we have no constraint on the shape of each row of those data structure. In a proper matrix, indeed, having the first row with, say, 3 entries and 5 on the second, wouldn't be possible at all. As we will see later on, arithmetic and boolean matrix expressions are correctly defined only on proper rectangular (or square) matrices. Anyway, for shortness, we will use in this document the term "matrix" as well as the term "2-dimensional array", even if it's not entirely appropriate.
The delimitation of arrays is obtained with the use of square brackets (e.g. '[1,4,7,12]'), while matrices are defined with double square brackets (e.g. '[[5,7,9],[2,5],[0,0,0,1,3]]').

We can do array and matrix assignments in order to change a single cell or a single row (in case of 2 dimensions). We can access as well the values of an array or a matrix: we can take the value of a single cell or even extract an entire row from a matrix.

On any 1-dimensional array (of any type) we have the **concatenation** operator (***++***), which simply takes two arrays (of the same type) $a_0$ and $a_1$, of length *n* and *m,* and creates a new one of length (*n+m*), having the first *n* elements of $a_0$ and the remaining *m* elements from $a_1$.

On any 2-dimensional array (of any type) we have defined the **horizontal matrix concatenation** operator (***++***), which takes two matrices (of the same type) $a_0$ (of n rows) and $a_1$ (of m rows), and returns a new matrix: this is constructed with *min(n,m)* rows with each of these rows composed as the array concatenation between each row in the two matrices.
Note that if the two matrices have a different number of rows, let's say 5 and 7, only the first 5 will be concatenated, while the other 2 of the second matrix will be lost. The same principle holds if the first matrix has more rows than the second one.
On 2-dimensional arrays (of any type) is defined the **vertical matrix concatenation** operator (#), which takes two matrices (of the same type) $a_0$ (with n rows) and $a_1$ (with m rows) and returns a new matrix with *n+m* rows, of which the first *n* are the rows from $a_0$ and the last *m* are the rows from $a_1$.

On integer arrays the following operations are defined:

- scalar multiplication (*);
- scalar division (/);
- term-by-term addition (+);
- term-by-term subtraction (-);
- term-by-term multiplication (*);

- term-by-term division (/);
- inner product (defined as @).

On integer matrices the following operations are defined:

- scalar product (*);
- scalar division (/);
- matrix addition (+);
- matrix subtraction (-);
- matrix multiplication, or dot product (@).

On boolean arrays are defined:

- term-by-term conjunction (&&);
- term-by-term disjunction (||);
- array negation (!);

On boolean matrices the same operations (conjunction, disjunction and negation) are defined.

## 2 Interpreter structure

In this section the structure of IMPterpreter will be described.

The interpreter is based on a folder structure that is reported in the following schema:

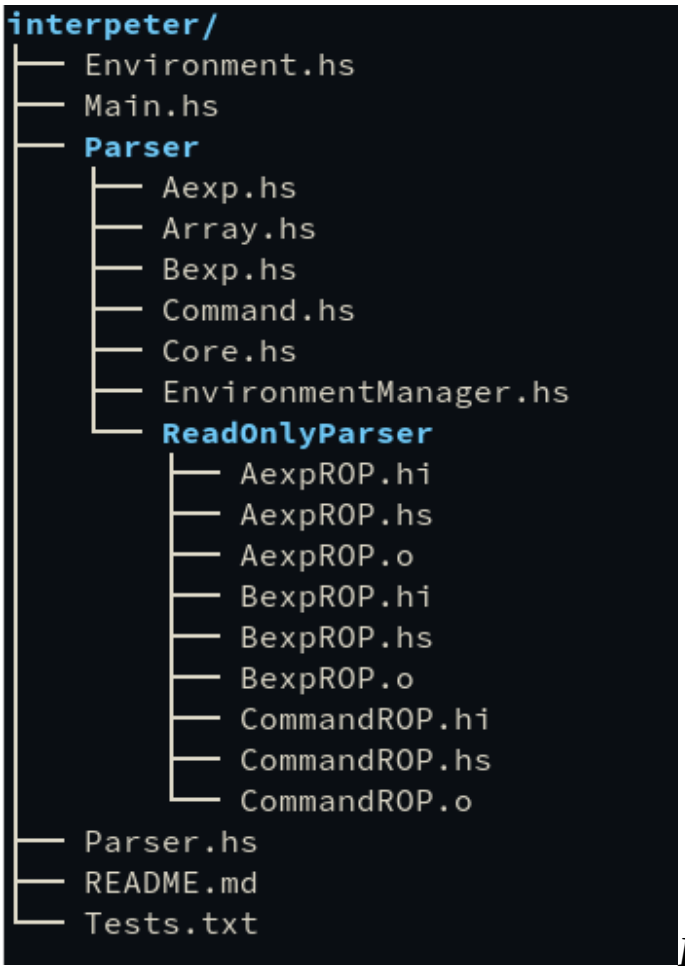


*Figure 4: Directory tree of IMPterpreter*

The base directory contains 3 files:

- Environment.hs contains the Environment module, which principally defines how the Env is defined;
- Main.hs which contains the implementation of the Command Line Interface of the interpreter and is therefore the access point for the user;
- Parser.hs links together all the submodules defined in the Parser module and contains the function *evaluate* which tries to interpret the user input.

The interpreter has been tested and compiled with Glasgow Haskell Compiler, Version 8.6.5.

### 2.1 Environment

The Environment module contains the code that manages the environment of the interpeter. The Env  type consists in an array of Variables, which are user defined data types which model the concept of “variable” of a programming language. A Variable is made of three fields:

- name : the name of the variable, stored as a String;
- *vtype* : the type of the variable, stored as a String; it can assume values “int”, “bool” and so on;
- *value* : the value of the variable; it is stored as a monadic type Either, with parameters Int and String: in this way we can easily store values of type Int using the Left constructor (used to store integers and bools) and String values in the Right constructor (used for strings and arrays).

```
data Variable = Variable {
    name :: String,
    vtype :: String,
    value :: Either Int String
} deriving Eq
```

*Figure 5: Variable and Env*

In this module are also present two functions that are needed to:

- modify the enviroment by adding a new variable to it, or just updating it if the variable already exists, which is called *modifyEnv* ;
- retrieve a variable from the Env (i.e. it searches for the variable name into the Env array), which is called *searchVariable* .

```
modifyEnv :: Env -> Variable -> Env
modifyEnv [] var = [var]
modifyEnv (x:xs) newVar = if (name x) == (name newVar)
                          then [newVar] ++ xs
                          else [x] ++ modifyEnv xs newVar

-- Searches for the value of a variable stored in the Env given the name
searchVariable :: Env -> String -> [(Int, String)]
searchVariable []     _         = []
searchVariable (x:xs) queryname = if (name x) == queryname
    then [(value x, vtype x)]
    else searchVariable xs queryname
```

*Figure 6: modifyEnv and searchVariable functions*

Finally, it is defined the *show* function to nicely print out variables.

## 2.2 Parser

The Parser folder contains the main code for the parsing of the program string into actual Haskell code that can be executed by the Haskell compiler.

The files contained into the Parser folder are therefore reported.

### 2.2.1 Parser.Core

The module *Parser.Core* contains, indeed, the core of the interpreter.

Here we can find the definition of the type *Parser*, which is defined as a type that in its constructor takes 2 fields:

- ➢ an Env (variables environment);
- ➢ a String representing the line to parse;

```
--Returns a Maybe value because the parsing can fail
newtype Parser a = P (Env -> String -> Maybe(Env, a, String))
```

*Figure 7: Parser type*

➢

It returns a *Maybe* Monad which can either fail (returning "Nothing") or succeed and return a triple *(Env, a, String)*:

- “Env” represents the environment resulting after the parsing, which intuitively changes after a variable assignment;
- “a” represents the result of the parsing, which can be, for example, a parsed integer value, or an array;
- “String” contains the eventual remainder string that has not been parsed; for example, if we apply the parser “command”, which only parses a single command, on a sequence of commands, the first command will be executed and the rest will be returned as a string.

The *Parser* type is a generic type that can be implemented by different kind of Parsers.

Indeed, in this module there are all the fundamental types of *Parser*, which are used to micro-manage the single strings of text, or even single characters, in order to be able to parse more complex types and structures.

For example we have the *nat* Parser, which succeeds only if the parsed encounters one or more digits; a *digit* Parser, in turn, is a Parser that succeeds if the encountered character is a digit, and fails otherwise.

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

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

digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char -> Parser Char
char x = sat (== x)

manychar :: String -> Parser String
manychar [] = return []
manychar (x:xs) =
    do
        char x;
        manychar xs;
        return (x:xs);
```

```haskell
identifier :: Parser String
identifier =
    do
        x <- lower
        xs <- many alphanum
        return (x:xs)

nat :: Parser Int
nat =
    do
        xs <- some digit
        return (read xs)

space :: Parser ()
space =
    do
        many (sat isSpace)
        return ()

int :: Parser Int
int =
    do
        char '-'
        n <- nat
        return (-n)
    <|>
    nat

token :: Parser a -> Parser a
token p =
    do
        space
        v <- p
        space
        return v
```

```
integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol xs = token (manychar xs)

notTerminator :: Char -> Bool
notTerminator c = c /= '\''

string :: Parser String
string =
    do
        symbol "'"
        stringVal <- many (sat notTerminator)
        symbol "'"
        return stringVal
```

*Figure 8: Fundamental types*

The function *parse* simply applies the Parser to an environment and a string:

```haskell
parse :: Parser a -> Env -> String -> Maybe(Env, a,String)
parse (P p) env input = p env input
```

*Figure 9: Function parse*

The Parser type is also an instance of *Functor*, *Applicative* and *Monad*.

```haskell
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap g p = P (\env input -> case parse p env input of
        Nothing -> Nothing
        Just (env, v, out) -> Just (env, g v, out))

instance Applicative Parser where
    --pure :: a -> Parser a
    pure v = P (\env input -> Just (env, v, input))
    -- <*> :: Parser (a -> b) -> Parser a -> Parser b
    pg <*> px = P (\env input -> case parse pg env input of
        Nothing  -> Nothing
        Just (env, g, out) -> parse (fmap g px) env out)

instance Monad Parser where
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    return v = P (\env input -> Just (env, v, input)) -- Has the same behaviour as "pure"

    p >>= f = P (\env input -> case parse p env input of
        Nothing -> Nothing
        Just (env, v, out) -> parse (f v) env out)
```

*Figure 10: Parser as instance of Functor, Applicative and Monad*

- Parser is an instance of Functor because we need to apply functions to the result (wrapped) values of a Parser, if it succeeds.
- Parser is an instance of Applicative because we need to apply functions that are wrapped in a Parser to wrapped values coming from a Parser.
- Parser is an instance of Monad because we need to apply sequentially a Parser using the resulting values of another Parser.

A Parser can fail, and that would need to execute another type of Parser instead of the one that just failed; that's why the Parser type is also an instance of *Alternative*.

```haskell
instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\env input -> Nothing)

    -- (<|>) :: Parser a -> Parser a -> Parser a
    p <|> q = P (\env input -> case parse p env input of
        Nothing -> parse q env input
        Just (env, v, out) -> Just (env, v, out))
```

*Figure 11: Parser as instance of Alternative*

### 2.2.2 Parser.Aexp

This module contains all the parsers used for the management of arithmetical expressions of IMP, as described in Section 1.

So here we have the functions that manage the execution of sums, subtractions, multiplications, divisions and grouping.

It's worth mentioning the *afactor* function, which in one of its alternatives can substitute the identifier of an integer variable with its numeric content. This operation is done by calling the function *readVariable* from the *EnvironmentManager*

```haskell
aexp :: Parser Int
aexp =
    do
        t <- aterm
        symbol "+"
        a <- aexp
        return (t+a)
    <|>
    do
        t <- aterm
        symbol "-"
        a <- aexp
        return (t-a)
    <|>
    aterm

aterm :: Parser Int
aterm =
    do
        f <- afactor
        symbol "*"
        t <- aterm
        return (f*t)
    <|>
    do
        f <- afactor
        symbol "/"
        t <- aterm
        if (t == 0) then empty
        else return (f `div` t)
    <|>
    afactor
```

*Figure 12: Aexp module*

module, and will often be done in the same way for every analogous situation.

*F*

```
afactor :: Parser Int
afactor =
    do
        symbol "("
        a <- aexp
        symbol ")"
        return a
    <|>
    do
        i <- identifier
        var <- readVariable i
        case var of
            Left var -> return var --var contains an integer
            Right var -> empty
    <|>
    do
        i <- identifier
        symbol "["
        index <- nat
        symbol "]"
        (var, vtype) <- readFullVariable i
        if vtype == t_arr_int
            then
            case var of
                Left var -> empty
                Right var -> return $ (read var :: [Int]) !! index
            else
                empty
    <|>
    do
        i <- identifier
        symbol "["
        rIndex <- nat
        symbol "]"
        symbol "["
        cIndex <- nat
        symbol "]"
        (var, vtype) <- readFullVariable i
        if vtype == t_arr_arr_int
            then
            case var of
                Left var -> empty
                Right var -> return $ ((read var :: [[Int]]) !! rIndex) !! cIndex
            else
                empty
    <|>
    integer
```

As we can see from the figure above, the interpreter is able to parse an expression of the form:

```
b = a[0]
```

if the variable *a* contains an array or:

```
b = a[1][3]
```

if the variable *a* contains a matrix, to get the value of a cell in the data structure and assign it to variable *b*. The same thing is allowed also for type boolean and string.

### 2.2.3 Parser.Bexp

In this module there are all the functions that parse boolean expressions and reflect what is defined in IMP as a *bexp*.

So we find all the

```
bexp :: Parser Bool
bexp =
    do  p1 <- bterm
        symbol "||"
        p2 <- bexp
        return (p1 || p2)
    <|>
    do  p1 <- bterm
        symbol "&&"
        p2 <- bexp
        return (p1 && p2)
    <|>
    bcompare
    <|>
    bterm
```

*Figure 14: Boolean expressions in the Bexp module*

operations defined in IMP, i.e. logic disjunction, conjunction, negation, equality, less-or-equal operator and boolean values *true* and *false*,.

In addition to these, we also have grouping (by parenthesis, as in Aexp) and other comparison operators, such as less-than, greater-than, different-from, greater-or-equal.

Here as well there is the parsing of a variable with its boolean value; it is worth noticing that any integer variable different from 0 is considered False, whereas the True value is stored as a 0: so e.g. if we

perform “True || 0” we obtain True as a result, while if we perform “True || 7” we obtain False.

Also arrays and matrices values can be accessed, as in Aexp.

```
bcompare :: Parser Bool
bcompare =
    do
        a <- aexp
        symbol "<"
        b <- aexp
        return (a < b)
    <|>
    do
        a <- aexp
        symbol "<="
        b <- aexp
        return (a <= b)
    <|>
    do
        a <- aexp
        symbol ">"
        b <- aexp
        return (a > b)
    <|>
    do
        a <- aexp
        symbol ">="
        b <- aexp
        return (a >= b)
    <|>
    do
        a <- aexp
        symbol "=="
        b <- aexp
        return (a == b)
    <|>
    do
        a <- aexp
        symbol "!="
        b <- aexp
        return (a /= b)
```

*Figure 15: Boolean comparisons in the Bexp module*

```
bterm :: Parser Bool
bterm =
    do
        symbol "!"
        b <- bterm
        return $ not b
    <|>
    do
        symbol "("
        b <- bexp
        symbol ")"
        return b
    <|>
    do
        symbol "true"
        return True
    <|>
    do
        symbol "false"
        return False
    <|>
    bcompare
    <|>
    do
        i <- identifier
        var <- readVariable i
        case var of
            Left var -> return (var == 0)
            Right var -> empty
```

```
    <|>
    do
        i <- identifier
        symbol "["
        index <- nat
        symbol "]"
        (var, vtype) <- readFullVariable i
        if vtype == t_arr_bool
            then
            case var of
                Left var -> empty
                Right var -> return $ (read var :: [Bool]) !! index
            else
                empty
    <|>
    do
        i <- identifier
        symbol "["
        rIndex <- nat
        symbol "]"
        symbol "["
        cIndex <- nat
        symbol "]"
        (var, vtype) <- readFullVariable i
        if vtype == t_arr_arr_bool
            then
            case var of
                Left var -> empty
                Right var -> return $ ((read var :: [[Bool]]) !! rIndex) !! cIndex
            else
                empty
```

*Figure 16: Parser bterm, in which we can also access values of array and matrix variables*

### 2.2.4

### 2.2.5 Parser.Command

This is where things get interesting. In the Command module are defined all the commands expressions of the IMP language.

Here it's defined the *program* Parser, which executes a series of *command* parsing, and the *command* Parser, which executes one between: **skip**, **assignment**, **if-then-else**, **while**. The ***sequence*** of instructions is intrinsically defined by the *program* Parser, since it sequentially parses different instructions.

```
program :: Parser String
program =
    do
        command
        program
    <|>
    command

command :: Parser String
command =
    assignment
    <|>
    skip
    <|>
    ifThenElse
    <|>
    while
```

*Figure 17: Program and command Parsers in the Command module*

In IMPterpreter a sequence is acknowledged by a semicolon ";" symbol at the end of a *skip* or an *assignment* or a closed curly bracket at the end of a *while* or an *else* in an if-then-else instruction.

### *Skip*

Skip is defined as a non-side effects instruction, which, therefore, does nothing at all! In our language an instruction of this type is defined as:

**"skip;"**

```
skip :: Parser String
skip =
    do
        symbol "skip"
        symbol ";"
```

*Figure 18: Skip Parser*

### *Assignments*

The Assignment is the most

complex operation, because it requires the definition of arrays and matrices assignments, which are quite a lot and require additional functions in the EnvironmentManager module (which we'll cover later).

The core of the assignment operations is the function *updateEnv*, found in the *EnvironmentManager* module. This function takes updates the environment with a new Variable that is passed to it. This variable has:

- the name as defined on the left-hand side of the assignment operation;
- the type which corresponds to "int", "bool", "string", depending on the assigned value;
- a value which is the result of the parsing of the aexp, bexp or stringExp parser.

```
assignment :: Parser String
assignment =
    --Aexp
    do
        varName <- identifier
        symbol "="
        aval <- aexp
        symbol ";"
        updateEnv Variable {name=varName, vtype=t_int, value = Left aval}
    <|>
    --Bexp
    do
        varName <- identifier
        symbol "="
        bval <- bexp
        symbol ";"
        if bval
        then updateEnv Variable {name=varName, vtype=t_bool, value = Left 0}
        else updateEnv Variable {name=varName, vtype=t_bool, value = Left 1}
    <|>
    -- String
    do
        varName <- identifier
        symbol "="
        stringVal <- stringExp
        symbol ";"
        updateEnv Variable {name=varName, vtype=t_string, value = Right stringVal}
```

*Figure 19: Basic assignments of Arithmetic Booleans and Strings expressions*

As we can spot from the Figure just above, the assignment Parser definition goes on with other alternatives. In fact, it is defined also the behaviour that the Parser must have with respect to one-dimensional and two-dimensional arrays of integers, booleans and strings.

We will take as a demonstrative example the case of arrays of ints, but the same logic applies also for booleans and strings.

### Arrays assignment

An assignment for an array of integers is defined on two cases:

I. The case in which we assign an entire array to a variable, as for example:

```
a = [1,5,8,9,0,2];
```

In this case the assignment is performed in the usual way, using the "updateEnv" function, giving as a parameter a Variable with type "[int]" and as value the array converted to a string (using the *show* function of Prelude);

II. The case in which we update a single array cell with a new integer, as for example in:

```
a[2] = 12/3 + 4;
```

In this case the assignemnt is done with the *updateArray* function, also defined in the *EnvironmentManager* module. This function takes three variables: the Variable name that contains the array, the index to be updated and the integer value (represented as a String) to insert at that index. So, from our example, we would have a runtime call to the function as:

```
updateArray a 2 "8";
```

```
updateArray :: String -> Int -> String -> Parser String
updateArray arrName index newElement =
```

*Figure 20: The updateArray function in EnvironmentManager, used to update an array entry*

as the value from the arithmetic expression would be equal to 8.

The array indexing in IMPterpreter goes from 0 to n-1, where n is the

```
--Int Array
do
    varName <- identifier
    symbol "="
    arrayVal <- aexpArray
    symbol ";"
    updateEnv Variable {name=varName, vtype=t_arr_int , value = Right (show arrayVal)}
<|>
do
    arrayName <- identifier
    symbol "["
    index <- nat
    symbol "]"
    symbol "="
    newValue <- aexp
    symbol ";"
    updateArray arrayName index (show newValue)
```

*Figure 21: Assignments of arrays of integers*

length of an array.

If, having an array *a* of, say, 5 elements we perform the operation:

```
a[5] = 4;
```

or

```
a[129] = 4;
```

we will obtain anyway an updated array with 5+1 = 6 elements, the last of which being equal to 4.

### Two-dimensional arrays assignment

For matrices (or better, two-dimensional arrays) of integers, we can have 3 cases:

I. The case in which we assign an entire 2-dimensional array to a variable, as for example:

```
a = [[1,5,8],[9,0,2],[2]];
```

In this case the assignment is performed in the usual way, using the "updateEnv" function, giving as a parameter a Variable with type "[[int]]" and as value the 2-dimensional array converted to a string (using, again, the *show* function of Prelude);

II. The case in which we update a **single row** with a new integer, as for example in:

```
a[2] = [3,2,1,5];
```

To do this, the function *updateArray* previously described is used, passing as a new value the entire array, converted to a string.

If we assign a value with an out-ouf-bound index, like

```
b[5] = [2,1,3];
```

to an array *b* of 2 rows, we will obtain an updated *b* array with 2+1 rows, the last being [2,1,3].

III. Finally, the case in which we update a single entry of the matrix, or, better, a single cell of one of the arrays in the data structure. For example, we could have:

```
a[2][3] = 30;
```

This assignment is performed using the function *updateMatrixEntry* (again, the term "matrix" is used for simplicity, even though it is not completely appropriate), defined in the *EnvironmentManager* module. This function takes 4 parameters in input: the name of the variable to modify, the index of the row to update, the index of the column to update (so we have a pair of coordinates) and the new value to enter (as a String).

So in our case the interpreter would call:

```
updateMatrixEntry a 2 3 "30";
```

```
updateMatrixEntry :: String -> Int -> Int -> String -> Parser String
updateMatrixEntry matName rowIndex colIndex newElement =
```

*Figure 22: Declaration of updateMatrixEntry function, in EnvironmentManager, used to update a cell of a two-dimensional array*

If we, again, perform an assignment with a column index out of bound, we will simply append the assigned integer to the specified row. So, having c = [[2,3,4],[5,6,7]], performing:

`c[0][24] = 10;`

will result in a variable c = [[2,3,4,10],[5,6,7]], with the specified row (at index 0) with one element more.If we try to perform an assignment with an outbound row index, we get a "**`Prelude.!!: index too large`**" error and the interpreter will shut down.

```
--Int Matrices / Array of Arrays
do
    varName <- identifier
    symbol "="
    arrayVal <- aexpMatrix
    symbol ";"
    updateEnv Variable {name=varName, vtype=t_arr_arr_int , value = Right (show arrayVal)}
<|>
do
    arrayName <- identifier
    symbol "["
    index <- nat
    symbol "]"
    symbol "="
    newValue <- aexpArray
    symbol ";"
    updateArray arrayName index (show newValue)
<|>
do
    matrixName <- identifier
    symbol "["
    indexR <- nat
    symbol "]"
    symbol "["
    indexC <- nat
    symbol "]"
    symbol "="
    newValue <- aexp
    symbol ";"
    updateMatrixEntry matrixName indexR indexC (show newValue)
```

*Figure 23: Assignments of two-dimensional arrays of integers*

If, in general, we try to change a value of an array or a two-dimensional array, the parser will refuse to execute the command as it won't find the variable to modify in the environment.

### *If then else*

The "if-then-else" command is defined, as in IMP, as the selection of the language.

We have a few rules for this command:

- ➢ The condition of selection appears after the **if** keyword and must be closed between round brackets and is any boolean expression that the interpter can understand.
- ➢ The **then** keyword is not in the language, but is kind of substituted by the presence of the curly brackets, which group the instructions of the if statement and those of the else statement.
- ➢ Every instruction in the curly brackets must end with a semi-colon ";".
- ➢ The semi-colon must not be typed after the closing of the else statement (i.e. after the closing curly bracket).
- ➢ The presence of the else branch is not mandatory.

```
ifThenElse :: Parser String
ifThenElse =
    do
        symbol "if"
        symbol "("
        b <- bexp
        symbol ")"
        symbol "{"
        if b
            then do {
                program;
                symbol "}";
                do {
                    symbol "else";
                    symbol "{";
                    CR.program;
                    symbol "}";
                    return "";
                }
                <|>
                return ""
            }
        else
            do {
            CR.program;
            symbol "}";
            do
                symbol "else"
                symbol "{"
                program
                symbol "}"
                return ""
            <|>
            return ""
        }
```

*Figure 24: If then else function*

As we can suspect from the ifThenElse Parser, the program is not completely parsed and executed: doing so would result in both

branches of the if-then-else to be executed, and we clearly don't want that (as we experienced during the programming of this function… oh dear, we did experience it indeed). So the instruction is initally parsed until the condition of the if statement: based on the result of the condition, the correct branch is executed and the other one is just parsed, (and not executed) by the *program* Parser in the Parser.ReadOnlyParser.CommandROP (that's a long name) module.

So if the condition is True, then the body of the *if* is executed, while the *else* is just parsed; else, if the condition is false, the body of the *if* is parsed and ignored, while the *else* body is parsed and executed.

### *While*

The while statement is defined as in IMP to perform iterations in our programs. The condition of the while is closed in round brackets (as in the if-then-else) and its body is closed in curly brackets.

The parsing of the while is slightly more complicated and it's performed in the following steps:

1. The entire while instruction is parsed (and not executed) by the *while* Parser in the Parser.ReadOnlyParser.CommandROP module.
2. The just parsed instruction (if syntaxically correct) is then appended as a copy of the original string via the *repeatWhileString* in Parser.ReadOnlyParser.CommandROP; so now we have that the original string has been consumed, but we have a copy of it on which the parser can operate.
3. Based on the result of the condition:
    - If True the interpreter executes the *while* body with the *program* Parser of *Command*, consuming the copy of the while statement previously attached; after this, another copy of it will be added to the parsing string, in order to allow subsequential executions. Finally the *while* function will be called again to repeat the whole process.

- If False the program will just be parsed and ignored with the *program* Parser of the Parser.ReadOnlyParser.CommandROP module.

```
while :: Parser String
while =
    do {
        w <- CR.while;
        CR.repeatWhileString w;
        symbol "while";
        symbol "(";
        condition <- bexp;
        symbol ")";
        symbol "{";
        if condition
            then do {
                program;
                symbol "}";
                CR.repeatWhileString w;
                while;
            }
        else
            do {
                CR.program;
                symbol "}";
                return "";
            }
    }
```

*Figure 25: While function in Command*

### 2.2.6 Parser.EnvironmentManager

The EnvironmentManager module in Parser contains various functions (as we have already seen) that are needed for the access (reading and writing) to the Environment.

The function *updateEnv* performs a writing of a variable to the environment array; if the name of the variable we pass to it is already present in the Env, then the value of the variable is overwritten. The

type of the variable is overwritten as well, so we can completely change a variable's nature after we instantiate one.

The functions *readVariable* and *readFullVariable* are used to access either just the value or both the value and the type of a variable. To do this, the function *searchVariable* of the Env module is used.

The function *updateArray,* as we've already seen, it's used to access the value of an array Variable in the environment in order to modify one of its values. Moreover, it is used to modify an entire row of a 2-dimensional array. As we can notice from the signature of the function, the newElement to insert into the array is passed as a string. In this way we can guarantee a sort of polymorphism to this function, so that it can be able to operate for arrays of integers, booleans or strings. The function works in this way:

- first the array to modify is retrieved from the Env using the function *searchVariable*;
- then the array, stored as a string in the variable, is parsed back to its array form using the function *read* from Prelude;
- the value in the array at the i-th index is changed with the new element (which has been parsed as well) using the *replaceInArray* function in this same module;
- the new array is converted back to a string (using the *show* function from Prelude);
- the value of the variable is updated with the new array and it's stored back in the Env.

The function *updateMatrixEntry* is used to update a matrix single entry, using a pair of coordinates rowIndex and colIndex. It operates in a similar way to the *updateArray* function: it uses the function *replaceInArray* as well to first get the new row and then to update the matrix with the just computed new row.

### 2.2.7 Parser.Array

The Parser.Array module is a very big submodule that contains all the functions needed for the management of String expressions, Arrays and 2-dimensional arrays of integers, boolean and strings.

It contains the logic of parsing arithmetic expressions involving arrays and matrices, as well as boolean expressions and string expressions.

To say it once and for all, an array variable of any type (int, bool or string) can be valorized with an assignment that involves a matrix, getting one of its rows as an array. For example, having a matrix *a* = [[2,4,6],[8,2,1],[0,9,3]], we can create a new array *b* :

```
b = a[1];
```

that will be *b* = [8,2,1], since the row at index 1 in *a* is [8,2,1] (remember that the indeces start from 0).

#### *Array and matrices of Integers*

The function *aexpArray* is used to parse arithmetic expressions involing arrays (of integers, obviously). It can parse every arithmetic expression:

- Scalar multiplication, which takes an array *a* and a scalar *s* in the form:

```
a * s
```
**or**
```
s*a
```

since it is commutative; this operation multiplies every entry of the array by the scalar *s.*

- Scalar division, which takes an array *a* and a scalar *s* in the form:

```
a / s
```

and performs the integer division of each element of *a* by the scalar *s*;

- Term by term addition (+), subtraction (-), multiplication (*) and division (/), which take two arrays and perform the corresponding arithmetic operation between each of their values having the same index: the result will have a number of elements equal to the minimum of the lengths of the two arrays.

- Inner product, which is performed with the '@' character between two arrays, and performs the mathematical inner product between them. The two arrays are needed to have the same length.
- Array concatenation, performed with the '++' symbol between two arrays of integers, which concatenates the second array to the end of the first one and returns the final array, as described in subsection 1.2. This operation is found also in the other two types of expressions, bexp and stringExp.
- Just a simple array of integers.

The parser *arrayInt* which parses an array of integers, or a variable name associated to a Variable in the Env containing an array of integers.

The parser *aexpMatrix* parses arithmetic expressions involing arrays of integers. The following operations can be parsed:

- scalar product (*), which takes an arithmetic expression (a number) and a matrix of integers and multiplies every entry of the matrix by that scalar; it is commutative.
- scalar division (/), which takes a matrix of integers and an arithmetic expression (a number) and performs the integer division on every entry of the matrix by that scalar.
- Matrix addition (+) and subtraction (-) which take 2 matrices and perform the term-by-term addition and subtraction, respectively. The shapes of the matrices must be compatible.
- Matrix multiplication (performed with @) which take 2 matrices $m_0$ (of size x by y) and $m_1$ (of size y by z) and performs the matrix multiplication (i.e. the dot product) between the two. The shapes of the matrices must be compatible.

- Matrix horizontal concatenation (performed with ++), as described in subsection 1.2.
- Matrix vertical concatenation (performed with #), as described in subsection 1.2.

This function can also parse a simple matrix of integers.

The parser *matrixInt* parses a 2-dimensional array of integers, or an identifier associated to a Variable in the Env containing an array of arrays of integers.

**Array and matrices of Booleans**

The parser bexpArray parses a boolean expression involving arrays of booleans to an array of booleans. It can parse:

- term-by-term logical conjunction (using '&&'): takes two arrays of booleans $b_0$ $b_1$ and returns another one which has as entries the result of the logical conjunction of the terms having the same indexes in $b_0$ $b_1$.
- term-by-term logical disjunction (using '||'): takes two arrays of booleans $b_0$ $b_1$ and returns another one which has as entries the result of the logical disjunction of the terms having the same indexes in $b_0$ $b_1$.
- Array concatenation (using '++').
- A result of the arrayBterm parser.
- A result of the arrayBool parser.

A parser *arrayBterm* is used to parse grouping of other array boolean expressions (using parenthesis), or the logical negation (using the '!' symbol), which takes an array of booleans b and returns the same array with the commuted truth values.

A parser *arrayBool* parses a string containing an array of booleans or a Variable identifier to an array of booleans.

The parser *bexpMatrix* parses a boolean expression regarding boolean matrices to a 2-dimensional arrays of booleans. It can parse:

- negation (performed with '!'), which negates all the entries of a boolean matrix;
- term-by-term conjunction (performed with '&&'), which takes two matrices of booleans $b_0$ $b_1$ and returns another one which has as entries the result of the logical conjunction of the terms having the same indexes in $b_0$ $b_1$. The matrices must be of a compatible shape.
- term-by-term disjunction (performed with '||'), which takes two matrices of booleans $b_0$ $b_1$ and returns another one which has as entries the result of the logical disjunction of the terms having the same indexes in $b_0$ $b_1$. The matrices must be of a compatible shape.
- Matrix horizontal concatenation (performed with ++);
- Matrix vertical concatenation (performed with '#');
- just a boolean matrix.

The parser *matrixBool* parses a string to an array of arrays of booleans. It can also parse an identifier associated with a Variable in the Env containing a matrix of booleans.

**Strings and Arrays (and matrices) of Strings**

In this section is defined the *stringExp* parser, which can parse expressions involving strings; the only managed expression is the string concatenation (performed using the symbols '++') which takes two strings and concatenates them together in one string (as described in subsection 1.2).

*stringTerm* is a parser that parses a string value (delimited by single quotes symbols ') or an identifier of a variable containing a string.

The parser *stringExpArray* is used to parse expressions involing arrays of strings. The only operation it can parse is the concatenation of arrays (++). Otherwise it can parse an *arrayString.*

The parser *arrayString* parses a string describing an array of strings (delimited with the square brackets) to an array of strings. Otherwise, it can parse a variable identifier associated with a Variable in the environment of type '[string]'.

The parsing of matrix of strings expression is done with the *stringExpMatrix* parser. This can parse:

- the matrix horizontal concatenation (performed with '++');
- the matrix vertical concatenation (performed with '#').
- a matrix of strings (with the *matrixString* parser).

The *matrixString* parser can parse a string describing an array of strings (delimited by double square brackets) to a 2-dimensional array of strings. It is also able to parse a variable identifier associated with a Variable in the environment of type '[[string]]'.

The Parser.Array module also contains other functions to operate on matrices, such as matrix transposition, matrix multiplication and vector product.

**2.2.8 Parser.ReadOnlyParse**

This module is a submodule of Parser and contains in turn other three submodules: AexpROP, BexpROP and CommandROP.

These three modules implement the logic of reading a string in order to just check the correctness and execute the if and while statements in the proper way. In fact, we may remember that the interpretation of an if-then-else command is done by ignoring either the if or the else body, according to the truthness of the condition. In the same way in the while interpretation we want an iteration to be executed only if the condition results true.

### *CommandROP*

This module contains several parsers:

- program and command, which are used, analogously to those in the Command module, to parse one or many commands, this time returning the parsed string to the input stream, so that it can be eventually executed later on;
- assignment, which parses an assignment instruction until a semicolon ";" is met, and returns the parsed command;
- ifThenElse, which parses the entire if-then-else statement (also the programs in the if and else bodies) and the boolean expression of the condition in it and returns the program string with the parsed boolean expression in it;
- while, which is used to parse a while condition and the program in it, and returns the parsed string;
- repeatWhileString, which is used, as we may remember, in the Command.while function in order to append the parsed program of the while to the input stream, so that it can be executed again.

### *AexpROP and BexpROP*

The AexpROP and BexpROP in Parser.ReadOnlyParser are quite uninteresting, since they just read arithmetic or boolean expressions in order to parse them and return them to the input stream for parsing.

So we find here the function to parse additions, subtractions, logic disjunctions and conjunctions, and so on (we can find the relative omonymous functions in Parser.Aexp and Parser.Bexp).

### 2.3 Main.hs

The Main module contains the command line interface part of IMPterpreter and, therefore, the access point of the interpreter for the user.

It provides an environment in which the user can simply type instructions that will be parsed and interpreted. Once the user has typed in a set of commands and presses enter, his program will be interpreted and the environment resulting from the computation will be shown.

The environment is saved for the user session, so that he can reuse all the variables he defines in previous commands. Once the session is closed, the environment is flushed and lost.

Main contains the function *printLogo* that prints the logo of IMPterpreter (done in ASCII art) and the user instructions for the interaction.

It also contains the function *repeatProgram* which:

- gets the user input (using the function *getInputProgram*);
- if the user didn't type ':q' or ':exit' (in that case IMPterpreter would stop its execution), calls the function *evaluate* of Parser in order to interpret what the user typed in;
- shows the resulting environment to the user if the parsing succeeded, while shows the eventual error otherwise;
- calls itself recursively to give the input the possibility to type in another program, or just exit; *repeatProgram* takes as an argument an Env: indeed the environment resulting from the last interpretation is passed in order to mantain the environment for the next computations.

Finally Main contains the function *main* which is the function that starts IMPterpreter: it simply executes the function *printLogo*, in

order to print the logo and the instructions to the UI, and the function *repeatProgram* giving a fresh empty Env as a parameter.

### 2.4 Parser.hs

Finally we have the Parser module! This module links all the other one together and contains a single function, *evaluate*.

This function takes a string to parse and an Environment and returns Either a String, containing an error text in case of parsing failure, or a Env, containing the environment resulting from the interpetation, in case of success.

```
module Parser (evaluate)
where
import Environment
import Parser.Core
import Parser.EnvironmentManager
import Parser.Aexp
import Parser.Bexp
import Parser.Command
import Parser.Array

evaluate :: String -> Env -> Either String Env
evaluate p inEnv = case parse program inEnv p of
    Nothing              -> Left "Error in the evaluation of the program.\nPlease check your syntax."
    Just (e, _, "")      -> Right e
    Just (e, _, residual) -> Left $ "Error in the evaluation of the program.\n"++
        "Please check your syntax.\nThe error occured around the statement:\n \""++
        (take 300 residual) ++"\""
```

*Figure 26: The Parser module*

## 3 Interface of IMPterpreter

IMPterpreter has a Command Line Interface (CLI) which lets a user type in its programs and observe the results in the environment that is printed out after an interpretation.

In order to run the interpreter, the application executable (simply called Main) must be run via a terminal application.

Launching this will show to the user the logo and the usage instructions, which are, by the way, very simple.


*Figure 27: CLI of IMPterpreter*

The user can decide to exit the program by typing ‘:q’ or ‘:exit’, or simply “CTRL+Z”; he can, instead, type in the program written according to the syntax of our language.

Once the user is satisfied with what he has written, he can press Enter and observe the resulting environment, containing the variables he defined in his program.

Once executed, the environment will be saved for all the next computations of the sessions, and only be emptied when IMPterpreter is rebooted. This way, the user can use and manipulate variables coming from previous computations.


*Figure 28: The results of two different computations*

## 4 Conclusions

We have explored how IMPterpreter works and seen some internal mechanisms. The level of abstraction could have been higher (or lower, sure), but I wanted to give enough detail so that the structure of the interpreter could have been as clear as possible, without being too boring.

Even though I implemented quite some features, the language could be extended in many many different ways and in many different direction. First of all, access to array and matrix elements should be implemented, which has not been implemented to try to limit

One thing that comes natural to me is the implementation of *functions*, so that we could have function identifiers and expressions, recursion, composition and all that comes with it. Some simple functions ready to be implement would be the function length, to get the length of an

array or a string, string manipulations functions, mathematical functions and so on. The sky's the limit.