Lab assessment for the exam in Formal Methods for computer science

a.a. 2020/2021

sIMP: A simple IMPerative programming language

Professor: Giovanni Pani

Student: Francesco Benedetti

Download or clone the project from the Github repo

Table of contents

1: Introduction to sIMP

sIMP, as the name suggests, is a simple programming language. It is an extension of the IMP language, and its syntax is very similar to the syntax used by the most popular programming language (especially Java). At the current state, sIMP allows the user to manage variables of **integer** or **boolean** type. When the variables are declared, the user can choose to immediately assign them a value, or to assign a value later on.
The operations that can be done on the integers are the classical arithmetical operations (sum, product, difference, division, power), and compare operations. For the booleans, the three logical operators are provided.

The language also provides the **array** and **stack** data structures, which at the moment can only contain integer values.

Here is a list of the command statements that can be used:

- while
- do while
- if – else (the else is not mandatory)
- skip

1.1: The grammar

This is the representation of the language's grammar in Extended Backus-Naur Form (EBNF):

```
program ::= com | program

com ::= declareBoolean ";"
    | declareInteger ";"
    | declareArray ";"
    | declareStack ";"
    | assignBoolean ";"
    | assignArrayPosition ";"
    | assignWholeArray ";"
    | assignInteger ";"
    | push ";"
    | pop ";"
    | ifelse
    | whiledo
    | dowhile ";"
    | skip ";"


digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
integer ::= {"-"} digit {digit}
letter ::= ["a"-"z" "A"-"Z"]
arithmeticIdentifier ::= letter {letter | digit | "_" }
booleanIdentifier ::= letter {letter | digit | "_" }
constArr ::= "[" {AExpr ","} AExpr "]"

declareBoolean ::= "bool" booleanIdentifier {"=" BExpr}
declareInteger ::= "int" arithmeticIdentifier {"=" AExpr}
declareArray ::= "array[" AExpr "]" arithmeticIdentifier {"=" constArr}
declareStack ::= "stack" arithmeticIdentifier
assignBoolean ::= booleanIdentifier "=" BExp
assignArrayPosition ::= arithmeticIdentifier "[" AExpr "]" "=" AExpr
assignWholeArray ::= arithmeticIdentifier "=" constArr
```

```
assignInteger ::= arithmeticIdentifier "=" AExpr
push ::= "push" "(" arithmeticIdentifier "," AExpr ")"
pop ::= "pop" "(" arithmeticIdentifier ")"
ifelse ::= "if" "(" BExp ")" "{" program "}" {"else" "{" program "}"}
whiledo ::= "while" "(" BExp ")" "{" program "}"
dowhile ::= "do" "{" program "}" "while" "(" BExp ")"
skip ::= "skip"

AExpr ::= arithmeticTerm {{"+" | "-"} arithmeticTerm}

arithmeticTerm ::= arithmeticFactor {{"*" | "/" | "^"} arithmeticFactor}
arithmeticFactor ::= "(" AExpr ")"
    | integer
    | constArr
    | "top" "(" arithmeticIdentifier ")"
    | arithmeticIdentifier
    | arithmeticIdentifier "[" AExpr "]"

BExpr ::= booleanTerm {"or" booleanTerm}
booleanTerm ::= booleanFactor {"and" booleanFactor}
booleanFactor ::= "True"
    | "False"
    | "empty" "(" arithmeticIdentifier ")"
    | booleanIdentifier
    | "(" BExpr ")"
    | "!" BExpr
    | AExpr "<" AExpr
    | AExpr ">" AExpr
    | AExpr "==" AExpr
    | AExpr "<>" AExpr
    | AExpr "<=" AExpr
    | AExpr ">=" AExpr
```

And here is an example of program written in sIMP (more examples can be found in the **test** section):

```
int a = 0;
int b;
if(a == 0){
    b = 1;
}
else {
    b = 2;
}
```

This is the grammar that sIMP recognizes. The **Values** data type identifies the types of data that the language is able to manage. It is possible to note that, even if the array and stack data structures are defined to contain integer values, in the language they are treated as standalone data types.

```
data Values =
    Integer Int
    | Boolean Bool
    | Array [Int]
    | Stack [Int]
```

AExpr and BExpr define, respectively, the arithmetic expressions (operations that return an integer) and Boolean expressions (operations that return a Boolean).

```
data AExpr =
    Const Int
    | ConstArr [Int]
    | Ar String AExpr
    | ArithmeticIdentifier String
    | Top String
    | Add AExpr AExpr
    | Diff AExpr AExpr
    | Div AExpr AExpr
    | Prod AExpr AExpr
    | Power AExpr AExpr
    deriving Show

data BExpr =
    BVal Bool
    | BooleanIdentifier String
    | Empty String
    | And BExpr BExpr
    | Or BExpr BExpr
    | Not BExpr
    | Lt AExpr AExpr
    | Gt AExpr AExpr
    | Eq AExpr AExpr
    | Different AExpr AExpr
    | Lte AExpr AExpr
    | Gte AExpr AExpr
    deriving Show
```

Com is for the various commands that the language allows.

```
data Com = DeclareBoolean String (Maybe BExpr)
    | DeclareInteger String (Maybe AExpr)
    | DeclareArray String AExpr (Maybe AExpr)
    | DeclareStack String
    | AssignBoolean String BExpr
    | AssignInteger String AExpr
    | AssignArrayPosition String AExpr AExpr
    | AssignWholeArray String AExpr
    | Push String AExpr
    | Pop String
    | Ifelse BExpr Program (Maybe Program)
    | Whiledo BExpr Program
    | Dowhile Program BExpr
    | Skip
    deriving Show
```

And lastly, the program, which is a list of commands.

```
type Program = [Com]
```

2: The structure of the language

sIMP is composed of two main modules, which work sequentially. The first module is the **parser**, whose main goal is to check the syntactic correctness of the code given in input. If any error is

found in the code, then the execution stops and an error message is displayed. The message contains the rows which weren't parsed (the first of which is the row containing the syntactic mistake that caused the interruption). The parser has also another role: it creates an internal representation of the program, suitable for the next module that comes in action: the **interpreter**. The interpreter takes care of the actual evaluation of the program. If some semantic mistake (such as trying to use an undeclared variable) is encountered, the execution is stopped and an error message is shown. If the evaluation of the whole program was successful, then the current state of the environment is printed as final output.

Both the parser and the interpreter were implemented in Haskell, using only the Prelude library.

2.1: The parser

The parser type is defined in this way:

```
newtype Parser a = P{
    runParser :: String -> Maybe(a, String)
}
```

The runParser function takes in input the string to parse. Since the string cannot be parsed all at once, the parser will return a pair, containing the parsed part (which can be anything), and the rest of the input, which will be parsed by further parsers. However, it could happen that the parser is not able to parse the input. This is why the returned value is wrapped in a Maybe. If the parsing went wrong, `Nothing` will be returned.

Since we want to combine more parsers and let them work in a sequential way, we need to create an instance of the Functor, Applicative, Monad and Alternative classes.

2.1.1: The functor

The functor class allows to apply a function on values that are wrapped, by using the **fmap** function. The "container" of the wrapped value is called **functor**. fmap unwraps the wrapped value, passes it to the function we want to use and finally rewraps the result. The fmap function is defined in this way in Haskell:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

As we can see, fmap takes a function (a -> b) and a wrapped value (f a) to return the value returned by the function, but wrapped in the same functor (f b).

Let's do an example to make the role of the functor more clear. Suppose we have an integer value wrapped in a Maybe. For example, we have `(Just 10)`. We want to add 4 to our number. Since the sum is defined on integers, we can't do `(Just 10) + 4` because we will get an error. In this situation, the fmap function comes in our aid, and we can use it to perform our operation. We can write `fmap (+4) (Just 10)` and our result will be `(Just 14)`
This is the implementation of the functor for our parser datatype:

```
instance Functor Parser where
    fmap f (P p) = P(\input -> case p input of
        Nothing -> Nothing
        Just(x, input') -> Just(f x, input'))
```

2.1.2: The applicative

What if, in the previous example, we wanted to do something like this:

```
Just (+4)(Just 10)?
```

We would have got an error even using the fmap operator, because it doesn't allow to work using wrapped functions. This is why Applicatives are very handful: because they allow to work on wrapped functions too. In Haskell, the applicative class is defined in this way:

```
class (Functor f) => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

The **pure** function just takes an input and wraps it in the functor. The **<*>** operator takes a wrapped function and a wrapped value, unwraps them and then runs the function on the unwrapped value. The result gets wrapped again and then is returned.

Since our parsers will work in chain, each parser will return a wrapped function that the next one will have to run on the portion of input of its competence. Hence we need to define the applicative for the Parser type. This is the implementation:

```
instance Applicative Parser where
    pure f = P((\input -> Just(f, input)))
    P(p1) <*> P(p2) = P(\input -> case p1 input of
        Nothing -> Nothing
        Just(f, input') -> case p2 input' of
            Nothing -> Nothing
            Just(x, input'') -> Just(f x, input'')
        )
```

2.1.3: The Monad

The Monad allows to define the actual sequencing procedure. Haskell defines the Monad as follows:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
```

And this is the implementation of the Monad for Parser:

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

Moreover, defining a Monad introduces the use of the **do notation**. This kind of notation is a syntactic sugar, that allows to use monads emulating, from a syntactical point of view, the writing style of the common imperative languages.

2.1.4: The Alternative

In Haskell the Alternative class implementation is the following:

```
class Applicative f => Alternative f where
  empty :: f a
```

```
(<|>) :: f a -> f a -> f a
some :: f a -> f [a]
many :: f a -> f [a]
many x = some x <|> pure []
some x = pure (:) <*> x <*> many x
```

The Alternative class introduces very useful operators:

- **empty** is an applicative computation with zero results
- **<|>** is a binary function, which combines two computations and returns the first non empty one. For our parser it becomes handful because when we have an input string, we don't know a priori which will be the actual parser that will be able to process it correctly. Therefore we can use this operator to create a "chain" of different parsers that will be ran one after the other on the same input, until a parser that doesn't return `Nothing`. If all the parsers return Nothing, then it means that something in the input was wrong.
- **some** and **many** are used to define the concept of repeated application. The first function applies the parser as many times as possible, but at least once. The second function is similar to the first, but allows for no parse as well.

Implementation of the Alternative related for our Parser:

```
instance Alternative Parser where
    empty = P(\input -> Nothing)
    P(p1) <|> P(p2) = P(\input -> p1 input <|> p2 input)
```

Lastly, the implementation of the **chain** operator, that allows to use the leftmost association when combining two or more parsers:

```
chain :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chain` op = do a <- p; rest a
    where
        rest a = (do
            f <- op
            b <- p
            rest (f a b))
            <|> return a
```

2.1.5: Implementation of the basic parsers

After the previous step, we can start implementing our basic parsers.

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

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

We first have to define a parser called **item**. This parser fails if the input string is empty, otherwise returns a parser that has parsed the first char. We also define a function **sat** that checks if a

character respects some property. We can now procede to the implementation the parsers for a digit, an uppercase letter, a lowercase letter, a letter, an alphanumeric character, a single character. Here I will report the implementation of just two of the above parsers, but the others can be implemented in the same way, using the functions defined in the **Utils.hs** file:

```
--Parser of a digit
digitP :: Parser Char
digitP = sat isDigit

-- Parser of a single char
charP :: Char -> Parser Char
charP x = sat (== x)
```

The following code snippet contains the implementation of the parser of spaces and the parser of tokens:

```
spacesP :: Parser ()
spacesP =
    do
        x <- many (sat isSpace)
        return ()

tokenP :: Parser a -> Parser a
tokenP p =
    do
        spacesP
        v <- p
        spacesP
        return v
```

The parser for tokens is important because allows to ignore the spaces that a programmer can use while writing a program. Thanks to this parser, the final user will be able to write both

```
int a = 0 ;
```

and

```
int a=0;
```

without getting errors.

The parsers that were just defined can now be easily combined to obtain parsers for strings, identifiers and integer numbers. The parser for arrays is more complex and is reported here:

```
arrayP :: Parser [Int]
arrayP = do
    symbolP "["
    n <- integerP
    ns <- many (
        do
            symbolP ","
            integerP)
    symbolP "]"
    return (n:ns)
```

We now have all the tools that we need to implement parsers for arithmetic expressions, boolean expressions and commands.

2.1.6: Arithmetic expressions parser

The parser for arithmetic expressions (aExprP) is defined in the following way:

```
aExprP :: Parser AExpr
aExprP = arithmeticTerm `chain` op where
    op = do
        symbolP "+"
        return Add
        <|>
        do
            symbolP "-"
            return Diff

arithmeticTerm :: Parser AExpr
arithmeticTerm = arithmeticFactor `chain` op where
    op = do
        symbolP "/"
        return Div
        <|>
        do
            symbolP "*"
            return Prod
        <|>
        do
            symbolP "^"
            return Power

arithmeticFactor :: Parser AExpr
arithmeticFactor = do
    symbolP "("
    expr <- aExprP
    symbolP ")"
    return expr
    <|>
    do
        (Const <$> integerP)
    <|>
    do
        (ConstArr <$> arrayP)
    <|>
    do
        symbolP "top"
        symbolP "("
        name <- identifierP
        symbolP ")"
        return (Top name)
    <|>
    do
        i <- identifierP
        do
            symbolP "["
            n <- aExprP
            symbolP "]"
            return (Ar i n)
            <|>
            return (ArithmeticIdentifier i)
```

We can notice how the parser uses two other parsers. This is done to ensure the correct priorities between arithmetic operations. In fact, **arithmeticFactor** is the one with highest priority. It parses expressions between brackets, constant integers, arrays, the "top" operation for stacks and the identifiers. These are the operations that have to be evaluated for first by the interpreter. **arithmeticTerm** has lower priority, since it parses the product, the division and the power. Lastly, **aExprP** parses the sum and difference operations, which are the ones with lowest priority grade. The chain operator was used to ensure the correct evaluation of expressions, in a left-most way.

2.1.7: Boolean expressions parser

For the boolean expressions parser (bExprP) I used an approach very similar to the one used for aExprP in order to grant the correct precedences. In this case, the "or" operation has the lowest rank of priority. Then comes the "and" operation. The operations with highest priority grade are the expressions around brackets, the "not" operation, the comparisons between arithmetic expressions and the "empty" operator for stacks.

```
bExprP :: Parser BExpr

bExprP = booleanTerm `chain` op where
    op = do
        symbolP "or"
        return Or

booleanTerm :: Parser BExpr
booleanTerm = booleanFactor `chain` op where
    op = do
        symbolP "and"
        return And

booleanFactor :: Parser BExpr
booleanFactor = do
    symbolP "True"
    return (BVal True)
    <|>
    do
        symbolP "False"
        return (BVal False)
    <|>
    do
        symbolP "empty"
        symbolP "("
        name <- identifierP
        symbolP ")"
        return (Empty name)
    <|>
    do
        symbolP "("
        bExpression <- bExprP
        symbolP ")"
        return bExpression
    <|>
    do
        symbolP "!"
        (Not <$> bExprP)
```

```
    <|>
    do
        a <- aExprP
        do
            symbolP "<"
            (Lt a <$> aExprP)
            <|>
            do
                symbolP ">"
                (Gt a <$> aExprP)
            <|>
            do
                symbolP "=="
                (Eq a <$> aExprP)
            <|>
            do
                symbolP "<>"
                (Different a <$> aExprP)
            <|>
            do
                symbolP "<="
                (Lte a <$> aExprP)
            <|>
            do
                symbolP ">="
                (Gte a <$> aExprP)
    <|>
    do
        i <- identifierP
        return (BooleanIdentifier i)
```

2.1.8: Commands parser

For the commands it is not necessary to define an hierarchy of priorities, therefor the commands parser (**commandP**) is just the alternative between the parsers of the single commands:

```
commandP :: Parser Com
commandP =
    integerDeclare
    <|>
    booleanDeclare
    <|>
    arrayDeclare
    <|>
    stackDeclare
    <|>
    integerAssign
    <|>
    booleanAssign
    <|>
    arrayAssignPosition
    <|>
    arrayAssignWhole
    <|>
    push
    <|>
    pop
```

```
    <|>
    ifThenElse
    <|>
    while
    <|>
    dowhile
    <|>
    skip
```

2.1.8.1: Declaration of integer variables

Recall that sIMP allows to declare a variable and immediately initialize it, or just declare it without assigning a value. Therefore, the **DeclareInteger** operation requires as input the name of the variable and a Maybe, that is `Nothing` if the user does not assign a value. The parser for the declaration of integer variables is implemented in this way:

```
integerDeclare :: Parser Com
integerDeclare = do
    symbolP "int"
    name <- identifierP
    do
        symbolP "="
        val <- aExprP
        symbolP ";"
        return (DeclareInteger name (Just val))
        <|>
        do
            symbolP ";"
            return (DeclareInteger name Nothing)
```

The language requires the user to specify the keyword "**int**", followed by the name of the variable. If, after the identifier, the parser meets the symbol "=", that means that the user is already assigning a value to the new variable. On the other hand, if the name of the variable is followed by the semicolon, the user is not initializing the variable.

2.1.8.2: Declaration of boolean variables

For boolean variables declarations, the idea is the same as the one for integer variables. In this case, the user is required to specify the keyword "**bool**" to state the intention to declare a boolean variable.

```
booleanDeclare :: Parser Com
booleanDeclare = do
    symbolP "bool"
    name <- identifierP
    do
        symbolP "="
        val <- bExprP
        symbolP ";"
        return (DeclareBoolean name (Just val))
        <|>
        do
            symbolP ";"

            return (DeclareBoolean name Nothing)
```

2.1.8.3: Declaration of arrays

For arrays declaration, the syntax is slightly different, since the user has to specify the dimension of the array being created. This parameter is specified between square brackets as in Java. The keyword, in this case, is "**array**". Also in this case, the user can choose to immediately assign a value to the new array.

```
arrayDeclare :: Parser Com
arrayDeclare = do
    symbolP "array"
    symbolP "["
    size <- aExprP
    symbolP "]"
    name <- identifierP
    do
        symbolP "="
        val <- aExprP
        symbolP ";"
        return (DeclareArray name size (Just val))
        <|>
        do
            symbolP ";"
            return (DeclareArray name size Nothing)
```

Here is an example of code that creates an array of 5 elements and initializes it:

```
array[5] a = [1,2,3,4,5];
```

2.1.8.4: Declaration of stacks

Normally, programming languages don't allow the user to initialize a stack within its declaration. sIMP follows the same approach, so a stack can only be populated after its declaration. This means that the code needed to declare a stack is just composed by the keyword "**stack**" and the identifier.

```
stackDeclare :: Parser Com
stackDeclare = do
    symbolP "stack"
    name <- identifierP
    symbolP ";"
    return (DeclareStack name)
```

2.1.8.5: Integer / boolean variables assignation

The parsers for the command of assignation of a value to a pre-declared variable, parse the name of the variable and the new value to assign. The procedure is basically the same for integer and boolean variables:

```
integerAssign :: Parser Com
integerAssign = do

    name <- identifierP
    symbolP "="
    val <- aExprP
    symbolP ";"
    return (AssignInteger name val)
```

```
booleanAssign :: Parser Com
booleanAssign = do
    name <- identifierP
    symbolP "="
    val <- bExprP
    symbolP ";"
    return (AssignBoolean name val)
```

2.1.8.6: Array assignation

We can either assign a new value to a specific position of a pre-declared array, or either redefine completely the array. The first case is covered by the **arrayAssignPosition** parser:

```
arrayAssignPosition :: Parser Com
arrayAssignPosition = do
    name <- identifierP
    symbolP "["
    position <- aExprP
    symbolP "]"
    symbolP "="
    val <- aExprP
    symbolP ";"
    return (AssignArrayPosition name position val)
```

For example, if we want to assign the value **7** at the 5th position of the array **a** we have to write something like this (indexes start from zero):

```
a[4] = 7;
```

Of course the size of **a** must be equal o bigger to 5, but the parser can't check this constraint. The interpreter will take care of that.

The case of "redefinition" of an array is covered by the **arrayAssignWhole** parser:

```
arrayAssignWhole :: Parser Com
arrayAssignWhole = do
    name <- identifierP
    symbolP "="
    val <- aExprP
    symbolP ";"
    return (AssignWholeArray name val)
```

Example:

```
a = [1,2,3,4];
```

We can do this operation only if the array **a** was declared as an array of size equal to 4. If the size is bigger or smaller, the interpreter will return an error.

2.1.8.7: Push and pop

2.1.8.8: If – else

2.1.8.9: While and do-while

2.1.8.10: Skip

2.1.9: Program parser

2.1.10: Additional functions

2.2: The interpreter

The interpreter executes the code provided by the user. It implements the eager evaluation strategy, meaning that an expression is evaluated as soon as it is met. To perform this approach, call by value is adopted.

The interpreter works with an internal representation of the programs, produced by the parser. For each of the possible expressions or commands is provided a function that allows the interpreter to correctly evaluate it. Each of these functions take as input the command to evaluate and the current state of the environment.

2.2.1: The environment

Programming languages allow the user to declare variables and store values in them. The variables can then be used at runtime to perform a multitude of tasks. When a variable is declared, the languages allocate in the memory a space in which the value of that variable will be stored for the entire execution of the program.

At this point, we introduce a new datatype, called **Variable**. It has the role to store one single variable. Variable is a structure composed of three fields: the variable name, the variable type and the value (which can be Nothing).

```
data Variable = Variable {
    var_name:: String,
    var_type:: String,
    var_value:: Maybe(Values)}
```

Since a program can (and almost certainly will) contain more than one variable, we need a structure to store a set of variables. Therefore, sIMP stores all the variables in a list, called environment (**Env**).

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

2.2.2: The "Output" type

The Output datatype is like a Maybe. It can either contain a value (Result) or either an error message, which obviously changes depending on the situation. This was done to help the user identifying the eventual semantic mistakes in the code that caused the interruption.

```
data Output a =
    Result a
    | Error String
```

For Output I also implemented the Functor, Applicative and Eq (Eq allows to confront two wrapped values and see if they are equal) in order to use function composition or pass a value wrapped in an Output to a function.

```
instance Functor Output where
    fmap _ (Error a) = Error a
    fmap f (Result a) = Result (f a)

instance Applicative Output where
    pure = Result
    (Result f) <*> (Result j) = Result (f j)
```

```
instance (Eq a) => Eq (Output a) where
    Result r1 == Result r2 = r1 == r2
    Error e == Result r = False
    Result r == Error e = False
    Error e1 == Error e2 = e1 == e2
```

The evaluator of commands does not return the Output data type, but another type, called OutputEnv. It is basically the same as Output. The only difference is that the error field contains both an error message and the command that caused the error to happen.

```
data OutputEnv =
    ResultEnv Env
    | ErrorEnv (String, Com)
```

2.2.3: Functions to manage the environment

Two functions were implemented to manage the Env type. The first is **readVariable**. It receives in input the name and the type of a variable and scans the environment until it finds, if exists, the corresponding variable and returns it.

```
readEnv :: Env -> String -> String -> Maybe(Variable)
readEnv [] n t = Nothing
readEnv (x:xs) n t = if (var_name x) == n && (var_type x) == t
    then Just x
    else readEnv xs n t
```

The second is **modifyEnv**. It takes as input a variable and adds it to the environment. If a variable with the same name and value already exists, the value gets overwritten.

```
modifyEnv :: Env -> Variable -> Env
modifyEnv [] newVar = [newVar]
modifyEnv (x:xs) newVar =  if (var_name x) == (var_name newVar) &&
(var_type x) == (var_type newVar)
    then [newVar] ++ xs
    else [x] ++ modifyEnv xs newVar
```

2.2.4: Arithmetic expressions evaluation

The evalAexpr function evaluates the arithmetic expressions that the grammar allows, but not all of them. This function takes care of evaluating only the operations that return a single value. In fact there is one operation that returns an array, and cannot be evaluated by this module.

```
evalAexpr :: AExpr -> Env -> Output Int
evalAexpr (Const k) _ = Result k
evalAexpr (Ar name pos) env =
    case (readEnv env name "array") of
        Nothing -> Error "The array does not exist"
        Just (v) -> case (var_value v) of
            Nothing -> Error "You are trying to read from an empty array"
            Just(Array v) -> Result (readArray v p)
                    where Result p = (evalAexpr pos env)
evalAexpr (ArithmeticIdentifier name) env =
    case (readEnv env name "int") of
        Nothing -> Error "The integer variable does not exist"
        Just (v) -> case (var_value v) of
            Nothing -> Error "Empty variable"
```

```
            Just(Integer i) -> Result i
evalAexpr (Top name) env =
    case (readEnv env name "stack") of
        Nothing -> Error "The stack does not exist"
        Just (v) -> case (length s == 0) of
            True -> Error "Empty stack!"
            False -> Result (readArray s ((length s) - 1))
            where Just (Stack s) = (var_value v)
evalAexpr (Add a b) env = (+) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalAexpr (Diff a b) env = (-) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalAexpr (Div a b) env = if (evalAexpr b env) == Result 0
    then error "Division by zero"
    else (ratio) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalAexpr (Prod a b) env = (*) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalAexpr (Power a b) env = (power) <$> (evalAexpr a env) <*> (evalAexpr b env)
```

It is possible to notice how the common arithmetic operations are evaluated by exploiting the advantages provided by the Applicative operator. Note also that the evaluation of the operation of reading from a specific cell of an array actually consists in doing one additional evaluation, since the index provided by the user can be an arithmetic expression.

The functions **readArray**, **ratio** and **power** are defined in the Utils.hs file.

2.2.5: Evaluation of operations that return an array

This function evaluates operations that return a whole array. It will be very useful during the evaluation of commands, for evaluating the assignments that involve arrays.

```
evalArrayOperation :: AExpr -> Env -> Output [Int]
evalArrayOperation (ConstArr ar) _ = Result ar
evalArrayOperation (ArithmeticIdentifier name) env =
    case (readEnv env name "array") of
        Nothing -> Error "The array does not exist"
        Just (v) -> case (var_value v) of
            Nothing -> Error "Empty array"
            Just(Array v) -> Result v
```

2.2.6: Evaluation of boolean operations

Same approach as the one used for the evaluation of arithmetic expressions. Also in this case the applicative operator was used for evaluating the logical operations and the comparisons.

```
evalBExpr :: BExpr -> Env -> Output Bool
evalBExpr (BVal b) _ = Result b
evalBExpr (BooleanIdentifier name) env =
    case (readEnv env name "bool") of
        Nothing -> Error "The boolean variable does not exist"
        Just(v) -> case (var_value v) of
            Nothing -> Error "Empty variable"
            Just(Boolean b) ->Result b
evalBExpr (Empty name) env = case (readEnv env name "stack") of
        Nothing -> Error "The stack does not exist"
        Just (v) -> Result (length s == 0)
            where Just (Stack s) = (var_value v)
evalBExpr (And a b) env = (&&) <$> (evalBExpr a env) <*> (evalBExpr b env)
evalBExpr (Or a b) env = (||) <$> (evalBExpr a env) <*> (evalBExpr b env)
evalBExpr (Not a) env = not <$> (evalBExpr a env)
evalBExpr (Lt a b) env = (<) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalBExpr (Gt a b) env = (>) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalBExpr (Eq a b) env = (==) <$> (evalAexpr a env) <*> (evalAexpr b env)
```

```
evalBExpr (Different a b) env = not <$> (evalBExpr (Eq a b) env)
evalBExpr (Lte a b) env = (<=) <$> (evalAexpr a env) <*> (evalAexpr b env)
evalBExpr (Gte a b) env = (>=) <$> (evalAexpr a env) <*> (evalAexpr b env)
```

2.2.7: Evaluation of commands

```
commandExec :: Com -> Env -> OutputEnv
```

The commands evaluator defines a different behavior for each of the possible configuration of the commands that the language provides (except for the if-else, while and do while statements, which are taken care of by the evaluator that will be discussed in the following section). For variable declarations, the interpreter will first check whether a variable with the same name and type already exists. If that happens, an error message is returned. Otherwise, a new variable will be added to the environment. Remember that it is possible to immediately initialize the variable. If the user does that, the **`var_value`** of the created variable will be (Just something), otherwise **`Nothing.`** As atated in the parser section, when declaring an array we have to specify the dimension. When initializing the array. the parser isn't able to check if the provided dimension and the actual size of the array coincide. The interpreter does that, as it is possible to see in the fourth implementation of the **commandExec** function.

```
commandExec (DeclareBoolean name expr) env = case (readEnv env name "bool") of
    Just(v) -> ErrorEnv("The variable has already been declared", (DeclareBoolean name expr))
    Nothing -> case expr of
        Nothing -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                     var_type = "bool",
                                                     var_value = Nothing})
        Just(v) -> case (evalBExpr v env) of
            Result b -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                          var_type = "bool",
                                                          var_value = (Just(Boolean b))})
            Error a -> ErrorEnv (a, (DeclareBoolean name expr))

commandExec (DeclareInteger name expr) env = case (readEnv env name "int") of
    Just(v) -> ErrorEnv ("The variable has already been declared", (DeclareInteger name expr))
    Nothing -> case expr of
        Nothing -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                     var_type = "int",
                                                     var_value = Nothing})
        Just(v) -> case (evalAexpr v env) of
            Result i -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                          var_type = "int",
                                                          var_value = (Just(Integer i))})
            Error a -> ErrorEnv (a, (DeclareInteger name expr))

commandExec (DeclareArray name dim val) env = case (readEnv env name "array") of
    Just(v) -> ErrorEnv ("The array has already been declared", (DeclareArray name dim val))
    Nothing -> case val of
        Nothing -> ResultEnv (modifyEnv env Variable{var_name = name,
        var_type = "array",
        var_value = (Just(Array(createEmptyArray s)))})
        Just(ConstArr arr) -> case (length arr == s) of
            True -> ResultEnv (modifyEnv env Variable {var_name =
                                                       name, var_type = "array",
                                                       var_value = (Just(Array ar))})
                                                       where Result ar = (evalArrayOperation (ConstArr arr) env)
            False -> ErrorEnv ("The given dimension and the actual length are not equal", (DeclareArray name dim val))
        where Result s = (evalAexpr dim env)

commandExec (DeclareStack name) env = case (readEnv env name "stack") of
    Just(v) -> ErrorEnv ("The array has already been declared", (DeclareStack name))
    Nothing -> ResultEnv (modifyEnv env Variable {var_name = name, var_type = "stack",  var_value = (Just(Stack []))})
```

If you want to test the commandExec function, you can paste the following code in the terminal, after importing the Interpreter.hs module. This line tries to declare a new array named "b". The environment only contains one variable, which is another array named "a". You can change the command to test the other behaviors of commandExec.

```
commandExec(DeclareArray "b" (Const 2) (Just(ConstArr
[1,2])))[Variable{var_name="a",var_type="array",var_value=(Just(Array [1]))}]
```

And this will be the updated environment:

```
[       "a"     "array"         Just array [1]
,       "b"     "array"         Just array [1,2]
]
```

When the interpreter has to evaluate an assignation, it looks in the environment for the variable to modify. If the variable is not found an error is returned. For array assignations, it is checked also the validity of the index.

```
commandExec (AssignBoolean name value) env = case (readEnv env name "bool") of
    Nothing -> ErrorEnv ("The boolean does not exist", (AssignBoolean name value))
    Just(v) -> case (evalBExpr value env) of
        Result b -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                      var_type = "bool",
                                                      var_value = (Just(Boolean b))})
        Error a -> ErrorEnv (a, (AssignBoolean name value))

commandExec (AssignInteger name value) env = case (readEnv env name "int") of
    Nothing -> ErrorEnv ("The integer does not exist", (AssignInteger name value))
    Just(v) -> case (evalAexpr value env) of
        Error a -> ErrorEnv (a, (AssignInteger name value))
        Result i -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                      var_type = "int",
                                                      var_value = (Just(Integer i))})

commandExec (AssignArrayPosition name pos val) env = case (evalAexpr val env) of
    Result r -> case (evalAexpr pos env) of
        Result p -> case (evalArrayOperation (ArithmeticIdentifier name) env) of
            Result ar -> case (writeArray ar p r) of
                Just (v) -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                              var_type = "array",
                                                              var_value = (Just(Array v))})
                Nothing -> ErrorEnv ("Index out of range", (AssignArrayPosition name pos val))
        Error a -> ErrorEnv (a, (AssignArrayPosition name pos val))
    Error b -> ErrorEnv (b, (AssignArrayPosition name pos val))

commandExec (AssignWholeArray name ar) env = case (evalArrayOperation (ArithmeticIdentifier name) env) of
    Result v -> case (evalArrayOperation ar env) of
        Result arr -> case (length v == length arr) of
            True -> ResultEnv (modifyEnv env Variable{var_name = name,
                                                      var_type = "array",
                                                      var_value = (Just(Array arr))})
            False -> ErrorEnv ("Mismatching length", (AssignWholeArray name ar))
        Error e1 -> ErrorEnv (e1, (AssignWholeArray name ar))
    Error e2 -> ErrorEnv (e2, (AssignWholeArray name ar))
```

When popping a value from a stack, the interpreter checks that the stack is not empty.

```
commandExec (Push name val) env = case (readEnv env name "stack") of
    Nothing -> ErrorEnv ("The stack does not exist", (Push name val))
    Just(v) -> case (evalAexpr val env) of
        Error a -> ErrorEnv (a, (Push name val))
        Result i -> ResultEnv (modifyEnv env Variable {var_name = name,
                               var_type = "stack",
                               var_value = (Just(Stack (s ++ [i])))})
                    where Just (Stack s) = (var_value v)

commandExec (Pop name) env = case (readEnv env name "stack") of
    Nothing -> ErrorEnv ("The stack does not exist", (Pop name))
    Just(v) -> case (length s == 0) of
        True -> ErrorEnv ("Can't pop from an empty stack!", (Pop name))
        False -> ResultEnv (modifyEnv env Variable {
                    var_name = name,
                    var_type = "stack",
                    var_value = (Just(Stack (removeElem s ((length s)-1))))})
```

```
        where Just (Stack s) = (var_value v)
```

2.2.8: Evaluation of programs

A program is a list of commands. Therefore, the **programExec** function works on lists in a recursive way. If receives as input an empty list, then the environment given in input is returned as output. If the list is not empty, the first element is evaluated. If no errors were found, the interpretation proceeds on the other elements until an error is found or all the elements have been interpreted. Each element is interpreted working on the environment modified by the previous one.

```
programExec :: [Com] -> Env -> OutputEnv
programExec [] env = ResultEnv env
programExec ((DeclareBoolean name expr) : cs) env =
    case (commandExec (DeclareBoolean name expr) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((DeclareInteger name expr) : cs) env =
    case (commandExec (DeclareInteger name expr) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((DeclareArray name dim val) : cs) env =
    case (commandExec (DeclareArray name dim val) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((DeclareStack name) : cs) env =
    case (commandExec (DeclareStack name) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((AssignBoolean name value) : cs) env =
    case (commandExec (AssignBoolean name value) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((AssignInteger name value) : cs) env =
    case (commandExec (AssignInteger name value) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((AssignArrayPosition name pos val) : cs) env =
    case (commandExec (AssignArrayPosition name pos val) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((AssignWholeArray name ar) : cs) env =
    case (commandExec (AssignWholeArray name ar) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((Push name value) : cs) env =
    case (commandExec (Push name value) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
programExec ((Pop name) : cs) env =
    case (commandExec (Pop name) env) of
        ErrorEnv (e, c) -> ErrorEnv (e, c)
        ResultEnv new_env -> programExec cs new_env
```

programExec also defines the behavior of the four statements that the language allows to use. For the **if-else** statement, the condition is evaluated and, depending on the outcome, the corresponding block of commands is put as next thing to evaluate.

```
programExec ((Ifelse cond progA progB) : cs) env =
    case (evalBExpr cond env) of
        Error a -> ErrorEnv (a, (Ifelse cond progA progB))
        Result True -> programExec (progA ++ cs) env
        Result False ->
            case progB of
                Nothing -> programExec cs env
                Just(com) -> programExec (com ++ cs) env
```

For the **while** and **do while** statements, the commands in the loop are executed and the condition is checked. These two steps are repeated until the condition is True. The difference between the two statements is that in the do while the block is executed at least one time. So in this case the condition is checked at the end of the execution. In the while loop, the condition is checked at the beginning, which could lead to not executing at all the commands in the loop.

```
programExec ((Whiledo cond prog) : cs) env =
    case (evalBExpr cond env) of
        Error a -> ErrorEnv (a, (Whiledo cond prog))
        Result True -> programExec (prog ++ [Whiledo cond prog] ++ cs)
env
        Result False -> programExec cs env
programExec ((Dowhile prog cond) : cs) env =
    case (programExec prog env) of
        ErrorEnv (msg, com) -> ErrorEnv (msg, (Dowhile prog cond))
        ResultEnv env -> case (evalBExpr cond env) of
            Error a -> ErrorEnv (a, (Dowhile prog cond))
            Result True -> programExec ([Dowhile prog cond] ++ cs) env
            Result False -> programExec cs env
```

We conclude by talking about the interpretation of the **skip** command, which is trivial since it simply consists in evaluating the next command.

```
programExec (Skip : cs) env = programExec cs env
```

3: Test

We can now test our programming language using the Main.hs module. First of all, install Haskell following the tutorial at this link.
Now clone or download the repository, go in the **src** folder and open the cmd prompt.

Type the following:

```
ghci
:l Main.hs
main
```

```
C:\Users\UTENTE\Desktop\sIMP\src>ghci
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Prelude> :l Main.hs
[1 of 5] Compiling Grammar          ( Grammar.hs, interpreted )
[2 of 5] Compiling Utils            ( Utils.hs, interpreted )
[3 of 5] Compiling Parser           ( Parser.hs, interpreted )
[4 of 5] Compiling Interpreter      ( Interpreter.hs, interpreted )
[5 of 5] Compiling Main             ( Main.hs, interpreted )
Ok, five modules loaded.
*Main> main
Welcome to...

      _    __    __   ____
     | |   | \\  // | | D__)
 ___ | |   | |\\// | | | |
(__  | |   | |    | | | |
___) |_|   |_|    |_| |_|

Insert the path to the file you want to use!
```

You are now ready to use sIMP. Type the path to the file containing your program and press **Enter** to run the code. In the test folder there are some files containing code to test the various functionalities of the program. For example, here we test the **testDoWhile** file that contains two do-while statements. At the current state, the file purposely contains an error, and this is the result:

```
Insert the path to the file you want to use!
../test/testDoWhile.txt

Input Program

int a = 0;
do{
    a = a+1;
} while (a<0);

array[2] c;
int i = 0;
do{
c[i] = i;
i = i+1;
}
while(i <= 2);



*** ERROR !!! "Index out of range" !!!***
 ON COMMAND: DoWhile [AssignArrayPosition "c" (ArithmeticIdentifier "i") (ArithmeticIdentifier "i"),AssignInteger "i"
 (Add (ArithmeticIdentifier "i") (Const 1))] (Lte (ArithmeticIdentifier "i") (Const 2))
```

An error-free code will be successfully evaluated, and the result will be the state of the environment:

```
Insert the path to the file you want to use!
../test/testGeneral.txt

Input Program

bool a = True;
int b = 2;
int c;
c = 21;
c = 4;
bool cb;
array[c] ar = [1,2,3,9];
if(c < b){
    cLessb = True;
}
int cb = c;


EXECUTION SUCCEDED!!

State of the memory:

        NAME     TYPE           VALUE
[       "a"     "bool"          Just bool True
,       "b"     "int"           Just int 2
,       "c"     "int"           Just int 4
,       "cb"    "bool"          Nothing
,       "ar"    "array"         Just array [1,2,3,9]
,       "cb"    "int"           Just int 4
]
```