1 Introduction

The goal of this project is to define an interpreter for an hypothetical programming language, with a syntax similar to the **C** language, thus using the concept of variables with types and array. According to the most common definition, we can affirm that an interpreter is a computer program that execute a script written in a specific programming language, without compiling it, thus, without be specific for the architecture of a machine.[1] In a general way, an interpreter does the following steps:

- Parses the code to check the correctness of it.
- Transforms the code in an internal representation of objects.
- Executes the code to produce an output, and modify the internal state of the machine.

Before introduce our language, we must specify what a parser is, a simple description of a parser is: "a **parser** is a program that takes a string as input and produces some form of a tree, that defines the syntactic structure of the string"[2], as indicated in Figure 1 Therefore, it is obvious that every programming language needs a parser to be executed. Let's take as example Haskell, the programming language used to implement this project, it uses a parser to interpret the input program and produce an output. Haskell is a functional programming language with a **lazy** evaluation strategy. The fact that Haskell is a functional programming language, helps us in define a parser, due to the use of the concept of **monad**. In the next chapter we will explain better what a monad is and how does it works in the parser.

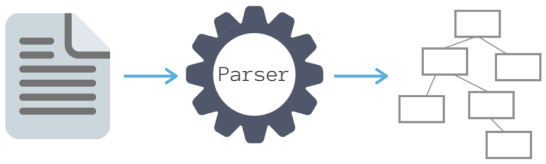

Figure 1: Graphical representation of a parser.

As said before the language that we are going to implements is a C-like language, thus, it must supports variables' definition with type:

```
int variable1 := 1;
bool variable1 := False;
```

assignment of arrays to variables, and the classic control structures such as while, and if-then-else:

```
int[] array = [1,2,3,4,5];
while(True){
    array[1]:= 0;
}
```

Indeed, we would like to use also the comments, thus a string with a initial character, that will be ignored during the execution.

[1] Wikipedia, Interpreter (Computing)
[2] Graham Hutton, Programming in Haskell, 2007, Cambdridge University Press

The project is organized in modules, as shown by the Figure 2 :

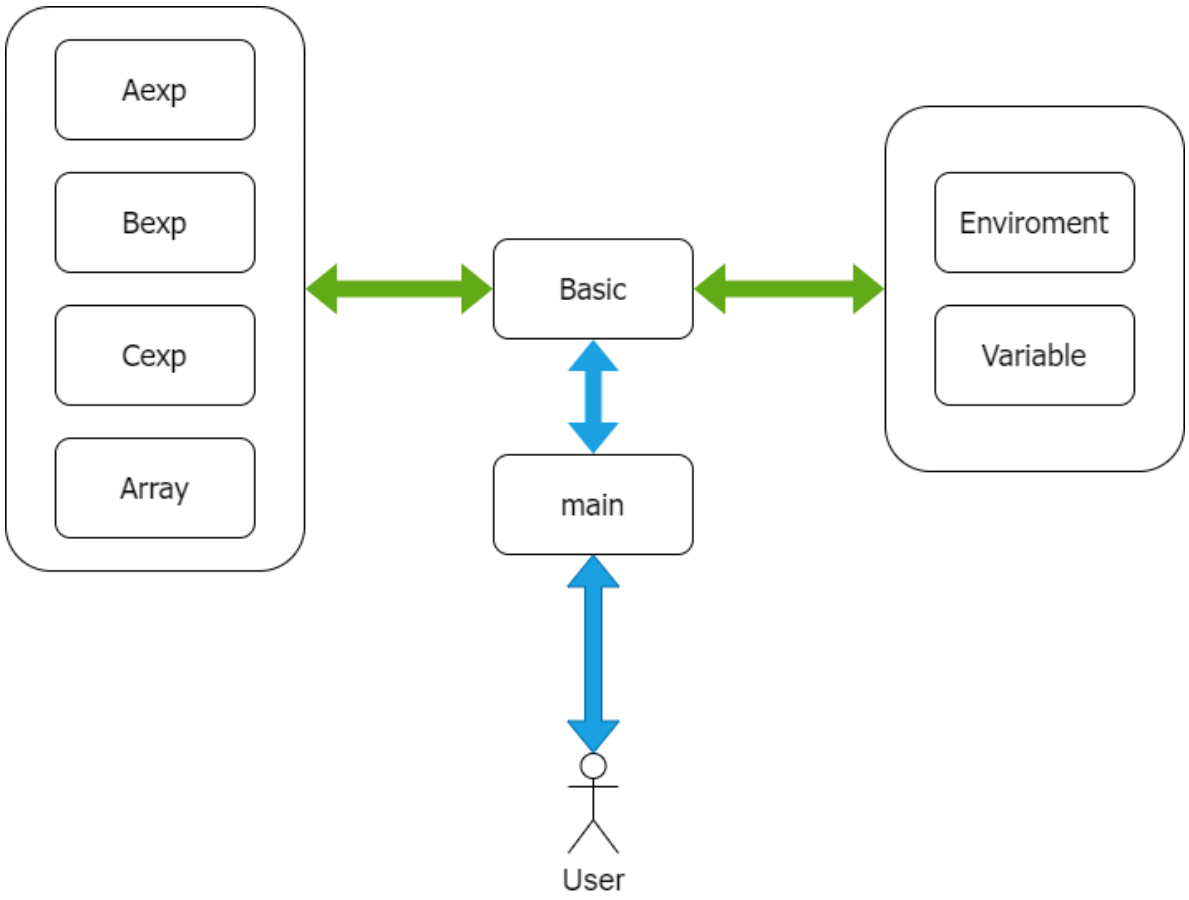

Figure 2: Internal architecture of the project.

Let's have a quick review of the project's architecture:

- The **main** module is the core of the interpreter, it define the basic IO primitives thanks to the user can communicate with the program, and the function to run the input code;
- The **Basic** module defines the concept of parser and the primitives used by the more complex expressions.
- The modules **Environment** and **Variable** define the concept of variable and environment, used by the parser to interpret the input code.
- The modules **Aexp**, **Bexp**, **Cexp**, **Array** define the arithmetic, boolean, command and array expressions that define the language, and how these must be parsed by the interpreter.

2 Environment, Variables and Parser

Our interpreter is composed of two main elements, the **parser** which defines how the input must be interpreter, producing an output, and the **environment** in which are stored all the variables and the references of the execution, the Figure 3 explains graphically how these components interact with each other.

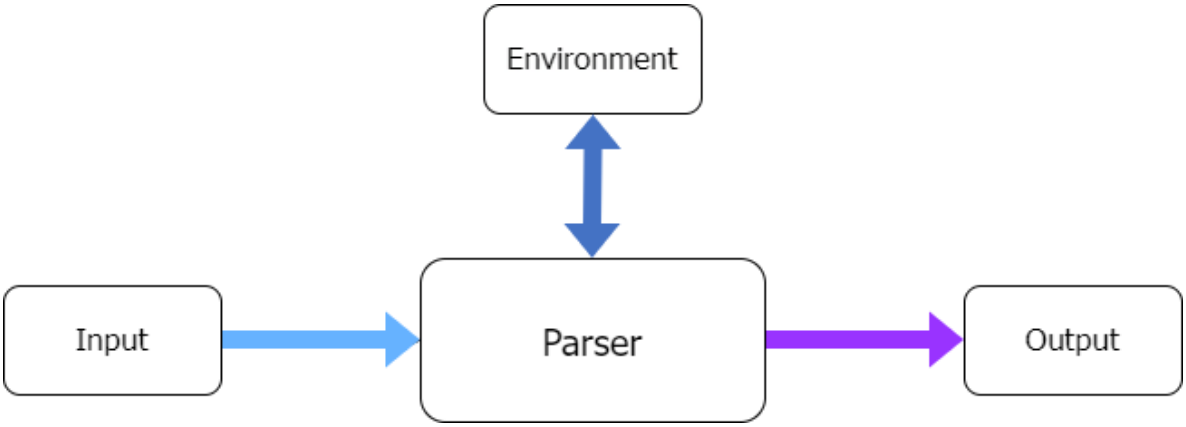

Figure 3: Main components of the project.

The input of the parser is just the input string that we type directly from the command line, or we load through a txt file, using the command ":load filename". In both cases the environment that is used by the parser is an empty array, that will be filled with the variables defined in the executed code, therefore, there is a bidirectional communication from the parser to the environment, differently to the other two components, in which the communication is unidirectional.

2.1 Variable

The smallest component of a program in our C-like language, is the **variable**. From an abstract point of view, a variable is just a triple of three elements:

- An unique **identifier**, which is a string, that must start with a lower case letter.
- A **type** that indicates the variable's type. In this language we suppose that there are just two types of variables, the **integer** variables and the **boolean** variables.
- A field that indicates the actual **value** of the variable.

Let's see how the concept of variable is defined in Haskell:

```
type Identifier = String
data DataType = Boolean | Integer | ArrayInt | ArrayBool
data Value a = Val a | Array [a]
data Variable = Variable {
        name :: Identifier,
        typeV :: DataType,
        value :: Value Int
}
```

Thus, using the **record notation** we represent a variable as composed of an **Identifier** which is a string, a **type** which is an own-defined data type and a **value**. Indeed, the actual value of the variable is wrapped in a context, to separate the variables of single elements, from the variables of array. As we said before, the variables can be of two types, boolean or integer. However, in this definitions the value of a variables is just an integer, but using the fact that **true** is represented by a 1, and **false** is represented by 0, we can emulate the behaviour of a boolean value using an integer.

Both the **Variable** and the **DataType** custom types have been made instances of classes: **Functor**, **Eq** and **Show**. Indeed, the following operators have been defined:

```
writeVariable :: Variable -> (Int, Int) -> Variable
replace :: [a] -> (Int, a) -> [a]
```

where, **writeVariable** takes as input a variable and two integers, that represents the index and the value to replace in the array of the variable, by using the function **replace**.

2.2 Environment and Parser

As we said before, an environment is a list of variables, or more specific, is the list of the variables defined in the input code. Thus, is quite intuitive that an environment is just a list of variables, as indicated in the Haskell code:

```
type Environment = [Variable]
```

There are some basic functions that manage the environment, these are:

```
modifyEnvironment :: Environment -> Variable -> Environment
modifyArray :: Environment -> Variable -> Int -> Int -> Environment
searchVariable :: Environment -> Identifier -> DataType -> [Value Int]

modifyEnvironment [] var = [var]
modifyEnvironment (x:xs) var = if (x == var) then
        [var] ++ xs
        else [x] ++ modifyEnvironment xs var

modifyArray [] var i v = [writeVariable var (i, v)]
modifyArray (x:xs) var i v = if (x == var) then
        [writeVariable x (i, v)] ++ xs
        else [x] ++ modifyArray xs var i v

searchVariable [] _ _ = []
searchVariable (x:xs) n t =  if ((n == (name x)) && (t == (typeV x)))
        then [value x]
        else searchVariable xs n t
```

Thus, the function **modifyEnviroment** adds a variable to the enviroment if it is not present inside it; the function **modifyArray** requires the enviroment, the variable to modify, the index of the array and the value to overwrite, and modify the variable if is present in the environment; finally, the function **searchVariable** returns an array of variables, if the enviroment is empty, then an empty array is returned, indicating that no variable has been found, on the other hand, if the function found the variable, then the array containing the variable's value is returned.

However, before explaining the others operators that defines the behaviour of the environment, we must introduce the parser. From the Figure 3 is quite intuitive that the parser must be a function with two parameters (the environment and the input string), that returns the evaluation of the input string. However, we want to improve this concept, by making this function returning not just one element, but three:

- The **evaluation** of the input string.
- The output **enviroment**.
- The **unparsed string**.

Indeed, how can indicate that the input string is not a valid string? Well, we can to this by using the **Maybe** type, defined in the basic Haskell's library, or by returning an array. Thus, if the array is empty, it indicates that the input string is not a valid string, and the array of just one element is the result. By combining all these concepts, we define the parser as:

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

Notice that we must use the "newtype" keyword to make the parser an instance of some classes that we will explain later. Indeed, we use the "newtype" kyeword instead "data", for an efficiency benefit, since the constructor does not require any cost when the program is evaluated. With respect to the "type" keyword, the "newtype" introduces a new type, rather than making the parser a synonim of an existing one, ensuring that the two types definitions can't be mixed up.

Let's define now the two basic functions used by the parser.

```
parse :: Parser a -> Environment -> String -> [(Environment, a, String)]
item :: Parser Char

parse (P p) env inp = p env inp

item = P (\env inp -> case inp of
        [] -> []
        (x:xs) -> [(env, x, xs)])
```

The function **parse** applies the parser function, given the initial environment and the input string, thus is simply just a remover of the dummy constructor P, defined by using the **newtype** keyword. The second function, **item** returns the first character of the input string, as indicated by the example:

```
parse item env "abcd"  -> [("a", env, "bcd")]
```

However, we would like to define our parser by combining different parsers and use a sort of "choice" operator, like happens in the definition of our BNF grammar for the language. Therefore, to define a parser as a sequence of operations, we must use the **do-notation**, and then make the parser type an instance of **applicative, functors** and **monads**.

2.3 Functor

Often we need to apply a function to an array, or in a more general way, to a data structure. A previous idea of mapping a function to an array, has been defined by the function **map**, however, the class **Functor** allows us to generalize the concept of apply a function to our parser. The Functor class is defined as:

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

Thus, the function **fmap** requires a function and an element wrapped in a context, then unwraps this value from a context, apply the function and returns the result of the function, wrapped in the same input context. From this definition, we can make our parser type an instance of Functor class as it follows:

```
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b

     fmap g p = P (\env inp -> case parse p env inp of
              [] -> []
              [(env,v,out)] -> [(env,g v, out)])
```

Therefore, the function fmap takes a function and a parser as input, and returns another parser. In a more detailed way, given a function **g** and a parser **p**, if the application of the parser p, to the input environment **env** and the string **inp** produces an empty array, it means that the parser has failed, and then we just return a failed parser. Otherwise, if the parser success with a triple (Enviroment, Output, Unparsed-string), we can apply the function g to the result value of the parser, returning another parser.

2.4 Applicative

If the Functor class abstracts the idea of mapping a function over each element of a structure, we would like to generalize the idea of apply a function with any number of arguments, rather than be restricted to a function with only one argument. This idea is implemented in the **Applicative** class, defined by the following built-in class declaration:

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

Therefore, the Applicative class inherits the behaviour of the Functor class, and implements two operators:

- **pure** wraps an element in a context f.
- The **applicative** function requires a function in the same context of the input, to produce an output with the same context.

The corresponding implementation of the Parser is:

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

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

From the following definition, **pure** just wrap a value in the context Parser. The applicative function, using the function parser, unwraps the function **pg** from its context, and the if the parse function success, applies the unwrapped function **g** to the input parser **px** by using the function **fmap**.

2.5 Monad

If the Functor class allows us to apply a function to an element wrapped in a context, the Applicative class goes further applying function that are wrapped in a context, the **Monads** allows us to apply function to the elements wrapped in a context, returning the resulting value wrapped in the same input context. This concept is implements in Haskell, by the class Monad, defined as follows:

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

That is, a type constructor m is a member of the class Monad if it is equipped with *return* and >>= operators, for the specified types. [3] The parser *return* succeeds without consuming any of the argument string, and returns the single value *a*. The >>= operator is a **sequencing** operator for parsers. The *return* and >>= functions for parsers satisfy some simple laws:

```
return a >>= f = f a
p >>= return = p
p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g
```

Thus the laws affirm that the bind operator is associative, and that return is a left and right unit for the bind operator. Therefore, a way to understand the Monad is to think about it as a **bridge** between two worlds. The first world contains the elements that are wrapped in a context, and the second is the world of functions that do not require as input an element wrapped in a context. A typical parser built using the >>= operator has the following structure:

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

[3] Monadic parsing in Haskell, Graham Hutton "" Erik Meijer, 1998, Cambridge University Press

Such a parser has a natural way of reading, that is: apply parser p1 and call its result value a1; then appy the parser p2, and returns the result a2,, finally combine all these results using the operator f. Haskell provides a more intuitive way to combine parsers, that is be using the **do-notation**:

```
do a1 <- p1
    a2 <- p2
    ...
    an <- pn
    f a1 a2 ... an
```

Or by using a single line:

```
do {
    a1 <- p1; a2 <- p2; ... an <- pn; f a1 a2 ... an;
}
```

Let's see with an example how the bind operator of Monad class, can be used. Notice that the type Maybe, is an instance of the Monad class, is this way:

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
    -- return :: a -> Maybe a
    return x = Just x

    -- (>>=) :: (a -> Maybe b) -> Maybe a -> Maybe b
    (>>=) g x = case x of
                    Nothing -> Nothing
                    Just a -> g a
```

Let's define now our function that returns the half of an integer, and let's calculate twice the half of 20:

```
half :: a -> Maybe a
Just 20 >>= half >>= half -> Just 5
```

Doing this, given a element wrapped in a context (in this case the context is Maybe), and a function that do not requires as input an element wrapped in a context, but returns another element wrapped in the same context. The bind function, unwrap the input value, apply the function, and returns a value wrapped in a context, this result value in then passes to the same function, that do the same operations, and so on and so forth... However, Haskell provides a special notation to simplify the sequence application of the bind operator. This special notation is called the **do-notation**, and is defined as:

```
half' :: a -> Maybe a
half' x = do {
    y <- half x;
    z <- half y;
    return z;
}
```

Thus, according to the do-notation, and given a value of type**a**:

- line 1 wraps x in a context, apply the function half, and returns the unwrapped value in the context, indicated by the left side of the arrow. In this way, the arrow function is a sort of unwrapping operator.
- line 2 do the same thing of the previous line.
- return z; wraps the value in the context, which is the returned value of the block.

Back to the Parser type, we make it an instance of the Monad class, in this way:

```haskell
instance Monad Parser where
    -- return :: a -> Parser a
    return = pure

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

The function returns do the same operation of the function pure, defined in the Applicative class. On the other hand, the bind operator, works as follows:

1. Given a parser **p** and a function **f**.
2. Apply the function parse using the parser **p**. This can return an empty array, indicating a failuer of the parser, or an array with an unique element, containing the result of the application of the parser to the input.
3. If the application of the parser fails, also the result parser must fail.
4. Othwerise, if the parse function success, a new Parser is returned, whose input is the application of the function **f** to the result of the parser **p**, of type **a**.

Now we can combine parsers to define other parses, indeed, we can make use of the do-notation, making the code more legible.

2.6 Alternative

If the do notation combines parsers in sequence, another natural way to combine parsers is to apply one parser to the input string, and if it fails to then apply another to the same input instead. In other words, we want to use a sort of choice operator between parsers. The concept of alternatives between parsers is captured by the following class declaration, in the **Control.Applicative** library:

```haskell
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

Where, the **empty** function always fails, and the alternative operator requires two values wrapped in a context, and returns one of these based on which one fails. This concept is easily implemented in our Parser type as:

```haskell
instance Alternative Parser where
        -- empty :: Parser a
        empty = P (\env inp -> [])

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

As we can see, the empty function returns a Parser that always fails, and the alterative operator requires two parsers **p** and **q**. If the first parser fails, then we returns the result of the parse function with the second parser, otherwise returns the result of the first parser. An example of the application of this class is:

```haskell
parse empty env "abcd"  -> []
parse (empty <|> return 'd') env "abcd" -> [(env, 'd', "abcd")]
```

2.7 Parser's primitives

Since now we defined three parsers, the parser **item** consumes the first character of the input string if this is not empty, the parser **return** that always succeeds with the input value, and the parser **empty** that always fails. Using these parsers and the function:

```
sat :: (Char -> Bool) -> Parser Char
```

that consumes the first char of the input string, if it satisfies the properties passed as input, we can define the parsers for the primitives of our language. Let's start with the primitives for manipulate the first character of the input string:

```
digit :: Parser Char
lower :: Parser Char
upper :: Parser Char
letter :: Parser Char
alphanum :: Parser Char
char :: Char -> Parser Char
```

Let's take as example the parser digit:

```
digit = sat isDigit
parse digit env "abcd" -> []
parse digit env "12ab" -> [(env, "1", "2ab")]
```

where the function **isDigit** is defined in the standard library Basic.Char. The parser digit applies the function isDigit to the input string "12ab", if the first character of the input string satisfies the property, then a new parser is returned, with the same environment and as result the first character, and the rest of the string as the unused string. On the other hand, if the first character does not satisfy the property, then an empty parser is returned. Now, to define the next parser, we need a way to apply the input parser as many time as possible, to consume the input string. In this case, the Alternative class helps us:

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
    many :: f a -> f [a]
    some :: f a -> f [a]
```

The operators **many** and **some** apply a parser as many times as possible, until it fails, with the result values from each of the successful application of the parser being returned in a list. The difference between these operators is that **many** permits zero or more applications of the parser, and **some** requires at least one successful application.
Using the previous basic parsers and the operators many and some, we can define more complex parsers.

```
string :: String -> Parser String
ident :: Parser String
varType :: Parser String
nat :: Parser Int
space :: Parser ()
int :: Parser Int
token :: Parser a -> Parser a
```

Therefore, the parser **string** returns the parser containing as result value the input string if it is a sequence of characters; the parser **ident** returns the input string if it represents an identifier (a sequence of characters starting with a lower case letter); the parser **varType** returns the parser contaning the input string "int" or "bool"; the parser nat returns the input string, if the first character is a natural number; the parser space returns the empty parser, with the input string as the unused string, therefore ignoring the spaces in the string:

```
parse env space " abc" -> [(env, (), "abc")]
```

The parser **int** returns the first element if this is an integer; On the other hand, most of the time the input string contains spaces between the single elements. Thus, to handle such spacing, we define the parser **token** that ignores any spaces between and after a given parser, and applies it:

```
token p = do {
            space;
            v <- p;
            space;
            return v;
    }
```

Using the token parser, we can define parsers that ignores spaces between identifier, numbers and other elements of the input string:

```
identifier :: Parser String
natural :: Parser Int
integer :: Parser Int
symbol :: String -> Parser String
typeVariable :: Parser String
```

Now, our language supports both comments and array's definitions, for this reason, the following parsers have been defined:

```
separator :: Parser a -> Parser [a]
comment :: Parser ()
brackets :: Parser a -> Parser a
rbrackets   :: Parser a -> Parser a

separator p = many (do symbol ","; p)

comment = do {
        string "--";
        many (sat (/= '\n'));
        return ();
}

brackets p = do {
            symbol "[";
            n <- p;
            symbol "]";
            return n;
   }
```

The **separator** parser takes as input the string that represents an array, such as ", 1, 2, 3", and returns the characters that satisfies the parser integer, separated by a comma. On the other hand, the comments are represented as a single line comment starting with the string "–" are just ignored from the interpreter, and then the empty parser is returned. In a similar way to token, the parser **brackets** returns the input string that is enclosed between square parenthesis and satisfies an input parser p. The parser **rbrackets** is similar to **brackets** but for round parenthesis.

Now that we know what a parser and an environment are, we can explain the other functions used to mainupale the enviroment inside a parser:

```
updateEnvironment :: Variable -> Parser String
updateArray :: Variable -> Int -> Int -> Parser String
readArrayVariable :: Identifier -> DataType -> Int -> Parser Int
readSingleVariable :: Identifier -> DataType -> Parser Int
writeArray  :: Identifier -> Int -> Int -> Parser [Int]
```

The function **updateEnviroment** returns a parser whose enviroment has been modified, adding the input variable and the same input string; the function **updateArray** returns a parser with a

modified environment, and the same input string; the function **readArrayVariable** returns the parser containing the value of the corresponding array; the function **readSingleVariable** returns a parser containing the value of the corresponding variable, stored inside the environment; finally, the function **writeArray** returns a parser whose value is the modified array.

3 Arithmetic Expressions

The basic expressions of our programming language are the arithmetic expressions, that supports the classic operators between integer numbers. First of all, we must define the grammar of our arithmetic expressions:

⟨*expression*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ + ⟨*expression*⟩ | ⟨*term*⟩ - ⟨*expression*⟩

⟨*term*⟩ ::= ⟨*atom*⟩ | ⟨*atom*⟩ * ⟨*term*⟩ | ⟨*atom*⟩ / ⟨*term*⟩

⟨*atom*⟩ ::= ⟨*factor*⟩ | ⟨*factor*⟩ ** ⟨*atom*⟩

⟨*factor*⟩ ::= (⟨*expression*⟩) | ⟨*identifier*⟩ | ⟨*integer*⟩ | ⟨*identifier*⟩[⟨*expression*⟩]

⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*natural*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*natural*⟩

⟨*integer*⟩ ::= ⟨*natural*⟩ | -⟨*natural*⟩

⟨*uppercase*⟩ ::= A | B | ... | Z

⟨*lowercase*⟩ ::= a | b | ... | z

⟨*string*⟩ ::= ⟨*uppercase*⟩ | ⟨*lowercase*⟩

⟨*identifier*⟩ ::= ⟨*lowercase*⟩ | ⟨*lowercase*⟩ ⟨*string*⟩

Notice that an expression can also be an evaluation of an array, of course this array must be an array of integers, otherwise the evaluation will fail. We can now translate these rules into a parser, using the do-notation and the alternative. The corresponding parsers have the following signature:

```
aexpr :: Parser Int
aterm :: Parser Int
aatom :: Parser Int
afactor :: Parser Int
```

Thus, each parser will returns the corresponding evaluation of the arithmetic expression, let's examine the implementation of the last parser:

```
afactor = do {
    e <- rbrackets aexpr;
    return e;
}<|> do {
    i <- identifier;
    readSingleVariable i Integer;
} <|> do {
    i <- identifier;
    n <- brackets aexpr;
    v <- readArrayVariable i ArrayInt n;
    return v;
} <|> integer
```

Thus, an arithmetic factor is:

- An arithmetic expression contained between the brackets.
- A variable inside the environment, whose name is the identifier.
- An evaluation of an array of integers.
- An integer.

Finally, before apply the parser and compute the relative result, we need to check the correctness of syntax of the arithmetic expression. We can do that by implementing parsers that return the input string, if the syntax is correct, otherwise fail.

```
parseAexp :: Parser String
parseAterm :: Parser String
parseAatom :: Parser String
parseAfactor :: Parser String

parseAfactor = do{
                e <- rbrackets parseAexp;
                return ("(" ++ e ++ ")");
        } <|> do {
                symbol "-";
                f <- parseAfactor;
                return ("-" ++ f);
        } <|> do {
                k <- integer;
                return (show k);
        } <|> do {
                i <- identifier;
                do {
                        n <- brackets parseAexp;
                        return (i ++ "[" ++ n ++ "]");
                } <|> return i;
        }
```

More details about how do these parser workd, will be explained in the last chapter, when we will talk about the execution of the interpreter.

4 Boolean Expressions

In a similar way to the Arithmetic Expressions, we can define the Boolean Expressions, starting from their grammar.

⟨*expression*⟩ ::= ⟨*term*⟩ `||` ⟨*expression*⟩ | ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩ `&&` ⟨*term*⟩ | ⟨*factor*⟩

⟨*factor*⟩ ::= True | False | !⟨*factor*⟩ | (⟨*expression*⟩) | ⟨*identifier*⟩ | ⟨*comparison*⟩ | ⟨*idetifier*⟩ [⟨*Aexpression*⟩]

⟨*comparision*⟩ ::= ⟨*Aexpression*⟩ `<` ⟨*Aexpression*⟩
| ⟨*Aexpression*⟩ `>` ⟨*Aexpression*⟩
| ⟨*Aexpression*⟩ ≥ ⟨*Aexpression*⟩
| ⟨*Aexpression*⟩ ≤ ⟨*Aexpression*⟩
| ⟨*Aexpression*⟩ == ⟨*Aexpression*⟩
| ⟨*Aexpression*⟩ ≠ ⟨*Aexpression*⟩

Then we define the parsers for the evaluation of a boolean expression:

```
bexp :: Parser Bool
bterm :: Parser Bool
bfactor :: Parser Bool
bcomparison :: Parser Bool
```

and finally the parsers to check the correctness of the boolean expressions' syntax:

```
parseBexp :: Parser String
parseBterm :: Parser String
parseBfactor :: Parser String
parseComparison :: Parser String
```

5 Array Expressions

In this section we are going to define the syntax and the parsers for the expressions that involve arrays. Notice that once we declare an array, it is initialized inside the environment as a dummy array with some starting values, and then we can populate it using for example a while loop.

⟨*Array*⟩ ::= ⟨*ArrayInitialization*⟩ | ⟨*ArrayAssignment*⟩

⟨*ArrayInitialization*⟩ ::= ⟨*type*⟩[] ⟨*identifier*⟩; | ⟨*type*⟩ [⟨*ArithmeticExpression*⟩] ⟨*identifier*⟩;

⟨*ArrayAssignment*⟩ ::= ⟨*identifier*⟩ [⟨*ArithmeticExpression*⟩] := ⟨*ArithmeticExpression*⟩;
| ⟨*identifier*⟩ [⟨*ArithmeticExpression*⟩] := ⟨*BooleanExpression*⟩;
| ⟨*type*⟩[] ⟨*identifier*⟩ := ⟨*ArrayExpression*⟩;
| ⟨*type*⟩[] ⟨*identifier*⟩ := ⟨*ArrayConcat*⟩;
| ⟨*identifier*⟩ := ⟨*ArrayConcat*⟩;
| ⟨*identifier*⟩ := ⟨*ArrayExpression*⟩;

⟨*ArrayExpression*⟩ ::= [] | [⟨*element*⟩]

⟨*element*⟩ ::= ⟨*intElement*⟩ | ⟨*boolElement*⟩

⟨*intElement*⟩ ::= ⟨*ArithmeticExpression*⟩ | ⟨*ArithmeticExpression*⟩, ⟨*intElement*⟩

⟨*boolElement*⟩ ::= ⟨*BooleanExpression*⟩ | ⟨*BooleanExpression*⟩, ⟨*boolElement*⟩

⟨*ArrayConcat*⟩ ::= ⟨*ArrayExpression*⟩ ++ ⟨*ArrayExpression*⟩

With the following parsers used to implement the grammar:

```
array :: Parser String
arrayInit :: Parser String
arrayAssignment :: Parser String
arrayExpr :: String -> String -> Parser String
arrayConcat :: String -> String -> Parser String

array = do {
        arrayInit <|>
        arrayAssignment
}

arrayInit = do {
    -- <type>[] <identifier>;
        t <- typeVariable;
        emptybrackets;
        i <- identifier;
        case t of
                "int" -> do {
                        symbol ";";
                        updateEnvironment Variable {name = i,
                            typeV = ArrayInt,
                            value = (Array [])
                        }
                }
                "bool" -> do {
                        symbol ";";
                        updateEnvironment Variable {name = i,
                            typeV = ArrayBool,
                            value = (Array [])}
                }
```

```
} <|> do {
    -- <type>[<Aexpr>] <identifier>;
    t <- typeVariable;
        n <- brackets aexpr;
        i <- identifier;
        symbol ";";
        case t of
                "int" -> do {
                            updateEnvironment Variable {name = i,
                                typeV = ArrayInt,
                                value = (Array (replicate n 0))
                            };
                }
                "bool" -> do {
                            updateEnvironment Variable {name = i,
                                typeV = ArrayBool,
                                value = (Array (replicate n 0))
                            }
                }
}
```

As we can see, the basic parser is the **array** parser, which states that an array is an initialization or an assignment operation. An initialization of an array requires the type of the array, and the square brackets, if the brackets are empty, an empty array is created. On the other hand, if we specify the size of the array between the square brackets, the interpreter initialize an array of default values. Notice that both in the case of integers array or booleans array, both initial value is zero, because in the interpreter's logic, the boolean values are emulated using the integers, where 0 indicates the False value and 1 the True Value. Indeed, we can also initialize an array specifying all the elements that the array have, or by concatenating two arrays in sequence.

As in the case of the boolean and arithmetic expression, we must check the syntax of the array, before apply the parsers. Therefore, we implements some parsers that returns the input string, if the syntax of this is correct.

```
parseArray :: Parser String
parseArrayInit :: Parser String
parseArrayAssignment :: Parser String
parseArrayExpr :: Parser String
parseArrayConcat :: Parser String
```

6 Command expressions

Now that we defined the basic elements of the interpreter, we are going to describe the core of it. In fact, the commands are the basic elements and each part of the input string is a command of the program itself. Once more, we start by describing the grammar of the commands:

⟨*program*⟩ ::= ⟨*command*⟩ ⟨*program*⟩ | ⟨*command*⟩ | ⟨*comment*⟩

⟨*command*⟩ ::= ⟨*initialization*⟩ | ⟨*assignment*⟩ | ⟨*if-then-else*⟩ | ⟨*while*⟩ | ⟨*for*⟩ | ⟨*skip*⟩ | ⟨*array*⟩

⟨*skip*⟩ ::= skip;

⟨*initialization*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩;

⟨*assignment*⟩ ::= int ⟨*identifier*⟩ := ⟨*ArithmeticExpression*⟩;
| bool ⟨*identifier*⟩ := ⟨*BooleanExpression*⟩
| ⟨*identifier*⟩ := ⟨*ArithmetixExpression*⟩;
| ⟨*identifier*⟩ := ⟨*BooleanExpression*⟩;

⟨*if-then-else*⟩ := if ⟨*BooleanExpression*⟩ { ⟨*program*⟩ }
| if ⟨*BooleanExpression*⟩ { ⟨*program*⟩ } else { ⟨*program*⟩ }

⟨*while*⟩ ::= while ⟨*BooleanExpression*⟩ { ⟨*program*⟩ }

⟨*for*⟩ ::= for(⟨*assignment*⟩ ⟨*BooleanExpression*⟩; ⟨*assigment*⟩){⟨*program*⟩}

⟨*type*⟩ ::= int | bool

⟨*comment*⟩ ::= – ⟨*string*⟩

Notice that once we have an initialization of a variable, it is possible to assign it only a value corresponding to the type of the variable, even if all the variables of the language are integers. On the other hand, it is possible to assign to an initialized variable, an expression that can be not only the type of the variable. However, in this case the interpreter understands that the initialized variable and the modified variable are different, and it will store another variable with the same name but different type. Let's have a look on the parsers that implement the grammar of the command expression:

```
command :: Parser String
program :: Parser String
skip :: Parser String
initialization :: Parser String
assignment :: Parser String
ifThenElse :: Parser String
while :: Parser String
repeateWhile :: String -> Parser String
```

Since now the parsers' behaviours are similar to the previous parser, however, the interesting part concerns the parser of the while loop.

```
while = do {
    w <- parseWhile;
    repeateWhile w;
              symbol "while";
              p <- bexp;
              symbol "{";

              if (p) then do {
                program;
                symbol "}";
                repeateWhile w;
                while;
              } else do {
                parseProgram;
                symbol "}";
                return "";
               }
    }

repeateWhile c = P(\env inp -> [(env, "", c ++ inp)])
```

From an abstract point of view, to implement a while loop, we need to repeat the code inside the parser, since the boolean condition is satisfied. Therefore, the first thing to do is take the actual code, by using the parser **parsewhile** (that checks the syntax of the code), and then use the parser **repeatWhile** that returns the same parser, appending to the input the while loop to repeat again. Doing this, if the boolean expression of the loop is true, then we parse the program inside the parenthesis and we repeat the while, appending once more the code to the unparsed string. On the other hand, if the boolean condition is false, then we return the parser with an empty string, indicating that the input is finished.

Indeed, notice that we do not define any parser for the for-loop, because it is possible to emulate a for-loop, using a while-loop.

Then, the parsers to check the correctness of the syntax of the code are:

```
parseProgram :: Parser String
parseSkip :: Parser String
parseCommand :: Parser String
parseInitialization :: Parser String
parseIfThenElse :: Parser String
parseWhile :: Parser String
parseAssignment :: Parser String
parseComment :: Parser String
```

As we said in the beginning, the language supports also the comments, in fact as we saw in the grammar, a comment is no more than a single-line string, starting with the symbols "–". Since the comments enrich the code, the interpreter will ignore them, eliminating from the input string each comment, and then parsing the rest. Following are defined the other parsers:

```
command =
        initialization <|>
        assignment <|>
         skip <|>
         ifThenElse <|>
         while <|>
        array

program = do {
        command;
        program;
} <|> command
```

```
initialization = do {
        t <- typeVariable;
        i <- identifier;
        symbol ";";
        case t of
                "int" -> updateEnvironment Variable {name = i,
                    typeV = Integer,
                    value = (Val 0)
                }
                "bool" -> updateEnvironment Variable {name = i,
                    typeV = Boolean,
                    value = (Val 0)
                }
}

assignment = do {
        t <- typeVariable;
        i <- identifier;
        symbol ":=";
        case t of
                "int" -> do {
                        v <- aexpr;
                           symbol ";";
                           updateEnvironment Variable {name = i,
                               typeV = Integer,
                               value = (Val v)
                           };
                }
                "bool" -> do {
                        v <- bexp;
                           symbol ";";
                           if (v == False) then
                               updateEnvironment Variable {name = i,
                                   typeV = Boolean,
                                   value = (Val 0)
                               } else updateEnvironment Variable {name = i,
                                   typeV = Boolean,
                                  value = (Val 1)
                               };
                }
}<|>do {
        i <- identifier;
        symbol ":=";
        do {
                v <- aexpr;
                   symbol ";";
                   updateEnvironment Variable {name = i,
                       typeV = Integer,
                       value = (Val v)
                   };
        } <|> do {
                v <- bexp;
                   symbol ";";
                   if (v == False) then
                       updateEnvironment Variable {name = i,
```

```
                            typeV = Boolean,
                            value = (Val 0)
                        } else updateEnvironment Variable {name = i,
                            typeV = Boolean,
                            value = (Val 1)
                        };
        }
}

skip = do {
        symbol "skip";
        symbol ";";
}

ifThenElse = do {
        symbol "if";
        b <- bexp;
          symbol "{";

          if (b) then
           do {
            program;
            symbol "}";
             do {
                      symbol "else";
                      symbol "{";
                      parseProgram;
                      symbol "}";
                      return "";
             } <|>
     return "";
    }
          else do {
            parseProgram;
            symbol "}";
            do {
                     symbol "else";
                     symbol "{";
                     program;
                     symbol "}";
                     return "";
            } <|>
            return "";
           }
}
```

7 Examples of use

In this section we are going to explain how the interpreter works, how the input string is parser, and we will show some examples of input programs.

7.1 Execution of the code

The **main** modules contains the basic functions used to run the interpreter and parse the code, these functions are:

```
main :: IO String
execute :: [(Environment, String, String)] -> String
getMemory :: [(Environment, String, String)] -> String
menu :: IO String
logo :: IO String
```

The main function, prints the logo and the interpreter's interactive line, where we can insert the code to be parsed:

```
main = logo;
logo = do {
    putStrLn $ "\t Welcom in EOLI (Edoardo Oranger Language Interpreter v1.0)!";
    putStrLn $ "\t Write the code to execute, or load it from file using :load <file-name>";
        menu;
}
```

Both function are IO monads of type strings, indicating that we can execute IO actions of strings in sequences.

```
menu = do {
    putStr $ "EOLI>";
    hFlush stdout;
    input <- getLine;

    if ((Prelude.head (wordsWhen (==' ') input)) == ":load") then do {
        fileContent <- readFile ((Prelude.head (Prelude.tail (wordsWhen (==' ') input))) ++ ".txt");
        putStrLn (execute (parse parseProgram [] fileContent));
        menu;
    }else if (input == "exit") then
        return "Bye!"
        else do {
            putStrLn (execute (parse parseProgram [] input));
            menu;
        }
}
```

The menu function prints the interactive line of the interpreter and checks the input. If the input string is ":load filename", the interpreter checks if there exits a file with that name, and executed the code inside them. Otherwise, if the user type "exit", the interpreter terminates the execution, or if the user type some code, the interpreter runs it. To execute the code, the function **execute** is used. This function requires the result of a parser as input, and when we asks the interpreter to run some code, it executes for the fist the parsers that checks the syntax of the code.

```
execute [] = "Invalid Input \n"

execute [(_, parsedString, "")] =
        "Parsed code: \n\n " ++ parsedString ++ "\n\n" ++
        "Memory: \n\n" ++ (getMemory (parse program [] parsedString))
```

```
execute [(_, parsedString, notParsedString)] =
        "Parsed code: \n\n" ++ parsedString ++ "\n\n" ++
        "Memory \n\n" ++ (getMemory (parse program [] parsedString )) ++
        "Error: \n\n Unused input '" ++ notParsedString ++ "'\n"
```

As we can see, the execute function analyze the results of the parser for the syntax. If the parser fails (it means that an empty list is returned), then the error message "Invalid input" is printed. On the other hand, if the parse success partially, with an unparsed string as result, the interpreter will run the correct code, but printing an error message, indicating that not all the code is correct. Finally, if the code is correct, the content of the memory is printed, indicating that all the code has been parsed successfully. However, we must notice two things:

- The first thing is that each time we run a code, an empty environment is initialized. It means that if we run a code and then another one immediately, the execution of the first code does not influence the second one, since the environment is a local variable for each execution.
- Them, we do not use immediately the parser that executes a code, first of all, we need to check the correctness of the syntax of the input string. Once we verified that the input string is correct, we can run the parser of the program.

Finally, let's have a look of the function to print the content of the memory:

```
getMemory [] = "Invalid input \n"
getMemory [(x:xs,parsedString, "")] = case typeV x of
        Boolean -> case value x of
                (Val 1) -> " Boolean: " ++ (name x) ++ " = True\n"
                    ++ (getMemory [(xs,parsedString,"")])
                (Val 0) -> " Boolean: " ++ (name x)
                    ++ " = False\n" ++ (getMemory [(xs,parsedString,"")])
        Integer -> " Integer: "
                    ++ (name x)
                    ++ " = " ++ (show (value x))
                    ++ "\n"
                    ++ (getMemory [(xs,parsedString,"")])
        ArrayInt -> " [Int]: "
                    ++ (name x)
                    ++ " = "
                    ++ (show (value x))
                    ++ "\n"
                    ++ (getMemory [(xs,parsedString,"")])
        ArrayBool -> " [Bool]: "
                    ++ (name x)
                    ++ " = "
                    ++ (showBoolean (value x))
                    ++ "\n"
                    ++ (getMemory [(xs,parsedString,"")])

getMemory [(env,parsedString,notParsedString)] = case notParsedString of
        "" -> ""
        otherwise -> " Error (unused input '"
                        ++ notParsedString ++ "')\n"
                        ++ getMemory [(env,parsedString, "")]

showBoolean :: Value Int -> String
showBoolean (Array []) = "[]"
showBoolean (Array xs) = "[" ++ showSingleBoolean (Array xs) ++ "]";

showSingleBoolean :: Value Int -> String
```

```
showSingleBoolean (Array [x]) | x == 1 = "True"
                              | x == 0 = "False"
showSingleBoolean (Array (x:xs)) | x == 1 = "True, " ++ showSingleBoolean (Array xs)
                                 | x == 0 = "False, " ++ showSingleBoolean (Array xs)
```

The function **getMemory** prints the result of the execution of the parser. Of course, if the parser returns an empty array, an error message is printed. If the parser returns an unparsed string as result, we also prints the content of the environment, but indicating that a part of the code has not been parsed successfully. Finally, when we are going to print the memory, we must distinguish between the type of the current variable the we are printing. In fact, if the variable is a boolean, we must convert the 0 in a False and the 1 in an Integer. Indeed, when we are going to print the content of an array of booleans, we use the function **showBoolean**, that returns an array, converting the integers in boolean variables.

7.2 Examples

Let's consider as a first example a program that given two integers, increase the smaller since it is smaller than the first, and decrease the greater.

```
int i := 10;
int j;
while(i > j){
    i := i - 1;
    j := j + 1;
}
```

with the corresponding result:

```
Memory:
    Integer -> i = 5;
    Integer -> j = 5;
```

Notice that we do not initialize the second variable, since the interpreter do that for us. Let's consider now another example, this time with the arrays. Given an array of boolean, we initialize an array of integers with the same length, and using a variable that starts from a number n, if the current value of the array of boolean is True, we increment the variable and write it in array of integers.

```
int length := 6;
bool[] x := [True, False, True, True, False, True];
int[length] y;
int i := 0;
int k := 1;
while(i < length){
    if (x[i]){
        y[i] := k;
        k := k + 1;
    }
    i := i + 1;
}
```

with the correspoding result:

```
Memory:
    Integer: length = 6
    [Bool]: x = [True, False, True, True, False, True]
    [Int]: y = [1, 0, 2, 3, 0, 4]
    Integer: i = 6
    Integer: k = 5
```

Therefore, when we declare an array of a specified length, the interpreter update the environment, adding a variable of zeros, and we can also declare the length of an array, and passing them inside the declaration of the array, since the number required to initialize an array is an arithmetic expression.

Let's consider a final example, this time using the comments, the concatenation with arrays and the for loop. We want to create a program that given an array of integers, if the sum of the values is greater then the square of the second element, divided by 2, then add to another array the squares of the first.

```
int length := 10;
int[] x := [1, 2, 3, 4, 5] ++ [6, 7, 8, 9, 10];
int[length] y;

-- Define the sum of the array
int sum := 0;
int i := 0;
while(i < length){
    sum := sum + x[i];
    i := i + 1;
}

bool condition := (sum > (x[1]^2)/2);
if (condition) {
    for(int i:= 0; i < length; i:=i+1;){
        y[i] := (x[i] ^ 2);
    }
}
```

with the corresponding result:

```
Integer: length = 10
[Int]: x = [1,2,3,4,5,6,7,8,9,10]
[Int]: y = [1,4,9,16,25,36,49,64,81,100]
Integer: sum = 55
Integer: i = 10
Boolean: condition = True
```