FORMAL METHODS FOR COMPUTER SCIENCE

I|m|P|T|e|R

Imperative Language Interpreter

Student: Ambra Urso
Student number: 767580
Professor: Giovanni Pani

Summary

1 – INTRODUCTION

The following document describes IMP TER, an interpreter for IMPt, a basic imperative language, written in Haskell.

IMPt syntax is close to the C language and include the most common types of data:

- Int : represents Integer numbers, that can have positive or negative values;
- Bool: represents Boolean values, which are True and False;
- Array: represents the Array, a data structure that contains a group of homogenous elements. In IMP TER, we can only have Arrays of Int values.

To make operations through this language, is it possible to use some control structures:

- Assignment: construct that is able to assign a value to a specific identifier. These identifiers can be:
  - Variables;
  - Arithmetic expression.
- Skip: performs a jump. This operation doesn't change the state or the execution flow;
- If-then-Else: a condition based control structure;
- While: a control flow statement that allows code to be executed repeatedly based on a given Boolean condition;
- For: iterative control structure that results in a portion of the program being executed repeatedly for a certain known number of times.
- Repeat Until: a control flow statement that allows code to be executed repeatedly based on a given Boolean condition;
- Do While: a control flow statement that allows code to be executed repeatedly based on a given Boolean condition;
- Operations Between Arrays

This documentation will describe and explain all the steps taken to implement the IMP TER interpreter, from the core representation of the program to the grammar of IMPt.

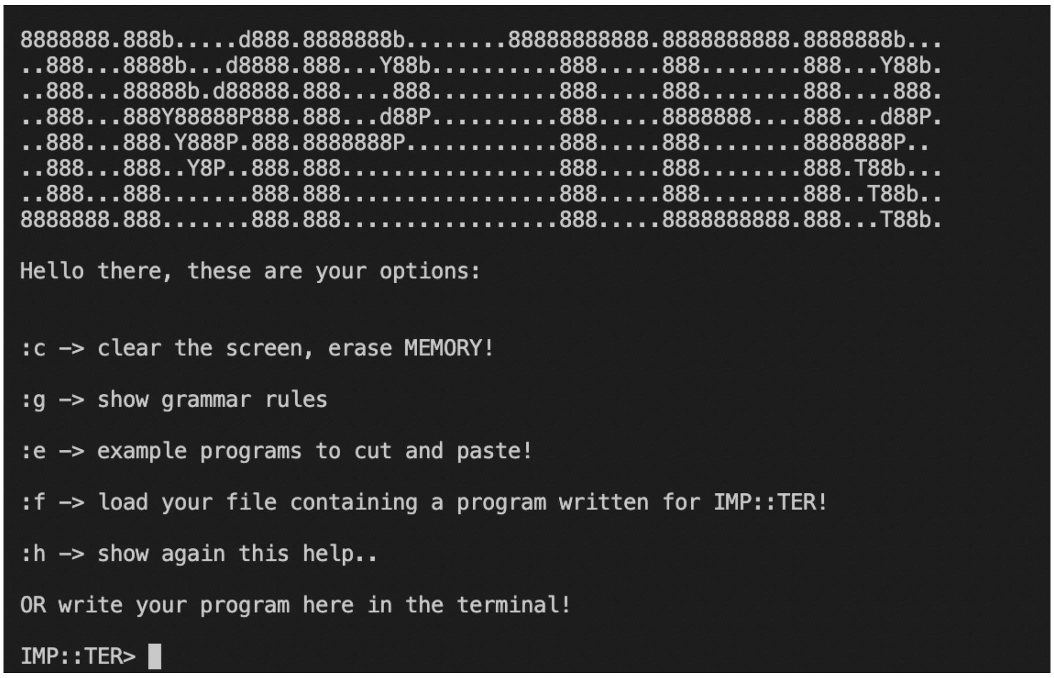
1.1 - SOMETHING ABOUT PARSERS

A Parser is the part of the program with the purpose of building the intermediate representation tree with all the expressions to interpretate, starting from a source code and creating its syntactical structure, according to the grammar.
To build a Parser in Haskell we need to introduce the type Parser in order to mark the mapping from Strings to a Parser object, that will be built as a tuple of a generic type a and the String not yet parsed.
To introduce a new Parser, we have to use Newtype. It is used to indicate that for the new type we're going to define, we will also define instances of Functor, Applicative and Monad classes for the Parser type constructed.

1.2 - INTERPRETER

An Interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring
them to be previously compiled into a machine language program.
Generally, an interpreter either parsers the source code and performs its
behavior directly, or translates source code into some efficient intermediate
representation and immediately executes it.

IMP TER has the following capabilities:



```
8888888.888b.....d888.8888888b........88888888888.8888888888.8888888b...
..888...8888b...d8888.888...Y88b..........888.....888........888...Y88b.
..888...88888b.d88888.888....888..........888.....888........888....888.
..888...888Y88888P888.888...d88P..........888.....8888888....888...d88P.
..888...888.Y888P.888.8888888P............888.....888........8888888P..
..888...888..Y8P..888.888.................888.....888........888.T88b...
..888...888.......888.888.................888.....888........888..T88b..
8888888.888.......888.888.................888.....8888888888.888...T88b.

Hello there, these are your options:


:c -> clear the screen, erase MEMORY!

:g -> show grammar rules

:e -> example programs to cut and paste!

:f -> load your file containing a program written for IMP::TER!

:h -> show again this help..

OR write your program here in the terminal!

IMP::TER>
```

A user can:

- write a program directly in the terminal
- Load a program from a file by inputting the file name
- See some example programs
- See the grammar rules

Files should be stored in the interpreter folder or an absolute path has to be provided.
For convenience, an executable file is provided in the folder.

Test files (t0 to t8) are provided.

2 - GRAMMAR

Grammar shows the internal representation of the program data that will be accepted by the Interpreter, defines the structure of the types, operations and commands that can be used by the interpreter.

- Arithmetic expression grammar:

```
<aexp> ::= <aterm> ‘+’ <aexp>
             | <aterm> ‘ –’ <aexp>
             | <aterm>

<aterm> ::= <afactor> ‘*’ <aterm> | <afactor>

<afactor> ::= (<aexp>) ‘*’ <integer> | <identifier>
```

- Boolean expression grammar:

```
<bexp> ::= <bterm> ‘||’ <bexp> | <bterm>

<bterm> ::= <bfactor> ‘&&’ <bterm> | <bfactor>

<bfactor> ::= ‘true’ | ‘false’
    | !<bfactor>
    | (boolExp)
    | <bcomparison>

< bcomparison> ::=        <aexp> ‘==‘   <aexp>   |
                          <aexp> ‘!=‘   <aexp>   |
                          <aexp> ‘<=‘   <aexp>   |
                          <aexp> ‘>=‘   <aexp>   |
                          <aexp> ‘ >’   <aexp>   |
                          <aexp>  ‘<‘   <aexp>   |
```

- Command grammar:

```
<program>  ::= <command> | <command> <program>


<command > ::= <assignment> | <ifThenElse> | <while> | <RepeatUntil>
<arrayassignment> | <forLoop> | <SpecialFun> | skip;

<assignment>  ::= <identifier>  ‘:=‘  <aexp>


<ArrayAssignment> ::= <identifier> ‘@‘ <array> ‘++’ <array>
                                   | ‘@‘ <array> ‘.’ <array>
                                   |‘@‘ <array> ‘+.’ <array>
```

| ‘@‘ <array> ‘*.’ <array>
| ‘:=’ <array>

<ifThenElse> ::= if (<boolExp>) then { <program> }
| if (<boolExp>) then {<program>} else {<program>}

<forLoop>::= for (<assignment> <boolExp> <assignment>) {<program>}

<While>::= while (<boolExp>) do {<program>}

<doWhile>::= do {<program>} while (<boolExp>)

<RepeatUntil>::= repeat {<program>} while (<boolExp>)

- Identifiers:

<lower> ::= a-z
<upper ::= A-Z
<integer> ::= [-]<nat>
<identifier> ::= <lower>| <lower><alphanum>| <lower>”[“<alphanum>”]”
<alphanum>::=<upper><alphanum>
| <lower> <alphanum>
|<natural> <alphanum>
| <upper> |<lower> | <natural>
<lower> ::= a-z
<upper>::= A-Z

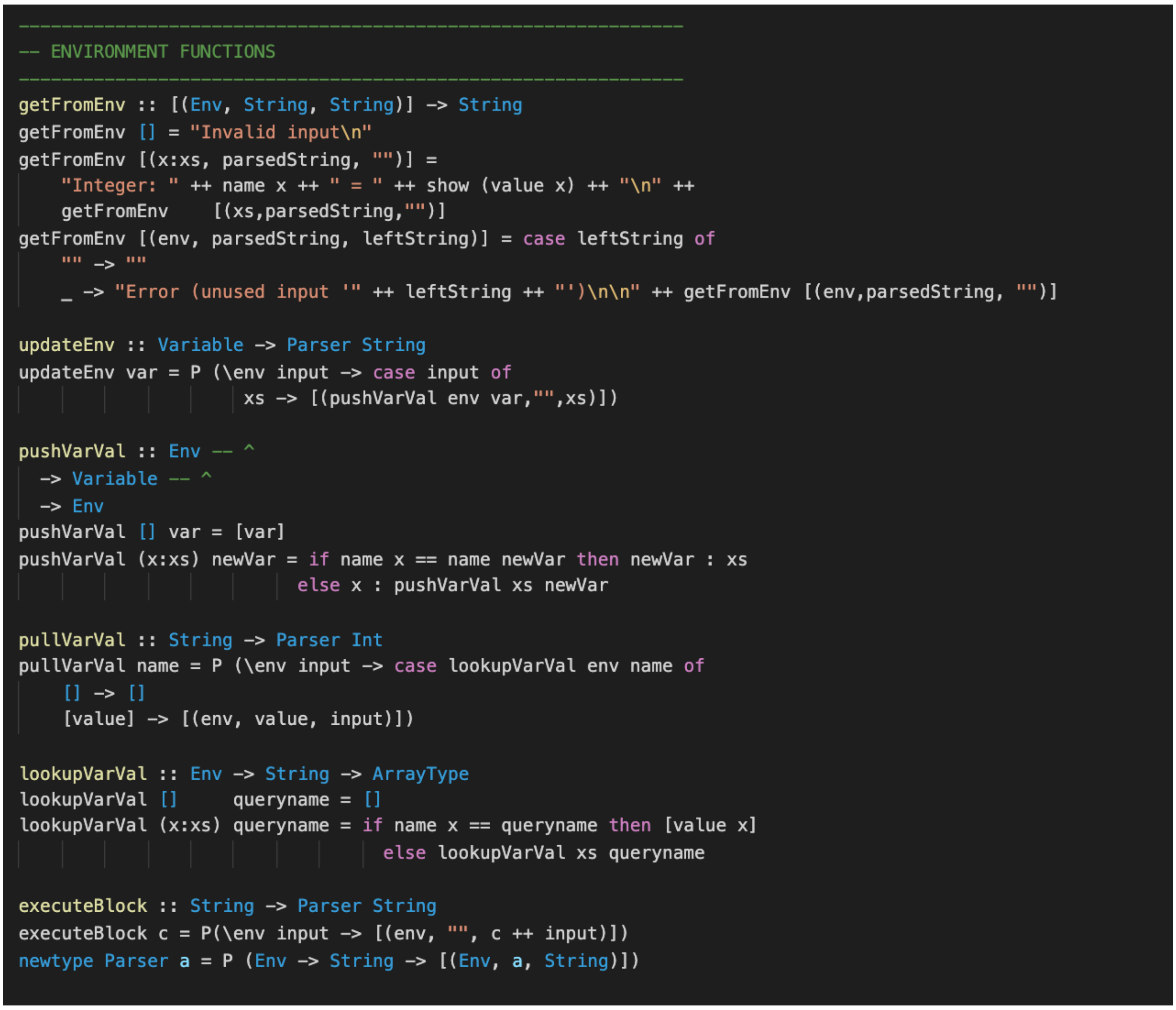
3 - ENVIRONMENT

IMP TER interpreter supports the adding of types in the program in a following extension. A Variable is a data structure composed by a name of String type, a string indicating the type of the variable, defined by a String type and a value with its corresponding Integer values:

The Environment, denoted as Env, is a list of variables, defined in IMP TER in this way:

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

The Environment is like a memory that can be updated (and read from) throughout the program execution. In the next step of the document we will see how we can manipulate and manage the environment.
To correctly use the Environment, we need functions that allow us to manipulate it:



```haskell
---------------------------------------------------------------
-- ENVIRONMENT FUNCTIONS
---------------------------------------------------------------
getFromEnv :: [(Env, String, String)] -> String
getFromEnv [] = "Invalid input\n"
getFromEnv [(x:xs, parsedString, "")] =
    "Integer: " ++ name x ++ " = " ++ show (value x) ++ "\n" ++
    getFromEnv    [(xs,parsedString,"")]
getFromEnv [(env, parsedString, leftString)] = case leftString of
    "" -> ""
    _ -> "Error (unused input '" ++ leftString ++ "')\n\n" ++ getFromEnv [(env,parsedString, "")]

updateEnv :: Variable -> Parser String
updateEnv var = P (\env input -> case input of
                                  xs -> [(pushVarVal env var,"",xs)])

pushVarVal :: Env -- ^
  -> Variable -- ^
  -> Env
pushVarVal [] var = [var]
pushVarVal (x:xs) newVar = if name x == name newVar then newVar : xs
                                       else x : pushVarVal xs newVar

pullVarVal :: String -> Parser Int
pullVarVal name = P (\env input -> case lookupVarVal env name of
    [] -> []
    [value] -> [(env, value, input)])

lookupVarVal :: Env -> String -> ArrayType
lookupVarVal []     queryname = []
lookupVarVal (x:xs) queryname = if name x == queryname then [value x]
                                        else lookupVarVal xs queryname

executeBlock :: String -> Parser String
executeBlock c = P(\env input -> [(env, "", c ++ input)])
newtype Parser a = P (Env -> String -> [(Env, a, String)])
```

4 - PARSERS

As told in the introduction, we can say that a parser is like a function that takes in input a string and return a generic type result in the output. The parser strategy employed for the implementation of this interpreter is based on a set of Parser-type monads.

We can summarize the progression of a parser starting with the basic parser that maps a string to a tree:

```haskell
type Parser = String -> Tree
```

Since a parser is not always able to process the entire input, we can make sure that it also returns the part of the input that it failed to consume:

```
type Parser = String -> (Tree , String )
```

Since a parser could fail, we can generalize it to return a list of result, assuming that an empty list denotes its failure:

```
type Parser = String -> [( Tree , String )]
```

Since we can have many different types of parser and different types of tree, we can further generalize it to return generic types:

```
type Parser = String -> [(a, String )]
```

The last adding done to this design, is the one of adding the environment.
It is our memory that allow us to do parsing with initial state and ending with some final state:

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

As we said at the beginning of the document, to introduce a new Parser, we can switch from type to Newtype. It is used to indicate that for the new type we're going to define, we will also define instances of Functor, Applicative and Monad classes for the Parser type constructed.

5 - FUNCTOR, APPLICATIVE AND MONAD

Making a Parser an instance of functor, applicative or monad, is it possible to:

- get advantages from the do notation;
- combine parsers in sequence.

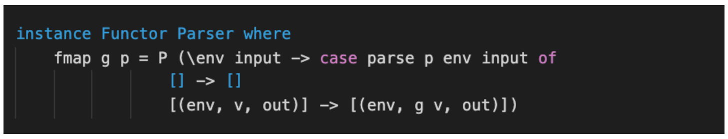
5.1 – FUNCTOR

A type f is a Functor if it provides a function fmap which, given any types a and b lets apply any function from a to be to turn an f a into an f b. Everything happens preserving the structure of the function f.
So, we can say that a Functor is a mapping between categories.

Furthermore, our function needs to adhere to the following:

- Identity : fmap id == id;
- Composition: fmap(f . g) == fmap f . fmap g

To make the Parser a Functor, we do it this way:

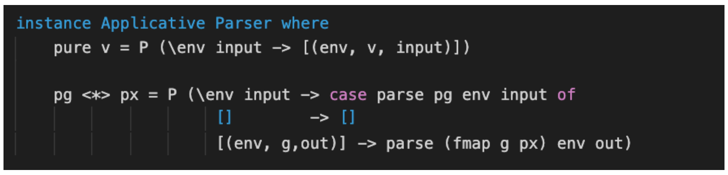


```
instance Functor Parser where
    fmap g p = P (\env input -> case parse p env input of
                    [] -> []
                    [(env, v, out)] -> [(env, g v, out)])
```

Applying fmap, we apply a function to the result value for a parser (if it succeed), and propagate the failure otherwise.
If it receive in input an empty list, it will return an empty list.

5.2 - APPLICATIVE

The Applicative is an intermediate structure between a Functor and a Monad. It is an instance that can be implemented only if a Functor is already implemented. The implementation of Applicative includes also the definition of Pure. Pure encapsulates a value into an arbitrary Applicative Functor. If we consider for example a "Pure a", we can say that can have many meaning:
Just a, [0], (empty, 0), etc.



```haskell
instance Applicative Parser where
    pure v = P (\env input -> [(env, v, input)])

    pg <*> px = P (\env input -> case parse pg env input of
                                 []             -> []
                                 [(env, g,out)] -> parse (fmap g px) env out)
```

In our case, pure transforms a value into a parser. It always succeeds giving this values as result, without consuming other part of the input string.

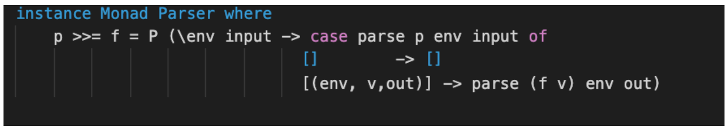
5.3 – MONADS

Monads are natural extension of Applicative Functors. They are used to apply a function that returns a wrapped value and to simulate, in the functional language, the behavior of an Imperative language. Monad is an abstract datatype of actions. Through the expression "do", Haskell gives us a convenient syntax for writing monadic expressions.

Every instance of Monad should satisfy the following properties:

- Left identity: return a >>= k = ka
- Right identity: m >>= return = m
- Associativity: m >>= (\x -> k x >> = H) = (m >>= k) >>= k

In our case:



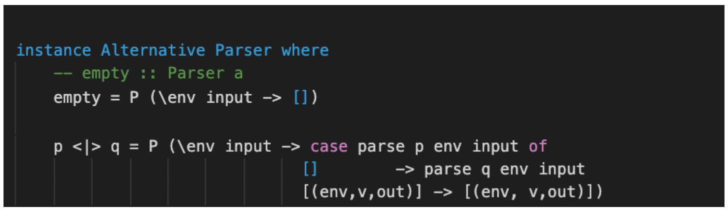
```haskell
instance Monad Parser where
    p >>= f = P (\env input -> case parse p env input of
                                        []             -> []
                                        [(env, v,out)] -> parse (f v) env out)
```

6 - ALTERNATIVES

Alternative allow us to apply more than one parser to an input simulating a choice. More precisely we apply a parser to an input string. If parsers fails, then we are able to apply another parser to the same input string.
There are two main primitives we can consider:

empty: represent a failed alternative;
<|>: represent an appropriate choice operator for the type.

Talking about our parser type, we can say that empty is a parser that fails, regardless of the input received. Differently <|> is a choice operator. It applies the first parser to the input and, if parser succeeds returns the result. Otherwise, it applies the second parser to the same input.

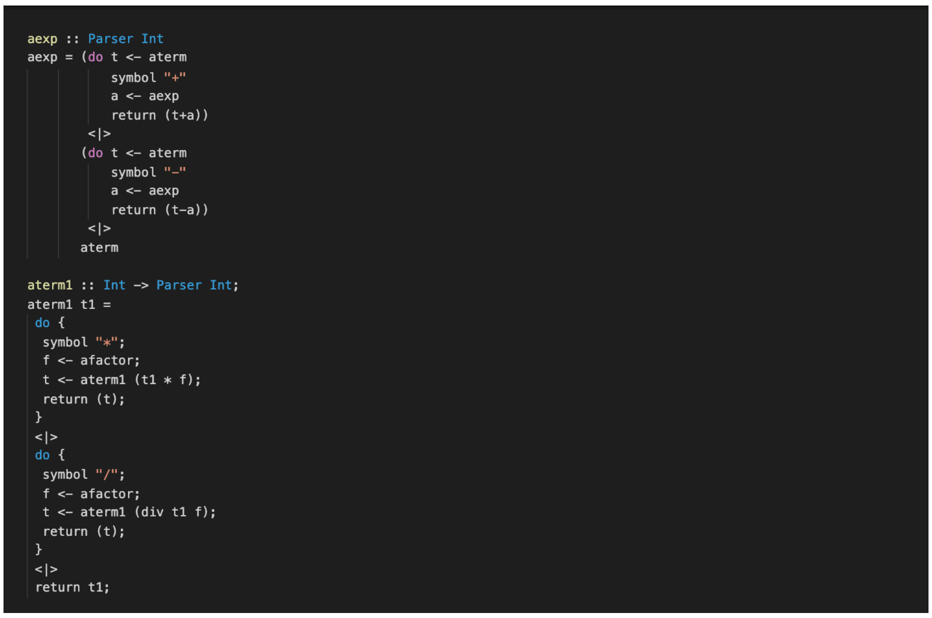


```haskell
instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\env input -> [])

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

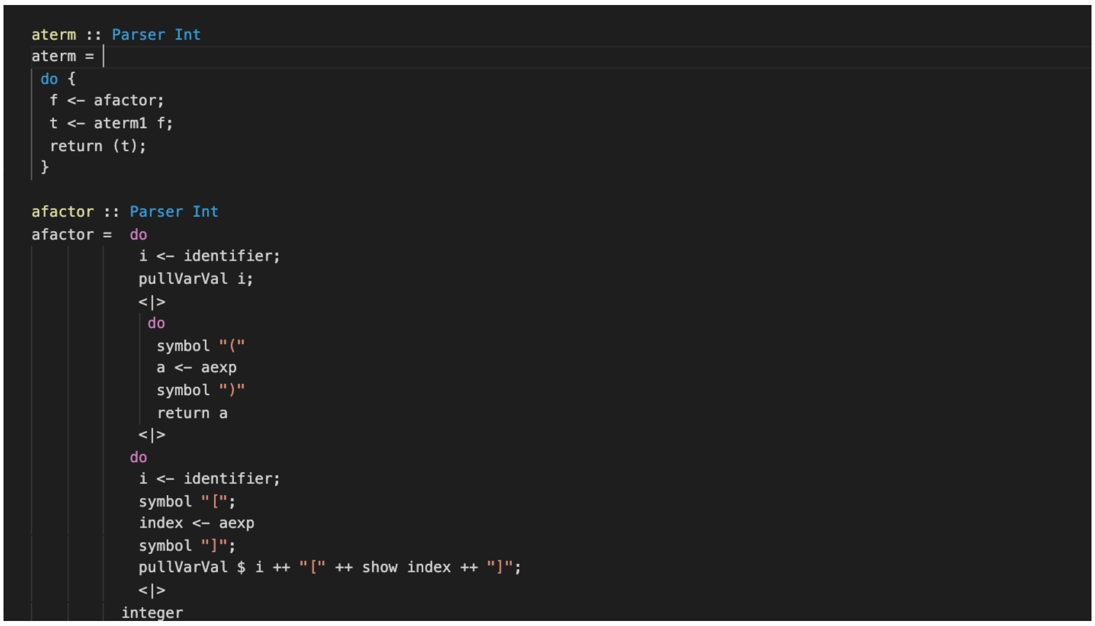
7 - ARITHMETIC PARSING

Just after completing the implementation of the main class instances, the Parser is carried on with the definition of the arithmetic parsing. It is divided into three sub-categories:

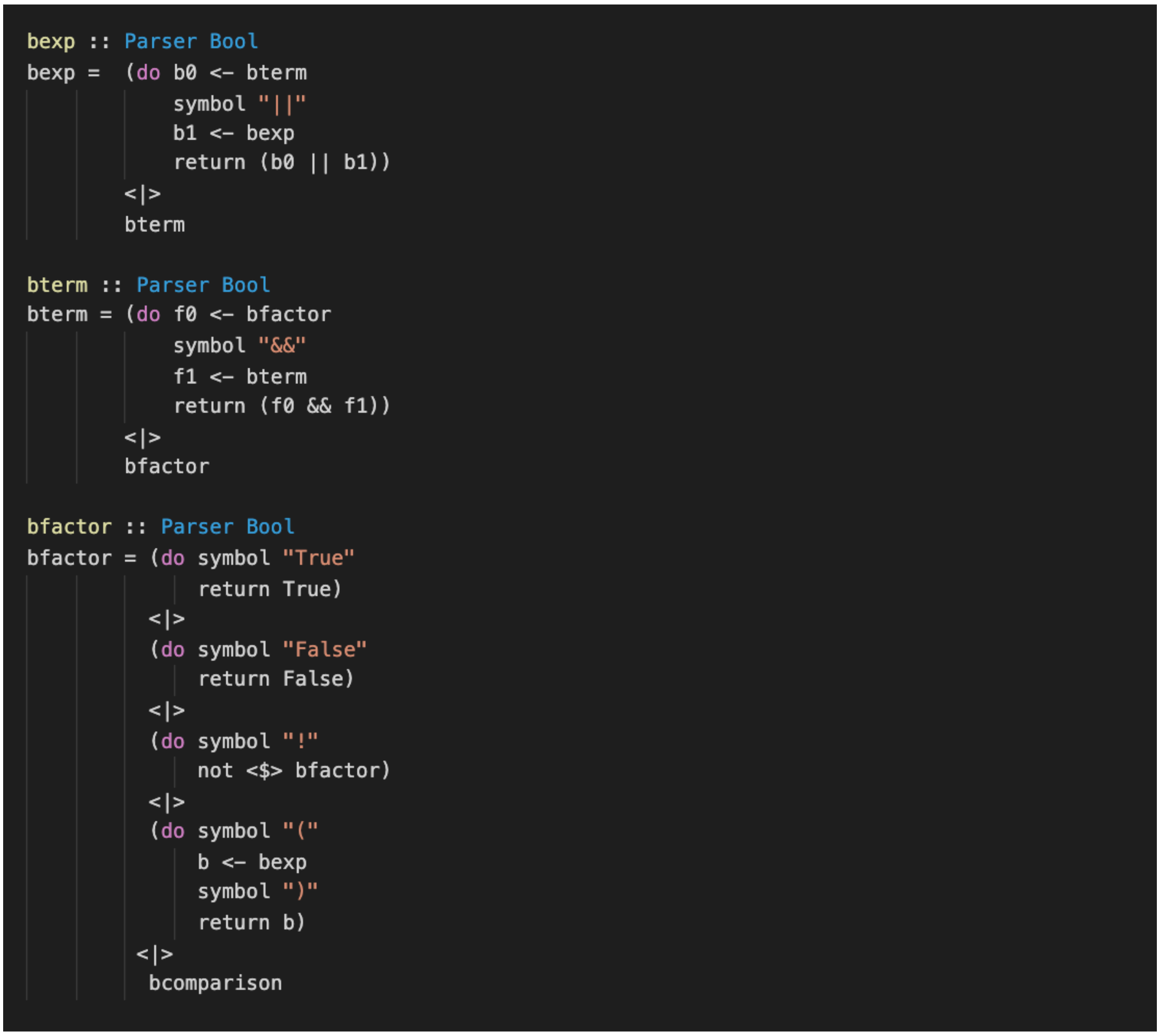
- aexp: a parser that manages additions, subtractions and the expansion of aterm;
- aterm/aterm1: parsers that manages multiplication between afactor and other aterm;
- afactor: parser that manages the recursion on aexp



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

aterm1 :: Int -> Parser Int;
aterm1 t1 =
 do {
  symbol "*";
  f <- afactor;
  t <- aterm1 (t1 * f);
  return (t);
 }
 <|>
 do {
  symbol "/";
  f <- afactor;
  t <- aterm1 (div t1 f);
  return (t);
 }
 <|>
 return t1;
```



```
aterm :: Parser Int
aterm =
 do {
  f <- afactor;
  t <- aterm1 f;
  return (t);
 }

afactor :: Parser Int
afactor =  do
            i <- identifier;
            pullVarVal i;
            <|>
             do
              symbol "("
              a <- aexp
              symbol ")"
              return a
            <|>
           do
            i <- identifier;
            symbol "[";
            index <- aexp
            symbol "]";
            pullVarVal $ i ++ "[" ++ show index ++ "]";
            <|>
          integer
```



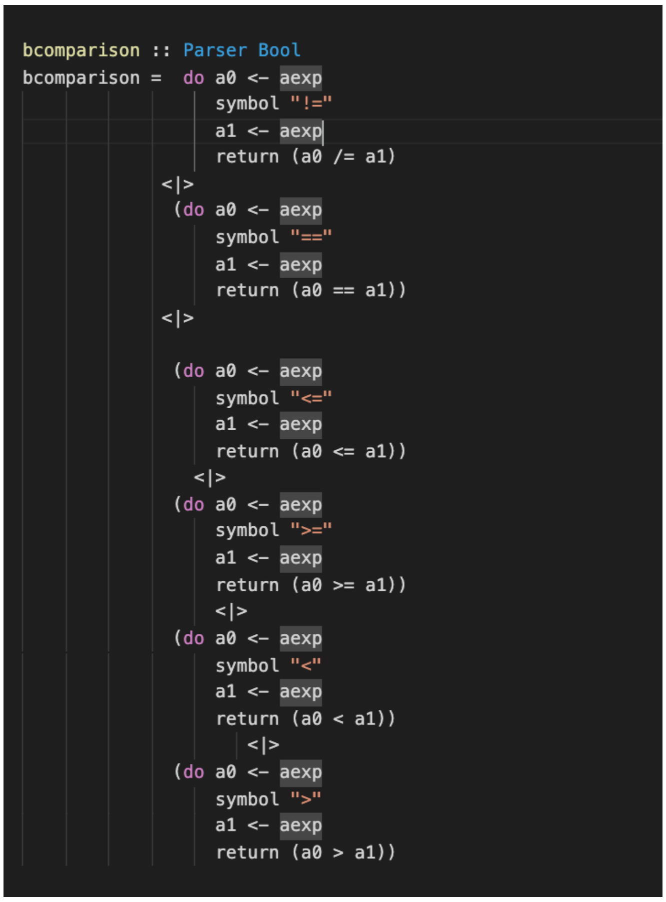
```
bexp :: Parser Bool
bexp =  (do b0 <- bterm
            symbol "||"
            b1 <- bexp
            return (b0 || b1))
        <|>
        bterm

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

bfactor :: Parser Bool
bfactor = (do symbol "True"
              return True)
          <|>
          (do symbol "False"
              return False)
          <|>
          (do symbol "!"
              not <$> bfactor)
          <|>
          (do symbol "("
              b <- bexp
              symbol ")"
              return b)
         <|>
          bcomparison
```

8 - BOOLEAN PARSING

In the same way of arithmetic, Boolean Parsing can be distinguish in many categories:

- bexp;
- bterm;
- bfactor;
- bcomparison

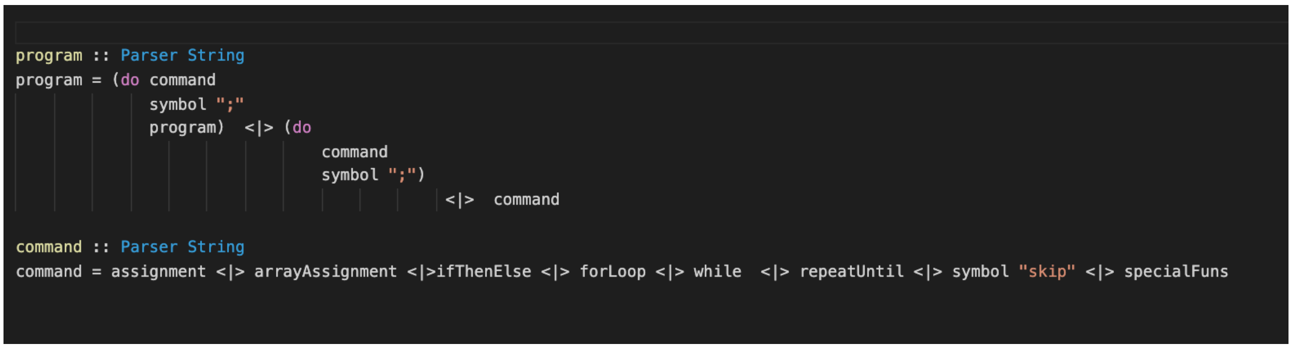
To remove ambiguities, && (AND) operator gets precedence on || (OR)



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

              (do a0 <- aexp
                  symbol "<="
                  a1 <- aexp
                  return (a0 <= a1))
                <|>
              (do a0 <- aexp
                  symbol ">="
                  a1 <- aexp
                  return (a0 >= a1))
                  <|>
              (do a0 <- aexp
                  symbol "<"
                  a1 <- aexp
                  return (a0 < a1))
                     <|>
              (do a0 <- aexp
                  symbol ">"
                  a1 <- aexp
                  return (a0 > a1))
```

9 - COMMAND PARSING

Commands supported in the language are Assignment, ArrayAssignment, While, DoWhile, For, ifThenElse, SpecialFuns and Skip



```
program :: Parser String
program = (do command
              symbol ";"
              program)  <|> (do
                                command
                                symbol ";")
                                            <|>  command

command :: Parser String
command = assignment <|> arrayAssignment <|>ifThenElse <|> forLoop <|> while  <|> repeatUntil <|> symbol "skip" <|> specialFuns
```

Assignments involves only identifiers and arithmetic expressions. The syntax is pretty simple:

- the notation " := " represents a value assignment;
- the notation “;” represent the end of the statement.

Variables are not declared but just assigned. Trying to read a never assigned variable will throw a memory error.

ArrayAssignment is used for assigning values to arrays, element by element or altogether.

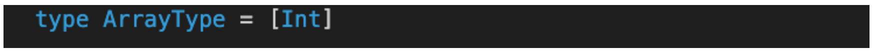
Supported assignments are of the following types:

x[idx]:=Value;
Value:=x[idx];
x[idx]:=y[idx1];
Z:=[v1,v2,v3..];

Where idx is an index and x,y,z are arrays

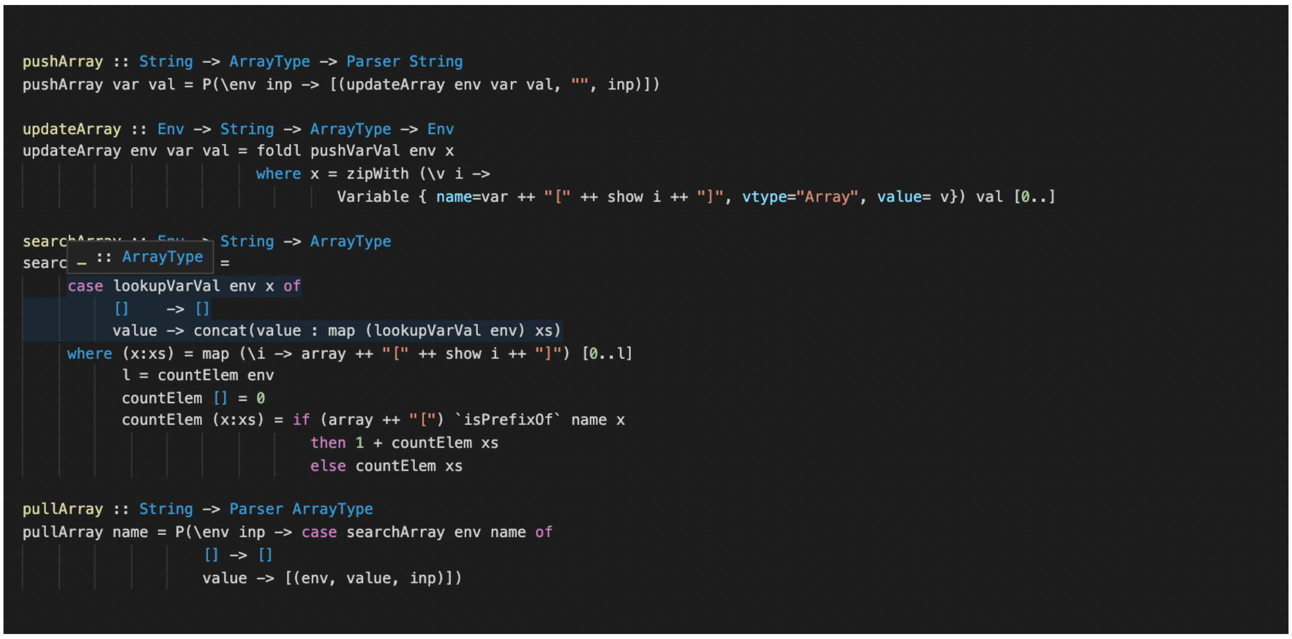
Conditional Statements and Loops are described in section 11 and 12 , while special functions are shown in section 14.

10 – ARRAY IMPLEMENTATION

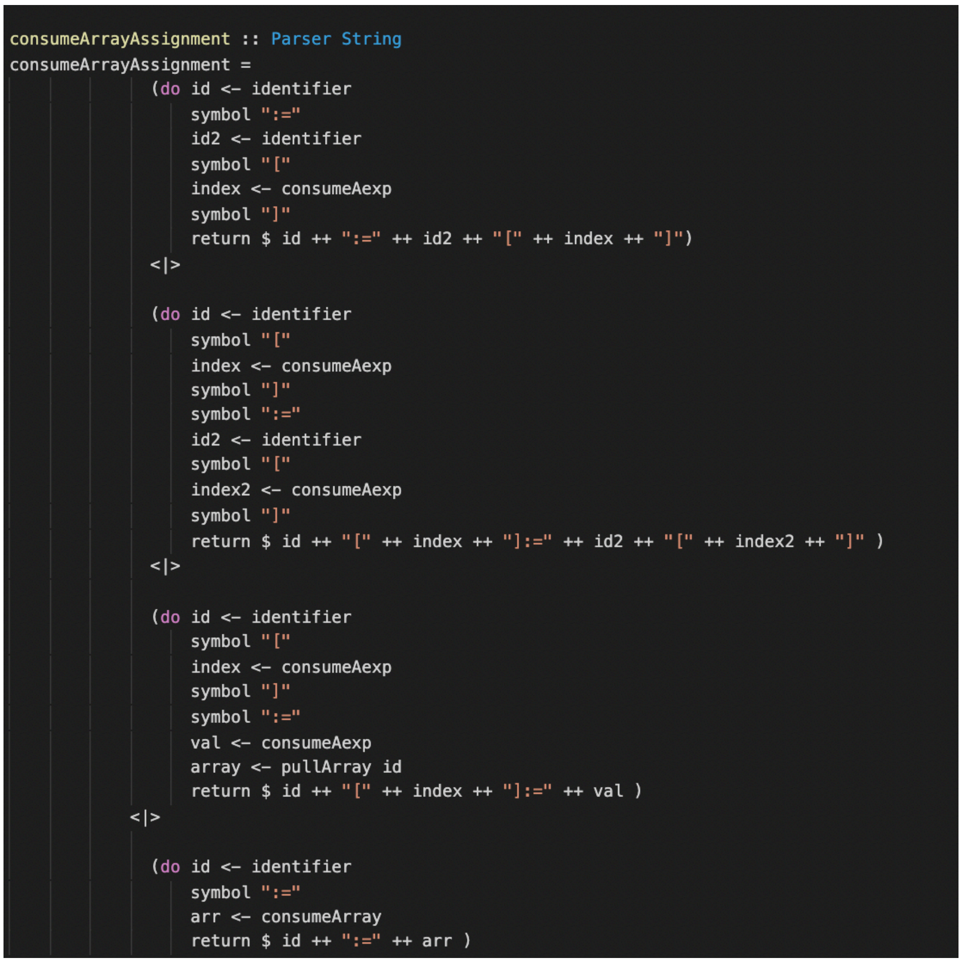


```
type ArrayType = [Int]
```

Arrays are defined as lists of Integer values, and some functions are provided to store and retrieve values of the elements to/from the environment.
Each element is stored as a single Integer value an identifier composed by the array name + the index of the element;

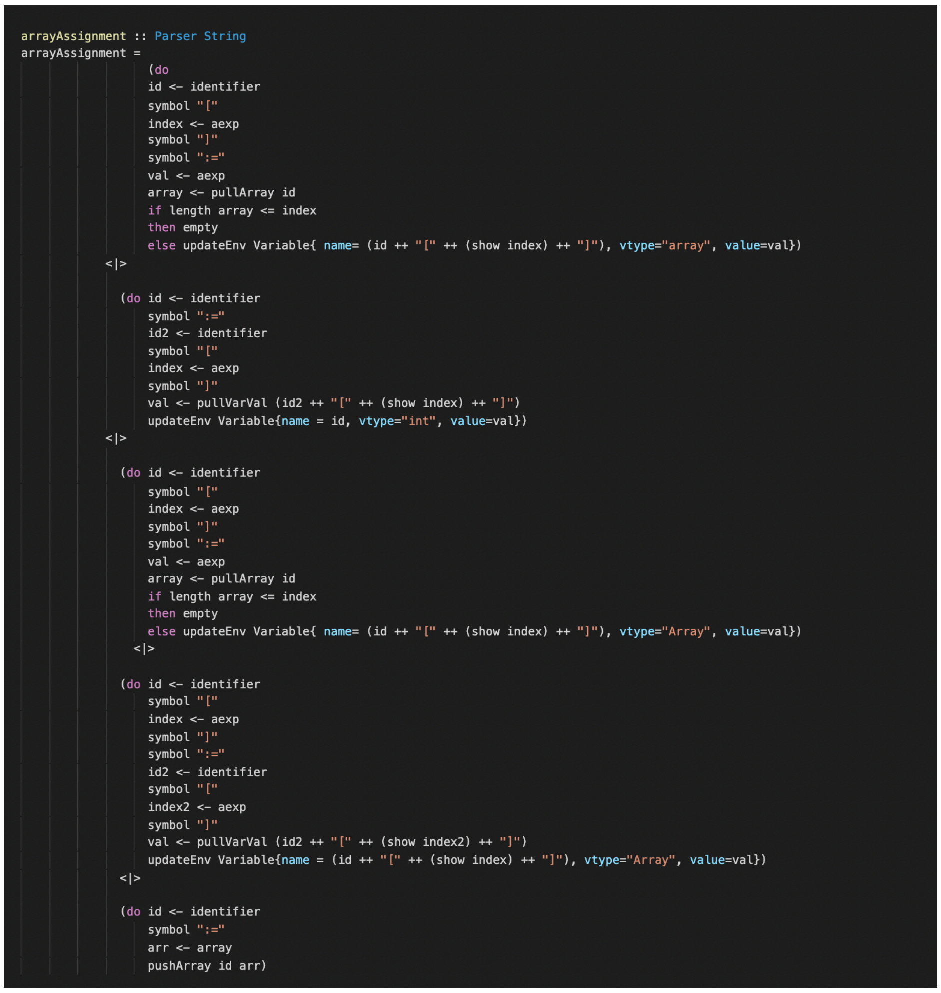


```
pushArray :: String -> ArrayType -> Parser String
pushArray var val = P(\env inp -> [(updateArray env var val, "", inp)])

updateArray :: Env -> String -> ArrayType -> Env
updateArray env var val = foldl pushVarVal env x
                          where x = zipWith (\v i ->
                                     Variable { name=var ++ "[" ++ show i ++ "]", vtype="Array", value= v}) val [0..]

searchArray :: Env -> String -> ArrayType
searc _ :: ArrayType =
    case lookupVarVal env x of
         []    -> []
         value -> concat(value : map (lookupVarVal env) xs)
    where (x:xs) = map (\i -> array ++ "[" ++ show i ++ "]") [0..l]
           l = countElem env
           countElem [] = 0
           countElem (x:xs) = if (array ++ "[") `isPrefixOf` name x
                                then 1 + countElem xs
                                else countElem xs

pullArray :: String -> Parser ArrayType
pullArray name = P(\env inp -> case searchArray env name of
                         [] -> []
                         value -> [(env, value, inp)])
```



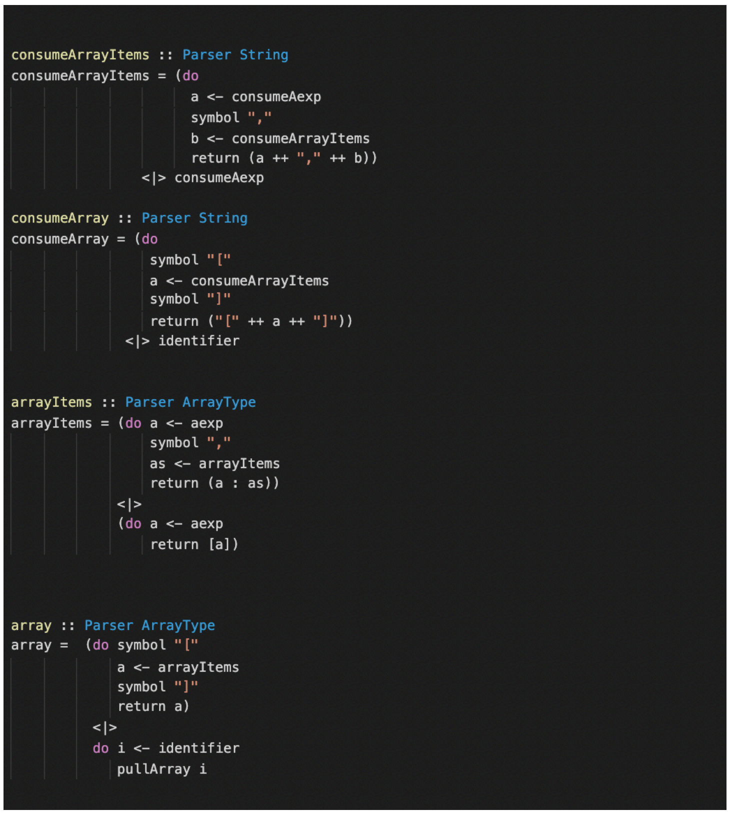
```haskell
consumeArrayAssignment :: Parser String
consumeArrayAssignment =
                (do id <- identifier
                    symbol ":="
                    id2 <- identifier
                    symbol "["
                    index <- consumeAexp
                    symbol "]"
                    return $ id ++ ":=" ++ id2 ++ "[" ++ index ++ "]")
                <|>

                (do id <- identifier
                    symbol "["
                    index <- consumeAexp
                    symbol "]"
                    symbol ":="
                    id2 <- identifier
                    symbol "["
                    index2 <- consumeAexp
                    symbol "]"
                    return $ id ++ "[" ++ index ++ "]:=" ++ id2 ++ "[" ++ index2 ++ "]" )
                <|>

                (do id <- identifier
                    symbol "["
                    index <- consumeAexp
                    symbol "]"
                    symbol ":="
                    val <- consumeAexp
                    array <- pullArray id
                    return $ id ++ "[" ++ index ++ "]:=" ++ val )
              <|>

                (do id <- identifier
                    symbol ":="
                    arr <- consumeArray
                    return $ id ++ ":=" ++ arr )
```

These are the functions defined to assign values to arrays :



```haskell
arrayAssignment :: Parser String
arrayAssignment =
                    (do
                    id <- identifier
                    symbol "["
                    index <- aexp
                    symbol "]"
                    symbol ":="
                    val <- aexp
                    array <- pullArray id
                    if length array <= index
                    then empty
                    else updateEnv Variable{ name= (id ++ "[" ++ (show index) ++ "]"), vtype="array", value=val})
                <|>

                  (do id <- identifier
                    symbol ":="
                    id2 <- identifier
                    symbol "["
                    index <- aexp
                    symbol "]"
                    val <- pullVarVal (id2 ++ "[" ++ (show index) ++ "]")
                    updateEnv Variable{name = id, vtype="int", value=val})
                <|>

                  (do id <- identifier
                    symbol "["
                    index <- aexp
                    symbol "]"
                    symbol ":="
                    val <- aexp
                    array <- pullArray id
                    if length array <= index
                    then empty
                    else updateEnv Variable{ name= (id ++ "[" ++ (show index) ++ "]"), vtype="Array", value=val})
                    <|>

                  (do id <- identifier
                    symbol "["
                    index <- aexp
                    symbol "]"
                    symbol ":="
                    id2 <- identifier
                    symbol "["
                    index2 <- aexp
                    symbol "]"
                    val <- pullVarVal (id2 ++ "[" ++ (show index2) ++ "]")
                    updateEnv Variable{name = (id ++ "[" ++ (show index) ++ "]"), vtype="Array", value=val})
                  <|>

                  (do id <- identifier
                    symbol ":="
                    arr <- array
                    pushArray id arr)
```

These functions parse the array and retrieve values from the env for computations:

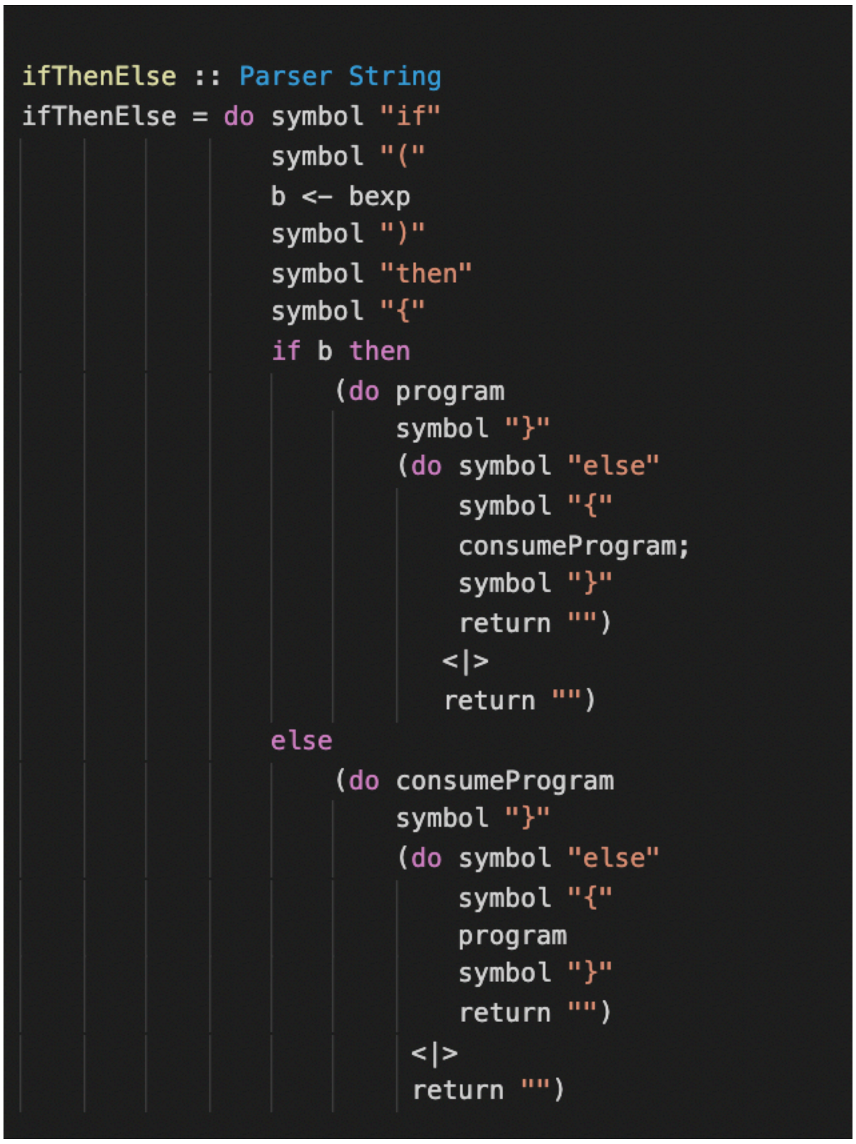


```haskell
consumeArrayItems :: Parser String
consumeArrayItems = (do
                      a <- consumeAexp
                      symbol ","
                      b <- consumeArrayItems
                      return (a ++ "," ++ b))
                   <|> consumeAexp

consumeArray :: Parser String
consumeArray = (do
                 symbol "["
                 a <- consumeArrayItems
                 symbol "]"
                 return ("[" ++ a ++ "]"))
              <|> identifier


arrayItems :: Parser ArrayType
arrayItems = (do a <- aexp
                 symbol ","
                 as <- arrayItems
                 return (a : as))
             <|>
             (do a <- aexp
                 return [a])


array :: Parser ArrayType
array =  (do symbol "["
             a <- arrayItems
             symbol "]"
             return a)
          <|>
          do i <- identifier
             pullArray i
```

11 - CONDITIONAL STATEMENTS

Conditional statements (if-then-else) in IMP TER do not need parentheses
around the boolean condition but for clarity, they are advised; else's are not mandatory.
Once a block is written, at least an instruction has to be present in order to have correct semantic. If block is expected to be empty (for some reason), skip; command will do just that.

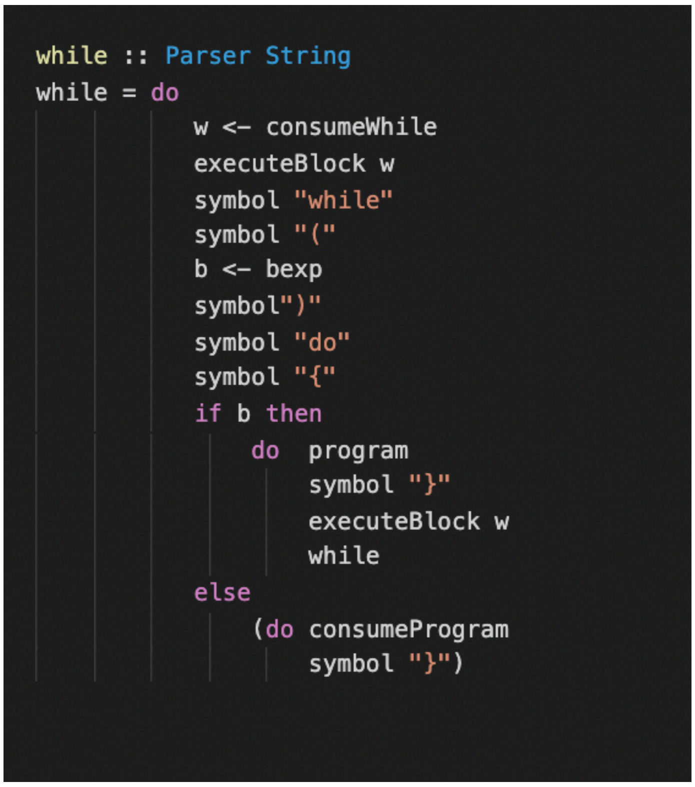
If the condition is not met, the block inside the then condition will be parsed without being executed, while the else block (if
present) will be parsed and executed; if the condition is met, the block inside
the then condition will be parsed and executed, and the block in the else
condition will be parsed but not evaluated.



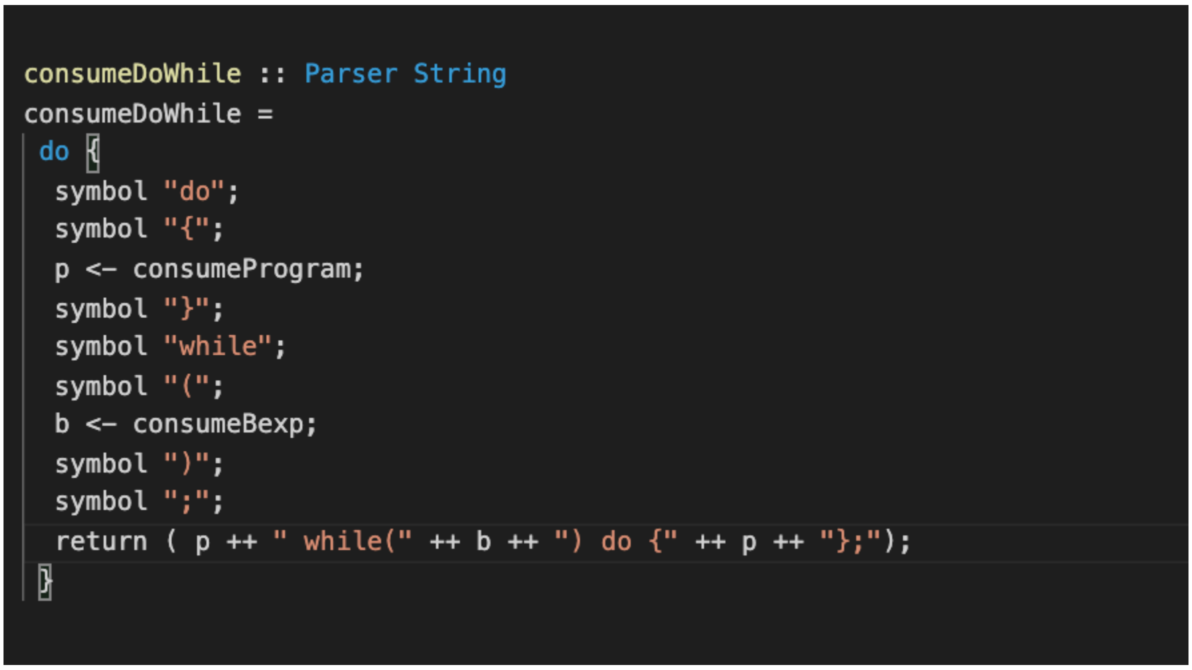
```haskell
ifThenElse :: Parser String
ifThenElse = do symbol "if"
                symbol "("
                b <- bexp
                symbol ")"
                symbol "then"
                symbol "{"
                if b then
                    (do program
                        symbol "}"
   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

12 - LOOP STATEMENTS

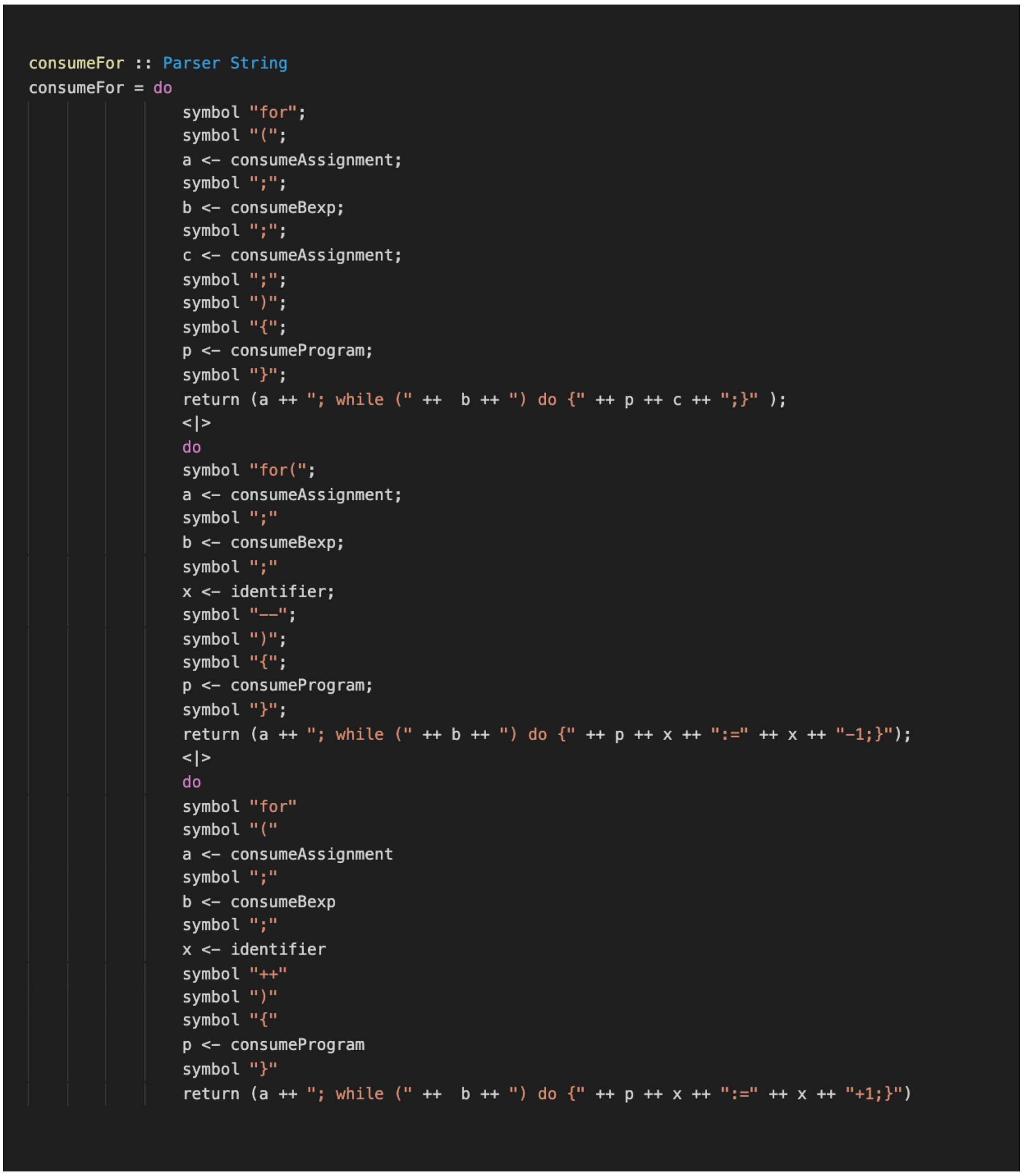
If the boolean condition is met (True), it parses and evaluates the block, adds the block back to unparsed string, and it calls another while parser to evaluate the expression; when the boolean condition is not met anymore, it parses without evaluating the block, and it returns a successful parser.



```haskell
while :: Parser String
while = do
          w <- consumeWhile
          executeBlock w
          symbol "while"
          symbol "("
          b <- bexp
          symbol")"
          symbol "do"
          symbol "{"
          if b then
              do  program
                  symbol "}"
                  executeBlock w
                  while
          else
              (do consumeProgram
                  symbol "}")
```

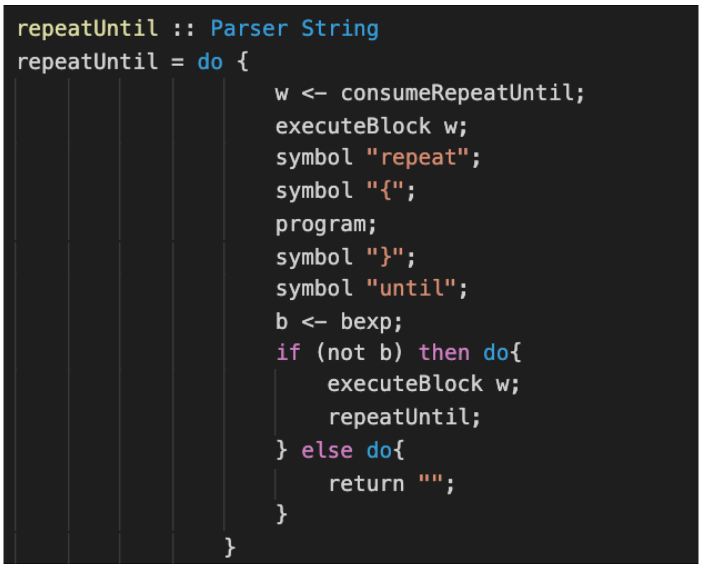


```haskell
consumeDoWhile :: Parser String
consumeDoWhile =
 do {
  symbol "do";
  symbol "{";
  p <- consumeProgram;
  symbol "}";
  symbol "while";
  symbol "(";
  b <- consumeBexp;
  symbol ")";
  symbol ";";
  return ( p ++ " while(" ++ b ++ ") do {" ++ p ++ "};");
 }
```



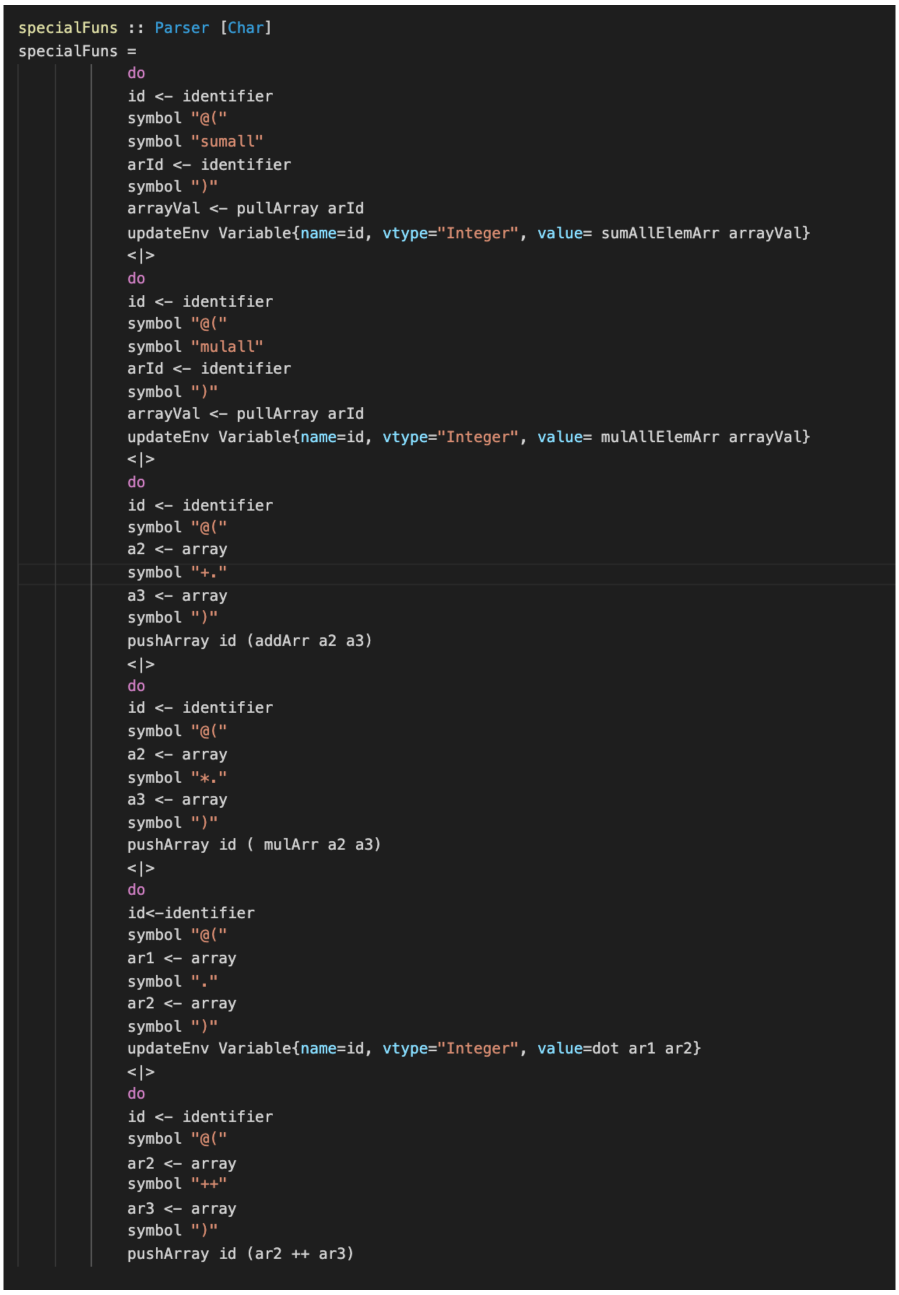
```haskell
consumeFor :: Parser String
consumeFor = do
                symbol "for";
                symbol "(";
                a <- consumeAssignment;
                symbol ";";
                b <- consumeBexp;
                symbol ";";
                c <- consumeAssignment;
                symbol ";";
                symbol ")";
                symbol "{";
                p <- consumeProgram;
                symbol "}";
                return (a ++ "; while (" ++  b ++ ") do {" ++ p ++ c ++ ";}" );
                <|>
                do
                symbol "for(";
                a <- consumeAssignment;
                symbol ";"
                b <- consumeBexp;
                symbol ";"
                x <- identifier;
                symbol "--";
                symbol ")";
                symbol "{";
                p <- consumeProgram;
                symbol "}";
                return (a ++ "; while (" ++ b ++ ") do {" ++ p ++ x ++ ":=" ++ x ++ "-1;}");
                <|>
                do
                symbol "for"
                symbol "("
                a <- consumeAssignment
                symbol ";"
                b <- consumeBexp
                symbol ";"
                x <- identifier
                symbol "++"
                symbol ")"
                symbol "{"
                p <- consumeProgram
                symbol "}"
                return (a ++ "; while (" ++  b ++ ") do {" ++ p ++ x ++ ":=" ++ x ++ "+1;}")
```

In IMP TER, only “While” and “RepeatUntil” loops are evaluated, instead “doWhile” and “For” loops are only parsed and reorganized to emulate a While statement, that is then evaluated.



```haskell
repeatUntil :: Parser String
repeatUntil = do {
                    w <- consumeRepeatUntil;
                    executeBlock w;
                    symbol "repeat";
                    symbol "{";
                    program;
                    symbol "}";
                    symbol "until";
                    b <- bexp;
                    if (not b) then do{
                        executeBlock w;
                        repeatUntil;
                    } else do{
                        return "";
                    }
                }
```

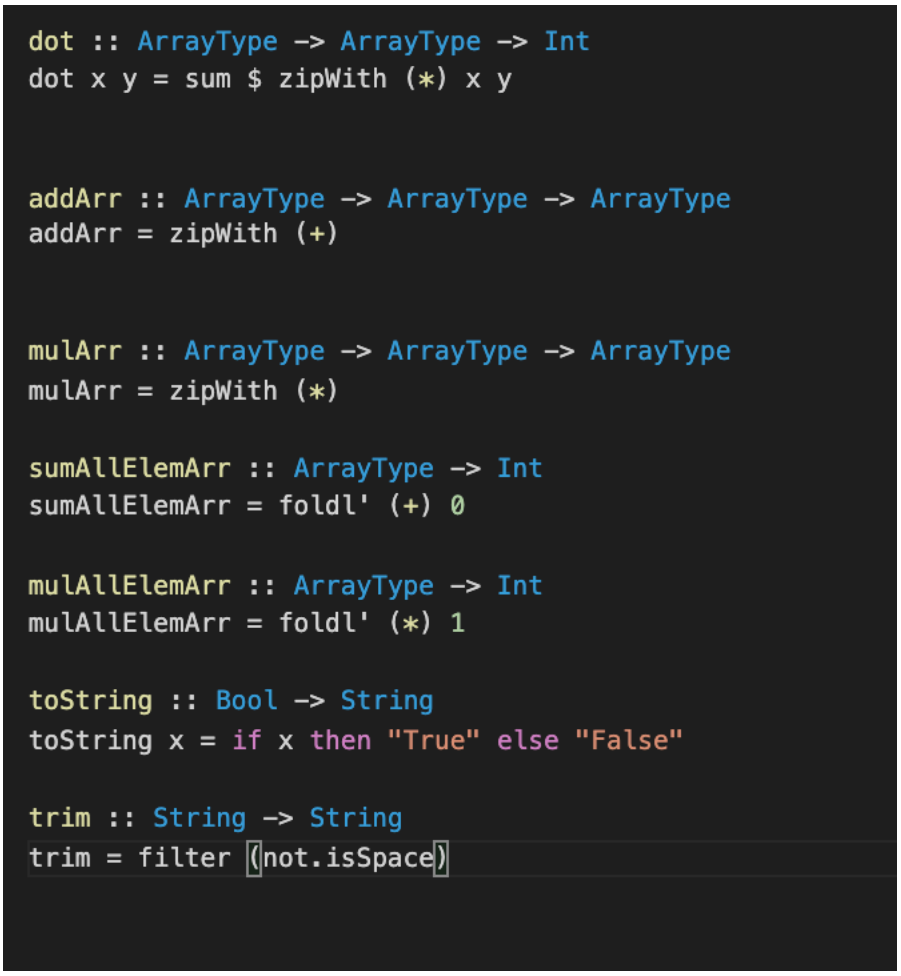
14 - SPECIAL FUNCTIONS



```haskell
specialFuns :: Parser [Char]
specialFuns =
            do
            id <- identifier
            symbol "@("
            symbol "sumall"
            arId <- identifier
            symbol ")"
            arrayVal <- pullArray arId
            updateEnv Variable{name=id, vtype="Integer", value= sumAllElemArr arrayVal}
            <|>
            do
            id <- identifier
            symbol "@("
            symbol "mulall"
            arId <- identifier
            symbol ")"
            arrayVal <- pullArray arId
            updateEnv Variable{name=id, vtype="Integer", value= mulAllElemArr arrayVal}
            <|>
            do
            id <- identifier
            symbol "@("
            a2 <- array
            symbol "+."
            a3 <- array
            symbol ")"
            pushArray id (addArr a2 a3)
            <|>
            do
            id <- identifier
            symbol "@("
            a2 <- array
            symbol "*."
            a3 <- array
            symbol ")"
            pushArray id ( mulArr a2 a3)
            <|>
            do
            id<-identifier
            symbol "@("
            ar1 <- array
            symbol "."
            ar2 <- array
            symbol ")"
            updateEnv Variable{name=id, vtype="Integer", value=dot ar1 ar2}
            <|>
            do
            id <- identifier
            symbol "@("
            ar2 <- array
            symbol "++"
            ar3 <- array
            symbol ")"
            pushArray id (ar2 ++ ar3)
```

These functions implement the following operations between arrays:

Concatenation (++), Dot Product (.), Sum of all elements of an array (sumall), Product of all elements of an array (mulall), Sum of all elements of two arrays PAIRWISE (+.), Product of all elements of two arrays PAIRWISE (*.)



```haskell
dot :: ArrayType -> ArrayType -> Int
dot x y = sum $ zipWith (*) x y


addArr :: ArrayType -> ArrayType -> ArrayType
addArr = zipWith (+)


mulArr :: ArrayType -> ArrayType -> ArrayType
mulArr = zipWith (*)

sumAllElemArr :: ArrayType -> Int
sumAllElemArr = foldl' (+) 0

mulAllElemArr :: ArrayType -> Int
mulAllElemArr = foldl' (*) 1

toString :: Bool -> String
toString x = if x then "True" else "False"

trim :: String -> String
trim = filter (not.isSpace)
```

15 – STORING VARIABLES BETWEEN INSTANCES

As the Environment, that is never discarded, but passed over as the initial state of further parsings, variables are stored in the same instance of IMP TER; is it always possible to force an empty environment for the next execution using clear (:c);