UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

DIPARTIMENTO DI INFORMATICA

Master's Degree in Computer Science

Formal Methods for Computer Science

FDInt

Student

Francesco Didio (**765767**)

Professor

Giovanni Pani

Summary

FDInt is a simple imperative programming language, implemented by Francesco Didio for the laboratory part of the course named “Formal Methods in Computer Science”.
The aim of this project was built an interpreter for an Imperative Language. An Interpreter is a program that directly executes instruction written by a programming language, without previously compile into a machine language.
FDInt interpreter used the **Eager-evaluation** strategy in which an expression is evaluated first executing the inner function and then the most external. This evaluation is called “call-by-value”.

In this interpreter the types used are:

- **Int**: Integer numbers that can have positive or negative value.
- **Bool**: Boolean values that can be True or False.
- **Array**: Data structure that contains a group of elements, in specific Int.

The control structures and the operations used are:

- **Assignment**: procedure that consist in the assignment of a value/arithmetic expression to a variable.
- **IfThenElse**: it consists in a conditional block in which if a Boolean condition is True then execute a specific operation, instead if the condition is False it execute another operation.
- **WhileDo**: it consists in a loop block. It can be seen as IfThenElse block repeated number of time as we want.
- **Ternary**: same thing of IfThenElse whit a simpler syntax.
- **Skip**: it perform a jump operation and does nothing.

In the next chapters we will see the definition of this arguments:

- **Grammar**: Arithmetic expression, Boolean expression and command using BNF.

- **Environment**: definition of what is an Environment and how it is defined.
- **Parser Implementation**: Definition of what means parsing and how in the interpreter the various operation/commands were parsed.

1. Grammar

As we said first, for the Grammar was used the Backus-Naur Form. This shows the internal representation of the program that will be accepted by the interpreter. In particular we find Arithmetic Expression, Boolean Expression and Commands.

1.1 Preliminary definitions

- **<integer>** ::= -<natural> | <natural>
- **<natural>** ::= <digit> | <digit><natural>
- **<digit>** ::= 0..9
- **<identifier>** ::= <lower><alphaNum>
- **<alphaNum>** ::= <lower><alphaNum> | <upper><alphaNum> | <digit><alphaNum> | <upper> | <lower>
- **<lower>** ::=a-z
- **<upper>** ::= A-Z

1.2 Arithmetic Expression

- **<aexp>** ::= <aterm> '+' <aexp> | <aterm> '-' <aexp> | <aterm>
- <**aterm**> ::=<factor> '*' <aterm> | <factor> '/' <aterm> | <aterm>
- <**factor**> ::= <identifier> | <integer> | '(' <aexp> ')' | <identifier> '{' <aexp> '}'

1.3 Boolean Expression

- <**bexp**> ::= <bterm> 'OR' <bexp> | <bterm>
- <**bterm**> ::= <bfactor> 'AND' <bterm> | <bfactor>
- <**bfactor**> ::= 'False' | 'True' | '!'<bfactor> | '(' <bexp> ')' | <bcomparison>
- <**bcomparison**> ::= <aexp> '==' <aexp> | <aexp> '<=' <aexp> | <aexp> '>=' <aexp> |<aexp> '>' <aexp> | <aexp> '<' <aexp>

- <**bterm**> ::= 'False' | 'True' | '!'<bterm> | '(' <bexp>')' | <bcomparison>

1.4 Commands

- <**prog**> ::= <command> ‘;’ <prog> | <command>
- <**command**> ::= <assignment> | <IfThenElse> | <while> | 'skip'
- <**assignment**> ::= <identifier> '=' <aexp> ';' | <identifier> ‘{‘ <aexp> ‘}=’ <aexp> | <identifier> ‘=’ <array> | <identifier> ‘=’ <array> ‘++’ <array> | <identifier> ‘{‘ <aexp> ‘}=’ <identifier> ‘{‘ <aexp> ‘}’ | <identifier> ‘{‘ <identifier> ‘{‘ <aexp> ‘}’
- <**IfThenElse**> ::= 'if' <bexp> 'then' <program> 'else' <program> ‘endif’
- <**whiledo**> ::= 'while' <bexp> 'do' <program>
- <**ternary**>::= ‘(‘<bexp>’)’’?’<prog>’:’<prog>

1.5 Array

- <**concArray**> ::= <array> ‘++’ <concArray> | <array>
- <**array**> ::= ‘{‘ <arrayItems> ‘}’ | <identifier>
- <**arrayItems**> ::= <integer> ‘,’ <arrayItems> | <integer>

2. Parser

The **FDInt**, as we said first, is an interpreter written in Haskell, an Imperative language.

“A parser for things is a function from strings to lists of pairs of things and strings” <Graham Hutton>.

A Parser is the part of the program whit the purpose of building the intermediate representation tree whit all the expressions to interpretate, starting from a source code and creating its syntactical structure, according to the grammar.
The Parser checks if the syntax of the source code is correct, following the structure of the language. A Parser is able to recognize the Arithmetic, Boolean or Command expressions.
When a Parser work, it returns a list in which there are the part of the string that it has parsed and the rest of the string.
The typical implementation can be summarized as this:

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

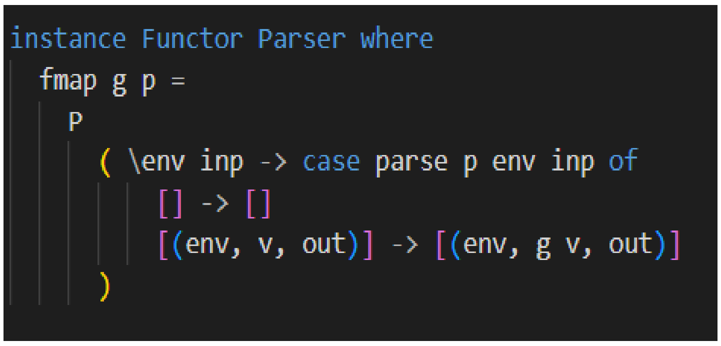
Instead, we need to handle an environment, so the definition became:

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

In order to combine parser in sequence we need to create an instance of the Functor, Applicative Monad and Alternative classes.

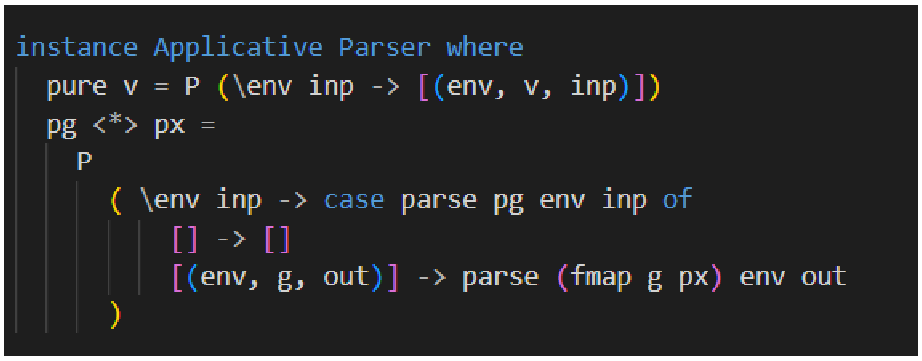
2.1 Functor

A Functor is a way to apply a function to a box or container. This class use the function **fmap** that takes in input a function and a box and returns another box whit inside the element after the function.


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

2.2 Applicative

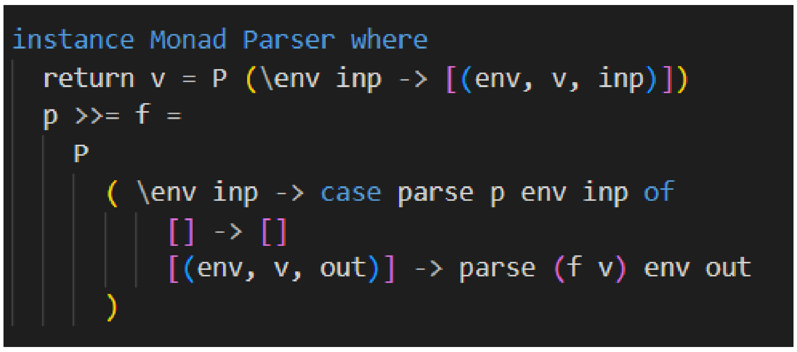
Applicative is an instance that can be implemented only if Functor is already implemented. In Applicative there are the definition of “pure”, that takes a generic element a and returns an applicative functor whit that value inside it and the operator <*> that allow the concatenation of different parsers.


```haskell
instance Applicative Parser where
  pure v = P (\env inp -> [(env, v, inp)])
  pg <*> px =
    P
      ( \env inp -> case parse pg env inp of
          [] -> []
          [(env, g, out)] -> parse (fmap g px) env out
      )
```

2.3 Monad

Monad are natural extension of Applicative Functors. They are used to apply a function that returns a wrapped value and to simulate the behavior of an Imperative language.

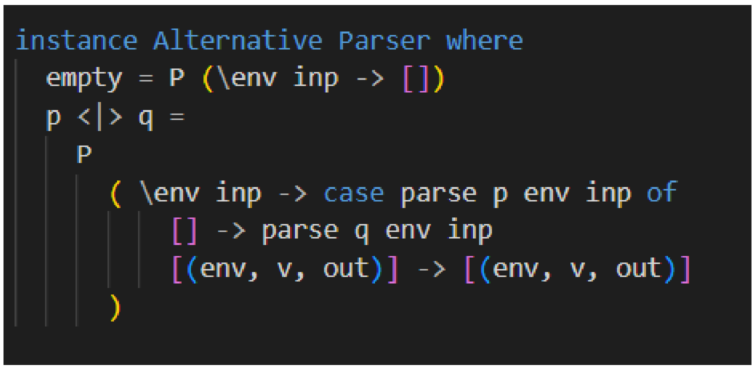
It works ad follow p>>f: if the application of the parser p to the input string inp returns a value v, it applies the function f to v, and then apply the parser f v to the original output string out of the first parser returning the final result.



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

2.4 **Alternative**

Alternative class allow the use of the operator <|> to simulate choices between parsers. More specifically, <|> returns the output of the first parser if it succeeds, otherwise it applies the same input to the second parser.



```haskell
instance Alternative Parser where
  empty = P (\env inp -> [])
  p <|> q =
    P
      ( \env inp -> case parse p env inp of
          [] -> parse q env inp
          [(env, v, out)] -> [(env, v, out)]
      )
```

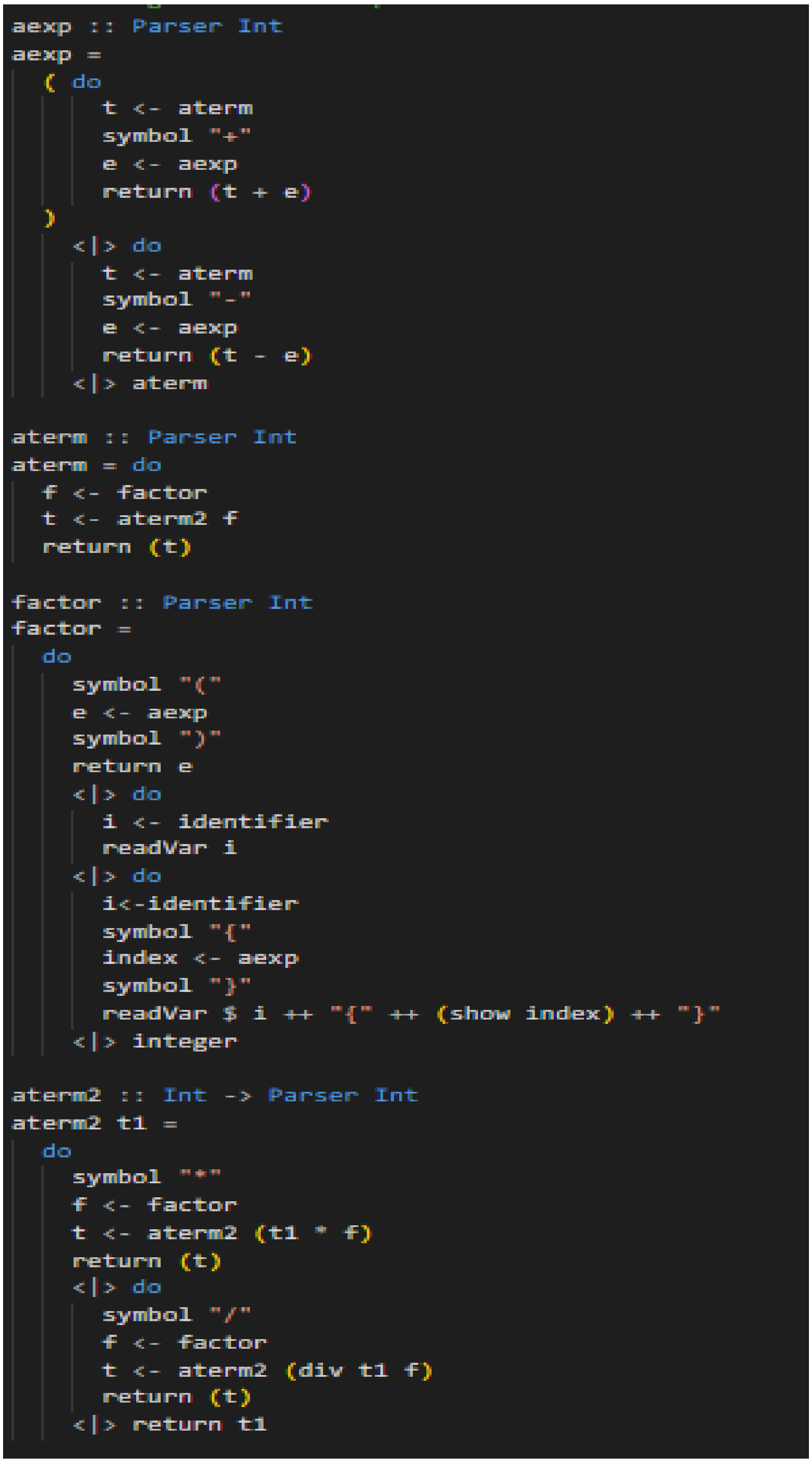
2.5 **Expression Parsing**

Now it will be shown the way in what the Parser detects the different kind of expression supported by the language.

2.5.1 **Arithmetic Expression**

The Parser for Arithmetic Expression is divided into three sub-parsers:

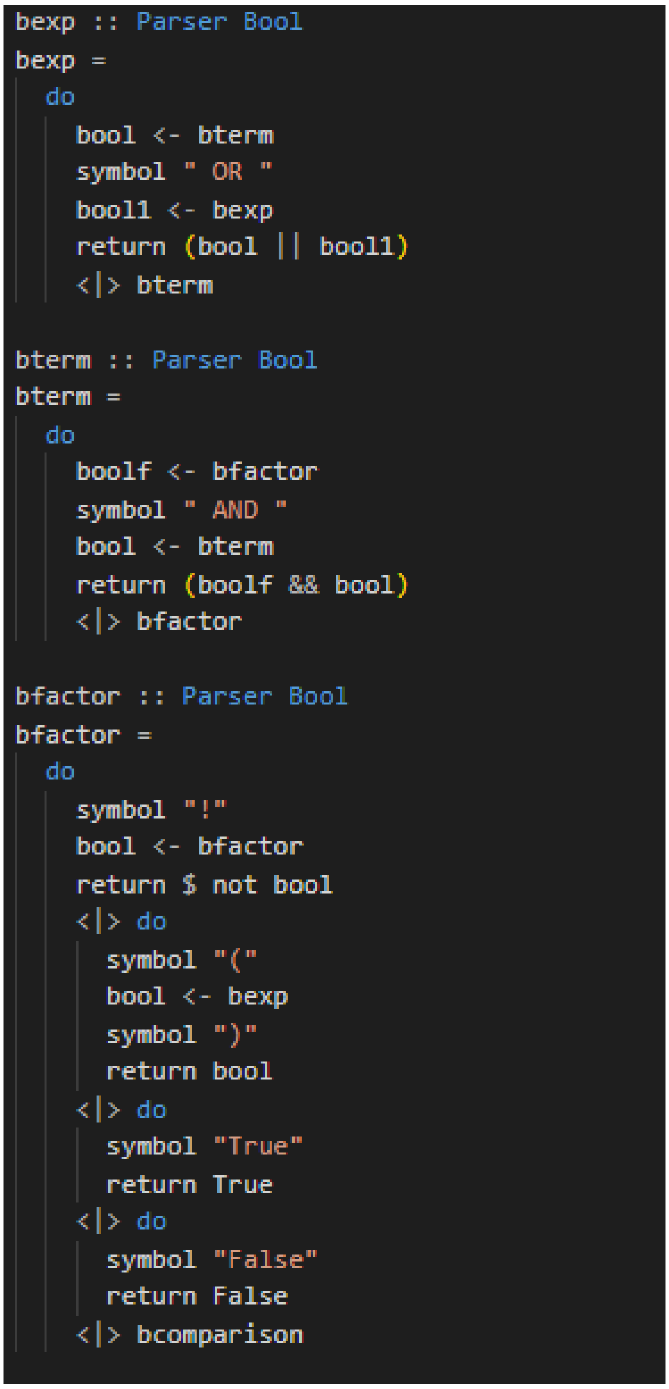
- Aexp: Is the main one, manage the addition and subtraction.
- Aterm: Manage the multiplication between afactor and aterm
- Afactor: Manage the recursion of Aexp and multiplication and divisions.



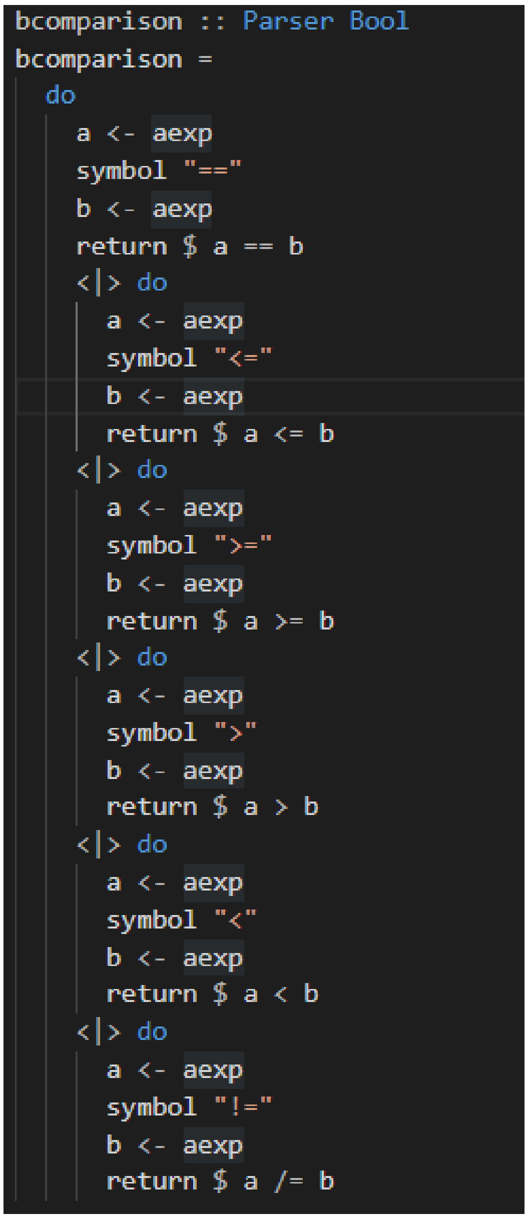
```haskell
aexp :: Parser Int
aexp =
  ( do
      t <- aterm
      symbol "+"
      e <- aexp
      return (t + e)
  )
    <|> do
      t <- aterm
      symbol "-"
      e <- aexp
      return (t - e)
    <|> aterm

aterm :: Parser Int
aterm = do
  f <- factor
  t <- aterm2 f
  return (t)

factor :: Parser Int
factor =
  do
    symbol "("
    e <- aexp
    symbol ")"
    return e
    <|> do
      i <- identifier
      readVar i
    <|> do
      i<-identifier
      symbol "{"
      index <- aexp
      symbol "}"
      readVar $ i ++ "{" ++ (show index) ++ "}"
    <|> integer

aterm2 :: Int -> Parser Int
aterm2 t1 =
  do
    symbol "*"
    f <- factor
    t <- aterm2 (t1 * f)
    return (t)
    <|> do
      symbol "/"
      f <- factor
      t <- aterm2 (div t1 f)
      return (t)
    <|> return t1
```

2.5.2 **Boolean Expression**

Same principle of Arithmetic Expression Parser. The operators that will be considered as first are always the round parenthesis, then all the Boolean operators like <,>,>=,<=,==,!=.



```haskell
bexp :: Parser Bool
bexp =
  do
    bool <- bterm
    symbol " OR "
    bool1 <- bexp
    return (bool || bool1)
    <|> bterm

bterm :: Parser Bool
bterm =
  do
    boolf <- bfactor
    symbol " AND "
    bool <- bterm
    return (boolf && bool)
    <|> bfactor

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



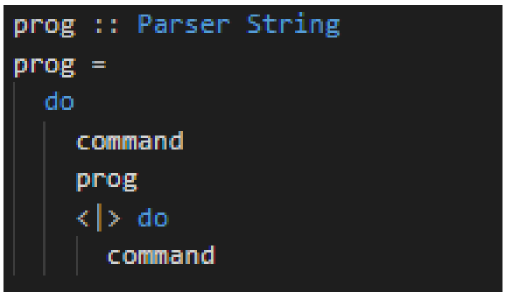
```haskell
bcomparison :: Parser Bool
bcomparison =
  do
    a <- aexp
    symbol "=="
    b <- aexp
    return $ a == b
    <|> do
      a <- aexp
      symbol "<="
      b <- aexp
      return $ a <= b
    <|> do
      a <- aexp
      symbol ">="
      b <- aexp
      return $ a >= b
    <|> do
      a <- aexp
      symbol ">"
      b <- aexp
      return $ a > b
    <|> do
      a <- aexp
      symbol "<"
      b <- aexp
      return $ a < b
    <|> do
      a <- aexp
      symbol "!="
      b <- aexp
      return $ a /= b
```

2.5.3 **Commands Parser**

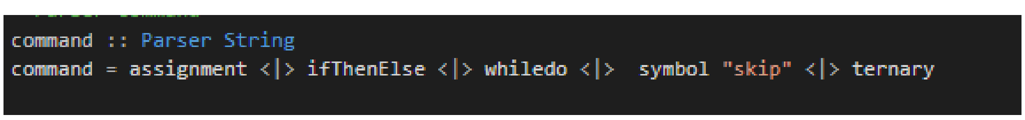
The Program Parser, named “prog” in the interpreter, is defined recursively as a sequence of commands, as we said first, that are respectively:

“Assignement, IfThenElse, Whiledo, Ternary and Skip”.

- Program “prog” and Command

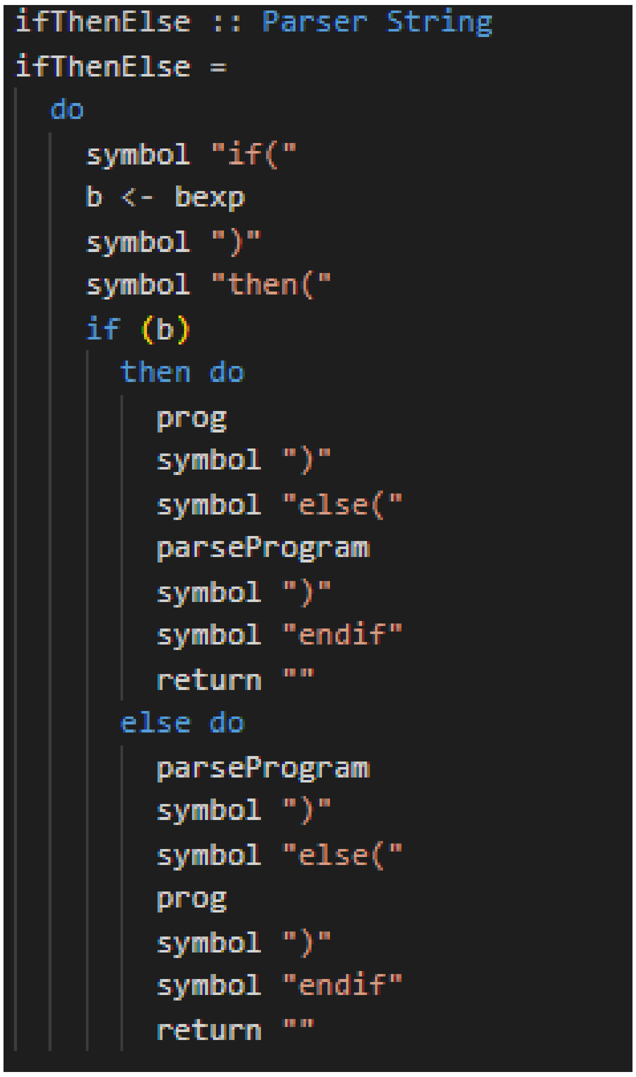


```
prog :: Parser String
prog =
  do
    command
    prog
    <|> do
      command
```



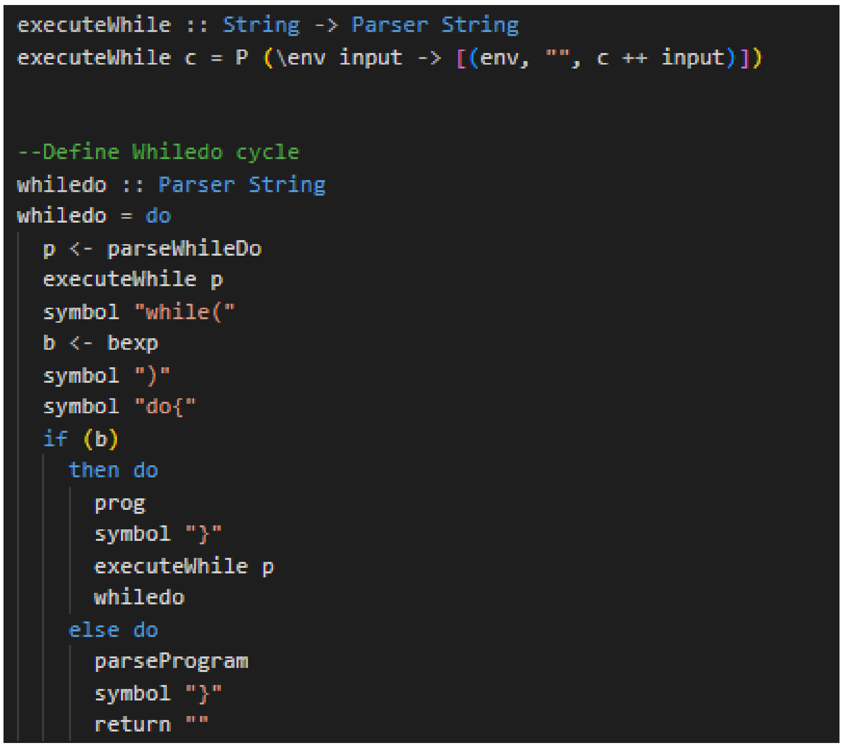
```
command :: Parser String
command = assignment <|> ifThenElse <|> whiledo <|>  symbol "skip" <|> ternary
```

- IfThenElse



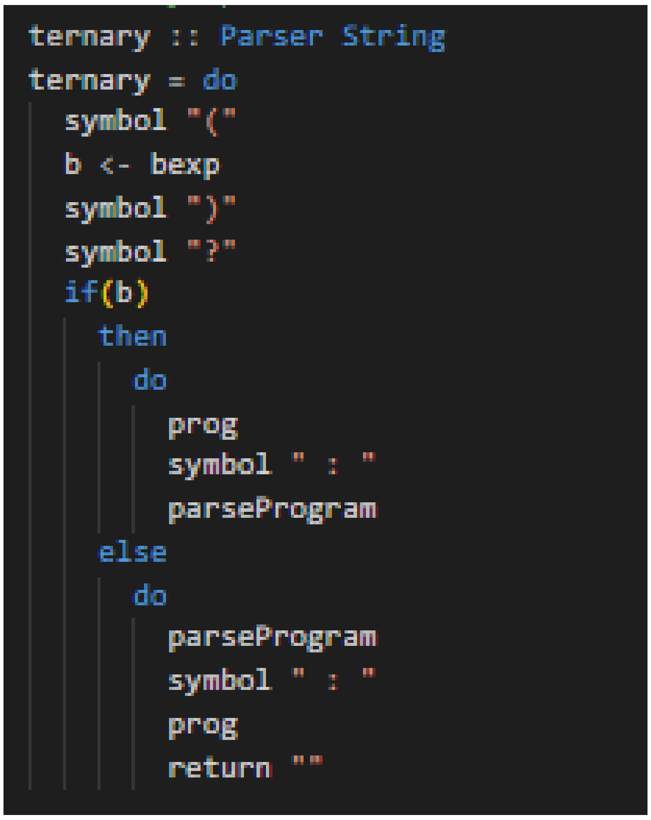
```
ifThenElse :: Parser String
ifThenElse =
  do
    symbol "if("
    b <- bexp
    symbol ")"
    symbol "then("
    if (b)
      then do
        prog
        symbol ")"
        symbol "else("
        parseProgram
        symbol ")"
        symbol "endif"
        return ""
      else do
        parseProgram
        symbol ")"
        symbol "else("
        prog
        symbol ")"
        symbol "endif"
        return ""
```

- Whiledo



```haskell
executeWhile :: String -> Parser String
executeWhile c = P (\env input -> [(env, "", c ++ input)])


--Define Whiledo cycle
whiledo :: Parser String
whiledo = do
  p <- parseWhileDo
  executeWhile p
  symbol "while("
  b <- bexp
  symbol ")"
  symbol "do{"
  if (b)
    then do
      prog
      symbol "}"
      executeWhile p
      whiledo
    else do
      parseProgram
      symbol "}"
      return ""
```

- Ternary operator



```haskell
ternary :: Parser String
ternary = do
  symbol "("
  b <- bexp
  symbol ")"
  symbol "?"
  if(b)
    then
      do
        prog
        symbol " : "
        parseProgram
    else
      do
        parseProgram
        symbol " : "
        prog
        return ""
```

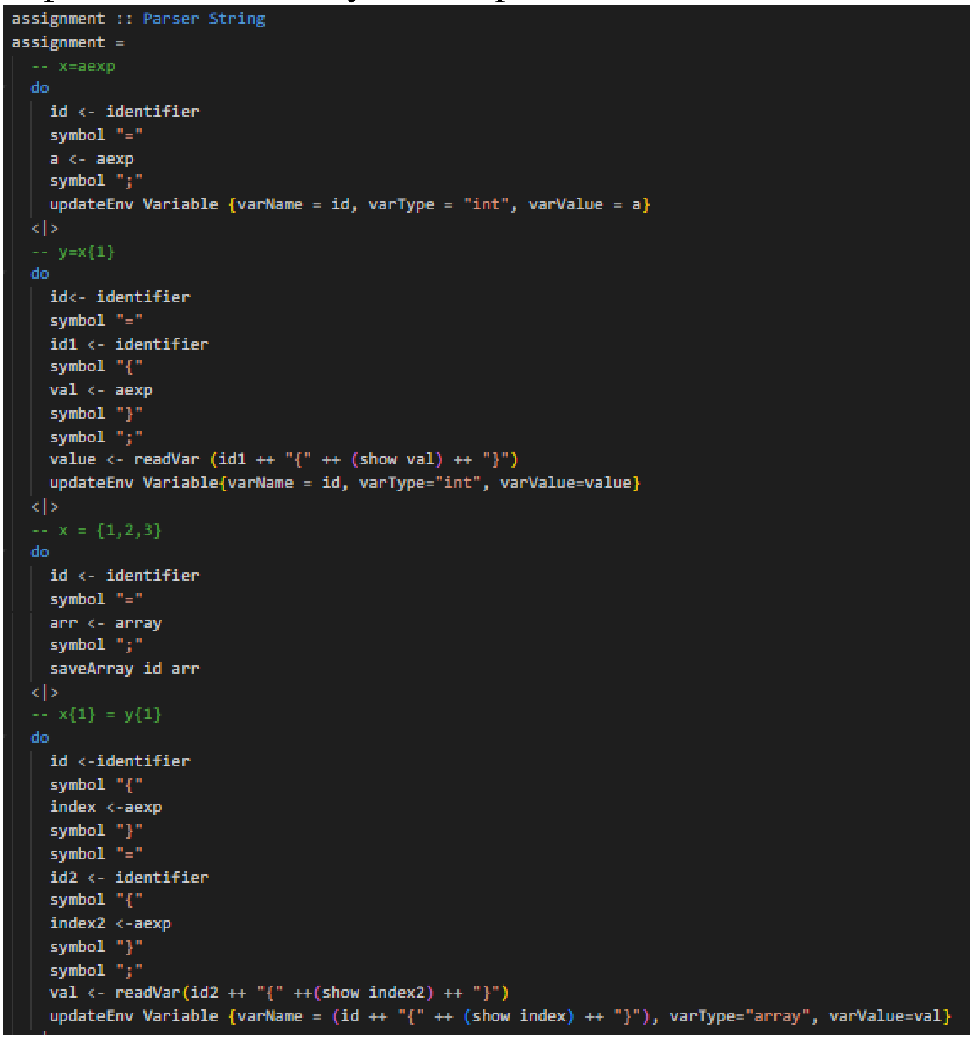
For the Assignment Parser we have to introduce first the structure used in this interpreter, the Array.

2.5.4 <u>**Arrays**</u>

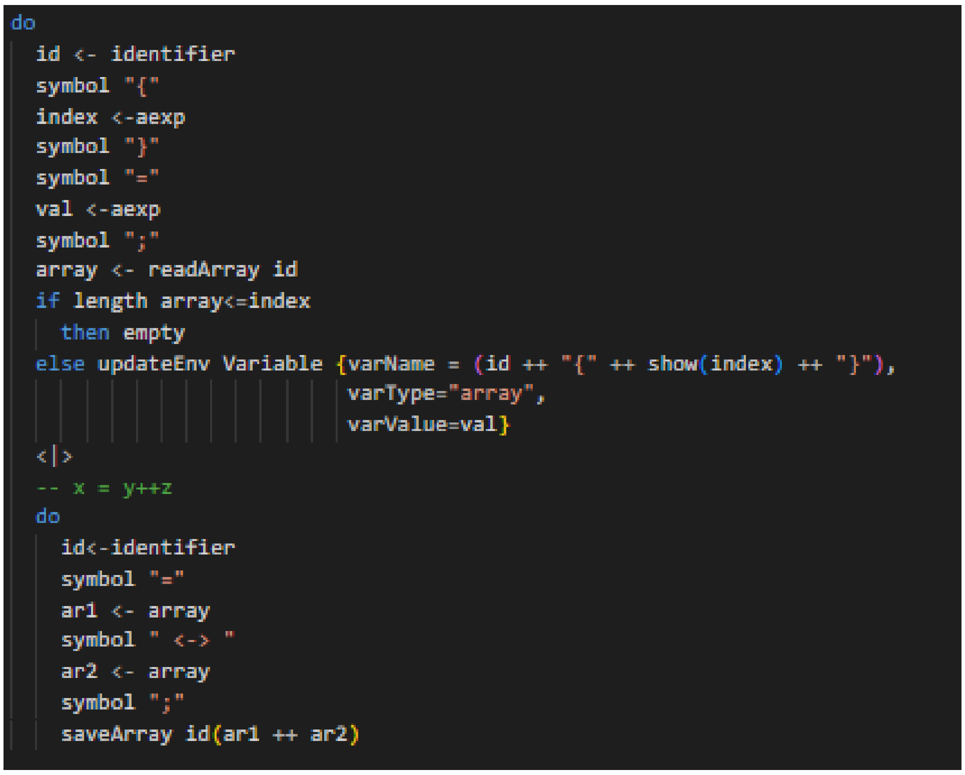
In **FDInt** was implemented the Array that manage a list of Integers. For management we say that we can do some manipulations:

- Create an array with "{}" brackets. Eg: x={1,2}
- Access through its index. Eg: y=x{0}
- Copy an array in a new variable. Eg: x={1,2};y=x;
- Change a specific value in a specific position. Eg: x={1,2};y={2,3};x{0}=y{0};
- Lastly, concatenate two arrays. Eg: x={1,2};y={3,4}z=x <-> y →{1,2,3,4}.

Now we can define Assignment Parser that consider Arithmetic expression and Array’s manipulations.



```
assignment :: Parser String
assignment =
  -- x=aexp
  do
    id <- identifier
    symbol "="
    a <- aexp
    symbol ";"
    updateEnv Variable {varName = id, varType = "int", varValue = a}
  <|>
  -- y=x{1}
  do
    id<- identifier
    symbol "="
    id1 <- identifier
    symbol "{"
    val <- aexp
    symbol "}"
    symbol ";"
    value <- readVar (id1 ++ "{" ++ (show val) ++ "}")
    updateEnv Variable{varName = id, varType="int", varValue=value}
  <|>
  -- x = {1,2,3}
  do
    id <- identifier
    symbol "="
    arr <- array
    symbol ";"
    saveArray id arr
  <|>
  -- x{1} = y{1}
  do
    id <-identifier
    symbol "{"
    index <-aexp
    symbol "}"
    symbol "="
    id2 <- identifier
    symbol "{"
    index2 <-aexp
    symbol "}"
    symbol ";"
    val <- readVar(id2 ++ "{" ++(show index2) ++ "}")
    updateEnv Variable {varName = (id ++ "{" ++ (show index) ++ "}"), varType="array", varValue=val}
```

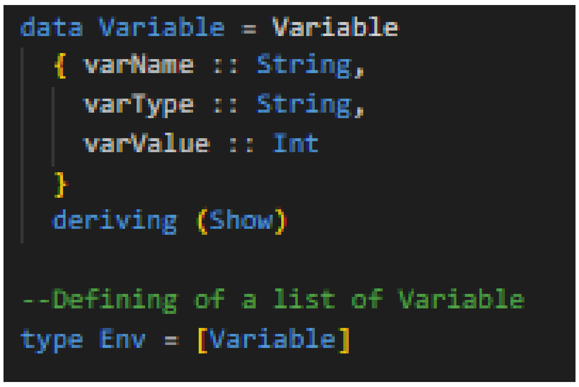


```
do
  id <- identifier
  symbol "{"
  index <-aexp
  symbol "}"
  symbol "="
  val <-aexp
  symbol ";"
  array <- readArray id
  if length array<=index
    then empty
  else updateEnv Variable {varName = (id ++ "{" ++ show(index) ++ "}"),
                           varType="array",
                           varValue=val}
  <|>
  -- x = y++z
  do
    id<-identifier
    symbol "="
    ar1 <- array
    symbol " <-> "
    ar2 <- array
    symbol ";"
    saveArray id(ar1 ++ ar2)
```

3. Environment

The interpreter, as we said before, represents the part of the program that decode the representation tree that has been built by the Parser. But to take into account of the parsed expression we need of an Environment.

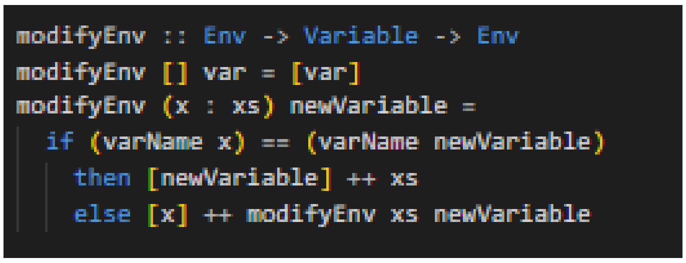
An Environment represents the memory of the program, that needs to be updated whit the program execution. It can be described as a list of elements of type Variable, represented as following:



```haskell
data Variable = Variable
  { varName :: String,
    varType :: String,
    varValue :: Int
  }
  deriving (Show)

--Defining of a list of Variable
type Env = [Variable]
```

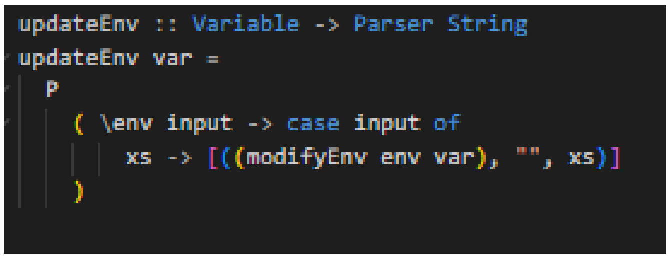
It is necessary, then, to write some functions that work with Environment and can read and write from it.

- ModifyEnv: take in input an Env and a Variable and gives back the Env with the new variable inserted.



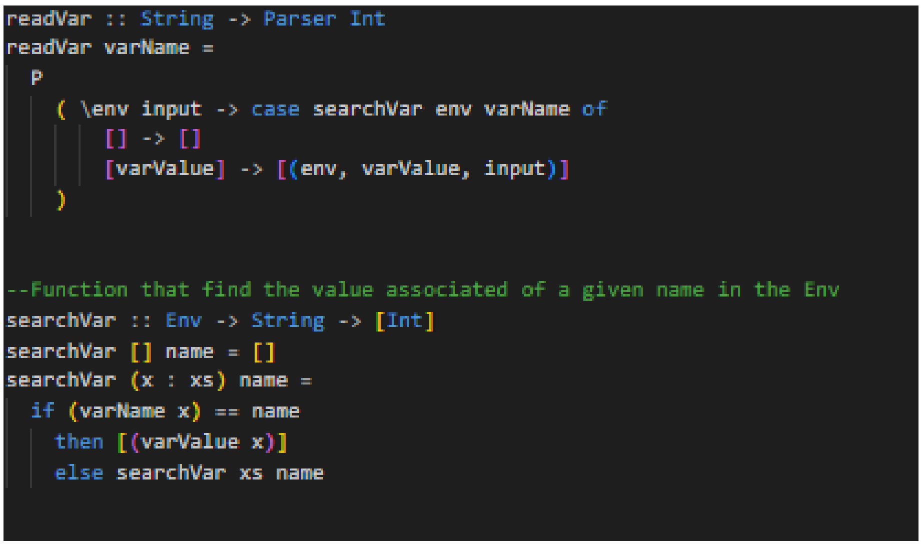
```haskell
modifyEnv :: Env -> Variable -> Env
modifyEnv [] var = [var]
modifyEnv (x : xs) newVariable =
  if (varName x) == (varName newVariable)
    then [newVariable] ++ xs
    else [x] ++ modifyEnv xs newVariable
```

- UpdateEnv: it needs the ModifyEnv function.



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

- SearchVariable and ReadVariable are necessary to find a Variable in the Environment given the name, and return its value if it’s find.

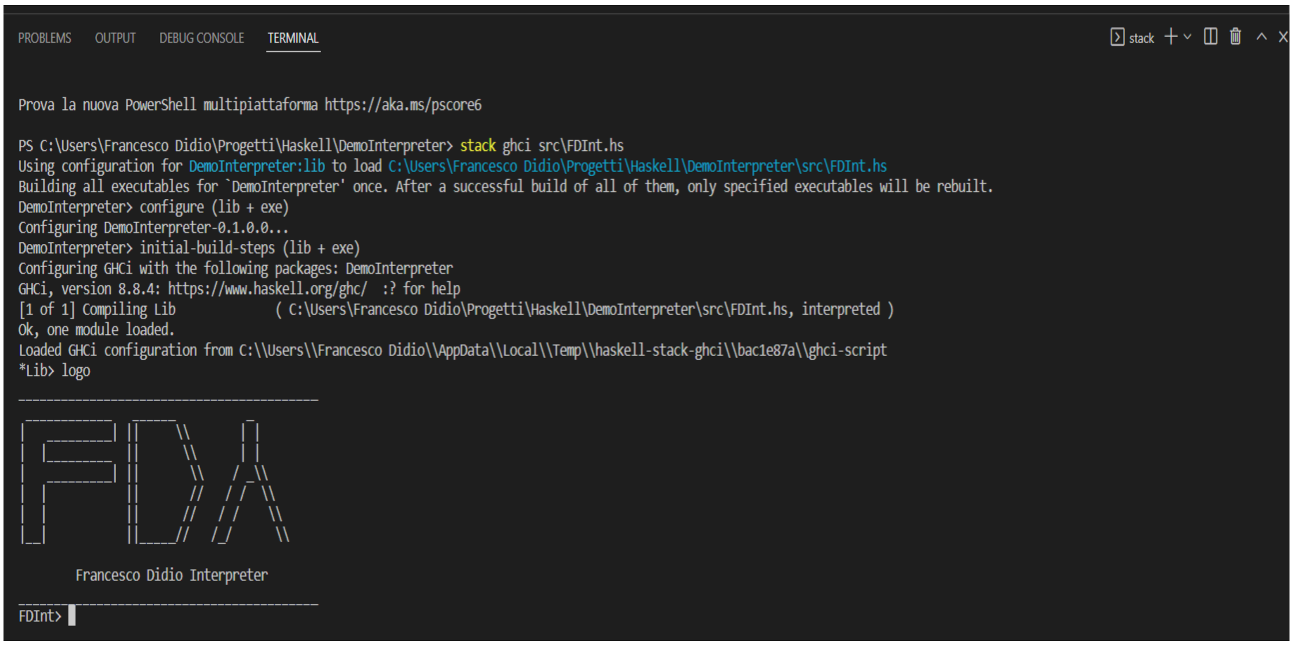


```haskell
readVar :: String -> Parser Int
readVar varName =
  P
    ( \env input -> case searchVar env varName of
        [] -> []
        [varValue] -> [(env, varValue, input)]
    )


--Function that find the value associated of a given name in the Env
searchVar :: Env -> String -> [Int]
searchVar [] name = []
searchVar (x : xs) name =
  if (varName x) == name
    then [(varValue x)]
    else searchVar xs name
```

4.FDInt Usage

The first thing to do to use **FDInt** is navigate to the FDInt.hs directory and use this command: stack ghci FDInt.hs.

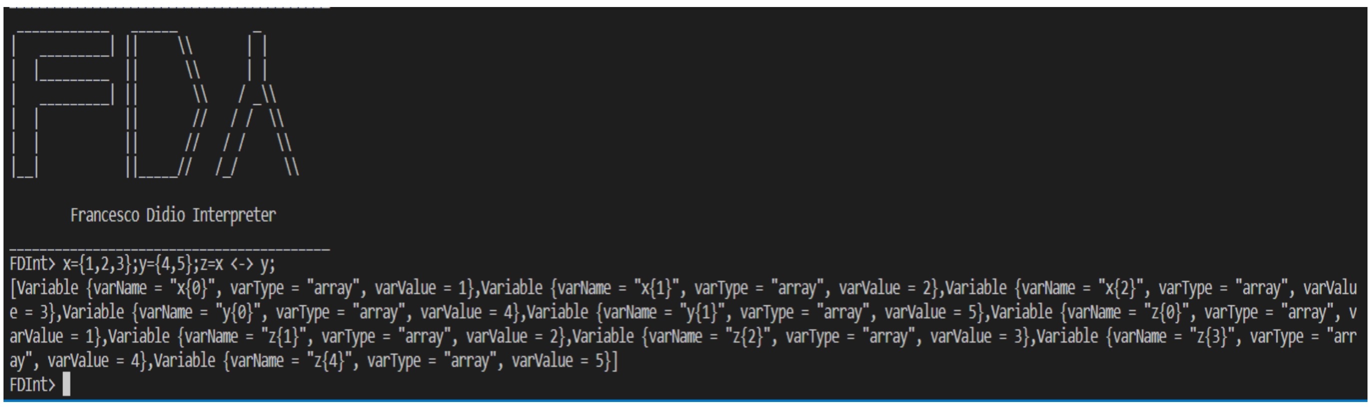
Now to start the program we need to write “logo” or “start” command. After this we will see this interface:



```
Prova la nuova PowerShell multipiattaforma https://aka.ms/pscore6

PS C:\Users\Francesco Didio\Progetti\Haskell\DemoInterpreter> stack ghci src\FDInt.hs
Using configuration for DemoInterpreter:lib to load C:\Users\Francesco Didio\Progetti\Haskell\DemoInterpreter\src\FDInt.hs
Building all executables for `DemoInterpreter' once. After a successful build of all of them, only specified executables will be rebuilt.
DemoInterpreter> configure (lib + exe)
Configuring DemoInterpreter-0.1.0.0...
DemoInterpreter> initial-build-steps (lib + exe)
Configuring GHCi with the following packages: DemoInterpreter
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Lib              ( C:\Users\Francesco Didio\Progetti\Haskell\DemoInterpreter\src\FDInt.hs, interpreted )
Ok, one module loaded.
Loaded GHCi configuration from C:\\Users\\Francesco Didio\\AppData\\Local\\Temp\\haskell-stack-ghci\\bac1e87a\\ghci-script
*Lib> logo

------------------------------------------
 ___________  ______       _
|  _________| ||    \\    | |
| |__________ ||     \\   | |
|  _________| ||      \\  / _\\
| |           ||      // / /  \\
| |           ||     // / /    \\
|__|          ||____//  /_/     \\

        Francesco Didio Interpreter

------------------------------------------
FDInt> 
```

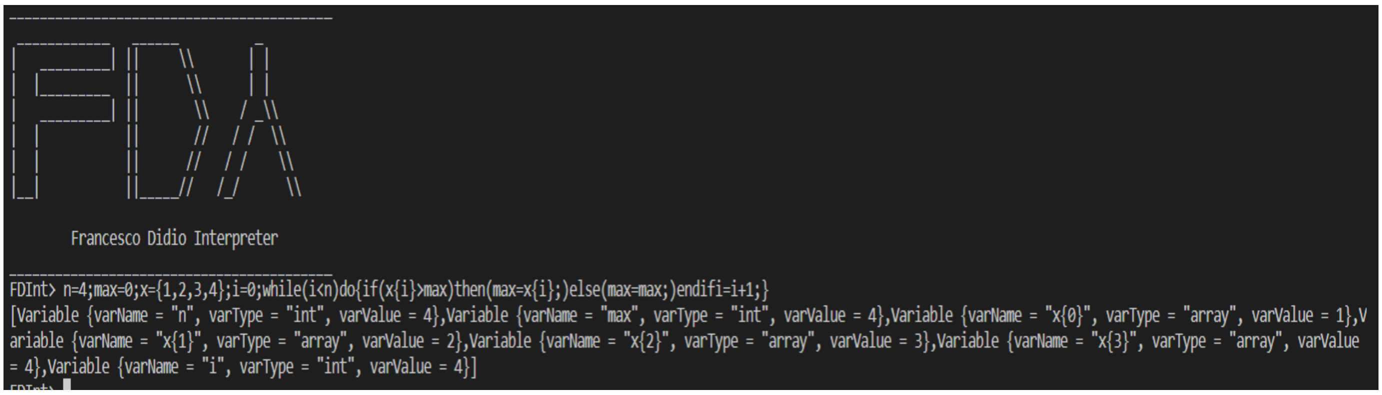
In the **FDInt**> we can write some commands. Let’s do some examples:

- Concatenate x={1,2,3}, y={4,5}→ z={1,2,3,4,5}



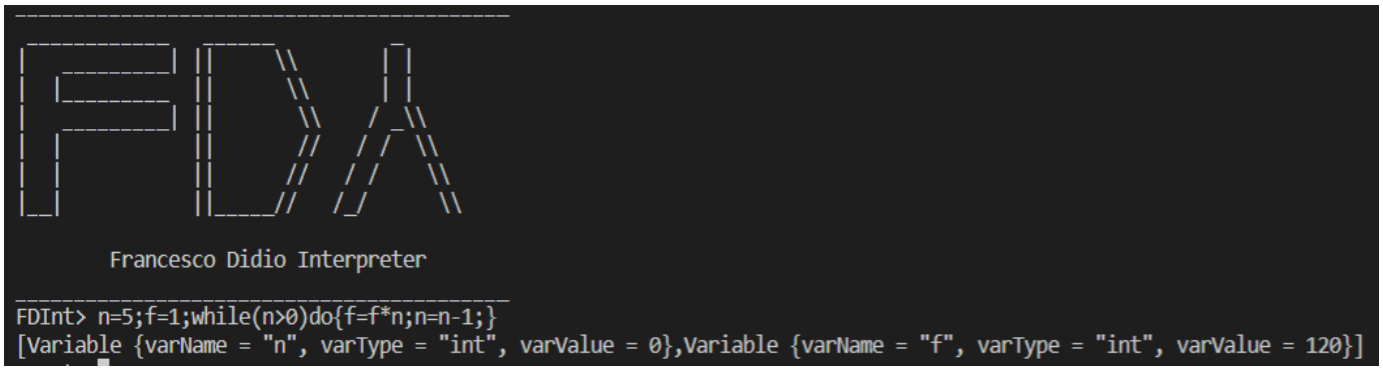
```
------------------------------------------
 ___________  ______       _
|  _________| ||    \\    | |
| |__________ ||     \\   | |
|  _________| ||      \\  / _\\
| |           ||      // / /  \\
| |           ||     // / /    \\
|__|          ||____//  /_/     \\

        Francesco Didio Interpreter

------------------------------------------
FDInt> x={1,2,3};y={4,5};z=x <-> y;
[Variable {varName = "x{0}", varType = "array", varValue = 1},Variable {varName = "x{1}", varType = "array", varValue = 2},Variable {varName = "x{2}", varType = "array", varValu
e = 3},Variable {varName = "y{0}", varType = "array", varValue = 4},Variable {varName = "y{1}", varType = "array", varValue = 5},Variable {varName = "z{0}", varType = "array", v
arValue = 1},Variable {varName = "z{1}", varType = "array", varValue = 2},Variable {varName = "z{2}", varType = "array", varValue = 3},Variable {varName = "z{3}", varType = "arr
ay", varValue = 4},Variable {varName = "z{4}", varType = "array", varValue = 5}]
FDInt> 
```

- Find max element in x={1,2,3,4}→ max = 4



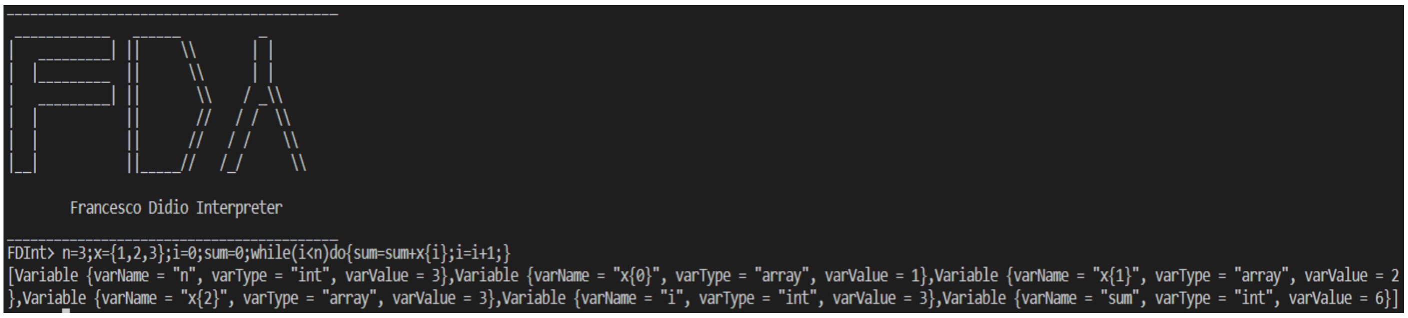
```
-----------------------------------------
 __________   _____         _
|  ________| ||    \\      | |
| |________  ||     \\     | |
|  ________| ||      \\   / _\\
| |          ||      //  / /  \\
| |          ||     //  / /    \\
|__|         ||____//  /_/      \\

        Francesco Didio Interpreter

-----------------------------------------
FDInt> n=4;max=0;x={1,2,3,4};i=0;while(i<n)do{if(x{i}>max)then(max=x{i};)else(max=max;)endifi=i+1;}
[Variable {varName = "n", varType = "int", varValue = 4},Variable {varName = "max", varType = "int", varValue = 4},Variable {varName = "x{0}", varType = "array", varValue = 1},V
ariable {varName = "x{1}", varType = "array", varValue = 2},Variable {varName = "x{2}", varType = "array", varValue = 3},Variable {varName = "x{3}", varType = "array", varValue
= 4},Variable {varName = "i", varType = "int", varValue = 4}]
FDInt>
```

- Factorial of 5 → 120



```
-----------------------------------------
 __________   _____         _
|  ________| ||    \\      | |
| |________  ||     \\     | |
|  ________| ||      \\   / _\\
| |          ||      //  / /  \\
| |          ||     //  / /    \\
|__|         ||____//  /_/      \\

        Francesco Didio Interpreter

-----------------------------------------
FDInt> n=5;f=1;while(n>0)do{f=f*n;n=n-1;}
[Variable {varName = "n", varType = "int", varValue = 0},Variable {varName = "f", varType = "int", varValue = 120}]
```

- Sum element of x={1,2,3} → 6



```
-----------------------------------------
 __________   _____         _
|  ________| ||    \\      | |
| |________  ||     \\     | |
|  ________| ||      \\   / _\\
| |          ||      //  / /  \\
| |          ||     //  / /    \\
|__|         ||____//  /_/      \\

        Francesco Didio Interpreter

-----------------------------------------
FDInt> n=3;x={1,2,3};i=0;sum=0;while(i<n)do{sum=sum+x{i};i=i+1;}
[Variable {varName = "n", varType = "int", varValue = 3},Variable {varName = "x{0}", varType = "array", varValue = 1},Variable {varName = "x{1}", varType = "array", varValue = 2
},Variable {varName = "x{2}", varType = "array", varValue = 3},Variable {varName = "i", varType = "int", varValue = 3},Variable {varName = "sum", varType = "int", varValue = 6}]
```