DIPARTIMENTO DI INFORMATICA

Formal Methods course - final assignment

byeSPACE

A Haskell interpreter for a C-like language

Professor
Giovanni Pani

Student
Giovanni Federico Poli
Mat: 766577

Academic year
2021-2022

Table of contents

1. Introduction

For this project has been build an interpreter based on the C-like language called CiMPLE. CiMPLE is a visual programming language that tries to visually mimic procedural languages like C/C++.

The idea is that C is a low-level programming language, thus whatever is expressible in the language is close in the resulting CPU instruction the compiler generates. CiMPLE stands for “C” “Simple”, meaning that is a simplified version of language C. It was in fact used to teach children robotics.

So, the focus of this project is to build an interpreter for this language (enriching it with the ability to handle arrays and the ternary operator), which is able to parse its instructions and execute them, always keeping track of an environment in which is saved every variable involved in the process.

CiMPLE allows the use of some typical constructs of imperative languages, such as assignment, if-then-else and while. More specifically, the language allows the following defined operations:

- **Arithmetic expressions**: a0 + a1 | a0 – a1 | a0 * a1 | a0 / a1

- **Boolean Expressions**: True | False | b0 < b1 | b0 == b1 | b0 > b1 | b0 >= b1<= b1 | b0 != b1 | !b0 | b0 && b1 | b0 || b1

- **Commands**: skip | X = a0 | if b0 then c0 else c1 | while b do c |

2. Grammar

Let’s now formally define the grammar of the language using the BNF.

Arithmetic Expressions:

<**aexp**> ::= <aterm> '+' <aexp> | <aterm> '-' <aexp> | <aterm>

<**aterm**> ::= <factor> '*' <aterm> | <factor> '/' <aterm> | <aterm>

<**factor**> ::= <identifier> | <integer> | '(' <aexp> ')' | <identifier> ‘{‘ <aexp> ‘}’

<**identifier**> ::= <lower> <alphaNum>

<**alphaNum**> ::= <lower> <alphaNum> | <upper> <alphaNum | <lower> | <upper>

<**lower**> ::= a-z

<**upper**> ::= A-Z

<**integer**> ::= '-' <natural> | <natural>

<**natural**> ::= <digit> <natural> | <digit>

<**digit**> ::= 0-9

<**concArray**> ::= <array> ‘++’ <array>

<**array**> ::= ‘{‘ <arrayItems> ‘}’ | <identifier>

<**arrayItems**> ::= <aexp> ‘,’ <arrayItems> | <aexp>

Boolean Expressions:

<**bexp**> ::= <bterm> 'OR' <bexp> | <bterm>

<**bterm**> ::= <bfactor> 'AND' <bterm> | <bfactor>

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

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

<**bcomparison**> ::= <aexp> '==' <aexp> | <aexp> '<=' <aexp> | <aexp> '>=' <aexp> |<aexp> '>' <aexp> |
<aexp> '<' <aexp>

Commands:

<**program**> ::= <command> ‘;’ <program> | <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;'

<**while**> ::= 'while' <bexp> 'do' <program> ‘end’ ';'

<**ternary**> ::= ‘(’ <bexp> ‘)’ ‘=’ <program> ‘:’ <program>

3. How the interpreter is structured

We can identify two main components in the system:
the parser and the evaluator.

The **Parser** that takes a string in input and syntactically validate it. The Parser analyzes each token of the input string and builds a tree that makes the syntactic structure of the string explicit. It is able therefore to recognize the Arithmetic or Boolean expressions and the Commands of the language.

The **Evaluator** has to execute the instructions based on the tree generated by the Parser. The evaluator thus must read and write the Environment, which is the set of local Variables necessary for the program, so it takes in input the formalized structure given by the Parser and returns an updated Environment.
The evaluator performs an EAGER evaluation (call-by-value), meaning that evaluates expressions in the exact moment in which they are assigned to a variable or when they are arguments of functions or other expressions.

Let's now talk about their implementation:

<u>Parser Implementation:</u>

First of all: what is a parser?

"A parser for things is a function from strings

to lists of pairs of things and strings"

At the most basic level, a parser takes a string in input and tries to recognize some kind of structure in the form of a tree.

Making some adjustments to the initial definition we can say that a Parser can be defined as a function that takes as input and produces a result of a generic type in output. In particular, it returns a list composed by the part of the string it has parsed or consumed and the rest of the string (which if it succeeds it's just nothing, but the parser does not always consume the entire given string, for this reason it's necessary to return the unconsumed part as output).

For our purposes though, we can't really define our parser like this:

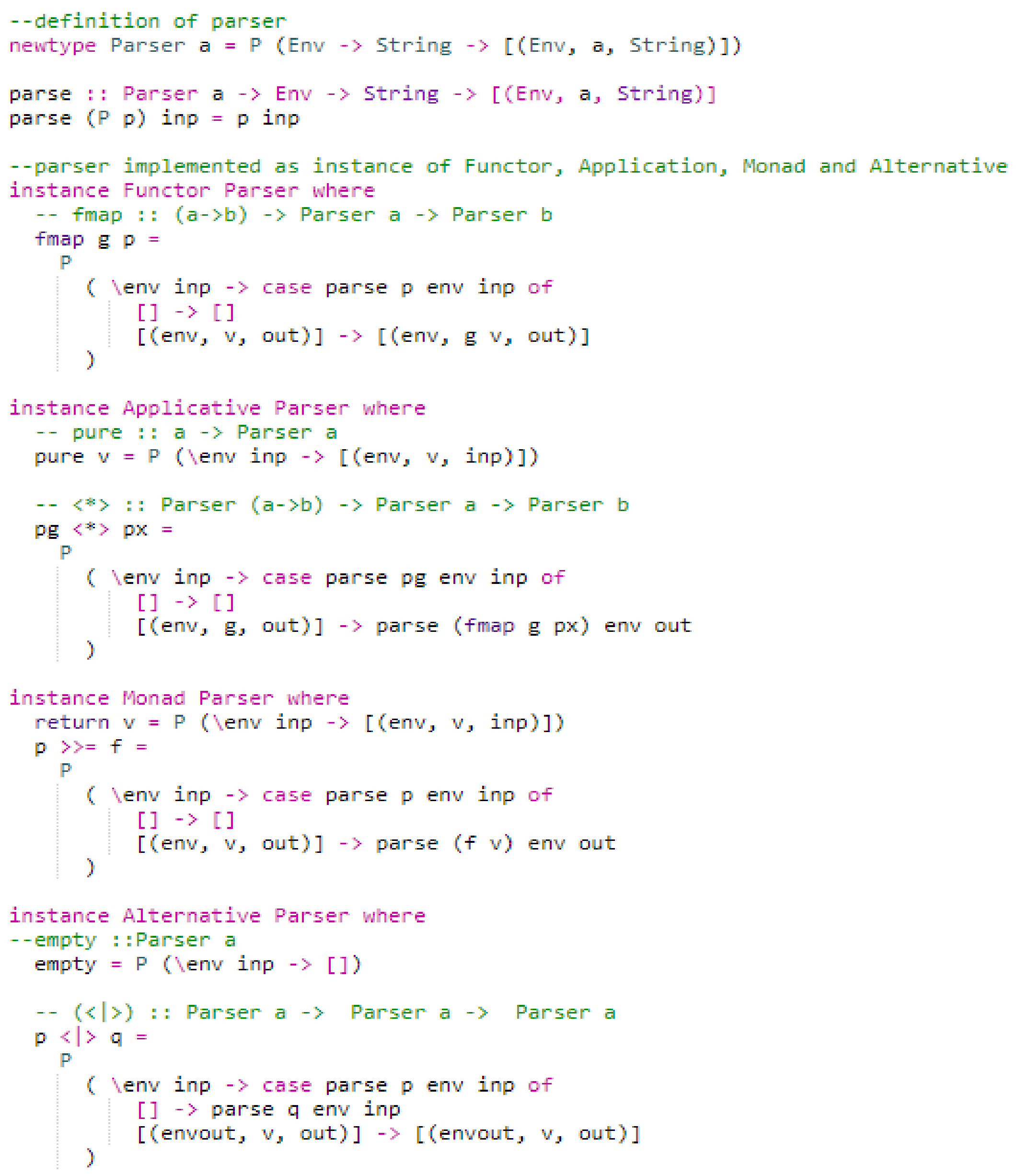
```
type Parser a = String -> [(a, string)]
```

and that's because we need to keep track of the changes that happen during the execution of our program. Thus, we have to further modify our definition by adding as input and output our Environment:
The final definition of the Parser is the following:

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

Using newtype allows us to make the type-constructor an instance of Functor, Monad, Alternative and Applicative classes in Haskell in order to take advantage of some useful operators.

• **Functor Parser** -> grants us the application of a function to the result of a Parser if it succeeds, otherwise propagating the failure.
• **Applicative Parser** -> allows the concat. of different parsers with the operator.
• **The Monad Parser** p `>>= f` :
if the application of the parser p to the input string inp returns a value v,
then it applies the function f to v, and then apply the parser f v to the original output string out and returns the result.
• **Alternative Parser** -> simulates choices between parsers; in particular:
if the first parser succeeds returns its output,
otherwise it applies the same input to the second parser.



```haskell
--definition of parser
newtype Parser a = P (Env -> String -> [(Env, a, String)])

parse :: Parser a -> Env -> String -> [(Env, a, String)]
parse (P p) inp = p inp

--parser implemented as instance of Functor, Application, Monad and Alternative
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)]
      )

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
      )

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
      )

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
          [(envout, v, out)] -> [(envout, v, out)]
      )
```

Environment Implementation:

We need to define our Environment in order to implement effectively our Parser, but what is an Environment?
Well, we cand imagine our Environment as the memory that needs to be updated along the program execution.
It can be described as a list of elements of type Variable (which contains it name, type and value).
So, it is necessary to define some functions in order to be able to read and write from/on the Environment, such as:

- `updateEnv` -> which is self-explanatory, it updates the current environment and if it finds a variable with the same name already in the Environment, it overrides it with the new value using the function `modifyEnv.`
- `searchVariable and readVariable ->` are useful to search a variable with a given name and return its value.

The same logic is also applied to arrays with the functions:

- `saveArray`
- `updateArray`
- `searchArray`
- `readArray`

With these functions we're enhancing the base version of the language, endowing it with some new functionalities useful in order to handle arrays.
In specific, our interpreter can manage arrays of integers defining them, assigning them some values, indexing them, and concatenating them together as follows:
Assignment:

```
x={1,2,3}
```

Once we assigned some values to the array variable, we can access its values using the index

```
y=x{1}
```

The index of an array can be the result of an Arithmetic expression:

```
x {1 + 1} = 2
```

the array can be copied into another variable:

```
y = x
```

and then we can also concatenate arrays:

```
z = x ++ y
```

For instance, if x={1,2} and y={3,4}, z = {1,2,3,4}

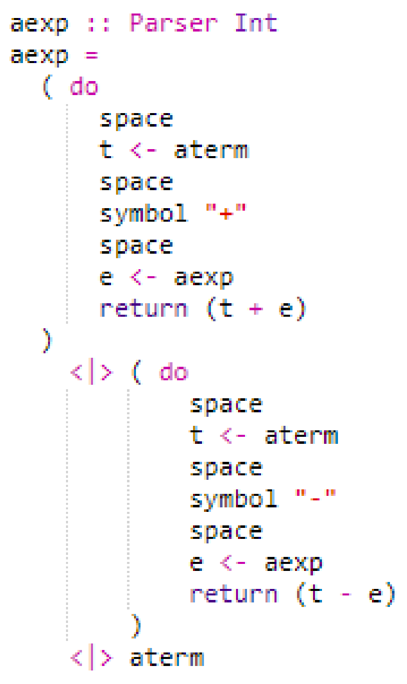
```haskell
--definition of environment as list of variable
data Variable = Variable
  { name :: String, --name of var
    vtype :: String, -- type of var
    value :: Int -- value of var, list of list of int for array and matrices
  }
  deriving (Show)

type Env = [Variable]

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

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

--simply reads a variable returning its value
readVariable :: String -> Parser Int
readVariable name =
  P ( \env input -> case searchVariable env name of
        [] -> []
        [value] -> [(env, value, input)])

--scans the environment in order to find a var with the same identifier
searchVariable :: Env -> String -> [Int]
searchVariable [] queryname = []
searchVariable (x : xs) queryname =
  if (name x) == queryname
    then [(value x)]
    else searchVariable xs queryname

-- @@@@@@@@@ ARRAY @@@@@@@@@

saveArray :: String -> [Int] -> Parser String
saveArray var val = P(\env input -> [(updateArray env var val, "", input)])

updateArray :: Env -> String -> [Int] -> Env
updateArray env var val = foldl (modifyEnv) env l
                            where l = zipWith (\a i ->
                                       Variable { name=var ++ "{" ++ (show i) ++ "}"
                                                , vtype="array"
                                                , value= a}) val [0..]

searchArray :: Env -> String -> [Int]
searchArray env array =
     case searchVariable env x of
          []    -> []
          value -> concat([value] ++ map (\var -> searchVariable env var) 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

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

4. Arithmetic and Boolean Parsers

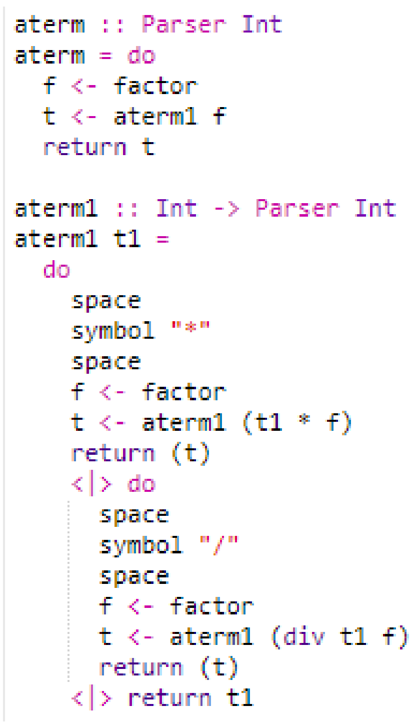
Note:
in the following code I've decided to give spaces the optionality property, (by which the name byeSPACE) meaning that:
it does not matter how many, if any, spaces you put into the input string of the interpreter, it will always be able to recognize it as long as it respects the correct syntax.

For these parsers we use the same logic, which implies a sub-division in 3 parsers, for instance in the Arithmetic Parser we have:

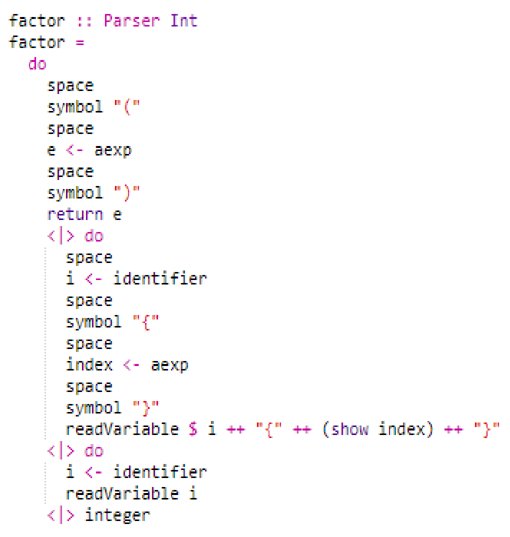
- `aexp ->` manages addition and subtraction
- `aterm ->` manages multiplication between a factor and an aterm
- `factor ->` manages the recursion



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



```
aterm :: Parser Int
aterm = do
  f <- factor
  t <- aterm1 f
  return t

aterm1 :: Int -> Parser Int
aterm1 t1 =
  do
    space
    symbol "*"
    space
    f <- factor
    t <- aterm1 (t1 * f)
    return (t)
    <|> do
      space
      symbol "/"
      space
      f <- factor
      t <- aterm1 (div t1 f)
      return (t)
    <|> return t1
```



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

The same can be said for the Boolean Parser, in which we have `bexp`, `bterm`, `bfactor` and also `bcomparison` that, as before, allow us to define a priority between operations:

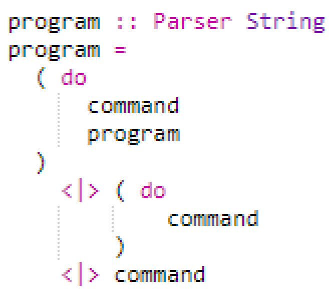
```haskell
bexp :: Parser Bool
bexp =
  ( do
      space
      bt <- bterm
      space
      symbol "OR"
      space
      b1 <- bexp
      return (bt || b1)
  )
    <|> bterm

bterm :: Parser Bool
bterm =
  ( do
      space
      bf <- bfactor
      space
      symbol "AND"
      space
      bt <- bterm
      return (bf && bt)
  )
    <|> bfactor

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

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

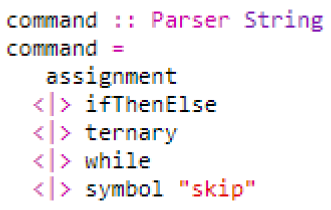
5. Commands

The program Parser is defined recursively as a sequence of specified commands, which are:

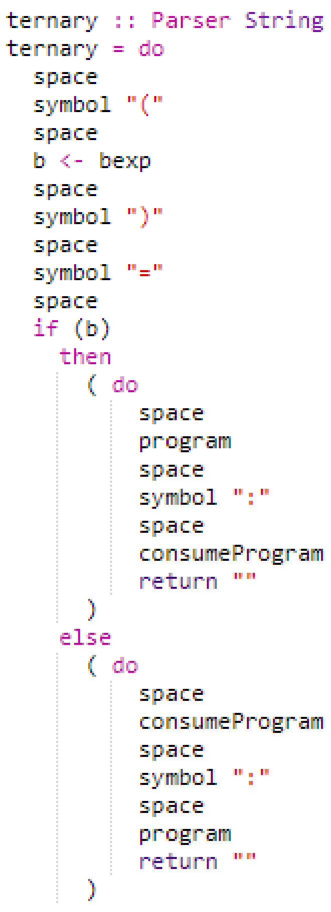
- **assignment**
- **ifThenElse** (in which can be specified only the “`if`” statement if there is no “`else`” statement)
- **while**
- **ternary**
- **skip**



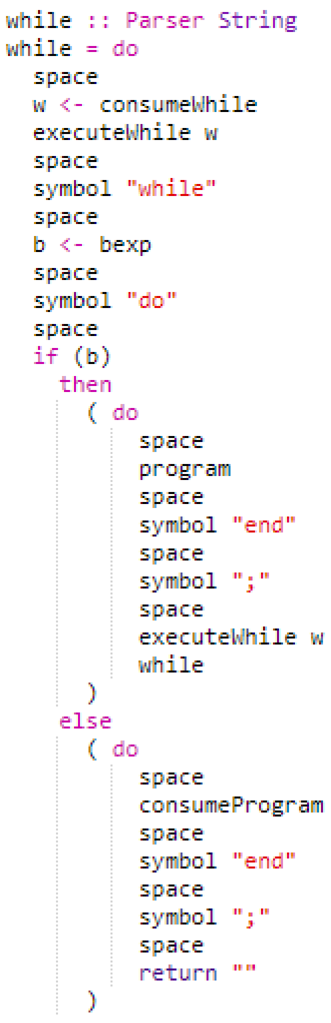
```
program :: Parser String
program =
  ( do
      command
      program
  )
   <|> ( do
           command
         )
   <|> command
```



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



```
ternary :: Parser String
ternary = do
  space
  symbol "("
  space
  b <- bexp
  space
  symbol ")"
  space
  symbol "="
  space
  if (b)
    then
      ( do
          space
          program
          space
          symbol ":"
          space
          consumeProgram
          return ""
      )
    else
      ( do
          space
          consumeProgram
          space
          symbol ":"
          space
          program
          return ""
      )
```



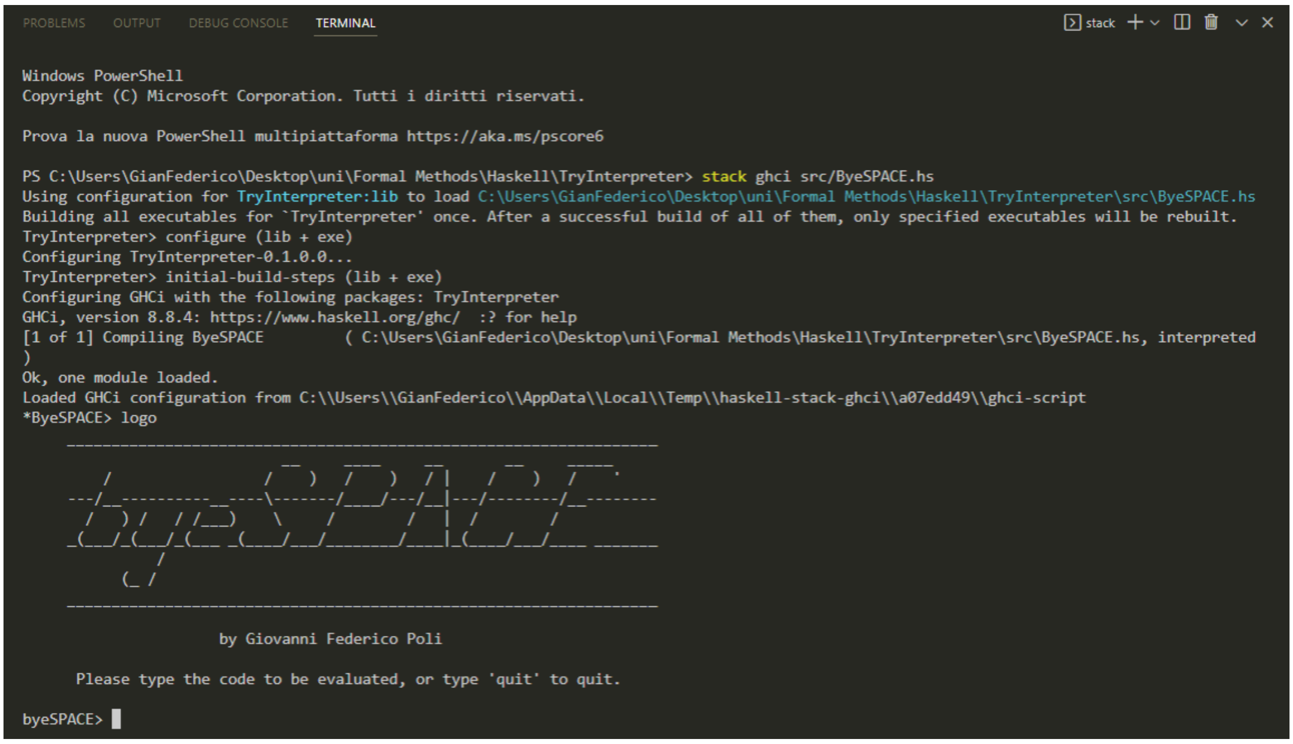
```
while :: Parser String
while = do
  space
  w <- consumeWhile
  executeWhile w
  space
  symbol "while"
  space
  b <- bexp
  space
  symbol "do"
  space
  if (b)
    then
      ( do
          space
          program
          space
          symbol "end"
          space
          symbol ";"
          space
          executeWhile w
          while
      )
    else
      ( do
          space
          consumeProgram
          space
          symbol "end"
          space
          symbol ";"
          space
          return ""
      )
```

6. Setup + Examples

byeSPACE has a command line interface that allows see the actual status of the memory (our Environment) and how it has changed. It needs ghci in order to work properly, so make sure it is installed and make also sure that the PATH of ghci is in the list of System Variables.

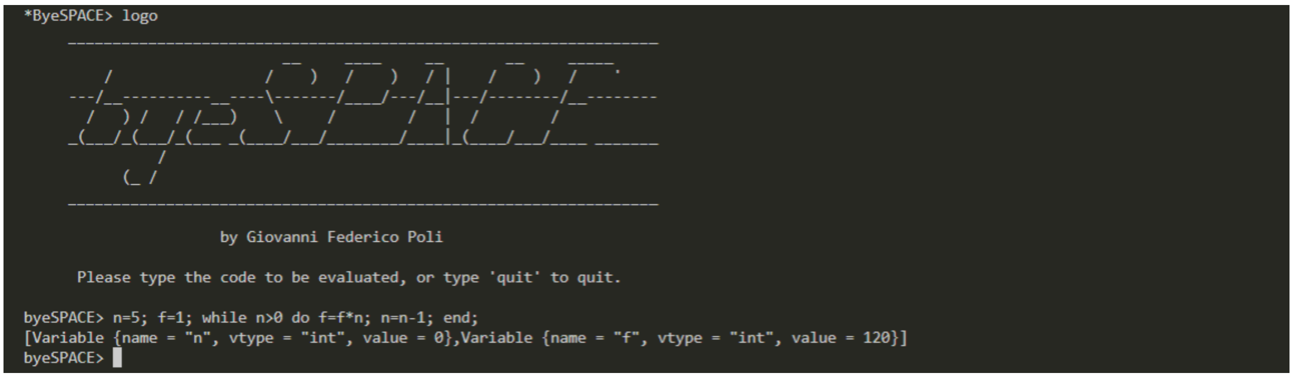
To run byeSPACE with ghci you can open the terminal and navigate to the byeSPACE.hs directory and then use the command: `stack ghci byeSPACE.hs`.

Then just call the function `logo` or `menu`, and then you can start writing your input string in which you can put as many spaces as you like.



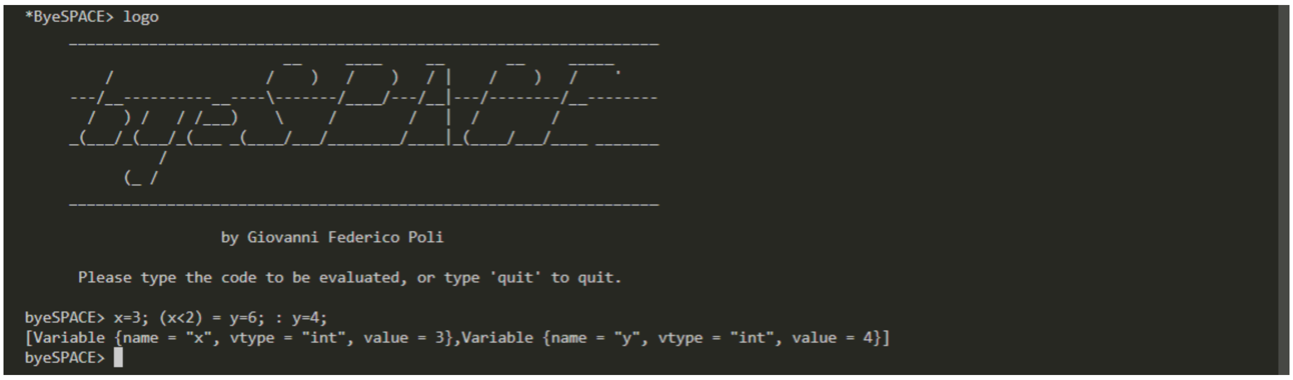
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

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

PS C:\Users\GianFederico\Desktop\uni\Formal Methods\Haskell\TryInterpreter> stack ghci src/ByeSPACE.hs
Using configuration for TryInterpreter:lib to load C:\Users\GianFederico\Desktop\uni\Formal Methods\Haskell\TryInterpreter\src\ByeSPACE.hs
Building all executables for `TryInterpreter' once. After a successful build of all of them, only specified executables will be rebuilt.
TryInterpreter> configure (lib + exe)
Configuring TryInterpreter-0.1.0.0...
TryInterpreter> initial-build-steps (lib + exe)
Configuring GHCi with the following packages: TryInterpreter
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling ByeSPACE         ( C:\Users\GianFederico\Desktop\uni\Formal Methods\Haskell\TryInterpreter\src\ByeSPACE.hs, interpreted
)
Ok, one module loaded.
Loaded GHCi configuration from C:\\Users\\GianFederico\\AppData\\Local\\Temp\\haskell-stack-ghci\\a07edd49\\ghci-script
*ByeSPACE> logo
     ------------------------------------------------------------------
                             __     ____     __       __     _____
         /                 /   )   /    )   / |    /    )   /    .
     ---/__----------__----\-------/____/---/__|---/--------/__--------
       /   ) /   / /___)    \     /        /   |  /        /
     _(___/_(___/_(___ _(____/___/________/____|_(____/___/____ _______
               /
           (_ /
     ------------------------------------------------------------------

                      by Giovanni Federico Poli

      Please type the code to be evaluated, or type 'quit' to quit.

byeSPACE> █
```

Let's try to calculate the factorial of 5:



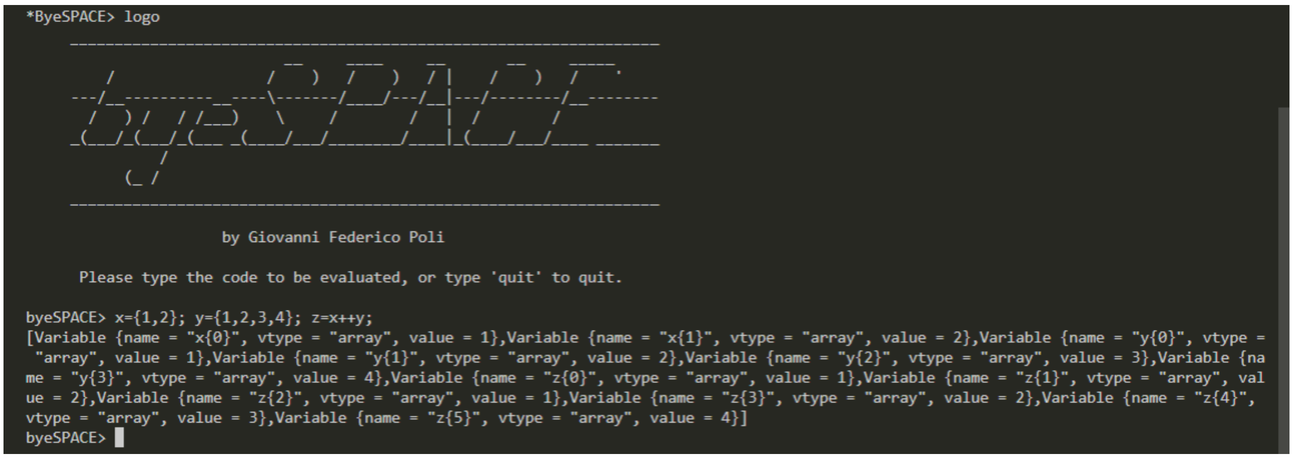
```
*ByeSPACE> logo
     -------------------------------------------------------------------
                                 __     ____     __      __     _____
         /                 /    )   /    )   / |    /    )   /    '
     ---/__----------__----\-------/____/---/__|---/--------/__---------
       /   ) /   / /___)    \     /        /   |  /        /
     _(___/_(___/_(___ _(____/___/________/____|_(____/___/____ ________
               /
           (_ /
     -------------------------------------------------------------------

                      by Giovanni Federico Poli

      Please type the code to be evaluated, or type 'quit' to quit.

byeSPACE> n=5; f=1; while n>0 do f=f*n; n=n-1; end;
[Variable {name = "n", vtype = "int", value = 0},Variable {name = "f", vtype = "int", value = 120}]
byeSPACE> █
```

Let’s try the ternary operator:



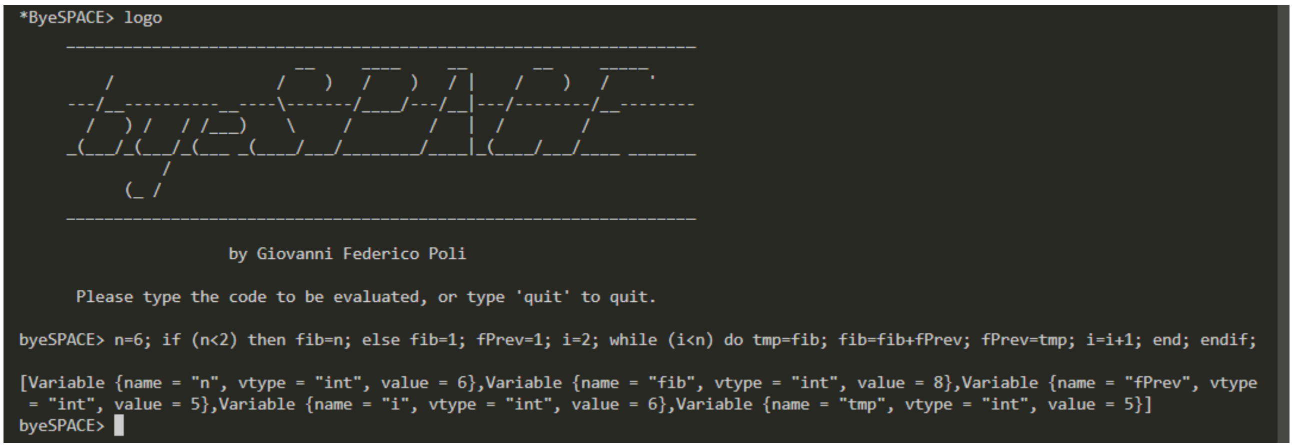
```
*ByeSPACE> logo
     -------------------------------------------------------------------
                                 __     ____     __      __     _____
         /                 /    )   /    )   / |    /    )   /    '
     ---/__----------__----\-------/____/---/__|---/--------/__---------
       /   ) /   / /___)    \     /        /   |  /        /
     _(___/_(___/_(___ _(____/___/________/____|_(____/___/____ ________
               /
           (_ /
     -------------------------------------------------------------------

                      by Giovanni Federico Poli

      Please type the code to be evaluated, or type 'quit' to quit.

byeSPACE> x=3; (x<2) = y=6; : y=4;
[Variable {name = "x", vtype = "int", value = 3},Variable {name = "y", vtype = "int", value = 4}]
byeSPACE> █
```

Let’s try some array assignments and concatenation:



```
*ByeSPACE> logo
     -------------------------------------------------------------------
                                 __     ____     __      __     _____
         /                 /    )   /    )   / |    /    )   /    '
     ---/__----------__----\-------/____/---/__|---/--------/__---------
       /   ) /   / /___)    \     /        /   |  /        /
     _(___/_(___/_(___ _(____/___/________/____|_(____/___/____ ________
               /
           (_ /
     -------------------------------------------------------------------

                      by Giovanni Federico Poli

      Please type the code to be evaluated, or type 'quit' to quit.

byeSPACE> x={1,2}; y={1,2,3,4}; z=x++y;
[Variable {name = "x{0}", vtype = "array", value = 1},Variable {name = "x{1}", vtype = "array", value = 2},Variable {name = "y{0}", vtype =
 "array", value = 1},Variable {name = "y{1}", vtype = "array", value = 2},Variable {name = "y{2}", vtype = "array", value = 3},Variable {na
me = "y{3}", vtype = "array", value = 4},Variable {name = "z{0}", vtype = "array", value = 1},Variable {name = "z{1}", vtype = "array", val
ue = 2},Variable {name = "z{2}", vtype = "array", value = 1},Variable {name = "z{3}", vtype = "array", value = 2},Variable {name = "z{4}",
vtype = "array", value = 3},Variable {name = "z{5}", vtype = "array", value = 4}]
byeSPACE> █
```

Let’s try to compute a Fibonacci sequence for n=6:



```
*ByeSPACE> logo
     ------------------------------------------------------------------
                             __     ____     __       __     _____.
         /                 /   )   /    )   / |     /   )   /     '
     ---/__----------__----\-------/____/---/__|---/--------/__--------
       /   ) /   / /___)    \     /        /   | /        /
     _(___/_(___/_(___ _(____/___/________/____|_(____/___/____ _______
               /
           (_ /
     ------------------------------------------------------------------

                      by Giovanni Federico Poli

      Please type the code to be evaluated, or type 'quit' to quit.

byeSPACE> n=6; if (n<2) then fib=n; else fib=1; fPrev=1; i=2; while (i<n) do tmp=fib; fib=fib+fPrev; fPrev=tmp; i=i+1; end; endif;

[Variable {name = "n", vtype = "int", value = 6},Variable {name = "fib", vtype = "int", value = 8},Variable {name = "fPrev", vtype
 = "int", value = 5},Variable {name = "i", vtype = "int", value = 6},Variable {name = "tmp", vtype = "int", value = 5}]
byeSPACE>
```