IMPure

A simple interpreter for the IMP language written in Haskell.
This parser-interpreter was realized for the course of "**Formal Method for Computer Science**" by Giuseppe Colavito.

IMP is a simple imperative language. It is composed by these basic structures:

- Assignment : assign a value to a variable
- If then else : if a boolean expression is true, then some instructions are executed. If not, some other instruction are executed
- While : loops executing the same command while a boolean condition is true
- Skip : does nothing

The IMPure interpreter uses eager evaluation strategy. To perform this kind of execution the interpreter uses the **call-by-value**.
The IMPure language can only accept variables of type Integer.

GRAMMAR:

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

command ::=   <aeVariableDeclaration> ";"
          |   <beVariableDeclaration> ";"
          |   <arVariableDeclaration> ";"
          |   <aeAssignment> ";"
          |   <beAssignment> ";"
          |   <arAssignment> ";"
          |   <assignment> ";"
          |   <ifThenElse> ";"
          |   <while> ";"
          |   <skip> ";"

aeVariableDeclaration ::=   "int" <identifier> "=" <aexp> ";"
beVariableDeclaration ::=   "bool" <identifier> "=" <bexp> ";"
arVariableDeclaration ::=   "array" <identifier> "=" <aexp> ";"
aeAssignment ::=    <identifier> "=" <aexp> ";"
beAssignment ::=    <identifier> "=" <bexp> ";"
arAssignment ::=    <identifier> "[ <aexp> "]" "=" <aexp>

ifThenElse ::=    "if" "("<bexp>")" "{" <program> "}"
            |     "if" "("<bexp>")" "{" <program>"}" "else" "{" <program> "}"

while ::=      "while" "("<bexp>")" "{" <program> "}"

skip ::=   "skip"

aexp ::=    <aterm> [{"+"|"-"} <aterm>]*

aterm ::=   <afact> ["*" <afact>]*

afact ::= <positiveterm> | <negativeterm> | "(" <aexp> ")"
        | <identifier> "[" <aexp> "]"

negativeterm ::=    "-" <positiveterm>

positiveterm ::=    <naturalnumber> | <identifier>

bexp ::=        <bterm> ["or" <bterm>]*

bterm ::=       <bfact> ["and" <bfact>]*

bfact ::=       <truthvalue> | "not" <bexp> | "(" <bexp> ")" | <comparison>
            |   <identifier>

comparison ::=  <aexp> <operator> <aexp>

truthvalue ::=    "True" | "False"

operator ::=    "<" | ">" | "==" | "<=" | ">=" | "!="
```

```
naturalnumber ::=   <digit> | <digit> <integer>

digit ::=     [0-9]*
identifier ::=    [a-zA-Z_][a-zA-Z_0-9]*
```

Design

The IMPure interpreter is splitted in two part:

- A parser
- An intepreter

The input file is passed to the parser, who creates an internal representation of the program.
The output of the parser is then passed to the interpreter that evaluates the program and updates the state of the memory that is empty at the start of the interpretation step. When the interpreter encounters a name of variable, he goes check into the state of the memory and uses the value of the variable.

Implementation

The environment (the state of the memory) is defined as a dictionary, where the value and the type associated with each name of variable that is declarated (and eventually updated) in the program are stored.

```
type Env = Dict String Type
```

The Type is defined as:

```
data Type = IntType Int | BoolType Bool | ArrayType [Int]
```

The arrays are defined only for integer values!

Environment Management

The environment must be kept updated within the execution of the program. For this purpose, the basic operation of the dictionary are used.

```haskell
newtype Dict key value = Dict [(key, value)]

--get the value for a given key
get :: (Eq key) => Dict key value -> key -> Maybe value
get (Dict []) _ = Nothing
get (Dict ((k, v) : ps)) key =
  if key == k then Just v else get (Dict ps) key

--insert into dictionary
insert :: (Eq key) => Dict key value -> key -> value -> Dict key value
insert (Dict []) key value = Dict [(key, value)]
insert (Dict ((k, v) : ps)) key value =
  if key == k then Dict ((key, value) : ps)
    else Dict ((k, v) : ds)
  where
    (Dict ds) = insert (Dict ps) key value
```

The interpreter operates on the internal representation of the program that is constructed from the code by the parser.

Internal structures

The internal structures used for this purpose are similar to the grammar's non-terminals :

```haskell
type Program = [Command]
```

The program is represented as a list of commands.

```haskell
data Command
  = AeVariableDeclaration String AExp
  | BeVariableDeclaration String BExp
  | ArVariableDeclaration String AExp
  | AeAssignment String AExp
  | BeAssignment String BExp
  | ArAssignment String AExp AExp
  | IfThenElse BExp [Command] [Command]
  | While BExp [Command]
  | Skip
```

The available commands are:

- Variable declaration, to declare a variable and assign to it a value,
- Assignment, to assign to a previously declared variable a new value,
- If-then-else, which executes the first list of commands if the boolean condition is true, otherwhise it executes the second list of commands,

- While, which executes the list of commands while the boolean condition is true,
- Skip, which goes to the next command without doing anything.

```
data AExp
  = Constant Int
  | AVariable String
  | AArray String AExp
  | Add AExp AExp
  | Sub AExp AExp
  | Mul AExp AExp
```

An arithmetic expression could be an integer constant, a name of variable, a value in an array or an operation between two arithmetic expressions.
The available operations on arithmetic expressions are addition, subtraction and multiplication.

```
data BExp
  = Boolean Bool
  | BVariable String
  | Not BExp
  | Or BExp BExp
  | And BExp BExp
  | Comparison AExp AExp Operator

data Operator
  = Lt
  | Le
  | Gt
  | Ge
  | Eq
  | Neq
```

A boolean expression could be a boolean constant, a boolean variable, an operation on boolean expression or a comparison between arithmetic expressions.
The available operations on boolean expressions are not, or and and.
The available comparison operators are less-then, less-equal, greater-then, greater-equal, equal, not-equal.

Interpreter Implementation

To implement all of the constructs of the IMP language, the interpreter will have to evaluate arithmetic expressions, boolean expressions and the commands we talked about (eg. if, while, ...).
The results of the evaluation of the interpreter are wrapped in a *Maybe* type.
This way we can easly represent failures and use the `error` function to interrupt the execution of the interpreter. We can also have a string in input to better understand which was the cause of the error!

Since the Functor and Applicative are already implemented for the Maybe type, we can easly unwrap the results, do some operation on them and then wrap them again and return it.

Arithmetic expression evaluation

The interpreter can evaluate an arithmetic expression given an environment and an *AExp* (that is defined in the internal structures). The output can be a `Just Int` or an `Nothing`. This evaluation is implemented using Functor(<$>) and Applicative (<*>).

```
aexpEval :: Env -> AExp -> Maybe Int
aexpEval _ (Constant i) = Just i
aexpEval e (AVariable s) =
  case get e s of
    Just (IntType v) -> Just v
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
aexpEval e (AArray s i) =
  case get e s of
    Just (ArrayType a) -> Just (readArray a j)
      where Just j = aexpEval e i
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
aexpEval e (Add a b) = (+) <$> aexpEval e a <*> aexpEval e b --Applicative
aexpEval e (Sub a b) = (-) <$> aexpEval e a <*> aexpEval e b
aexpEval e (Mul a b) = (*) <$> aexpEval e a <*> aexpEval e b
```

Boolean expression evaluation

The interpreter can evaluate a boolean expression given an environment and a *BExp* (that is defined in the internal structures). The output can be a `Just Bool` or an `Nothing`. This evaluation is implemented again using Functor(`<$>`) and Applicative (`<*>`).

```
bexpEval :: Env -> BExp -> Maybe Bool
bexpEval _ (Boolean b) = Just b
bexpEval e (BVariable s) =
  case get e s of
    Just (BoolType v) -> Just v
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
bexpEval e (Not b) = not <$> bexpEval e b --Functor
bexpEval e (Or a b) = (||) <$> bexpEval e a <*> bexpEval e b --Applicative
bexpEval e (And a b) = (&&) <$> bexpEval e a <*> bexpEval e b
bexpEval e (Comparison a b op) = compEval e a b op

compEval :: Env -> AExp -> AExp -> Operator -> Maybe Bool
compEval e a b Lt = (<) <$> aexpEval e a <*> aexpEval e b
compEval e a b Le = (<=) <$> aexpEval e a <*> aexpEval e b
compEval e a b Gt = (>) <$> aexpEval e a <*> aexpEval e b
compEval e a b Ge = (>=) <$> aexpEval e a <*> aexpEval e b
compEval e a b Eq = (==) <$> aexpEval e a <*> aexpEval e b
compEval e a b Neq = (/=) <$> aexpEval e a <*> aexpEval e b
```

Commands Execution

Given an environment and a list of commands, we can execute the commands in the list (the program).

```
programExec :: Env -> [Command] -> Env
programExec e [] = e
programExec e (Skip : cs) = programExec e cs
programExec e ((AeVariableDeclaration s ex) : cs) =
  case aexpEval e ex of
    Just ex' -> case get e s of
      Just _ -> error "MultipleDeclaration"
      Nothing -> programExec (insert e s (IntType ex')) cs
    Nothing -> error "InvalidArithmeticExpression"
programExec e ((BeVariableDeclaration s ex) : cs) =
  case bexpEval e ex of
    Just ex' -> case get e s of
      Just _ -> error "MultipleDeclaration"
      Nothing -> programExec (insert e s (BoolType ex')) cs
    Nothing -> error "InvalidBooleanExpression"
programExec e ((ArVariableDeclaration s i) : cs) =
  case get e s of
    Just _ -> error "MultipleDeclaration"
    Nothing -> programExec (insert e s (ArrayType (declareArray j))) cs
    where Just j = aexpEval e i
programExec e ((AeAssignment s ex) : cs) =
  case get e s of
    Just (IntType _) -> programExec (insert e s (IntType ex')) cs
      where
        Just ex' = aexpEval e ex
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
programExec e ((BeAssignment s ex) : cs) =
  case get e s of
    Just (BoolType _) -> programExec (insert e s (BoolType ex')) cs
      where
        Just ex' = bexpEval e ex
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
programExec e ((ArAssignment s i ex) : cs) =
  case get e s of
    Just (ArrayType a) ->
    programExec (insert e s (ArrayType (writeArray a j ex'))) cs
      where
        Just ex' = aexpEval e ex
        Just j = aexpEval e i
    Just _ -> error "TypeMismatch"
    Nothing -> error "UndeclearedVariable"
programExec e ((IfThenElse b nc nc') : cs) =
  case bexpEval e b of
    Just True -> programExec e (nc ++ cs)
    Just False -> programExec e (nc' ++ cs)
    Nothing -> error "InvalidBooleanExpression"
programExec e ((While b c) : cs) =
  case bexpEval e b of
    Just True -> programExec e (c ++ [While b c] ++ cs)
```

```
    Just False -> programExec e cs
    Nothing -> error "InvalidBooleanExpression"
```

And here is our interpreter. If we give in input a program (written as the internal representation of the program of the interpreter), the interpreter will evaluate the program and give us in output the state of the memory at the end of the program!
If something goes wrong, the execution gets interrupted and some basic information about the error are shown.

This is how a legal input for the interpreter looks like. At the end of the computation, x will be the result of the factorial of 5.

```
[ VariableDeclaration "i" (Constant 1),
  VariableDeclaration "n" (Constant 5),
  VariableDeclaration "x" (Constant 1),
  While
    (Comparison (AVariable "i") (AVariable "n") Le)
    [ Assignment "x" (Mul (AVariable "x") (AVariable "i")),
      Assignment "i" (Add (AVariable "i") (Constant 1))
    ]
]
```

Of course this "language" is too hard to write and understand. For more complex programs it would be really heavy to read and write.
To avoid this problems we define a more friendly language and implement a parser that will transform the new language in the language of the interpreter.

Parser Implementation

Parser Type

First of all we define our type Parser:

```
newtype Parser a = P (String -> Maybe (a, String))
```

The parser contains a function from string to a `Maybe` couple `(a, String)`, where a is a parametrized type and the string is the part of the input string that is not parsed yet.
The type a is the type of value returned in case of correctly parsed input.
We use the `Maybe` just to have the value of `Nothing` that will mean to us that the parser failed to parse.

Functor, Applicative, Monad, Alternative

We want to build a chain of parsers that will parse all the code we give in input, and to do this in haskell we will need to implement some methods:

Functor

```
instance Functor Parser where
  fmap g (P p) =
    P
      ( \input -> case p input of
          Nothing -> Nothing
          Just (v, out) -> Just (g v, out)
      )
```

The Functor is usefull to apply a function to a value wrapped in a Parser.
Functor in Haskell is a kind of functional representation of different types which can be mapped over. A Functor is an inbuilt class and is defined this way:

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

Applicative

```
instance Applicative Parser where
  pure v = P (\input -> Just (v, input))
  (P pg) <*> px =
    P
      ( \input -> case pg input of
          Nothing -> Nothing
          Just (g, out) -> case fmap g px of
            (P p) -> p out
      )
```

The Applicative is used when we have a function wrapped in a Parser and a Parser. We want to apply the function in the first argument to the second argument of the applicative.
An Applicative Functor is a normal Functor with some extra features provided by the Applicative Type Class. It is a way to map a function which is defined inside a Functor with another Functor
The Applicative has this definition:

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

Monad

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

The Monad is used to apply a function that returns a wrapped parser to a wrapped parser.
Monads apply a function that returns a wrapped value to a wrapped value. Monads have a function >>= (called bind) to do this. The monad is defined this way:

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

In our case, we have a `Parser a` and a function `a -> m b`. With the monad we specify the behaviour of the program in the case we want to apply the function to a wrapped parser, even if the function accepts only the type `a`.
Thanks to the monad implementation we are able to use the return and do statements.

Alternative

```haskell
class Monad f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  many :: f a -> f [a]
  some :: f a -> f [a]
  many x = some x <|> pure []
  some x = (:) <$> x <*> many x
  chain :: f a -> f (a -> a -> a) -> f a
  chain p op = do a <- op; rest a
    where
      rest a = (do f <- op; b <- p; rest (f a b)) <|> return a

instance Alternative Maybe where
  empty = Nothing
  Nothing <|> my = my
  (Just x) <|> _ = Just x

instance Alternative Parser where
  empty = P (const Nothing)

  (P p) <|> (P q) =
    P
      ( \input -> case p input of
          Nothing -> q input
          Just (v, out) -> Just (v, out)
      )
```

It's useful to implement the class alternative that will allow us to concatenate more parsers and use some cool functions like many and some.
If we have two parsers P and Q, using the `<|>`, if the first fails we will get as output the output of the second parser, else we will get the output of the first. We also implement the chain operator which let us use the leftmost associative property of arithmetic expressions. Without the chain operator, we can have really messy results like:

```
>>> int n = 1+1-1-1
>>> "n" = 2
```

The chain method definition can be found on "Monadic Parsing in Haskell" by Grahm Hutton.

Arithmetic Expression Parsing

```
aexp :: Parser AExp
aexp = do chain aTerm o
  where
    o =
      (do symbol "+"; return Add)
        <|> do symbol "-"; return Sub


aTerm :: Parser AExp
aTerm = do chain aFactor o
    where o = do symbol "*"; return Mul


aFactor :: Parser AExp
aFactor =
  (Constant <$> integer)
    <|> do
      i <- identifier
      do
        symbol "["
        n <- aexp
        symbol "]"
        return (AArray i n)
        <|> return (AVariable i)
    <|> do
      symbol "("
      a <- aexp
      symbol ")"
      return a
```

The aexp Parser does all the parsing on the arithmetic expressions by using aTerm that uses aFactor. This two other parser are useful to ensure the precedence on the multiplication operation but also giving the higher precedence on the expressions written between round brackets.

Boolean Expression Parsing

```
bexp :: Parser BExp
bexp = chain bTerm o
  where o = do
            symbol "or"
            return Or


bTerm :: Parser BExp
bTerm = chain bFact o
  where o = do
            symbol "and"
            return And
```

```
bFact :: Parser BExp
bFact =
  do
    symbol "True"
    return (Boolean True)
    <|> do
      symbol "False"
      return (Boolean False)
    <|> do
      symbol "not"
      Not <$> bexp
    <|> do
      symbol "("
      b <- bexp
      symbol ")"
      return b
    <|> do comparison
    <|> (BVariable <$> identifier)
```

The bexp Parser does all the parsing on the boolean expressions by using bTerm that uses bFactor. This two other parser are useful to ensure the precedence on the and operation but also giving the higher precedence on the not operation and the expressions written between round brackets. The bexp parser also uses the comparison parser, which handles all the possible comparisons between arithmetic expressions.

```
comparison :: Parser BExp
comparison =
  do
    a1 <- aexp
    do
      symbol "<"
      a2 <- aexp
      return (Comparison a1 a2 Lt)
      <|> do
        symbol "<="
        a2 <- aexp
        return (Comparison a1 a2 Le)
      <|> do
        symbol ">"
        a2 <- aexp
        return (Comparison a1 a2 Gt)
      <|> do
        symbol ">="
        a2 <- aexp
        return (Comparison a1 a2 Ge)
      <|> do
        symbol "=="
        a2 <- aexp
        return (Comparison a1 a2 Eq)
      <|> do
        symbol "!="
        a2 <- aexp
        return (Comparison a1 a2 Neq)
```

Commands Parsing

```
command :: Parser Command
command =
  aeVariableDeclaration
    <|> beVariableDeclaration
    <|> arVariableDeclaration
    <|> aeAssignment
    <|> beAssignment
    <|> arAssignment
    <|> ifThenElse
    <|> while
    <|> skip
```

The command parser is basically an or between all the parsers of the possible commands.

```
aeVariableDeclaration :: Parser Command
aeVariableDeclaration =
  do
    symbol "int"
    i <- identifier
    symbol "="
    r <- AeVariableDeclaration i <$> aexp
    symbol ";"
    return r
```

This parser is used to parse the declaration of integer variables. They have the `int` prefix

```
beVariableDeclaration :: Parser Command
beVariableDeclaration =
  do
    symbol "bool"
    i <- identifier
    symbol "="
    r <- BeVariableDeclaration i <$> bexp
    symbol ";"
    return r
```

This parser is used to parse the declaration of boolean variables. They have the `bool` prefix

```
arVariableDeclaration :: Parser Command
arVariableDeclaration =
  do
    symbol "array"
    i <- identifier
    symbol "="
    j <- aexp
    symbol ";"
    return (ArVariableDeclaration i j)
```

This parser is used to parse the declaration of arrays of integers. They have the `array` prefix. The array is of fixed length, so we have to write `array x = 2` to have the array x of two integers. By default those values are set to 0.

It is not possible to declare a variable without assigning a value to it.

```
aeAssignment :: Parser Command
aeAssignment =
  do
    i <- identifier
    symbol "="
    r <- AeAssignment i <$> aexp
    symbol ";"
    return r


beAssignment :: Parser Command
beAssignment =
  do
    i <- identifier
    symbol "="
    r <- BeAssignment i <$> bexp
    symbol ";"
    return r


arAssignment :: Parser Command
arAssignment =
  do
    i <- identifier
    symbol "["
    j <- aexp
    symbol "]"
    symbol "="
    r <- ArAssignment i j <$> aexp
    symbol ";"
    return r
```

The assignment parser is like the variable declaration parser but without the prefixes. For the arrays we can assign a value to the i-th position of the array by using the square brackets: `x[2] = 1816`

```
skip :: Parser Command
skip =
  do
    symbol "skip"
    symbol ";"
    return Skip
```

The skip parser is probabily the most trivial parser!

```
ifThenElse :: Parser Command
ifThenElse =
  do
    symbol "if"
    symbol "("
    b <- bexp
    symbol ")"
    symbol "{"
    thenProgram <- program
    do
      symbol "}"
      symbol "else"
      symbol "{"
      elseProgram <- program
      symbol "}"
      return (IfThenElse b thenProgram elseProgram)
      <|> do
        symbol "}"
        return (IfThenElse b thenProgram [Skip])
```

The if-then-else parser parses the "if" and the boolean expression between round brackets and saves the programs in both branches, returning the internal representation of this construct. The interpeter will then decide which branch to execute, based on the boolean expression.

```
while :: Parser Command
while =
  do
    symbol "while"
    symbol "("
    b <- bexp
    symbol ")"
    symbol "{"
    p <- program
    symbol "}"
    return (While b p)
```

The while parser parses the "while" and the boolean expression between round brackets and saves the program inside the curl braces, returning the internal representation of this construct. The interpeter will then decide how many times the program inside the while will be executed, based on the boolean condition.

Program Parsing

```
program :: Parser [Command]
program =
  do many command
```

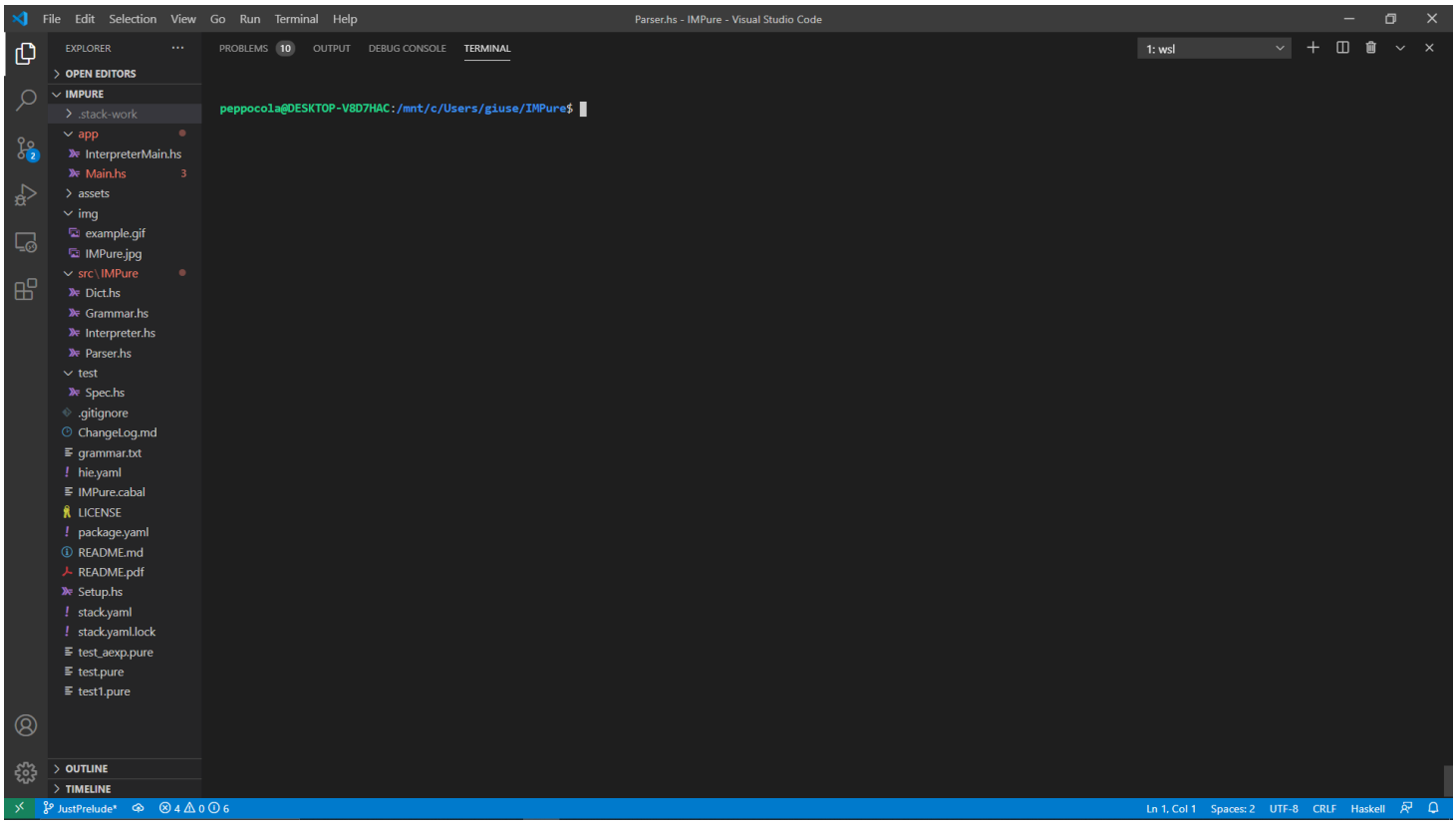
The program parser parses many commands, meaning that a program can be composed of 0 or potentially infinite commands.

```
parse :: String -> ([Command], String)
parse s = (first, second)
  where
    (P p) = program
    result = p s
    first = fst (head result)
    second = snd (head result)

parseFailed :: ([Command], String) -> Bool
parseFailed (_ , "") = False
parseFailed (_ , _) = True

getParsedCommands :: ([Command], String) -> [Command]
getParsedCommands (c , _) = c

getRemainingInput :: ([Command], String) -> String
getRemainingInput (_ , s) = s
```

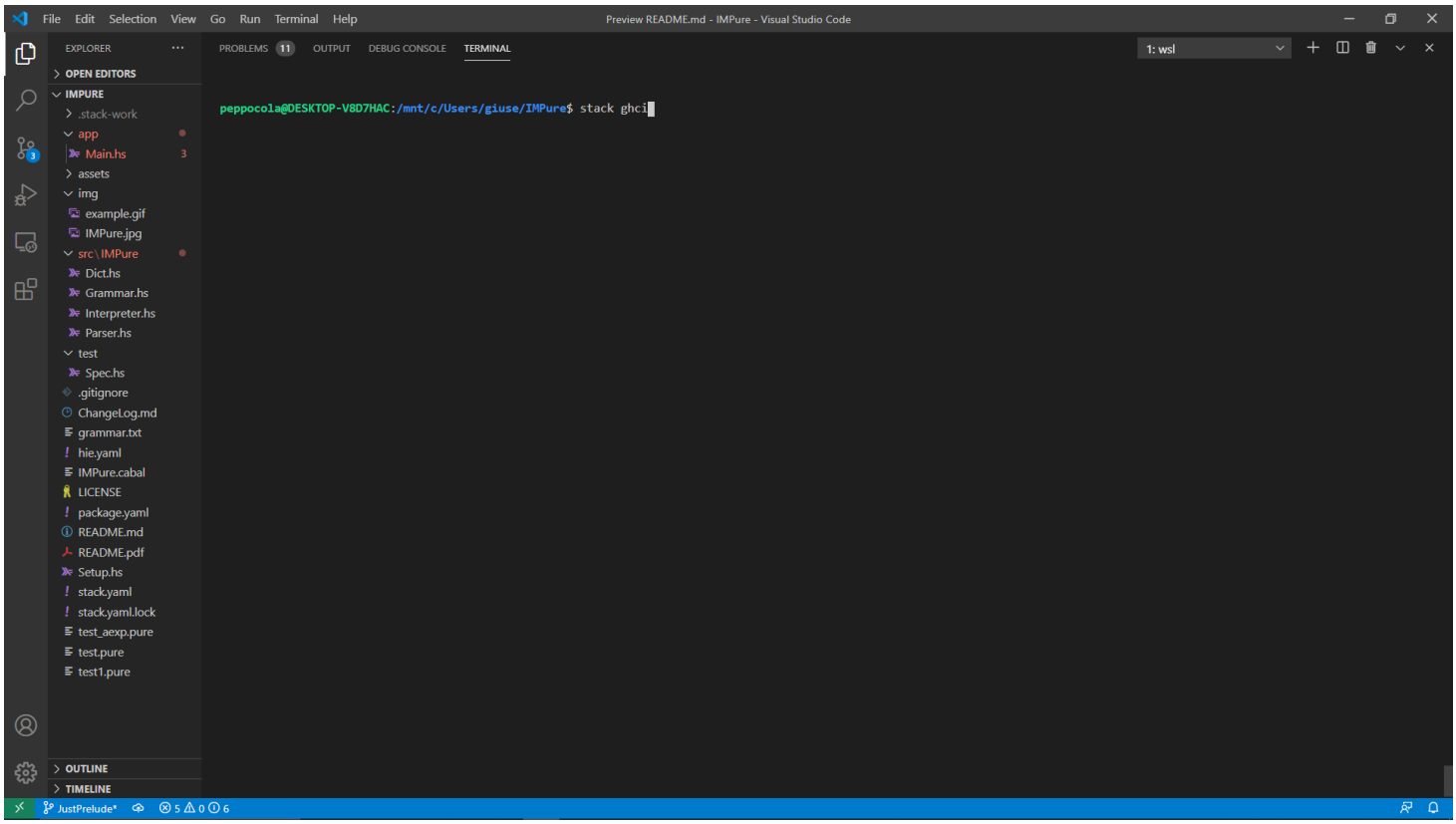
Here are some operation to use the parser.
The parse method parses an entire program and returns as output the list of commands written in the internal representation and the string of unconsumed output. If this string is not empty, the parsing failed!

Execution Example

First, we run the shell in the folder of the project.



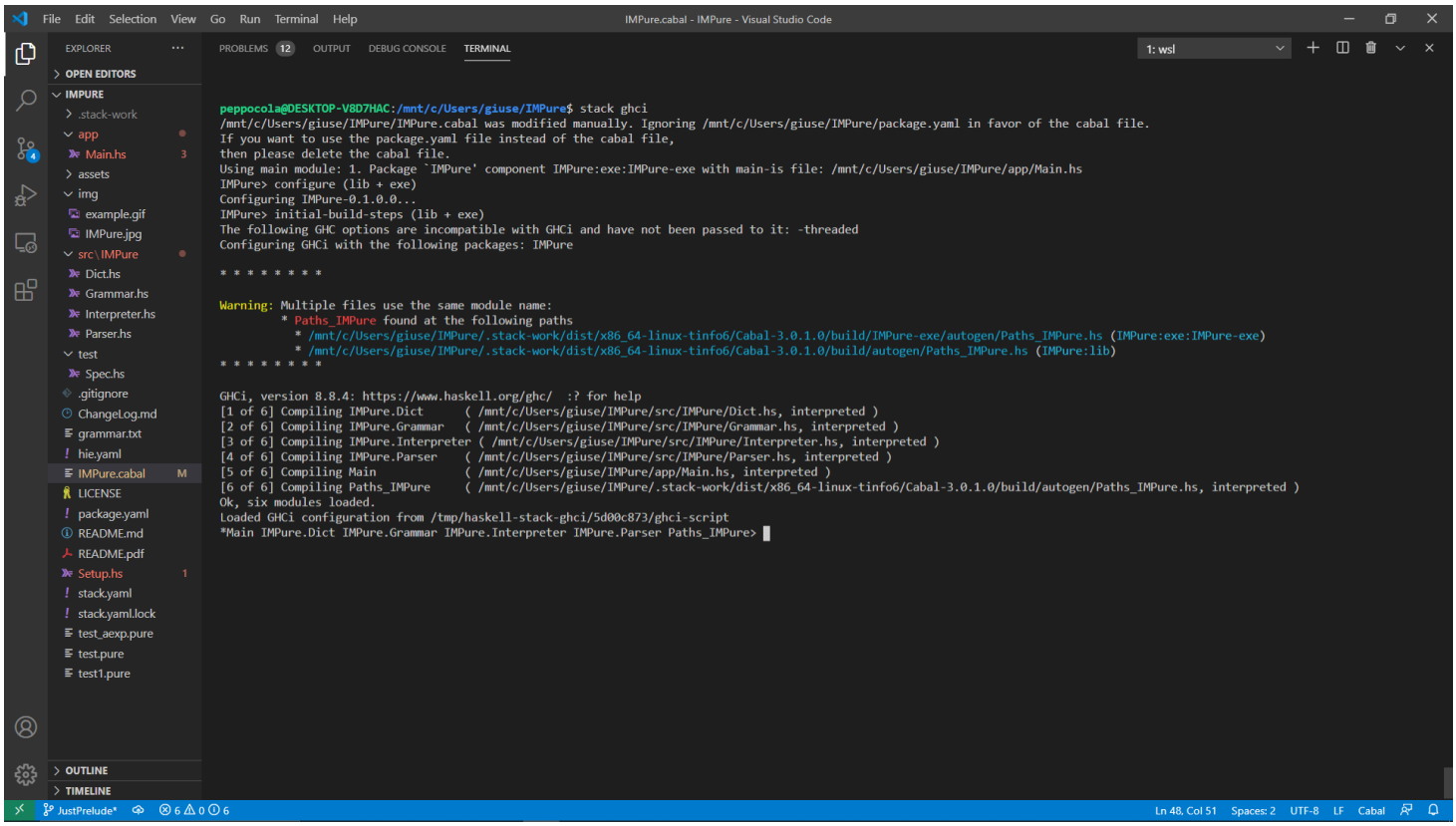
Then we type ghci.



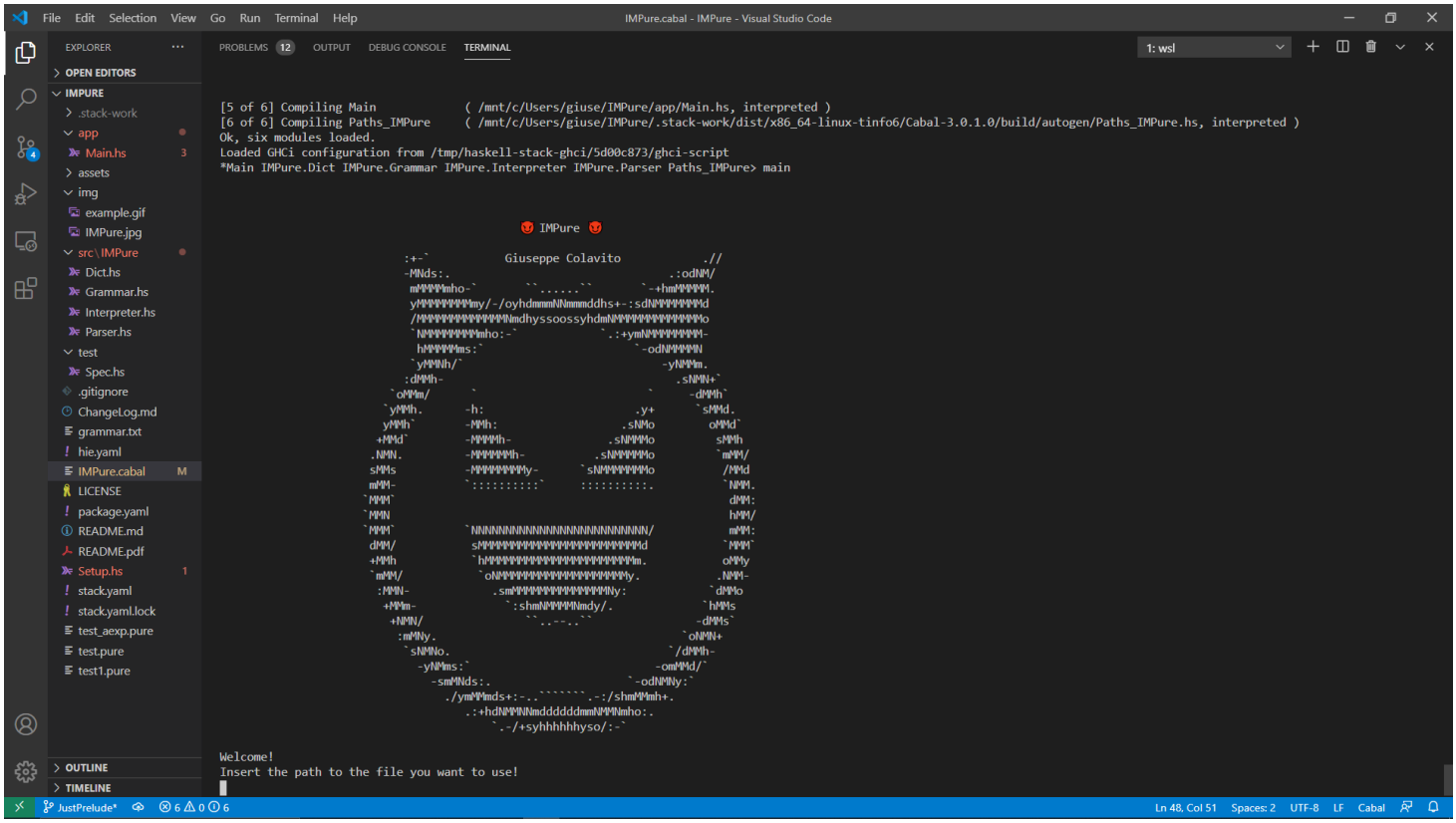
Once in ghci we import all the modules.

:l app/Main.hs src/IMPure/Dict.hs src/IMPure/Grammar.hs src/IMPure/Interpreter.hs src/IMPure/Parser.hs

(type this all in a single line!)

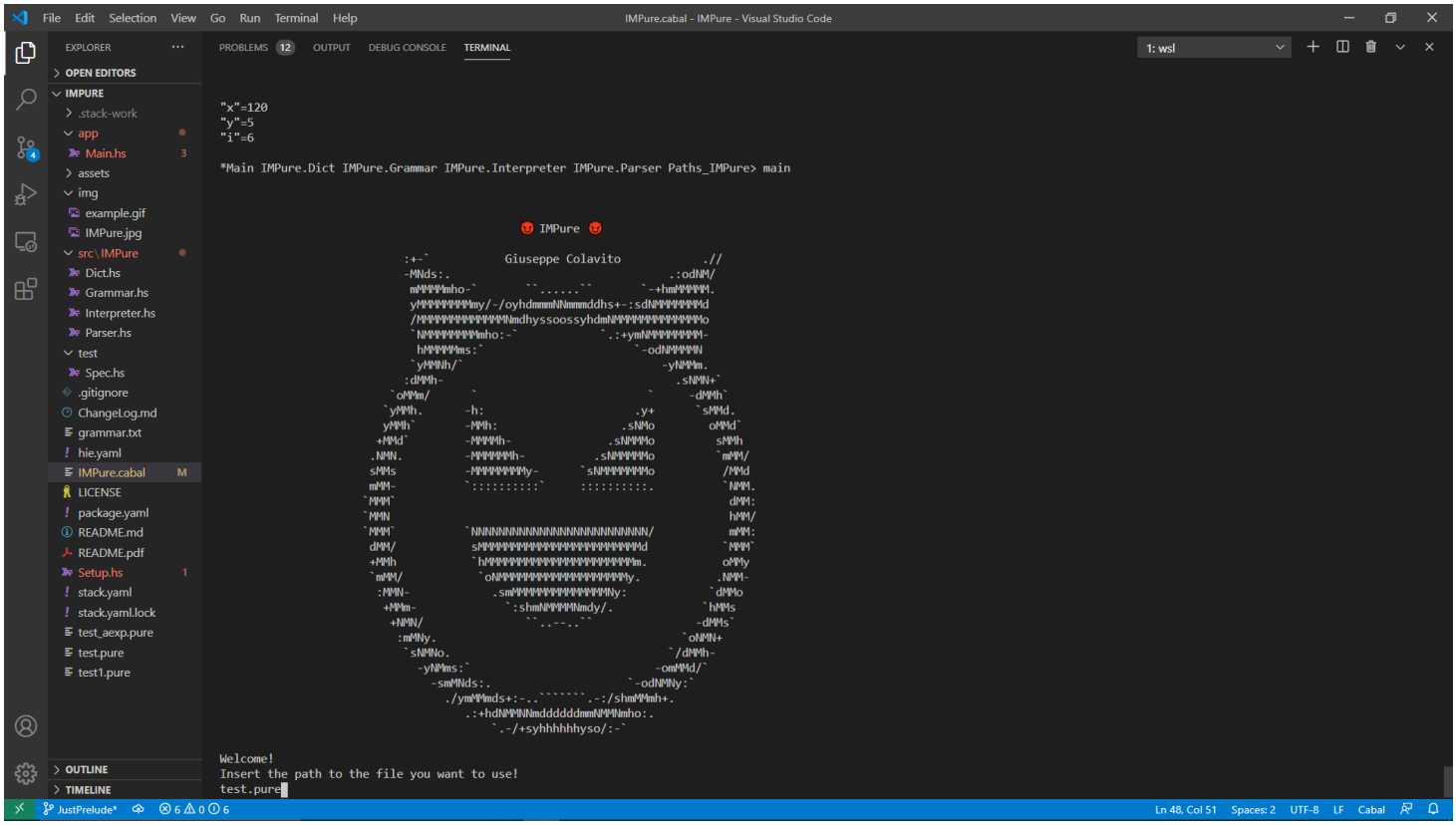


Once we have imported all the modules we type `main` and then press Enter!

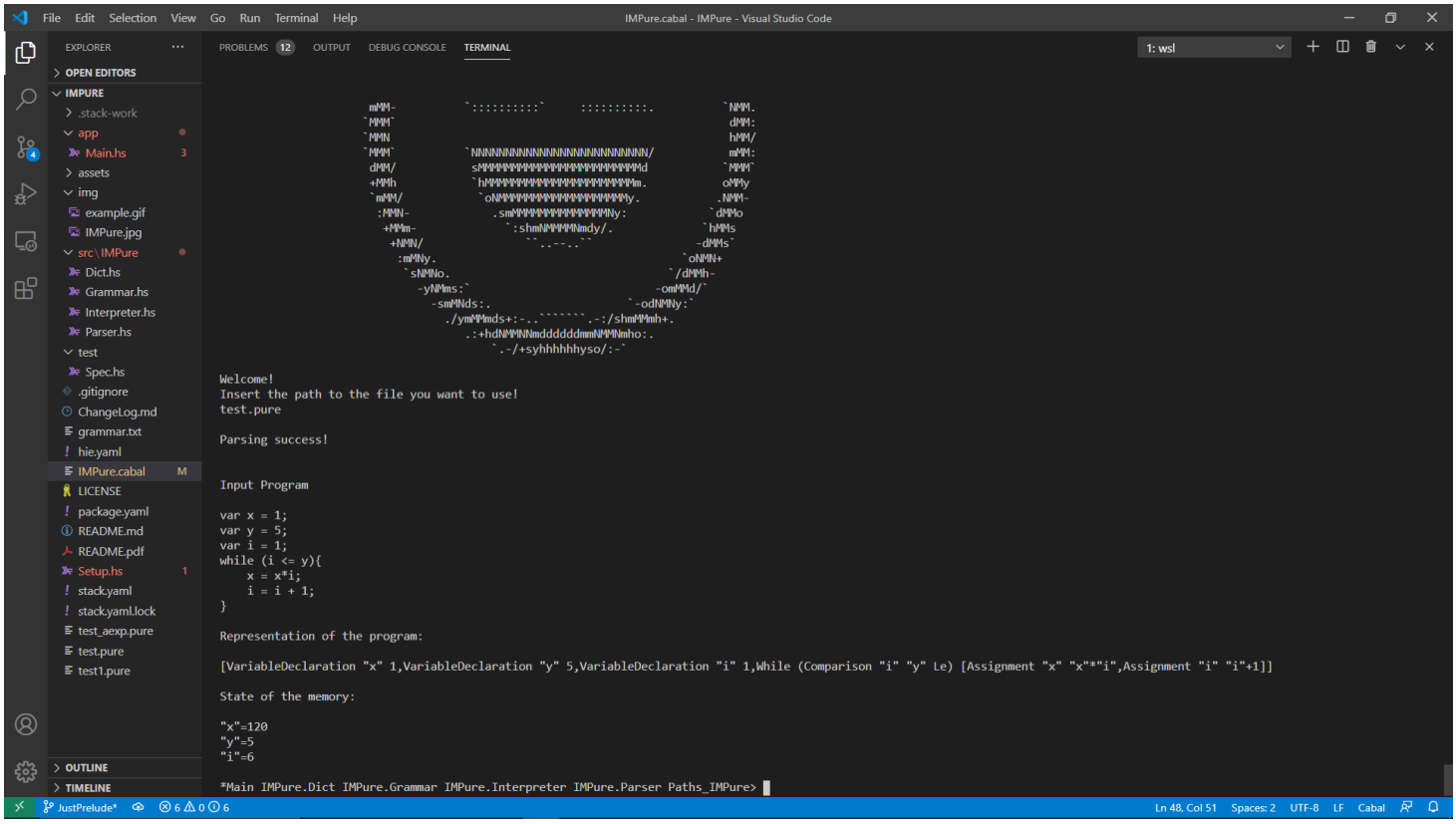


Now we should write the path to the file we want to give in input to the interpreter.

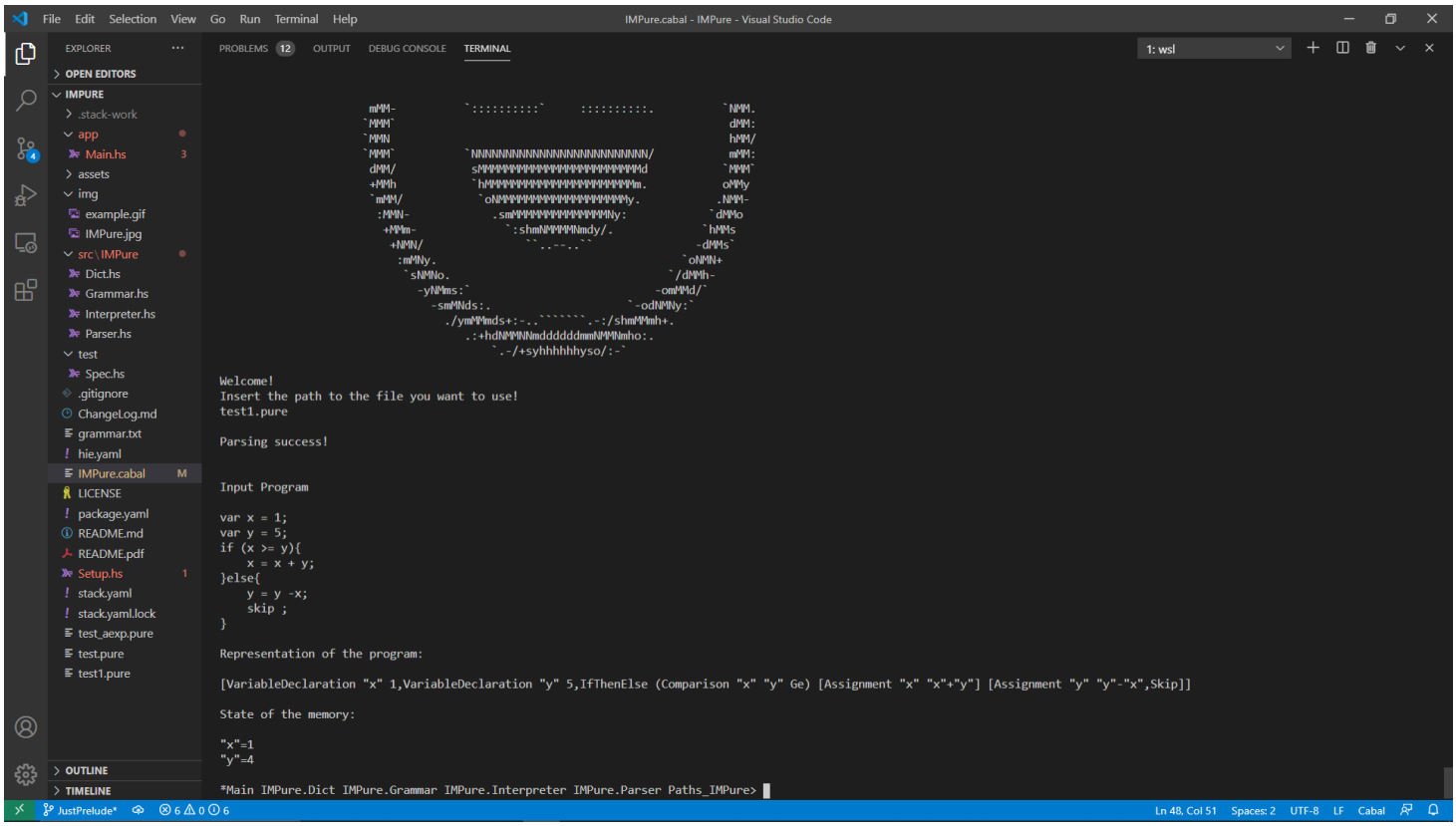
(you can use `test.pure` *and* `test1.pure` *)*



In this example the IMPure interpreter evaluates the factorial (the code used can be found in the file **test.pure**)



In this other example the IMPure interpreter evaluates a random program with an If-then-else (the code used can be found in the file **test1.pure**)



Have fun with IMPure!!!!!!