

Formal Methods in Computer Science

Haskell Interpreter for an Imperative Language

Fiorentino Nicola - 765448

A.Y. 2021-2022

I. Introduction

The defined language is a variant of the IMP language, enriched with additional commands and data structures. The following constructs will be provided:

- Assignment: instruction that assigns a value to a variable identified by a name. Variables represent integers or arrays of integers.
- Skip: instruction that changes neither the state nor the flow of execution.
- If-Then-Else: control structure that performs different computations depending on the evaluation of a boolean condition. The else block is optional.
- While: control structure used to repeat a computation until a specific condition is met.
- For: control structure that allows code to be executed repeatedly. It requires an initialization expression, a termination condition and an incremental step.

The syntax of the language is formalized using BNF. An interpreter for the language will be implemented in Haskell. We recall that an interpreter is a program that directly executes instructions written in a programming language, without requiring them to be previously compiled into a machine language program. To obtain the desired behavior, the source code must be parsed. The parser takes a string of characters as input and produces some form of tree that makes the syntactic structure of the string explicit.

II. BNF Grammar

The syntactic structure of the language is formalized in the following grammar, which describes how strings of the language can be constructed.

<code><digit> ::= 0 1 2 3 4 5 6 7 8 9</code>
<code><natural> ::= <digit> <natural> <digit></code>
<code><integer> ::= - <natural> <natural></code>
<code><lower> ::= a b c d e f g h i j k l m n o p q r s t u v w x y z</code>
<code><upper> ::= A B C D E F G H I J K L M N O P Q R S T U V W X Y Z</code>
<code><alphanum> ::= <upper> <alphanum> <lower> <alphanum> <natural> <alphanum> <upper> <lower> <natural></code>
<code><identifier> ::= <lower> <alphanum> <lower></code>
<code><aexp> ::= <aterm> + <aexp> <aterm> - <aexp> <aterm></code>
<code><aterm> ::= <afactor> * <aterm> <afactor> / <aterm> <afactor> % <aterm> <afactor></code>
<code><afactor> ::= (<aexp>) <identifier> <identifier> <index> <identifier> . len <integer></code>
<code><index> ::= [<aexp>]</code>
<code><bexp> ::= <bterm> <bexp> <bterm></code>
<code><bterm> ::= <bfactor> && <bterm> <bfactor></code>
<code><bfactor> ::= (<bexp>) ! <bfactor> true false <aexp> < <aexp> <aexp> <= <aexp> <aexp> > <aexp> <aexp> >= <aexp> <aexp> == <aexp> <aexp> != <aexp></code>
<code><program> ::= <command> <program> <command></code>
<code><command> ::= <assignment> <ifThenElse> <while> <for> skip ;</code>
<code><assignment> ::= <identifier> = <aexp> ; <identifier> = Array <index> ; <identifier> = [<arraycontent>] ; <identifier> <index> = <aexp> ;</code>
<code><arraycontent> ::= <aexp> , <arraycontent> <aexp></code>
<code><ifThenElse> ::= if <bexp> { <program> } else { <program> } if <bexp> { <program> }</code>
<code><while> ::= while <bexp> { <program> }</code>
<code><for> ::= for(<assignment> <bexp> ; <assignment>) { <program> }</code>

III.Environment

To store and manipulate variables we introduce an environment, capable of simulating the different states of a program. The environment can be seen as a memory that is read and updated along the program execution. It is implemented as a list of variables. Each variable will be structured as a pair, where the first element denotes the name and the second element denotes the value. The value of a variable is represented as a list of integers so that arrays can be easily implemented. Note that empty arrays are not allowed and arrays with a single element will be considered as integer variables.

```
type Variable = (String, [Int])
```

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

The updating of the environment and the reading of the variables are performed with the `updateEnv` and `readVariable` functions. To modify arrays that have already been created, we introduce a function `updateArray` that allows to assign a new value to the n-th element of a given array. Using parsers, the three functions can be integrated into the monadic parsing logic. Note that the function `lookup` returns the value to which a specified key is mapped within an association list.

```
readVariable :: String -> Parser [Int]
```

```
readVariable name = P (\env inp -> case lookup name env of
    Nothing -> []
    Just ns -> [(env, ns, inp)])
```

```
updateEnv :: Variable -> Parser String
```

```
updateEnv var = P (\env inp -> [(modify env, "", inp)])
```

```
  where
```

```
    modify [] = [var]
```

```
    modify (x:xs) = if fst x == fst var then var:xs else x:modify xs
```

```
updateArray :: String -> Int -> Int -> Parser String
```

```
updateArray name ind val = P (\env inp -> case lookup name env of
    Nothing -> []
```

```
    Just ns -> [(modify env, "", inp) | ind < length ns])
```

```
  where
```

```
    modify (x:xs) = if fst x == name then
```

```
      (name, take ind (snd x) ++ val:drop (ind + 1) (snd x)):xs
```

```
    else x:modify xs
```

IV. Parser

In Haskell, a parser can be seen as a function that takes a string and produces a tree. In general, it might not always consume its entire argument string. For this reason, we generalize it to also return any unconsumed part of the argument string. Since the parser might not always succeed, it should return a list of results, with the convention that the empty list denotes failure and a singleton list denotes success. Moreover, different parsers return different kinds of trees, or more generally, any kind of value. Hence, it is useful to abstract from the specific type of result values and make this into a parameter of the parser. The final requirement concerns the environment: a parsing operation must begin with an initial state and should end with a final state.

In summary, a parser of type **a** is a function that takes an environment and an input string and produces a list of results, each of which is a triple comprising an environment, a result value of type **a** and an output string. To allow the parser type to be made into instances of classes, it is defined using **newtype** with a dummy constructor **P**. The parser is then applied to an input string using a function that simply removes **P**.

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

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

We need to make the parser type into an instance of the functor, applicative and monad classes. In this way, we can use the **do** notation to combine parsers in sequence.

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

Note that **fmap** applies a function to the result value of a parser if the parser succeeds and propagates the failure otherwise; **pure** transforms a value into a parser that always succeeds with this value as result, without consuming any of the input string; **<*>** applies a parser that returns a function to a parser that returns an argument to give a parser that returns the result of applying the function to the argument and only succeeds if all the components succeed; **p >= f** fails if the application of the parser **p** to the input string fails, otherwise applies the function **f** to the result value **v** to give another parser **f v** which is then applied to the output string produced by the first parser. We also recall that the monadic function **return** is another name for the applicative function **pure**.

The **do** notation allows to combine parsers in sequence, with the output string from each parser in the sequence becoming the input string for the next. Another way of combining parsers is to apply one parser to the input string and if this fails to apply another parser to the same input. This is achieved by making the parser type into an instance of the alternative class.

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

In this case, **empty** is the parser that always fails regardless of the input string, and **<|>** is a choice operator that returns the result of the first parser if it succeeds on the input and applies the second parser to the same input otherwise.

V. Utility

Now it is possible to define a number of useful parsers. First, we consider a basic parser **item**, which fails if the input string is empty and succeeds with the first character as the result value otherwise. It represents the building block from which all other

parsers will be constructed. We will also define a parser `sat p` for single characters that satisfy the predicate `p`.

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

sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
    if p x then return x else empty
```

Using `sat` and appropriate predicates, we can define parsers for single digits, lower-case letters, upper-case letters, alphanumeric characters and specific characters. In this way, it is possible to define a parser `string xs` for a string of characters `xs`, with the string itself returned as the result value. Note that `string` only succeeds if the entire target string is consumed from the input.

```
digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char -> Parser Char
char x = sat (== x)

string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
    string xs
    return (x:xs)
```

We will also use the default parsers **many** *p* and **some** *p*, which apply a parser *p* as many times as possible until it fails, with the result values from each successful application being returned in a list. The difference between the two is that **many** permits zero or more applications, whereas **some** requires at least one successful application. Using them, we can implement parsers for identifiers, natural numbers and spacing. Specifically, **nat** converts the number that was read into an integer, while **space** returns the empty tuple `()` as a dummy result value. Using **nat** it will be easy to define a parser for integers.

```
ident :: Parser String
ident = do x <- lower
        xs <- many alphanum
        return (x:xs)

nat :: Parser Int
nat = do xs <- some digit
      return (read xs)

space :: Parser ()
space = do many (sat isSpace)
        return ()

int :: Parser Int
int = do char '-'
        n <- nat
        return (-n)
    <|> nat
```

Parsers should allow spacing to be freely used around the tokens of an input string. To handle such spacing, we define a primitive that ignores any space before and after applying a parser for a token. Then, we can define parsers that ignore spacing around identifiers, natural numbers, integers and special symbols.

```
token :: Parser a -> Parser a
token p = do space
            v <- p
            space
            return v
```

```

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol xs = token (string xs)

```

Finally, to properly handle arrays we define two more parsers: **index** parses a numeric value enclosed in square brackets; **arraycontent** parses a series of comma-separated values, returning them as a list of integers. Both use the primitive **aexp** that will be introduced in the next paragraph.

```

index :: Parser Int
index = do symbol "["
          n <- aexp
          symbol "]"
          return n

arraycontent :: Parser [Int]
arraycontent = do n <- aexp
                 do symbol ","
                   ns <- arraycontent
                   return (n:ns)
                 <|> return [n]

```

VI. Arithmetic Expressions

It is now possible to translate the language grammar into a parser, by rewriting the rules using the parsing primitives introduced. Sequencing in the grammar is translated into the **do** notation, choice **|** is translated into the **<|>** operator and special symbols are handled using the **symbol** function.

For arithmetic expressions we consider different priority levels and right associativity of operators. For this reason, we define the following primitives: **aexp** manages addition and subtraction; **aterm** manages multiplication, division and modulus;

afactor manages parenthesized expressions, (possibly indexed) identifiers, array lengths and integers. Note that trying to read a never assigned variable will raise an error.

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

aterm :: Parser Int
aterm = do f <- afactor
        do symbol "*"
            t <- aterm
            return (f * t)
        <|> do
            symbol "/"
            t <- aterm
            return (f `div` t)
        <|> do
            symbol "%"
            t <- aterm
            return (f `mod` t)
        <|> return f

afactor :: Parser Int
afactor = do symbol "("
            e <- aexp
            symbol ")"
            return e
        <|> do
            i <- identifier
            ns <- readVariable i
            do k <- index
                if k >= 0 && k < length ns then return (ns !! k)
                else empty
```

```

    <|> do
      symbol "."
      symbol "len"
      return (length ns)
    <|> return (head ns)
  <|> integer

```

VII. Boolean Expressions

Boolean expressions are handled in a similar way. To consider different priority levels between operators the following primitives are introduced: **bexp** manages disjunctions; **bterm** manages conjunctions; **bfactor** manages parenthesized expressions, negations, truth values and arithmetic comparisons.

```

bexp :: Parser Bool
bexp = do t <- bterm
      do symbol "||"
        e <- bexp
        return (t || e)
    <|> return t

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

bfactor :: Parser Bool
bfactor = do symbol "("
      e <- bexp
      symbol ")"
      return e
    <|> do
      symbol "!"
      f <- bfactor
      return (not f)
    <|> do
      symbol "true"

```

```

    return True
<|> do
    symbol "false"
    return False
<|> do
    a <- aexp
    do symbol "<="
        b <- aexp
        return (a <= b)
    <|> do
        symbol "<"
        b <- aexp
        return (a < b)
    <|> do
        symbol ">="
        b <- aexp
        return (a >= b)
    <|> do
        symbol ">"
        b <- aexp
        return (a > b)
    <|> do
        symbol "=="
        b <- aexp
        return (a == b)
    <|> do
        symbol "!="
        b <- aexp
        return (a /= b)

```

VIII.Commands

By following the grammar, a program and a command can be easily parsed as follows. Note that the skip command is used as a placeholder for an empty program.

```

program :: Parser String
program = do command
          program
          <|> command

```

```

command :: Parser String
command = assignment
    <|> ifThenElse
    <|> while
    <|> for
    <|> do symbol "skip"
            symbol ";"
            return ""

```

The assignment operation can link: a (possibly indexed) identifier with an arithmetic expression; an identifier with the keyword **Array** followed by an index (to create an array of a given size, filled with zeros); an identifier with a sequence of comma separated values, enclosed in square brackets. The syntax requires the use of the symbol = with a semicolon at the end of the statement. Each assignment will update the environment.

```

assignment :: Parser String
assignment = do i <- identifier
              do symbol "="
                  n <- aexp
                  symbol ";"
                  updateEnv (i, [n])
              <|> do
                  symbol "="
                  symbol "Array"
                  k <- index
                  symbol ";"
                  updateEnv (i, replicate k 0)
              <|> do
                  symbol "="
                  symbol "["
                  ns <- arraycontent
                  symbol "]"
                  symbol ";"
                  updateEnv (i, ns)
              <|> do
                  k <- index
                  symbol "="
                  n <- aexp
                  symbol ";"
                  updateArray i k n

```

In the if-then-else construct, brackets around the boolean condition are not mandatory. The else branch is optional. Each block of statements must be enclosed in curly brackets and cannot be empty. If according to the condition a block should not be executed, it will be parsed without being evaluated.

```
ifThenElse :: Parser String
ifThenElse = do symbol "if"
               b <- bexp
               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 ""
```

Similarly, in the while construct parentheses around the boolean condition are not mandatory and the block of statements must be enclosed in curly brackets. If the condition is met, the block is executed and the while is added to the unparsed string for a new evaluation. When the condition is not met anymore, the block is parsed without being evaluated.

```

while :: Parser String
while = do w <- consumeWhile
          P (\env inp -> [(env, "", w ++ inp)])
          symbol "while"
          b <- bexp
          symbol "{"
          if b then do
            program
            symbol "}"
            P (\env inp -> [(env, "", w ++ inp)])
            while
          else do
            consumeProgram
            symbol "}"
            return ""

```

In the for construct, initialization, condition and incremental step must be enclosed in round brackets. Note that each of the three components must end with a semicolon. After executing the initialization step, the for loop is translated into a while construct, adding the incremental step at the end of the block.

```

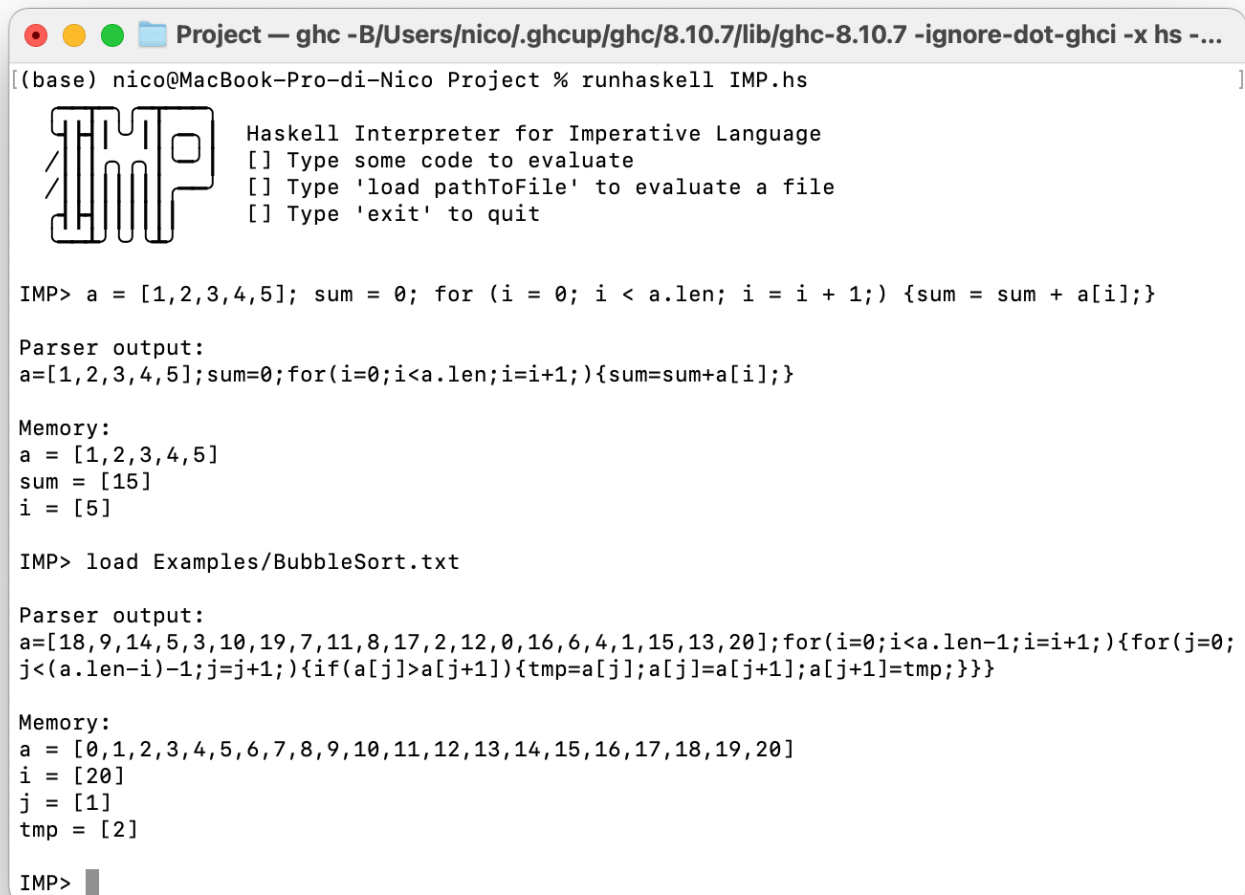
for :: Parser String
for = do symbol "for"
          symbol "("
          assignment
          c <- consumeBexp
          symbol ";"
          i <- consumeAssignment
          symbol ")"
          a <- symbol "{"
          p <- consumeProgram
          b <- symbol "}"
          P (\env inp -> [(env, "", "while" ++ c ++ a ++ p ++ i ++ b ++ inp)])
          while

```


We often need to read an input string without executing it. For this reason, we define a number of consume functions that parse the input without evaluation, leaving the environment unchanged. Due to their simplicity, these functions are not reported in the documentation. They can be found in `IMP.hs`.

IX. Execution

To run the interpreter just type in the console `runhaskell path/IMP.hs`. A simple interface will be displayed. Now you can type some code to evaluate. Alternatively, you can execute a program stored on a text file by typing the command: `load pathToFile`. Some simple programs to run are provided in the Examples folder. For each run, the parser output and the final state of the memory will be shown. An example of use is provided below.



```
Project — ghc -B/Users/nico/.ghcup/ghc/8.10.7/lib/ghc-8.10.7 -ignore-dot-ghci -x hs -...
[(base) nico@MacBook-Pro-di-Nico Project % runhaskell IMP.hs]

 Haskell Interpreter for Imperative Language
[] Type some code to evaluate
[] Type 'load pathToFile' to evaluate a file
[] Type 'exit' to quit

IMP> a = [1,2,3,4,5]; sum = 0; for (i = 0; i < a.len; i = i + 1;) {sum = sum + a[i];}

Parser output:
a=[1,2,3,4,5];sum=0;for(i=0;i<a.len;i=i+1;){sum=sum+a[i];}

Memory:
a = [1,2,3,4,5]
sum = [15]
i = [5]

IMP> load Examples/BubbleSort.txt

Parser output:
a=[18,9,14,5,3,10,19,7,11,8,17,2,12,0,16,6,4,1,15,13,20];for(i=0;i<a.len-1;i=i+1;){for(j=0;
j<(a.len-i)-1;j=j+1;){if(a[j]>a[j+1]){tmp=a[j];a[j]=a[j+1];a[j+1]=tmp;}}}

Memory:
a = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
i = [20]
j = [1]
tmp = [2]

IMP> █
```