Interpreter for IMP Language

Roberto Lorusso

September 2022 - Università degli Studi di Bari Aldo Moro

Indice

Introduction

This work is focused on the development of an interpreter for extending the imperative language IMP with new types and commands. The interpreter is written in th Haskell, a functional programming language suited for this purpose. In the first chapter we give an overview of the language IMP and its enriched grammar used for our interpreter, defined in Backus-Naur Form. Afterwards we start going deeper, describing the parser and its parsing strategy written in Haskell. Then we proceed with the description of the Environment type and the operations that make possible the manipulation of data once the interpreter consumed the input strings. At the end there are some execution examples, giving a glance on the possible operation allowed by the grammar.

Capitolo 1

Grammar for IMP

In this chapter we introduce the language IMP in order to extend it, resulting in the formalization of the language used by our interpreter

1.1 The language IMP

In this section we'll briefly introduce the language IMP, its syntax and formation rules, and then we'll specify the formation rules for our grammar: an the extended version of IMP, comprising complex data structure such as arrays, matrices, queues and stacks. **IMP** is called an "imperative" language because program execution involves carrying out a series of explicit commands to change state. Formally, IMP's behaviour is described by rules which specify how its expressions are evaluated and its commands are executed. Those rules can be defined by means of operational or denotational semantics. Firstly, we list the syntactic sets associated with **IMP**:

- numbers **N**, consisting of positive and negative integers with zero,
- truth values **T** = **true, false**,
- location **Loc**,
- arithmetic expressions **Aexp**,
- boolean expressions **Bexp**,
- commands **Com**.

We assume the syntactic structure of numbers and locations is given. For instance, the set **Loc** might consist of non-empty strings of letters or such strings followed by digits, while **N** might be the set of signed decimal numerals for positive and negative whole number. For the other syntactic sets we need to define the formation rules by means of **BNF**(Backus-Naur form). We consider meta-variables to range over the syntactic sets: where a, b, c, n, X, are meta-variables associated with the respective syntactic sets.

For **Aexp**:

$$a ::= n|X|a_0 + a_1|a_0 - a_1|a_0 \times a_1 \tag{1.1}$$

For **Bexp**:

$$b ::= true|false|a_0 = a_1|a_0 \leq a_1|\neg b|b_0 \wedge b_1|b_0 \vee b_1 \tag{1.2}$$

For **Com**:

$$c ::= skip|X := a|c_0; c_1|if\ b\ then\ c_0\ else\ c_1|\ while\ b\ do\ c \tag{1.3}$$

Here we explicit the behaviour of the commands above:

- **Skip**: it doesn't execute anything, skipping to the next instruction,
- **Assignment**: the result of the evaluation of the right side of ":=" is assigned to the variable identified by the string on the left side.
- **Sequence**: two or more instructions are executed one after the other until there is nothing more to execute,
- **If-Then-Else**: the standard conditional instruction,
- **While-Loop**: executes a sequence of instructions many times, until a condition is verified.

As anticipated, the interpreter manages data structures like **Arrays, Matrices, Queues and Stacks**, and so commands for their manipulation have been implemented:

- **For**: executes a sequence of instructions a given number of times, through the use of a counter,
- **Repeat-Until**: also called Do-While, it executes a sequence of instructions at least one time, until a condition is reached.

1.2 Grammar in BNF

Now we are able to give the formation rules for our extended version of the language IMP, making its structure explicit, thus defining possible operations on the types. We start from the definition of arithmetical expressions.

1.2.1 Arithmetical expressions

Below the grammar for **Aexp**:

$$\begin{array}{lll} \langle aexp \rangle & ::= & \langle aterm \rangle \text{ '+' } \langle aexp \rangle \\ & | & \langle aterm \rangle \text{ '-' } \langle aexp \rangle \\ & | & \langle aterm \rangle \end{array}$$

$$\begin{array}{lll} \langle aterm \rangle & ::= & \langle afactor \rangle \text{ '*' } \langle aterm \rangle \\ & | & \langle afactor \rangle \text{ '/' } \langle aterm \rangle \\ & | & \langle afactor \rangle \text{ '\^{}' } \langle afactor \rangle \\ & | & \langle afactor \rangle \text{ '\%' } \langle afactor \rangle \\ & | & \langle afactor \rangle \end{array}$$

⟨*afactor*⟩ ::= ‘(‘ ⟨*aexp*⟩ ‘)’
| ⟨*identifier*⟩ ‘[‘ ⟨*aexp*⟩ ‘]’
| ⟨*identifier*⟩ ‘[‘ ⟨*aexp*⟩ ‘]’ ‘[‘ ⟨*aexp*⟩ ‘]’
| ⟨*identifier*⟩
| ⟨*integer*⟩

1.2.2 Boolean expressions

Below the grammar for **Bexp**:

⟨*bexp*⟩ ::= ⟨*bterm*⟩ ‘OR’ ⟨*bexp*⟩
| ⟨*bterm*⟩

⟨*bterm*⟩ ::= ⟨*bfactor*⟩ ‘AND’ ⟨*bterm*⟩
| ⟨*bfactor*⟩

⟨*bfactor*⟩ ::= ‘true’
| ‘false’
| ‘!’ ⟨*bfactor*⟩
| ‘(’ ⟨*bexp*⟩ ‘)’
| ⟨*bcomparison*⟩

⟨*bcomparison*⟩ ::= ⟨*aexp*⟩ ’ $>$ ’ ⟨ *aexp* ⟩
| ⟨*aexp*⟩ ‘$\leq$’ ⟨*aexp*⟩
| ⟨*aexp*⟩ ‘$>$’ ⟨*aexp*⟩
| ⟨*aexp*⟩ ‘$\geq$’ ⟨*aexp*⟩ | ⟨*aexp*⟩ “==” ⟨*aexp*⟩”

1.2.3 Data structures

Here we define the grammar for complex data structure such as arrays, matrices, stacks and queues.

Arrays, Stacks and Queues

An array is a sequence of aexp elements. It’s worth to notice that empty arrays are not permitted.

⟨*array*⟩ := ‘[’ ⟨*arrayItems*⟩ ‘]’ ⟨*arrayItems*⟩ := ⟨*aexp*⟩ ‘,’ ⟨*arrayItems*⟩
| ⟨*aexp*⟩

The formation rule for the arrays encompasses that of **stacks** and **queues**, the main difference is in the operation with which these different data types are equipped and their elements’ organization.

Stack’s operations

- **push**(identifier, value): inserts a new element inside the array-shaped stack.
- **pop**(identifier): deletes an element and its location from the array-shaped stack.

Queue's operations

- **enqueue**(identifier, value): inserts a new element in the array-shaped queue.
- **dequeue**(identifier): deletes an element and its location from the array-shaped queue.

Matrices

A matrix is defined as an array of arrays.

⟨*matrix*⟩ := '[' ⟨*matrixItems*⟩ ']' ⟨*matrixItems*⟩ := ⟨*array*⟩ ',' ⟨*matrixItems;*⟩
| ⟨*array*⟩

1.2.4 Commands

As we can see, the grammar fro **Aexp** and **Bexp** resembles the original grammar of IMP. In this case we give a new definition for the commands. Below the grammar for **Comm**.

Program

A program is defined by a command or a sequence of commands, where the listed commands are described in the next sections. For the sake of brevity we assume that the skip command does not affect the state of the memory.

⟨*program*⟩ ::= ⟨*command*⟩
| ⟨*command*⟩ ⟨*program*⟩

⟨*command*⟩ ::= ⟨*ifThenElse*⟩
| ⟨*While*⟩
| ⟨*forLoop*⟩
| ⟨*repeatUntil*⟩
| 'skip' ';'

Assignment

This command has many different definitions, accordingly to the possibility of managing complex data structures such as arrays, matrices, queues and stacks. Below the BNF grammar for the assignment command:

⟨*assignment*⟩ ::= ⟨*identifier*⟩ ':=' ⟨*aexp*⟩ ';'
| ⟨*identifier*⟩ ':=' ⟨*bexp*⟩ ';'
| ⟨*identifier*⟩ ':=' ⟨*array*⟩ ';'
| ⟨*identifier*⟩ ':=' ⟨*array*⟩ '++' ⟨*array*⟩ ';'
| ⟨*identifier*⟩ ':=' ⟨*identifier*⟩ '++' ⟨*identifier*⟩ ';'
| 'createArray('⟨*identifier*⟩ '[' ⟨*aexp*⟩ ']);'
| ⟨*identifier*⟩ ':=' ⟨*matrix*⟩ ';'
| ⟨*identifier*⟩ '[' ⟨*aexp*⟩ ']' '[' ⟨*aexp*⟩ ']' ':=' ⟨*aexp*⟩ ';'
| ⟨*identifier*⟩':=' ⟨*identifier*⟩ '[' ⟨*aexp*⟩ ']' '[' ⟨*aexp*⟩ '];'
| ⟨*identifier*⟩':=' ⟨*identifier*⟩ '[' ⟨*aexp*⟩ '];'

| ⟨*identifier*⟩ '[' ⟨*aexp*⟩ ']:=' ⟨*identifier*⟩ '[' ⟨*aexp*⟩ '];'
| 'createMatrix('⟨*identifier*⟩ '[' ⟨*aexp*⟩ ']' '[' ⟨*aexp*⟩ ']);'
| ⟨*identifier*⟩ ':=' ⟨*stack*⟩ ';'
| 'push('⟨*identifier*⟩','⟨*aexp*⟩');'
| 'pop('⟨*identifier*⟩');'
| ⟨*identifier*⟩ ':=' ⟨*queue*⟩ ';'
| 'enqueue(' ⟨*identifier*⟩','⟨*aexp*⟩');'
| 'dequeue('⟨*identifier*⟩ ');'

If-Then-Else

The traditional If-Then-Else command, extended with the possibility of being declared without explicitly use the 'else' block, By means of BNF grammar:

⟨*ifThenElse*⟩ ::= 'if' ⟨*bexp*⟩ '' ⟨*program*⟩ ''
| 'if' ⟨*bexp*⟩ '' ⟨*program*⟩ '' 'else' '' ⟨*parseProgram*⟩ ''
| 'if' ⟨*bexp*⟩ '' ⟨*parseProgram*⟩ ''
| 'if' ⟨*bexp*⟩ '' ⟨*parseProgram*⟩ '' 'else' '' ⟨*program*⟩ ''

While-Loop

The while-loop command allows us to carry out the instructions inside its body multiple times, until a certain condition is verified. By means of BNF grammar:

⟨*while*⟩ ::= ⟨*parsewhile*⟩ 'while' "(⟨*bexp*⟩ ")" '' ⟨*program*⟩ '' ⟨*while*⟩
| ⟨*parsewhile*⟩ 'while' "(⟨*bexp*⟩ ")" '' ⟨*parseProgram*⟩ ''

Repeat-Until

The repeatUntil command, also called Do-While, is the opposite of the previously mentioned while-loop command. The main difference is that this command carries out the instructions in its body at least one time, and its execution keeps looping until a condition is verified. By means of BNF grammar:

⟨*repeatUntil*⟩ ::= ⟨*parseRepeatUntil*⟩ 'repeat' '' ⟨*program*⟩ '' 'until' ⟨*bexp*⟩ ⟨*repeatUntil*⟩
| ⟨*parseRepeatUntil*⟩ 'repeat' '' ⟨*program*⟩ '' 'until' ⟨*bexp*⟩

For-Loop

This is a common command in imperative languages like C. executes a sequence of instructions a given number of times, through the use of a counter. Here we propose the same syntax of the C language, according to the following structure:

- assignment of a starting value for the counter variable,
- condition on the variable counter,
- action to be carried out on the variable counter at every step of execution of the command,
- body of the command containing the instructions to be carried out.

By means of BNF grammar:

$$
\begin{array}{lcl}
\langle \textit{forLoop} \rangle & ::= & \text{`for('} \ \langle \textit{assignment} \rangle \ \langle \textit{bexp} \rangle \ \text{`;'} \ \langle \textit{identifier} \rangle \ \text{`++)\{'} \ \langle \textit{program} \rangle \ \text{`\}'} \\
 & | & \text{`for('} \ \langle \textit{assignment} \rangle \ \langle \textit{bexp} \rangle \ \text{`;'} \ \langle \textit{identifier} \rangle \ \text{`--) \{'} \ \langle \textit{program} \rangle \ \text{`\}'} \\
 & | & \text{`for('} \ \langle \textit{assignment} \rangle \ \langle \textit{bexp} \rangle \ \text{`;'} \ \langle \textit{assignment} \rangle \ \text{`)'}
\end{array}
$$

Capitolo 2

Parsers

In this chapter we define the parser for the already defined grammar, highlighting the implementation details concerning the Haskell language. A parser is a program that takes a string of characters, and produces some form of tree that makes the syntactic structure of the string explicit. Haskell can be used to program simple ones, and by using them in combination we can obtain a more complex program able to preprocess complex inputs such as arithmetic expressions. Parsers can be naturally viewed as functions, leading to their formal definition:

$$type\ Parser\ a = String \rightarrow [(a, String)]$$

Since in our language we introduced **variables**, we need to define an **Environment** for them in order to store their names and values over the time. The function describing the parser becomes

$$newtype\ Parser\ a = P\ (Env \rightarrow String \rightarrow [(Env, a, String)])$$

where **Env** refers to environment, which is a list of variables. Later, at chapter 3, we give a more precise definition for these data types; for the moment let's focus on how the parsers pre-processes an input string. We can start by defining two essential parsers such as **parse** and **item**. The former consent to us to apply a parser to an Environment and an input string without using the dummy constructor, while the latter will consume only the first character of an input string or, in the case that the input is an empty string, it will return an empty string.

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

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

```

2.1 Functor, Monad and Applicative

In order to apply parsers in sequence we need to define the **Parser** type as instance of Functor, Applicative and Monad. This allows us to apply a parser to the output of another, leading to a sequence of transformation of the input string. In this way we can use different parsers in the **do** notation, also introducing a **non-deterministic** behaviour which will be handled by the Haskell Language Server.

Parser as a Functor

Declaring the parser type as an instance of a Functor makes possible to apply functions to the result value of a successful parser. That is, fmap applies a function to the result value of a parser if the parser succeeds, and propagates the failure otherwise. Below its implementation:

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

```

Parser as an Applicative

Declaring the parser as an instance of Applicative makes possible to construct complex parsers from simpler ones, by means of higher-order functions. In Haskell, one of the ways in which parsers can be elegantly combined is using applicative style. First we give an implementation for the **pure** function, which maps an element from its domain to the domain of parsers always succeeding, without consuming any of the input string. Then we define an implementation for the applicative operator $<*>$ in order to apply 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. By the introduction of this operator we can apply different parser in sequence without defining many different **fmap** for the Functor instance.

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

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

```

Parser as a Monad

Parsers can also be expressed and combined using monadic style. In this way we can exploit the power of **do** notation and being able to apply a function f to a value v and then return $f\ v$ to another parser. The implementation for the sequencing operator is:

```
instance Monad Parser where
-- (>>=) :: Parser a -> (a -> Parser b) -> Parser b

p >>= f = P (\env input -> case parse p env input of
```

```
    [] -> []
    [(env, v, out)] -> parse(f v) env out)

```

That is, the parser $p \quad >>= \quad f$ fails if the application of the parser p to the input string *inp* fails, and otherwise applies the function f to the result value v to give another parser fv,which is then applied to the output string out that was produced by the first parser to give the final result.

Parser as an Alternative

The **do** notation combines parsers in sequence, with the output string from each parser in the sequence becoming the input string for the next. Another natural way of combining parsers is to apply one parser to the input string, and if this fails to then apply another to the same input instead. We now consider how such a choice operator can be defined for parsers. In this way we can introduce a **non-deterministic** behaviour to the parsers.

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

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

```

Where **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.

2.2 Parsers for types

In this section we discuss the implementation of Parsers for the types defined in the grammar. Every section for types defined in our grammar has the same structure: a first part concerning the implementation of the grammar and a second part describing the parsing strategy for the input strings. Later it will be clear the necessity of just reading the string without processing it. This kind of operation is necessary when we have to print many times certain part of the program in the input string, such as when we are performing a loop, in order to correctly evaluate such commands. The functions that perform such parsing strategies are quite similar to the functions that represent the grammar of the expressions.

First, we need to introduce some utility parsers which are essential for parsing our types.

2.2.1 Utility parsers

We can exploit the power of built-in functions of Haskell of type $Char \rightarrow Bool$ by using them with a parser that verifies that their condition is satisfied on the input string.

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

```

Following the same pattern, we can write a parser that returns a blank for every input string:

```
space :: Parser ()
space = do {
  many(sat isSpace);
  return ();}

```

Now we can embed some utility functions, such as isLower, inside our parsers by employing our parser **sat** as follows:

```
lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

natural :: Parser Int
natural = token nat

digit :: Parser Char
digit = sat isDigit

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);}


```

In this way we identify strings that encode integers, natural numbers, variable's identifier and so on. These parsers will simply return the input string if the predicate is satisfied.

The token parser

This parser, and its derivates, deserves to be highlighted since it is essential for matching identifiers, variables and so on, without considering the spaces with which they can be wrapped.

```
token :: Parser a -> Parser a
token p = do {
  Space;
```

```
    v <- p;
    Space;
    return v;}
```

As we can see it drops spaces from the input and considers only the string between them. Now we can define other parser that use **token**.

```
  identifier :: Parser String
  identifier = token ident

  integer :: Parser Int
  integer = token int

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

Where **ident** is the parser for identifiers, and int is that for integers. They look like this:

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

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

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

The pattern is becoming very clear: we can combine very simple parser to build complex ones. Let's move to define parsers for our grammar.

2.2.2 Arithmetical expressions

Here we give the implementation for the Arithmetical expression, that is, how our interpreter recognizes the **aexp** defined in our grammar, and the parsing strategy for them.

Implementation for Aexp

The necessity behind the use of aexp and aterm is to impose a priority ordering in the evaluation of the operators '*' and '/' with respect to '+' and '-'.

```
  aexp :: Parser Int
  aexp = (do
  t <- aterm
  symbol "+"
  a <- aexp
```

```
return (t+a))
<|>
(do
t <- aterm
symbol "-"
a <- aexp
return (t-a))
<|>
aterm

aterm :: Parser Int
aterm = do {
  f <- afactor;
  symbol "*";
  t <- aterm;
  return (t * f);    }
<|>
do {
  f <- afactor;
  symbol "/";
  t <- aterm;
  return (f `div` t);    }
<|>
do {
  f <- afactor;
  symbol "^";
  t <- afactor;
  return(power f t);    }
<|>
do {
  f <- afactor;
  symbol "%";
  t <- afactor;
  return(modul f t);    }
<|>
afactor

afactor :: Parser Int
afactor = (do
symbol "("
a <- aexp
symbol ")"
return a)
<|>
(do
i <- identifier
readVariable i)
<|>
integer
```

The first part of the definition of afactor introduces the possibility to use the parenthesis to impose an ordering in the evaluation of the arithmetic expressions.

Parser for Aexp

Here we define the parsing strategy for the Arithmetical expressions.

```
parseAexp :: Parser String
parseAexp =
```

```
  do {
    t <- parseAterm;
    do {
      symbol "+";
      e <- parseAexp;
      return (t ++ "+" ++ e);         }

    <|> do {
      <|> do {
        t <- parseAterm;
        return t;    }


      parseAterm :: Parser String
      parseAterm = do {
        f <- parseFactor;
        do {
          symbol "*";
          t <- parseAterm;
          return(f ++ "*" ++ t);
        } <|> do {
          symbol "/";
          t <- parseAterm;
          return(f ++ "/" ++ t);
        } <|> do {
          symbol "^";
          t <- parseAterm;
          return(f ++ "^" ++ t);
        }  <|> do {
          symbol "%";
          t <- parseAterm;
          return(f ++ "%" ++ t);
        }  <|>
        return f;}

      parseFactor :: Parser String
      parseFactor =
      do {
        symbol "(";
        e <- parseAexp;
        symbol ")";
        return ("(" ++ e ++ ")")
      } <|> do {
        symbol "-";
        f <- parseFactor;
        return("-" ++ f);
      } <|> do {
        i <- identifier;
        return i;
      } <|> do {
        i <- integer;
        return (show i);    }
      symbol "-";
      e <- parseAexp;
      return (t ++ "-" ++ e);
  } }
```

As we can see, every function simply returns the string it has read, and so as a result of ParseAexp we obtain the string corresponding to the exact arithmetic expression contained in the input string.

2.2.3 Data Structures

Here we report the implementation for the data structure and their parsing strategy.

Implementation for Arrays

```
array :: Parser [Int]
array =  (do symbol "{"
  a <- arrayItems
  symbol "}"
return a)

arrayItems :: Parser [Int]
arrayItems = (do a <- aexp
symbol ","
as <- arrayItems
return ([a] ++ as))

<|>
(do a <- aexp
return [a])
```

The relative parsing strategy:

```
parseArray :: Parser String
parseArray = (do symbol "{"
  a <- parseArrayItems
  symbol "}"
return ("{" ++ a ++ "}"))
<|> identifier

parseArrayItems :: Parser String
parseArrayItems = (do a <- parseAexp
symbol ","
b <- parseArrayItems
return (a ++ "," ++ b))
<|> parseAexp
```

Implementation for Matrices

```
matrix :: Parser [[Int]]
matrix = (do symbol "{"
  a <- matrixItems
  symbol "}"
return a)
matrixItems :: Parser [[Int]]
matrixItems = do {
  do {
    a <- array;
    symbol ",";
    as <- matrixItems;
    return ([a] ++ as);       }
```

```
    <|>
    do{

      a <- array;
      return ([a]);    }}



```

The relative parsing strategy:

```
parseMatrix :: Parser String
parseMatrix = do{do {symbol "{";
      a <- parseMatrixItems;
      symbol "}";
    return ("{" ++ a ++ "}");}
  <|> do {identifier }}
parseMatrixItems :: Parser String
parseMatrixItems = do {do { a <- parseArray;
    symbol ",";
    b <- parseMatrixItems;
    return (a ++ "," ++ b);}
  <|> do { parseArray;}}


```

Creating matrices and arrays

Methods for creating matrices and arrays filled with zeros have been implemented. They are:

```
createArray :: [Int] -> Int -> [Int]
createArray [] 0 = []
createArray [] col = [0] ++ (createArray [] (col-1))

createMatrix :: [[Int]] -> Int -> Int -> [[Int]]
createMatrix [[]] 0 col = []
createMatrix [[]] row col = (createArray [] col) : createMatrix [[]] (row
-1) col

```

2.2.4 Boolean expressions

Here we give the implementation for the Boolean expression, that is, how our interpreter recognizes the **bexp** defined in our grammar.

Implementation for Bexp

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


bterm :: Parser Bool
```

```
bterm = (do
f0 <- bfactor
symbol "AND"
f1 <- bterm
return (f0 && f1)
<|>
bfactor)


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


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

```
    a1 <- aexp
    return (a0 /= a1))
```

Parser for Bexp

```
consumeBexp :: Parser String
consumeBexp = do
b <- parseBexp
return b


parseBexp :: Parser String
parseBexp = do {
  p1 <- parseBexp2;
  symbol "OR";
  p2 <- parseBexp;
  return (p1 ++ " OR " ++ p2);
} <|> do {
  p <- parseBexp2;
  return p;    }


parseBexp2 :: Parser String
parseBexp2 =
do {
  p1 <- parseBexp3;
  symbol "AND";
  p2 <- parseBexp2;
  return (p1 ++ " AND " ++ p2);
} <|> do {
  c <- parseCompareTo;
  return c;
}    <|>
do {
  symbol "True";
  return "True";
}    <|>
do {
  symbol "False";
  return "False";
}    <|>
do {
  i <- identifier;
  return i;
} <|> do {
  symbol "!";
  p <- parseBexp3;
  return ("!" ++ p)
}


parseBexp3 :: Parser String
parseBexp3 =
do {
  symbol "(";
  p <- parseBexp;
  symbol ")";
  return ("(" ++ p ++ ")");
} <|> do {
```

```
      c <- parseCompareTo;
      return c;
    }    <|>
    do {
      symbol "True";
      return "True";
    }
    <|> do {
      symbol "False";
      return "False";
    }    <|>
    do {
      i <- identifier;
      return i;
    } <|> do {
      symbol "!";
      p <- parseBexp3;
      return ("!" ++ p)
    }


    parseCompareTo :: Parser String
    parseCompareTo = do {
      a1 <- parseAexp;
      symbol "==";
      a2 <- parseAexp;
      return (a1 ++ "==" ++ a2);
    } <|> do {
      a1 <- parseAexp;
      symbol "<=";
      a2 <- parseAexp;
      return (a1 ++ "<=" ++ a2);
    } <|> do {
      a1 <- parseAexp;
      symbol "<";
      a2 <- parseAexp;
      return (a1 ++ "<" ++ a2);
    } <|> do {
      a1 <- parseAexp;
      symbol ">=";

      a2 <- parseAexp;
      return (a1 ++ ">=" ++ a2);
    } <|> do {
      a1 <- parseAexp;
      symbol ">";
      a2 <- parseAexp;
      return (a1 ++ ">" ++ a2);
    } <|> do {
      a1 <- parseAexp;
      symbol "!=";
      a2 <- parseAexp;
      return (a1 ++ "!=" ++ a2);    }
```

As we notice, the **parseCompareTo** function uses the **parseAexp** function to properly obtain the exact string which expresses the arithmetic expression used in the boolean expression.

2.2.5 Commands

Since the definition for commands is very rich and complex, here we describe the more general notion of **command** while in the following sections we give an accurate description for every possible command provided by our grammar

Implementation for commands

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

command :: Parser String
command = assignment
<|>
ifThenElse
<|>
While
<|>
forLoop
<|>
repeatUntil
<|>
(do
symbol "skip"
symbol ";")


```

2.2.6 Assignment command

The assignment comes in very different flavours, on the basis of which data type being used, such as arrays, matrices and so on. Inside the implementation of this command we call some functions such as **updateEnv, saveArray, saveMatrix** which will be described in the next Chapter. Since the actions they carry out are quite straightforward, we can stay focused on the description of the grammar implementation details.

```
assignment :: Parser String
assignment = (do
--x:=10;
x <- identifier
symbol ":="
v <- aexp
symbol ";"
updateEnv Variable{name = x, vtype = "Integer", value = [[v]]})
<|> (do
--x:=True;
x <- identifier
symbol ":="
v <- bexp
symbol ";"
updateEnv Variable{name = x, vtype = "Boolean", value =
  [[(fromBoolToInt v)]]})
<|>
-- x:={1,2,3};
```

```
    (do id <- identifier
    symbol ":="
    arr <- array
    symbol ";"
    saveArray id arr)
    <|>
    -- x := {1,2,3}++{4,5,6};
    (do id <- identifier
    symbol ":="
    ar1 <- array
    symbol "++"
    ar2 <- array
    symbol ";"
    saveArray id (ar1 ++ ar2))
    <|>
    -- y:= {1,2,3}; z:={4,5,6}; x := y++z;
    (do id <- identifier
    symbol ":="
    id1 <- identifier

    symbol "++"
    id2 <- identifier
    symbol ";"
    arr1 <- readValuesArray id1
    arr2 <- readValuesArray id2
    saveArray id (arr1++arr2))
    <|>
    -- createArray(v[5]);
    (do symbol "createArray("
    id <- identifier
    symbol "["
    col <- aexp
    symbol "])"
    symbol ";"
    saveArray id (createArray [] col))
    <|>
    -- x={{1,2,3},{1,2,3}}
    (do id <- identifier
    symbol ":="
    mat <- matrix
    symbol ";"
    saveMatrix id mat)
    <|>
    -- x:={{1,2,3},{1,2,3}}; x[1][1]:= 3;
    (do id <- identifier
    symbol "["
    row <- aexp
    symbol "]"
    symbol "["
    col <- aexp
    symbol "]"
    symbol ":="
    val <- aexp
    symbol ";"
    mat <- readValues id

    saveMatrix id (writeMatrix mat val row col))
    <|>
    -- x:={{1,2,3},{4,5}}; y:=x[1][1];
    (do id <- identifier
    symbol ":="
    id2 <- identifier
```

```
    symbol "["
    row <- aexp
    symbol "]"
    symbol "["
    col <- aexp
    symbol "]"
    symbol ";"
    val <- readMatrixVariable id2 row col;
    updateEnv Variable{name = id, vtype = "Integer", value =
      [[val]]})
    <|>
    -- x:={1,2,3}; x[1]:= 3;
    (do id <- identifier
    symbol "["
    col <- aexp
    symbol "]"
    symbol ":="
    val <- aexp
    symbol ";"
    arr <- readValuesArray id
    saveArray id (updateArray arr val col))
    <|>
    -- x:={1,2,3}; y:= {1,2,3}; x[1]:=y[2];
    (do id <- identifier
    symbol "["
    index <- aexp
    symbol "]"
    symbol ":="
    id2 <- identifier

    symbol "["
    index2 <- aexp
    symbol "]"
    symbol ";"
    arr <- readValuesArray id
    val <- readArray id2 index2
    saveArray id (updateArray arr val index))
    <|>
    -- createMatrix(v[5][5]);
    (do symbol "createMatrix("
    id <- identifier
    symbol "["
    row <- aexp
    symbol "]"
    symbol "["
    col <- aexp
    symbol "])"
    symbol ";"
    saveMatrix id (createMatrix [[]] row col))
    <|>
    -- x:={1,2,3}; push(x,2);
    (do
    symbol "push("
    id <- identifier
    symbol ","
    val <- aexp
    symbol ");"
    stck <- readValuesArray id
    saveStack id ([val] ++ stck)
    )
    <|>
    -- x:={1,2,3}; pop(x);
```

```
(do
symbol "pop("
id <- identifier

symbol ");"
stck <- readValuesArray id
saveStack id (tail stck)
)
<|>
-- x:={1,2,3}; enqueue(x,2);
(do
symbol "enqueue("
id <- identifier
symbol ","
val <- aexp
symbol ");"
q <- readValuesArray id
saveQueue id ([val] ++ q)
)
<|>
-- x:={1,2,3}; dequeue(x);
(do
symbol "dequeue("
id <- identifier
symbol ");"
q <- readValuesArray id
saveQueue id (init q)            )
```

Parser for assignment

As done before, these parsers just return the string representation of the input, without consume it.

```
parseAssignment :: Parser String
parseAssignment = -- y := x[1]
(do id <- identifier
symbol ":="
id2 <- identifier
symbol "["
index <- parseAexp

";")
symbol "]"
symbol ";"
return $ id ++ ":=" ++ id2 ++ "[" ++ index ++ "]" ++
<|>
-- x[1] := y[1];
(do id <- identifier
symbol "["
index <- parseAexp
symbol "]"
symbol ":="
id2 <- identifier
symbol "["
index2 <- parseAexp
symbol "]"
symbol ";"
return $ id ++ "[" ++ index ++ "]:=" ++ id2 ++ "[" ++
index2 ++ "]" ++ ";")
```

```
    <|>
    (do id <- identifier
    symbol ":="
    a <- parseAexp
    symbol ";"
    return $ id ++ ":=" ++ a ++ ";")
    <|>
    -- x[1] := y
    (do id <- identifier
    symbol "["
    index <- parseAexp
    symbol "]"
    symbol ":="
    val <- parseAexp
    array <- parseArray
    symbol ";"
    return $ id ++ "[" ++ index ++ "]:=" ++ val ++ ";" )
    <|>

    ")"++";")
    <|>
    -- x:={1,2,3}
    (do id <- identifier
    symbol ":="
    arr <- parseArray
    symbol ";"
    return $ id ++ ":=" ++ arr ++ ";" )
    <|>
    -- x={{1,2,3},{1,2,3}}
    (do id <- identifier
    symbol ":="
    m <- parseMatrix
    symbol ";"
    return $ id ++ ":=" ++ m ++ ";" )
    <|>
    -- x = y++z
    (do id <- identifier
    symbol ":="
    ar1 <- parseArray
    symbol "++"
    ar2 <- parseArray
    symbol ";"
    return $ id ++ ":=" ++ ar1 ++ "++" ++ ar2 ++ ";")
    <|>
    -- x:=qsort ({1,4,5});
    (do id <- identifier
    symbol ":="
    symbol "qsort"
    symbol "("
    id2 <- parseArray
    symbol ")"
    symbol ";"
    return $ id ++ ":=" ++ "qsort" ++ "(" ++ id2 ++

    -- x={{1,2,3},{1,2,3}};
    (do id <- identifier
    symbol ":="
    symbol "{"
      m <- parseMatrix
      symbol "}"
    symbol ";"
    return $ id ++ ":=" ++ "{" ++ m ++ "}"++";")
```

```
<|>
--y={{1,2,3},{1,2,3}}; x:=y[1][1];
(do id <- identifier
symbol ":="
id2 <- identifier
symbol "["
row <- parseAexp
symbol "]"
symbol "["
col <- parseAexp
symbol "]"
symbol ";"
return $ id ++ ":=" ++ id2 ++"[" ++ row ++ "]" ++"["
++ col ++ "]" ++";")
<|>
-- x={{1,2,3},{1,2,3}}; x[1][1]:= 3;
(do id <- identifier
symbol "["
row <- parseAexp
symbol "]"
symbol "["
col <- parseAexp
symbol "]"
symbol ":="
val <- parseAexp
symbol ";"
return $ id ++ "[" ++ row ++ "]" ++"[" ++ col ++ "]"
++ ":=" ++ val ++";")

<|>
-- x:={1,2,3}; push(x,2);
(do
symbol "push("
id <- identifier
symbol ","
val <- parseAexp
symbol ");"
return $ "push(" ++ id ++ "," ++ val ++ ");"
)
<|>
-- x:={1,2,3}; pop(x);
(do
symbol "pop("
id <- identifier
symbol ");"
return $ "pop(" ++ id ++ ");"
)
<|>
-- x:={1,2,3}; enqueue(x,2);
(do
symbol "enqueue("
id <- identifier
symbol ","
val <- parseAexp
symbol ");"
return $ "enqueue(" ++ id ++ "," ++ val ++ ");"
)
<|>
-- x:={1,2,3}; dequeue(x);
(do
symbol "dequeue("
id <- identifier
```

```
    symbol ");"
    return $ "dequeue(" ++ id ++ ");"

    )
    <|>
    -- createArray(v[5]);
    (do symbol "createArray("
    id <- identifier
    symbol "["
    col <- parseAexp
    symbol "])"
    symbol ";"
    return $ "createArray(" ++ id ++ "[" ++ col ++ "]);")
    <|>
    -- createMatrix(v[5][5]);
    (do symbol "createMatrix("
    id <- identifier
    symbol "["
    row <- parseAexp
    symbol "]"
    symbol "["
    col <- parseAexp
    symbol "])"
    symbol ";"
    return $ "createMatrix(" ++ id ++ "[" ++ row ++ "]" ++
    "[" ++ col ++ "]);")

```

2.2.7 If-Then-Else

Here we give the implementation details and parsing strategy for If-Then-Else command

Implementation of If-Then-Else

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

```
  program
  symbol "}"
return "")
<|>
return "")        )
```

Parser for If-Then-Else

```
parseIfThenElse :: Parser String
parseIfThenElse = do {
  symbol "if";
  symbol "(";
  b <- parseBexp;
  symbol ")";
  symbol "{";
    p1 <- parseProgram;
    symbol "}";
  do {
    symbol "else";
    symbol "{";
      p2 <- parseProgram;
      symbol "}";
    return ("if(" ++ b ++ "){" ++ p1 ++ "}else{" ++ p2 ++ "}")
  }
  <|>
  return ("if(" ++ b ++ "){" ++ p1 ++ "}");}

```

2.2.8 While command

Here we report the implementation fro the while command. As mentioned before, now becomes clear the aim of have separate parsers for consuming the input or just to read it. Here we can see that the two parsers work together in order to loop correctly over a determined set of instructions.

```
 while :: Parser String
 while = do
 w <- parseWhile
 repeatWhile w
 symbol "while"
 symbol "("
 b <- bexp
 symbol ")"
 symbol "{"
   if (b) then (
   do
   program
   symbol "}"
 repeatWhile w
 while)
 else ( do
 parseProgram
 symbol "}"
return "")

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

Parsing the While

As we can see, the function parseWhile returns the string "w" containing the string related to the while-loop block, this string is then printed again, using repeatWhile, in the input text and its execution starts. If the condition is met, the whole process continues until the condition becomes false, then the rest of the block is read without execution using parseProgram.

```

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

2.2.9 Repeat Until

The repeatUntil command is basically the opposite of the while-loop command. The difference is related to the fact that the portion of the program contained in RepeatUntil is executed at least one time, and its execution keeps looping until the condition of the until part is met. Its implementation is:

```
repeatUntil :: Parser String
repeatUntil = do
w <- parseRepeatUntil;
parseProgram
symbol "}"
return "")

repeatWhile w;
symbol "repeat";
symbol "{";
program;
symbol "}";
symbol "until";
symbol "(";
b <- bexp;
symbol ")";
if (b) then do{
repeatWhile w;
repeatUntil;
} else do{
return ""; }
```

Parser for Repeat-Until

```
parseRepeatUntil :: Parser String
parseRepeatUntil = do{
symbol "repeat";
symbol "{";
p <- parseProgram;
symbol "}";
symbol "until";
```

```
symbol "(";
b <- consumeBexp;
symbol ")";
return("repeat {" ++ p ++ "} until " ++ "( " ++ b ++ " ) " );}
```

2.2.10 For command

Implementation of For command

```
forLoop :: Parser String
forLoop = do
f <- parseForLoop
repeatWhile f
program
return ""
```

Parser of For Command

As we can see in the assignment we call the parsing method “parseForLoop” and this will be repeated as in the while method:

```

parseForLoop :: Parser String
parseForLoop = do {
symbol "for";
symbol "(";
a <- parseAssignment;
b <- parseBexp;
symbol ";";
x <- identifier;
symbol "++";
symbol ")";
symbol "{";
p <- parseProgram;
symbol "}";
return (a ++ " while(" ++ b ++ ") {" ++ p ++ x ++ ":=" ++ x ++
"+1;}");
} <|> do {
symbol "for";
symbol "(";
a <- parseAssignment;
b <- parseBexp;
symbol ";";
x <- identifier;
symbol "--";
symbol ")";
symbol "{";
p <- parseProgram;
symbol "}";
return (a ++ " while(" ++ b ++ ") {" ++ p ++ x ++ ":=" ++ x ++
"-1;}");
} <|> do {
symbol "for";
symbol "(";
a <- parseAssignment;
b <- parseBexp;
symbol ";";
c <- parseAssignment;
symbol ")";
symbol "{";
```

```
p <- parseProgram;
symbol "}";
return (a ++ " while(" ++ b ++ ") {" ++ p ++ c ++ "}");}
```

Capitolo 3

Environment management

As stated before, we need an environment to store variables and their values, in order to produce **state transitions** by the use of **commands**. Here we report again the definition of **Variable** data type and its relative Environment:

```
data Variable = Variable {
name :: String,
vtype :: String,
value :: [[Int]]
} deriving Show.

type Env = [Variables].
```

We need the clause **"deriving Show"** to assure that the variables can be displayed on the screen. Now, given a variable **x**, we can access its fields by simply placing before the name of the field we are interested in, i.e. to access the name we can type **name x**, and so on. Notice that the **value** field is a matrix, the choice is motivated by the fact that our grammar manipulates complex data structure. In particular arrays, stacks and queues will be treated as one dimensional matrices equipped with different operations, accordingly to the data type.

3.1 Reading the Environment

Once we have set our definition of environment, we need ways to deal with the various operations that concern the variables, and so the environment itself. First of all, we need ways to simply read a variable, which means defining a function that given the identifier for that variable is able to return the associated value:

```

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

In order to find the variable that needs to be read, the function performs a search in all the environment, in order to verify if there is some variable that has the same identifier; this is done through a scan of the list of variables contained in the environment:

```
searchVariable :: Env -> String -> [[Int]]
searchVariable [] queryname = []
searchVariable (x:xs) queryname = if (name x) == queryname
then [[((value x) !! 0) !! 0]] else searchVariable xs queryname
```

It is worth to notice that the returned value is a list of a list, this because our IMP language also deals with arrays and matrices.

3.1.1 Reading data structures

Now we can specify functions for reading data structures in a similar way.

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


readMatrixVariable :: String -> Int -> Int -> Parser Int
readMatrixVariable name j k = P (\env input -> case searchMatrixVariable
env name j k of
[[]] -> []
[[value]] -> [(env,value, input)])
```

As we see, the main difference is the presence of the indices j and k, that are necessary to identify the elements contained in arrays and matrices. So, we modify the function to search variables in this way:

```
searchArrayVariable :: Env -> String -> Int -> [[Int]]
searchArrayVariable [] queryname j = [[]]
searchArrayVariable (x:xs) queryname j = if ((name x) == queryname)
then [[((value x) !! 0) !! j]] else searchArrayVariable xs
queryname j

searchMatrixVariable :: Env -> String -> Int -> Int -> [[Int]]
searchMatrixVariable [] queryname j k = []
searchMatrixVariable (x:xs) queryname j k = if ((name x) == queryname)
then [[((value x) !! j) !! k]] else searchMatrixVariable xs queryname jk
```

Reading a Matrix

This function allows to read a matrix variable, giving a Parser as output.

```
readValues :: String -> Parser [[Int]]
readValues name = P (\env input -> case searchValues env name of
[[]] -> []
(xs:ys) -> [(env, xs:ys , input)])

searchValues :: Env -> String -> [[Int]]
searchValues [] queryname = []
searchValues (x:xs) queryname = if (name x) == queryname then value x else
↪ searchValues xs queryname
```

Reading an Array

This function allows to read an array variable, giving a Parser as output.

```
readValuesArray :: String -> Parser [Int]
readValuesArray name = P (\env input -> case searchValuesArray env name of
[] -> []
(xs:ys) -> [(env, xs:ys , input)])

searchValuesArray :: Env -> String -> [Int]
searchValuesArray [] queryname = []
searchValuesArray (x:xs) queryname = if (name x) == queryname
then ((value x) !! 0) else searchValuesArray xs queryname
```

3.2 Updating the environment

Now we need functions to update the environment, which means to either add a new variable or replace one. This is made in the following way:

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

Notice that we do not replace the value of the variable, instead we directly overwrite it through a replacement. Some functions have been implemented to easily manage the access of arrays and matrices that write in a specific position of the value of variable (that is a list of a list of integer):

```
updateArray ::[Int] -> Int -> Int ->[Int]
updateArray [] val col = []
updateArray (x:xs) val 0 = val:xs
updateArray (x:xs) val col = x:(updateArray xs val (col-1))
updateMatrix ::[[Int]] -> Int -> Int -> Int ->[[Int]]
updateMatrix [[]] val row col = [[]]
updateMatrix (xs:ys) val 0 col = (updateArray xs val col):ys
updateMatrix (xs:ys) val row col = xs:(updateMatrix ys val (row-1) col)
```

3.2.1 Saving data structures

In order to improve code readability, we implemented some functions to save different data structure through the same call to function **updateEnv**. The definitions follows:

```
saveArray :: String -> [Int] -> Parser String
saveArray var val = updateEnv Variable{name=var, vtype="Array", value= [val]}

saveStack :: String -> [Int] -> Parser String
saveStack var val = updateEnv Variable{name=var, vtype="Stack", value= [val]}

saveQueue :: String -> [Int] -> Parser String
saveQueue var val = updateEnv Variable{name=var, vtype="Queue", value= [val]}

saveMatrix :: String -> [[Int]] -> Parser String
saveMatrix var val = updateEnv Variable{name=var, vtype="Matrix", value= val}
```

As we can see, the function call is the same, the only difference is on the “vtype” attribute and on the input that is a list of lists for the matrix type, but for the others is only a list.

Capitolo 4

Tests and conclusions

Finally we can parse strings matching our grammar with the functions described until now. The **parser** function is responsible of the interaction with user and accumulates all the user inputs verifying that they respect the grammar through the call to **parseProgram** and then, if the function succeeds, the function **program** is invoked to parse and compute the effective values. This process continues until the user decides to quit the interpreter.

There are some keywords with which we can display different information, they are:

- **printmem**: displays the memory and all the parsed code since the start of interpreter,
- **syntax**: displays the formation rules for the grammar,
- **examples**: displays some examples to the user,
- **quit | exit | bye** : interrupts the interpreter and clears the memory,
- **help**: displays all the keywords listed until know.

If none of these keywords is inserted, then the interpreter checks if the string matches with the grammar, otherwise will display an error message and continues to wait for input. For the sake of brevity we report only the final part of the parser function, which manages the calls to the main parsers.

```

case parse parseProgram [] line of
[] -> do
putStrLn "Syntax error! Please read the syntax typing \"help\" "
parser xs
otherwise -> do
parser(xs ++ line)
```

As we can see, the parser appends every matched input line to the previous ones by a recursive call. In this way, as already said, we accumulate all the parsed code and only when **printmem** is invoked the entire parsed code is consumed, resulting in the final Environment.

```

parser :: String -> IO String
parser xs =  do
```

```
putStr "IMPInt#>"
hFlush stdout
line <- getLine
case line of
"printmem" ->do
putStrLn  ""
putStrLn  " ***** Parsed code ***** "
if xs == [] then putStrLn ""  else
putStrLn (getCode (parse parseProgram [] xs))
putStrLn ""
putStrLn "***** Memory *****"
putStrLn (getMemory (parse program [] xs))
putStrLn ""
parser(xs)
```

The two last functions that are responsible for the displaying of information about the parsed code and memory are **getCode** and **getMemory**. The former takes as input the the result of the Parser **parseProgram**, while the latter takes as input the the result of the Parser **program**, extracting from its environment all the varibles information. Their implementation is given below:

```

getCode :: [(Env, a, String)] -> a
getCode [(_, x, _)]  =  x


getMemory :: [(Env, a, String)] -> String
getMemory [] = " Invalid input\n"

getMemory [(x:xs, parsedString, "")] = case Main.getVarType x of
"Boolean" -> case Main.getVarValue x of
[[1]] -> " Boolean: " ++ (Main.getVarvname x) ++ " = True\n" ++ (getMemory [(xs,parsedString,"")])

[[0]] -> " Boolean: " ++ (Main.getVarvname x) ++ " = False\n" ++ (getMemory [(xs,parsedString,"")])


"Integer" -> " Integer: " ++ (Main.getVarvname x) ++ " = " ++ (show ((Main.getVarValue x !! 0) !! 0) ) ++ "\n" ++ (getMemory[(xs,parsedString,"")])


"Array" -> " Array: " ++ (Main.getVarvname x) ++ " = " ++ (show (Main.getVarValue x !! 0) ) ++ "\n" ++ (getMemory[(xs,parsedString,"")])


"Matrix" -> " Matrix: " ++ (Main.getVarvname x) ++ " = " ++ (show (Main.getVarValue x)) ++ "\n" ++ (getMemory[(xs,parsedString,"")])


"Stack" -> " Stack: " ++ (Main.getVarvname x) ++ " = " ++ (show (Main.getVarValue x !! 0) ) ++ "\n" ++ (getMemory[(xs,parsedString,"")])


"Queue" -> " Queue: " ++ (Main.getVarvname x) ++ " = " ++ (show (Main.getVarValue x !! 0) ) ++ "\n" ++ (getMemory[(xs,parsedString,"")])

"" -> "Empty data type but we have following values: " ++ (Main.getVarvname x) ++ " = " ++ (show (Main.getVarValue x)) ++ "\n" ++ (getMemory[(xs,parsedString,"")])

```

```haskell

getMemory [(env, parsedString, notParsedString)] = case notParsedString of
"" -> ""
otherwise -> " Error (unused input '" ++ notParsedString ++ "')\n" ++ getMemory [(env,parsedString, "")]


getVarType :: Variable -> String
getVarType = vtype

getVarvname :: Variable -> String
getVarvname = vname

getVarValue :: Variable -> [[Int]]
getVarValue = value
```

4.1 Testing results

In this section we report some examples of usage of the interpreter and their relative results. As we can see from the images, the parsed code is consumed only when **printmem** is invoked.

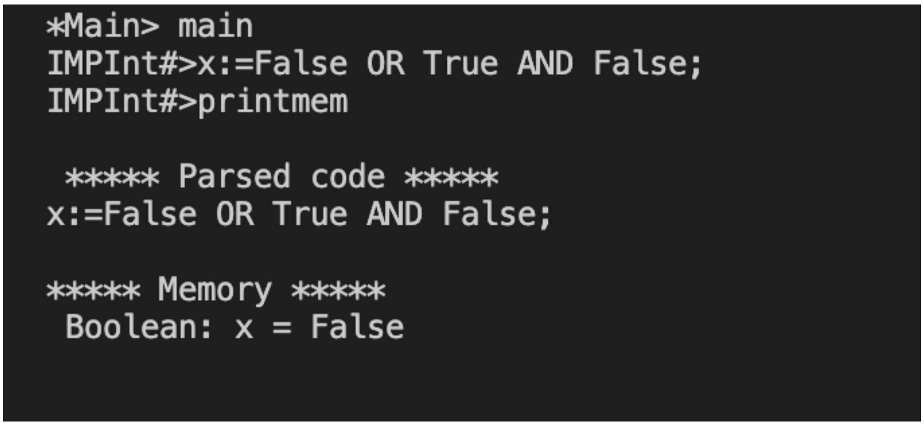
4.1.1 Arithmetical expressions

Below an example of parsing an arithmetical expression, and the state of the memory after the execution.

```
*Main> main
IMPInt#>x:=1+1;
IMPInt#>y:=x-5;
IMPInt#>z:=x+y+(-1);
IMPInt#>a:=3^5;
IMPInt#>b:=5%20;
IMPInt#>c:=3*4 + 4/2 + (10*3)/4;
IMPInt#>d:= x - z + (b*a)/c;
IMPInt#>printmem

 ***** Parsed code *****
x:=1+1;y:=x-5;z:=x+y+(-1);a:=3^5;b:=5%20;c:=3*4+4/2+(10*3)/4;d:=x-z+(b*a)/c;

***** Memory *****
 Integer: x = 2
 Integer: y = -3
 Integer: z = -2
 Integer: a = 243
 Integer: b = 5
 Integer: c = 21
 Integer: d = -53
```

Figura 4.1: Aexp execution results

4.1.2 Boolean expressions

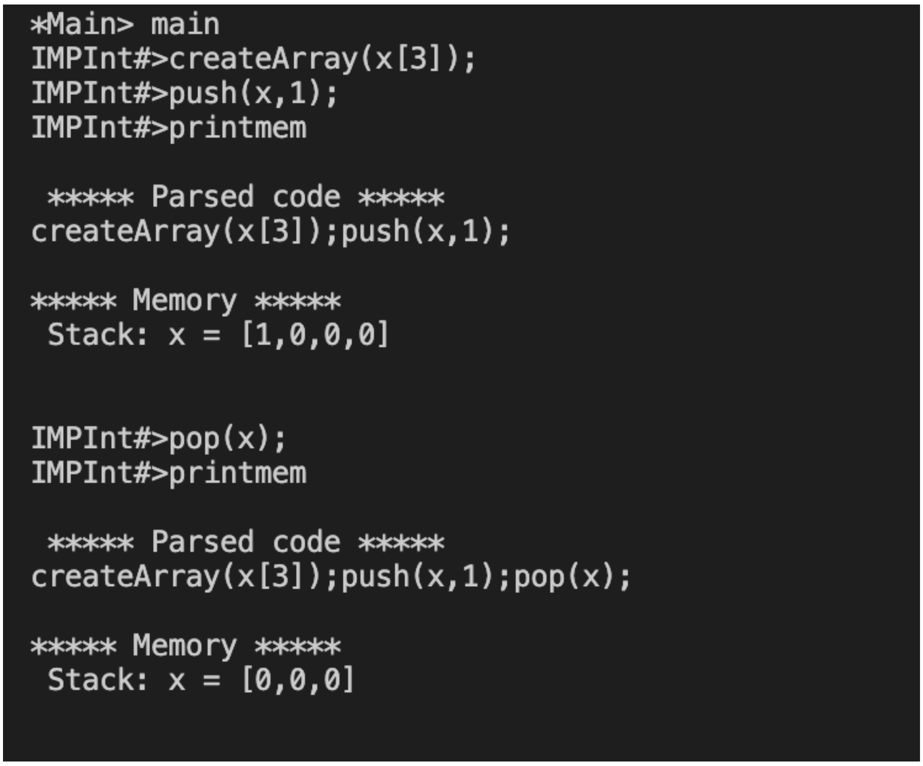


```
*Main> main
IMPInt#>x:=False OR True AND False;
IMPInt#>printmem

 ***** Parsed code *****
x:=False OR True AND False;

***** Memory *****
 Boolean: x = False
```

Figura 4.2: Bexp execution results

4.1.3 Data structures

As we can see from the images below, the stack operations respect the paradigm of LIFO while the queues ones that of FIFO.



```
*Main> main
IMPInt#>createArray(x[3]);
IMPInt#>push(x,1);
IMPInt#>printmem

 ***** Parsed code *****
createArray(x[3]);push(x,1);

***** Memory *****
 Stack: x = [1,0,0,0]


IMPInt#>pop(x);
IMPInt#>printmem

 ***** Parsed code *****
createArray(x[3]);push(x,1);pop(x);

***** Memory *****
 Stack: x = [0,0,0]
```

Figura 4.3: Stack manipulation

```
*Main> main
IMPInt#>x := {0,1,2};
IMPInt#>enqueue(x,4);
IMPInt#>printmem

 ***** Parsed code *****
x:={0,1,2};enqueue(x,4);

***** Memory *****
 Queue: x = [4,0,1,2]


IMPInt#>dequeue(x);
IMPInt#>printmem

 ***** Parsed code *****
x:={0,1,2};enqueue(x,4);dequeue(x);

***** Memory *****
 Queue: x = [4,0,1]
```

Figura 4.4: Queue manipulation

4.1.4 Commands

In this sectin we report the results obtained by the execution of commands provided by our grammar. Notice that also the assignment is tested.

```
*Main> main
IMPInt#>x := 3; y := 4; if (x <= 4) { x := 76; } else { x := 88; }
IMPInt#>printmem

 ***** Parsed code *****
x:=3;y:=4;if(x<=4){x:=76;}else{x:=88;}

***** Memory *****
 Integer: x = 76
 Integer: y = 4
```

Figura 4.5: If-Then-Else execution results

```
*Main> main
IMPInt#>n := 0; i := 0; while (i < 10) {n := n + 1; i := i + 1;}
IMPInt#>printmem

 ***** Parsed code *****
n:=0;i:=0;while(i<10){n:=n+1;i:=i+1;}

***** Memory *****
 Integer: n = 10
 Integer: i = 10
```

Figura 4.6: While execution results

```
*Main> main
IMPInt#>x:=0; repeat{x:=x+1;} until (x<10)
IMPInt#>printmem

 ***** Parsed code *****
x:=0;

***** Memory *****
 Integer: x = 10
```

Figura 4.7: IRepeat Until execution results

```
*Main> main
IMPInt#> a:=0; for (i:=10; i>0; i--)  {a:=a+1;}
IMPInt#>printmem

 ***** Parsed code *****
a:=0;i:=10; while(i>0) {a:=a+1;i:=i-1;}

***** Memory *****
 Integer: a = 10
 Integer: i = 0
```

Figura 4.8: For-Loop execution