

Università degli studi di Bari “Aldo Moro”

Master’s Degree in Computer Science

*Formal Methods in Computer Science*

# LIMP: Lightweight Imperative Language Interpreter



*Student:*

Gianfranco Demarco

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Grammar</b>	<b>3</b>
2.1	BNF . . . . .	3
2.2	Haskell . . . . .	6
<b>3</b>	<b>Parser</b>	<b>10</b>
3.1	Functor . . . . .	11
3.2	Applicative . . . . .	11
3.3	Monads . . . . .	12
3.4	Alternative . . . . .	14
3.5	Basic parsers . . . . .	14
3.6	Arithmetic Expressions . . . . .	17
3.7	Boolean Expressions . . . . .	19
3.8	Commands . . . . .	20
3.9	Arrays . . . . .	23
3.10	Stacks . . . . .	26
3.11	Queues . . . . .	26
<b>4</b>	<b>Interpreter</b>	<b>28</b>
4.1	Arithmetic Expression Evaluation . . . . .	29
4.2	Boolean Expression Evaluation . . . . .	30
4.3	Array Expression Evaluation . . . . .	32
4.4	Program Execution . . . . .	34
<b>5</b>	<b>Using the interpreter</b>	<b>38</b>
5.1	Interactive shell . . . . .	39
5.2	Running files . . . . .	41
5.2.1	Bubble Sort . . . . .	41
5.2.2	Stack . . . . .	43
5.2.3	Queue . . . . .	44
5.2.4	Other test programs . . . . .	45



# List of Figures

5.1	Execution of a single command in the LIMP interactive shell . . . . .	39
5.2	Execution of multiple command in the LIMP interactive shell. The Environment from previous command is retained. A comment is present between the commands . . . . .	40
5.3	Execution of a program from source file . . . . .	41
5.4	Result of the bubble sort program . . . . .	42
5.5	Result of the test program for stacks . . . . .	43
5.6	Result of the test program for queues . . . . .	44

# Chapter 1

## Introduction

LIMP - Lightweight Imperative is an extension of IMP, the imperative language defined by Glynn Winskel in its book "The Formal Semantics of Programming Languages".

It implements the basic types and control structures of an imperative language. Moreover, it is provided with the basic implementation of some data structures.

### Types:

- **Int**: represents Integer numbers, both positive and negative
- **Bool**: represents Boolean values: True and False
- **Array**: represent the Array data structure, implemented as a collection of Int values
- **Stack**: represent a LIFO data structure
- **Queue**: represent a FIFO data structure

### Control Structures:

- **Declaration**: a Declaration is a command that tells the LIMP interpreter to assign some memory to a particular variable, of one of the types listed above
- **Assignment**: an Assignment is a statement used to give a value to a variable already defined; it has many forms, depending on the type of the variable (Int and Bool have simple assignments, Arrays can have single-value and full assignment, Stack and Queue have their own assignment mechanisms)

- **Skip**: represents a statement that does nothing; it is mainly an internal tool for representing things like if statements without the else statement and comments
- **IfElse**: represents a conditional block, composed by a Boolean condition and two subprograms; if the condition is evaluated to True, the first subprogram is executed; otherwise, the second program is executed.
- **While**: represent a loop, composed by a Boolean condition and a subprogram; the subprogram is executed until the condition becomes False.

This document goes through every detail of the implementation of the LIMP language. The main components here discussed are:

- **Grammar**: The grammar defines the legal syntax for the language. This syntax provides an internal representation of the source program, which is executed from the interpreter.
- **Parser**: The parser is a program that takes the source code as an input, and returns the internal tree representation of the code, based on the Grammar of the language. If the source is well-formed, the Parser succeeds and returns a tree representation; otherwise it fails and an error is raised.
- **Interpreter**: The interpreter is a computer program that directly executes the instruction written in a high-level language, using another language. It takes the tree provided by the parser and executes each command. These commands have the effect of updating an internal memory, called Environment, which represents the state of the program: it is the set of the variables declared. In this case, the high-level language is the LIMP language and the interpreter is written in Haskell.

# Chapter 2

## Grammar

### 2.1 BNF

The first informal listing of the grammar of LIMP is given using the BNF (Backus-Naur Form). It is presented in a top-down perspective.

```
1
2 program ::=    <command> | <command> <program>
3
4 command ::=    <skip> ";"
5                | <ifElse> ";"
6                | <while> ";"
7                | <boolDeclare> ";"
8                | <arithDeclare> ";"
9                | <boolAssign> ";"
10               | <arithAssign> ";"
11               | <arrayDeclare> ";"
12               | <arrayAssign> ";"
13               | <arrayFullAssign> ";"
14               | <stackDeclare> ";"
15               | <stackPush> ";"
16               | <stackPop> ";"
17               | <queueDeclare> ";"
18               | <queueEnqueue> ";"
19               | <queueDequeue> ";"
20
21
22 skip ::= "skip" | "/"* [<character>]* "/"
23
24
25 ifElse ::= "if" "("<bExp>)" "{" <program> "}"
26          | "if" "("<bExp>)" "{" <program>}" "else" "{" <program> "}"
27
28 while ::= "while" "("<bExp>)" "{" <program> "}"
29
30
31
```

```

32 boolDeclare ::= "bool" <identifier> "=" <bExp> ";"
33
34 bExp ::= <bTerm> ["Or" <bTerm>]*
35
36 bTerm ::= <bFact> ["And" <bFact>]*
37
38 bFact ::= True
39         | False
40         | "Not" <bExp>
41         | "(" <bExp> ")"
42         | "empty" <identifier>
43         | "qempty" <identifier>
44         | "bool(" <identifier> ")"
45         | <aExp> < <aExp>
46         | <aExp> > <aExp>
47         | <aExp> <= <aExp>
48         | <aExp> >= <aExp>
49         | <aExp> == <aExp>
50         | <aExp> != <aExp>
51
52
53 arithDeclare ::= "int" <identifier> "=" <aExp> ";"
54
55 identifier ::= [a-zA-Z_][a-zA-Z_0-9]*
56
57 aExp ::= <aTerm> [{"+"|" -"} <aTerm>]*
58
59 aTerm ::= <aFactor> [{"*"|" /"|" ^"} <aFactor>]*
60
61 aFactor ::= "top" <identifier>
62          | "first" <identifier>
63          | "length" <identifier>
64          | <identifier> "[" <aExp> "]"
65          | <identifier>
66          | <integer> | "(" <aExp> ")"
67
68
69 integer ::= - <natInt> | <natInt>
70
71 natInt ::= <digit> <natInt> | <digit>
72
73 digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
74
75
76
77 arithAssign ::= <identifier> "=" <aExp> ";"
78
79 boolAssign ::= <identifier> "=" <bExp> ";"
80
81
82

```



```

83 arrayDeclare ::= "array" <identifier> "["<aExp>"]"
84   | "array" <identifier> "=" <arrayFull>
85   | "array" <identifier> "=" <arrayScalarProduct>
86   | "array" <identifier> "=" <arrayConcat>
87   | "array" <identifier> "=" <arrayDotProduct>
88
89 arrayFull ::= "[" <aExp> ["," <aExp>]* "]"
90
91 arrayScalarProduct ::= "scalar" <identifier> <aExp>
92
93 arrayConcat ::= "concat" <identifier> <identifier>
94   | <identifier> "++" <identifier>
95
96 arrayDotProduct ::= "dot" <identifier> <identifier>
97
98 arrayAssign ::= <identifier> "[" <aExp> "]" "=" <aExp>
99
100 arrayFullAssign ::= <identifier> "=" <arrayFull>
101   | <identifier> "=" <arrayScalarProduct>
102   | <identifier> "=" <arrayConcat>
103   | <identifier> "=" <arrayDotProduct>
104
105
106 stackDeclare ::= "stack" <identifier>
107
108 stackPush ::= "push" <identifier> <aExp>
109
110 stackPop ::= "pop" <identifier>
111
112
113 queueDeclare ::= "queue" <identifier>
114
115 queueEnqueue ::= "enqueue" <identifier> <aExp>
116
117 queueDequeue ::= "dequeue" <identifier>

```

## 2.2 Haskell

Now we discuss in detail the grammar as it is used by the parser and the interpreter. The grammar can be found in the file Grammar.hs of the LIMP source code.

```
data Type =
    IntType Int
  | BoolType Bool
  | ArrayType [Int]
  | StackType [Int]
  | QueueType [Int]
  deriving Show
```

The Type data type represents all of the (variable) types that are supported by LIMP. We can see that the IntType and the BoolType are respectively mapped directly onto Int and Bool types. Instead, Arrays, Stacks and Queues are all implemented as lists of Int.

```
data ArithExpr =
    Constant Int
  | ArithVariable String
  | Add ArithExpr ArithExpr
  | Sub ArithExpr ArithExpr
  | Mul ArithExpr ArithExpr
  | Div ArithExpr ArithExpr
  | Power ArithExpr ArithExpr
  | StackTop String
  | QueueFirst String
  | ArrayLength String
  | ArrayPos String ArithExpr
  deriving Show
```

The ArithExpr data type represents arithmetic expressions, which are those expression that when completely evaluated, will yield an Int value. In addition to the expressions that one is expected to find, like sum, subtraction or division, there are others which derive from the data structure that have been implemented in LIMP:

- **StackTop**: represents the first (top) element of a Stack
- **QueueFirst**: represents the first element of a Queue
- **ArrayLength**: represent the length of an Array

- **ArrayPos**: represents the value retained by an Array at a particular position

We can see that the definition is recursive, i.e. we can have an `ArrayLength` as the first value of an `Add`, and a `StackTop` as the second value. `ArithExpr` can be used in boolean expressions.

```
data BoolExpr =
    Boolean Bool
  | BoolVariable String
  | Lt ArithExpr ArithExpr
  | Gt ArithExpr ArithExpr
  | Eq ArithExpr ArithExpr
  | Neq ArithExpr ArithExpr
  | Lte ArithExpr ArithExpr
  | Gte ArithExpr ArithExpr
  | And BoolExpr BoolExpr
  | Or BoolExpr BoolExpr
  | Not BoolExpr
  | StackEmpty String
  | QueueEmpty String
  deriving Show
```

The `BoolExpr` data type represents boolean expressions, which are those expression that when completely evaluated, will yield `True` or `False`. In addition to the classic boolean expressions, there are others which derive from the data structure that have been implemented in LIMP:

- **StackEmpty**: will be `True` if the referred Stack is empty
- **QueueEmpty**: will be `True` if the referred Queue is empty

`BoolExpr` can be assigned to Boolean variables or can be used as conditions for `If Else` or `While` statements.

```

data ArrayExpr =
    ArrayInit ArithExpr
  | ArrayFull [ArithExpr]
  | ArrayConcat String String
  | ArrayDotProduct String String
  | ArrayScalarProduct String ArithExpr
deriving Show

```

The `ArrayExpr` data type represent all of those expressions which yield an array as result. As these are completely custom, we briefly describe each of them:

- **ArrayInit:** This represent the basic declaration of an Array. In this case, we just define the length of the array: an array of that length will be created and filled with 0s. Unlike other languages (like c), the length of an array is dynamic: it will be overwritten by successive full declaration.
- **ArrayFull:** This represents an entire array, which is a list of arithmetic expressions.
- **ArrayConcat:** This represents the concatenation of two arrays, which is an array composed by all of the values of the first array, followed by all of the values of the second array.
- **ArrayDotProduct:** This represents the dot product of two array, which is an array having at the position *i* the value of the product of the element at position *i* of the first array and the element at position *i* of the second array. This operation can be performed only between arrays of the same size.
- **ArrayScalarProduct:** This represents the product of an array by a scalar, which is an array having at the position *i* the value of the element at position *i* of the original array times the scalar.

```
data Command =  
    Skip  
  | IfElse BoolExpr [Command] [Command]  
  | While BoolExpr [Command]  
  | BoolDeclare String BoolExpr  
  | ArithDeclare String ArithExpr  
  | BoolAssign String BoolExpr  
  | ArithAssign String ArithExpr  
  | ArrayDeclare String ArrayExpr  
  | ArrayAssign String ArithExpr ArithExpr  
  | ArrayFullAssign String ArrayExpr  
  | StackDeclare String  
  | StackPush String ArithExpr  
  | StackPop String  
  | QueueDeclare String  
  | QueueEnqueue String ArithExpr  
  | QueueDequeue String  
deriving Show
```

Finally, these are the commands supported by LIMP. Each of these represent operations that might (or might not) update the Environment, which is the internal state of the program. Apart from the classical control structures, declaration and assignment commands have been here implemented for all of the data structures of LIMP.

It is worth to note that the code blocks which constitute the branches of an IfElse statement and the body of the While statement are represented as a list of Commands, which is the representation of a Program.

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

# Chapter 3

## Parser

The parser is that part of the program that reads the source code with the purpose of building an intermediate representation of the program. It checks the syntax of the code, piece by piece: when a known syntax is recognised, its representation is added to the parser tree and the parsing goes on. If a certain syntax doesn't match any of the rules of the grammar of the language, the parsing is interrupted and an error is returned.

To build a Parser in Haskell for LIMP we introduce the custom type *Parser*.

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

The definition above makes it clear that our Parser is a function that takes a string and returns a list of pairs. The pair comprehends the result of type *a*, and the rest of the string that has to be parsed. This definition allows us to consider as a failed parsing a pair without the first element.

*A parser for things  
Is a function from strings  
To lists of pairs  
Of things and strings  
- Dr Seuss*

To implement our desired functionalities, we need to be able to make the Parser instance of classes. To do so, we slightly change our definition to the following:

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

Then we can implement the function *parse*, that applies the Parser, by simply removing the dummy constructor:

```
parse :: String -> ([Command], String)
parse s = case p s of
  [] -> ([], "")
```

```
[(c, s)] -> (c, s)
where
  (P p) = program
```

The first parsing primitive we define, which will be the building block for all of the other, is *item*, which fails if the string is empty, or succeeds with the first character as the result value otherwise:

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

### 3.1 Functor

The idea of mapping a function over each element of a data structure isn't specific to the type of lists, but can be abstracted further to a wide range of parameterised types. The class of types that support such a mapping function are called *functors*. *Functors* will be useful given the wide usage of the Maybe data type in our program. In Haskell, this concept is captured by the following class declaration in the standard prelude:

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

To make *Parser* into a functor, we need to implement the function *fmap*:

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

That is, *fmap* applies a function to the result value of a parser if the parser succeeds, and propagates the failure otherwise.

### 3.2 Applicative

Applicatives are the tools to allow functions with any number arguments to be mapped, rather than being restricted to functions with a single argument.

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

A version of *fmap* for functions with any desired number of arguments can be constructed in terms of two basic functions, *pure* and (<\*>).

*Pure* converts a value of type *a* into a structure of type *f a*, while (<\*>) is a generalised form of function application for which the argument function, the argument value, and the result value are all contained in *f* structures.

The class of functors that support *pure* and (<\*>) functions are called *applicative functors*, or *applicatives* for short.

The Parser type can then be made into an applicative functor as follows:

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

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

In this case, *pure* transforms a value into a parser that always succeeds with this value as its result, without consuming any of the input string.

In turn, (<\*>) 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.

The applicative machinery automatically ensures that the parser fails if the input string is too short, without the need to detect or manage this ourselves.

### 3.3 Monads

The goals of using Monads is to be able to manage failures in a concise way. Using the *Maybe* data type, a common pattern emerge: we usually map *Maybe* with itself, and *Just x* to some values. Abstracting out this pattern gives a new operator (*>=>*) that is defined as follows:

```
(>=>) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >=> f = case mx of
  Nothing -> Nothing
  Just x -> f x
```



That is, `»=` takes an argument of type  $a$  that may fail and a function of type  $a \rightarrow b$  whose result may fail, and returns a result of type  $b$  that may fail. If the argument fails we propagate the failure, otherwise we apply the function to the resulting value. In this manner, `»=` integrates the sequencing of values of type *Maybe* with the processing of their results. The `»=` operator is often called *bind*, because the second argument binds the result of the first.

Haskell provides a special notation for expressions of the above form, allowing them to be written in a simpler manner as follows:

```
do  x1 <- m1
    x2 <- m2
    .
    .
    .
    xn <- mn
    f x1 x2 ... xn
```

The `do` notation can be used with any applicative type that forms a monad. In Haskell, the concept of a monad is captured by the following built-in declaration:

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

  return = pure
```

That is, a monad is an applicative type  $m$  that supports *return* and `»=` functions of the specified types. The default definition `return = pure` means that *return* is normally just another name for the applicative function *pure*.

Finally, we make the *Parser* type into a monad:

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

That is, the parser  $p \gg f$  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  $f\ v$ , which is then applied to the output string *out* that was produced by the first parser to give the final result.

### 3.4 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. Making a choice between two alternatives isn't specific to parsers, but can be generalised to a range of applicative types. This concept is captured by the following class declaration in the library *Control.Applicative*

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

That is, for an applicative functor to be an instance of the `Alternative` class, it must support `empty` and `</>` primitives of the specified types.

`Empty` represents an alternative that has failed, and `</>` is an appropriate choice operator for the type.

The instance for the `Parser` type is a natural extension of this idea, 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:

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

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

### 3.5 Basic parsers

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

digit :: Parser Char
digit = sat isDigit

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

```
upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

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

First of all, we define a parser `sat p` for single characters that satisfy the predicate `p`. Using `sat` and appropriate predicates from the library `Data.Char`, we can now define parsers for single digits, lower-case letters, upper-case letters, arbitrary letters, alphanumeric characters, and specific characters.

We can define a parser `string xs` for the string of characters `xs`, with the string itself returned as the result value:

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

Our next two parsers, *many p* and *some p*, apply a parser `p` as many times as possible until it fails (some requires at least one success). In fact, these are already provided in the `Alternative` class. Using `many` and `some`, we can now define parsers for identifiers (variable names) comprising a lower-case letter followed by zero or more alphanumeric characters, natural numbers comprising one or more digits, integers and spacing comprising zero or more space, tab, and newline characters:

```
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
```

To handle such spacing, we define a new primitive that ignores any space before and after applying a parser for a token:

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

Using token, we can now define parsers that ignore spacing around identifiers, natural numbers, integers and special symbols:

```
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)
```

## 3.6 Arithmetic Expressions

```
aExp  :: Parser ArithExpr
aExp = do chain aTerm op
      where
        op =
          do
            symbol "+";
            return Add
          <|>
          do
            symbol "-";
            return Sub

aTerm :: Parser ArithExpr
aTerm = do chain aFactor op
      where
        op =
          do
            symbol "*"
            return Mul
          <|>
          do
            symbol "/"
            return Div
          <|>
          do
            symbol "^"
            return Power

aFactor :: Parser ArithExpr
aFactor = do
  value <- integer
  return (Constant value)
  <|>
  do
    symbol "top"
    i <- identifier
    return (StackTop i)
  <|>
  do
    symbol "first"
    i <- identifier
```

```
    return (QueueFirst i)
<|>
do
  symbol "length"
  i <- identifier
  return (ArrayLength i)
<|>
do
  i <- identifier
  symbol "["
  pos <- aExp
  symbol "]"
  return (ArrayPos i pos)
<|>
do
  i <- identifier
  return (ArithVariable i)
<|>
do
  symbol "("
  a <- aExp
  symbol ")"
  return a
```

Notice how Stack, Queue and Array operations have been declared as ArithExpr and so they can be used in arithmetic expressions.

### 3.7 Boolean Expressions

```

bExp :: Parser BoolExpr
bExp = chain bTerm op
  where op = do
    symbol "Or"
    return Or

bTerm :: Parser BoolExpr
bTerm = chain bFact op
  where op = do
    symbol "And"
    return And

bFact :: Parser BoolExpr
bFact =
  do
    symbol "True"
    return (Boolean True)
  <|> do
    symbol "False"
    return (Boolean False)
  <|> do
    symbol "not"
    b <- bExp
    return (Not b)
  <|> do
    symbol "("
    b <- bExp
    symbol ")"
    return b
  <|> do
    symbol "empty"
    i <- identifier
    return (StackEmpty i)
  <|> do
    symbol "qempty"
    i <- identifier
    return (QueueEmpty i)
  <|> do
    symbol "bool("
    i <- identifier
    symbol ")"

```

```

    return (BoolVariable i)
<|> do
  a1 <- aExp
  do
    symbol "<"
    a2 <- aExp
    return (Lt a1 a2)
  <|> do
    symbol ">"
    a2 <- aExp
    return (Gt a1 a2)
  <|> do
    symbol "<="
    a2 <- aExp
    return (Lte a1 a2)
  <|> do
    symbol ">="
    a2 <- aExp
    return (Gte a1 a2)
  <|> do
    symbol "=="
    a2 <- aExp
    return (Eq a1 a2)
  <|> do
    symbol "!="
    a2 <- aExp
    return (Neq a1 a2)

```

Notice how Stack, Queue and Array operations have been declared as BoolExpr and so they can be used in boolean expressions.

## 3.8 Commands

```

command :: Parser Command
command =
  skip <|>
  ifElse <|>
  while <|>
  boolDeclare <|>
  arithDeclare <|>
  boolAssign <|>
  arithAssign <|>

```



```
arrayDeclare <|>
arrayAssign <|>
arrayFullAssign <|>
stackDeclare <|>
stackPush <|>
stackPop <|>
queueDeclare <|>
queueEnqueue <|>
queueDequeue

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

skip :: Parser Command
skip =
  do
    symbol "skip"
    symbol ";"
    return Skip
  <|> comment

comment :: Parser Command
comment =
  do
    symbol "/*"
    endComment

endComment =
  do
    symbol "*/"
    return Skip
  <|>
  do
    item
    endComment

ifElse :: Parser Command
ifElse =
  do
    symbol "if"
    symbol "("
```

```

    b <- bExp
    symbol ")"
    symbol "{"
    ifBranch <- program
    symbol "}"
    do
        symbol "else"
        symbol "{"
        elseBranch <- program
        symbol "}"
        return (IfElse b ifBranch elseBranch)
    <|> do
        return (IfElse b ifBranch [Skip])

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

arithDeclare :: Parser Command
arithDeclare =
    do
        symbol "int"           -- int id = 4;
        i <- identifier
        symbol "="
        value <- aExp
        symbol ";"
        return (ArithDeclare i value)

boolDeclare :: Parser Command
boolDeclare =
    do
        symbol "bool"         -- bool id=True;
        i <- identifier
        symbol "="

```

```

    value <- bExp
    symbol ";"
    return (BoolDeclare i value)

arithAssign :: Parser Command
arithAssign =
    do
        i <- identifier
        symbol "="
        value <- aExp
        symbol ";"
        return (ArithAssign i value)

boolAssign  :: Parser Command
boolAssign  =
    do
        i <- identifier
        symbol "="
        value <- bExp
        symbol ";"
        return (BoolAssign i value)

```

### 3.9 Arrays

```

arrayDeclare :: Parser Command
arrayDeclare =
    do
        symbol "array"
        i <- identifier
        symbol "["
        length <- aExp
        symbol "]"
        symbol ";"
        return (ArrayDeclare i (ArrayInit length))
    <|>
    do
        symbol "array"
        i <- identifier
        symbol "="
        arrayValues <- (
            do
                arrayFull

```

```

        <|>
        arrayScalarProduct
        <|>
        arrayConcat
        <|>
        arrayDotProduct
    )
    symbol ";"
    return (ArrayDeclare i arrayValues)

arrayFull :: Parser ArrayExpr
arrayFull =
    do
        symbol "["
        first <- aExp
        rest <- many (
            do
                symbol ","
                aExp
            )
        symbol "]"
        return (ArrayFull (first : rest))

arrayScalarProduct :: Parser ArrayExpr
arrayScalarProduct =
    do
        symbol "scalar"
        source <- identifier
        scalar <- aExp
        return (ArrayScalarProduct source scalar)

arrayConcat :: Parser ArrayExpr
arrayConcat =
    do
        symbol "concat"
        headArray <- identifier
        tailArray <- identifier
        return (ArrayConcat headArray tailArray)
    <|>
    do
        headArray <- identifier
        symbol "++"
        tailArray <- identifier

```

```
return (ArrayConcat headArray tailArray)
```

```
arrayDotProduct :: Parser ArrayExpr
arrayDotProduct =
  do
    symbol "dot"
    headArray <- identifier
    tailArray <- identifier
    return (ArrayDotProduct headArray tailArray)
```

```
arrayAssign :: Parser Command
arrayAssign =
  do
    i <- identifier
    symbol "["
    length <- aExp
    symbol "]"
    symbol "="
    value <- aExp
    symbol ";"
    return (ArrayAssign i length value)
```

```
arrayFullAssign :: Parser Command
arrayFullAssign =
  do
    i <- identifier
    symbol "="
    arrayValues <- (
      do
        arrayFull
        <|>
        arrayScalarProduct
        <|>
        arrayConcat
        <|>
        arrayDotProduct
      )
    symbol ";"
    return (ArrayFullAssign i arrayValues)
```

## 3.10 Stacks

```
stackDeclare :: Parser Command
stackDeclare =
  do
    symbol "stack"
    i <- identifier
    symbol ";"
    return (StackDeclare i)
```

```
stackPush :: Parser Command
stackPush =
  do
    symbol "push"
    i <- identifier
    value <- aExp
    symbol ";"
    return (StackPush i value)
```

```
stackPop :: Parser Command
stackPop =
  do
    symbol "pop"
    i <- identifier
    symbol ";"
    return (StackPop i)
```

## 3.11 Queues

```
queueDeclare :: Parser Command
queueDeclare =
  do
    symbol "queue"
    i <- identifier
    symbol ";"
    return (QueueDeclare i)
```

```
queueEnqueue :: Parser Command
queueEnqueue =
```

```
do
  symbol "enqueue"
  i <- identifier
  value <- aExp
  symbol ";"
  return (QueueEnqueue i value)

queueDequeue :: Parser Command
queueDequeue =
  do
    symbol "dequeue"
    i <- identifier
    symbol ";"
    return (QueueDequeue i)
```

# Chapter 4

## Interpreter

The interpreter is the part of the program which has the task of reading the intermediate tree representation generated by the parser, and carry out the meaning of that representation using its native language.

The interpreter associates a semantics to the symbols yield by the parser.

The most important part of the interpreter its the Environment. The Environment represents the internal state of the source program, and the main purpouse of the interpreter is to update it according to the semantics of the parsed source program.

```
data Variable = Variable {name :: String, value :: Type }
instance Show Variable where
  show x = "\n" ++ (name x) ++ " = " ++ (show (value x))
```

The basic element of the Environment is the Variable, which represent a single piece of data, and is described by a name and a value. Here we derive the class Show to have a prettier print of the state.

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

The Environment can be now defined simply as a list of Variables.

The two main functions to manipulate the Environment respectively allow the interpreter to read a Variable from the state, or write a new Variable (or update the value of an existing one).

```
readEnv :: Env -> String -> Maybe Type
readEnv [] varName = Nothing
readEnv (x:xs) varName | name x == varName = Just (value x)
                        | otherwise         = readEnv xs varName
```



```

writeEnv :: Env -> Variable -> Env
writeEnv [] var = [var]
writeEnv (x:xs) var | (name x == name var) = [var] ++ xs
                    | otherwise             = [x] ++ writeEnv xs var

```

## 4.1 Arithmetic Expression Evaluation

The Interpreter gives meaning to all of the structures decoded by the Parser.

We have defined many data types that have to be considered as arithmetic expression, and here we describe, for each of them, which value to assign.

Because the evaluation is bound to the current state of the program, the function to evaluate the expressions also take the environment as an input.

```

arithExprEval :: Env -> ArithExpr -> Maybe Int

arithExprEval env (Constant i) = Just i

arithExprEval env (ArithVariable i) =
    case readEnv env i of
        Just (IntType v) -> Just v
        Just _ -> error "type mismatch"
        Nothing -> error "undeclared variable"

arithExprEval env (StackTop i) =
    case readEnv env i of
        Just (StackType (v:vs)) -> Just v
        Just _ -> error "type mismatch in StackTop"
        Nothing -> error "undeclared variable in StackTop"

arithExprEval env (QueueFirst i) =
    case readEnv env i of
        Just (QueueType (v:vs)) -> Just v
        Just _ -> error "type mismatch in QueueFirst"
        Nothing -> error "undeclared variable in QueueFirst"

arithExprEval env (ArrayLength array) =
    case readEnv env array of
        Just (ArrayType arrayValues) -> Just (length arrayValues)
        Just _ -> error "type mismatch in QueueFirst"
        Nothing -> error "undeclared variable in QueueFirst"

arithExprEval env (ArrayPos array posExp) =

```

```

    case readEnv env array of
      Just (ArrayType arrayValues) -> Just (arrayValues !! pos)
      where Just pos = arithExprEval env posExp
      Just _ -> error "type mismatch in QueueFirst"
      Nothing -> error "undeclared variable in QueueFirst"

arithExprEval env (Add a b) = pure (+) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

arithExprEval env (Sub a b) = pure (-) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

arithExprEval env (Mul a b) = pure (*) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

arithExprEval env (Div a b) = pure (div) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

arithExprEval env (Power a b) = pure (^) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

```

## 4.2 Boolean Expression Evaluation

The Boolean Expression Evaluation comes into place when we have to assign meaning to expressions which have a Boolean has return value.

```

boolExprEval :: Env -> BoolExpr -> Maybe Bool

boolExprEval env (Boolean b) = Just b

boolExprEval env (BoolVariable s) =
  case readEnv env s of
    Just (BoolType v) -> Just v
    Just _ -> error "type mismatch"
    Nothing -> error "undeclared variable"

boolExprEval env (StackEmpty s) =
  case readEnv env s of
    Just (StackType []) -> (Just True)
    Just (StackType _) -> (Just False)
    Just _ -> error "type mismatch"

```

```

    Nothing -> error "undeclared variable"

boolExprEval env (QueueEmpty s) =
    case readEnv env s of
        Just (QueueType []) -> (Just True)
        Just (QueueType _) -> (Just False)
        Just _ -> error "type mismatch in QueueEmpty"
        Nothing -> error "undeclared variable in QueueEmpty"

boolExprEval env (Lt a b) = pure (<) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (Gt a b) = pure (>) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (Eq a b) = pure (==) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (Neq a b) = pure (/=) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (Lte a b) = pure (<=) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (Gte a b) = pure (>=) <*> (arithExprEval env a)
                                <*> (arithExprEval env b)

boolExprEval env (And a b) = pure (&&) <*> (boolExprEval env a)
                                <*> (boolExprEval env b)

boolExprEval env (Or a b) = pure (||) <*> (boolExprEval env a)
                                <*> (boolExprEval env b)

boolExprEval env (Not a) = not <$> boolExprEval env a

```

## 4.3 Array Expression Evaluation

Array Expression Evaluation is needed each time there is an expression that returns an Array as result. This evaluation tells the program how to give meaning to such expression and how to build the values that construct a new array. This new type allows us to declare and assign arrays, and potentially to use this expressions everywhere an array can appear.

```
arrayExprEval :: Env -> ArrayExpr -> Maybe [Int]
arrayExprEval env (ArrayScalarProduct source scalaraExp) =
  case readEnv env source of
    Just (ArrayType sourceArrayValues) -> Just values
      where values = [x * scalar | x <- sourceArrayValues]
            where Just scalar = arithExprEval env scalaraExp
    Just _ -> error "Type mismatch in arrayExprEval ArrayScalarProduct"
    Nothing -> error "Undeclared source in arrayExprEval ArrayScalarProduct"

arrayExprEval env (ArrayInit lengthaExp) = Just values
  where values = getFilledArray length
        where Just length = arithExprEval env lengthaExp

-- array a = [1,2,3,4];
arrayExprEval env (ArrayFull valuesaExps) = Just values
  where values = map fromJust (map (arithExprEval env) valuesaExps)
        where fromJust (Just x) = x

-- concat a b OR a ++ b
arrayExprEval env (ArrayConcat headArray tailArray) =
  case readEnv env headArray of
    Nothing -> error "headArray not in env in ArrayConcat"
    Just (ArrayType headArrayValues) ->
      case readEnv env tailArray of
        Nothing -> error "tailArray not in env in ArrayConcat"
        Just (ArrayType tailArrayValues) -> Just (headArrayValues ++
          tailArrayValues)
        Just _ -> error "type mismatch for tailArray in ArrayConcat"
    Just _ -> error "type mismatch for headArray in ArrayConcat"

\newpage
-- destination = dot arr1 arr2;
arrayExprEval env (ArrayDotProduct headArray tailArray) =
  case readEnv env headArray of
    Nothing -> error "headArray not in env in ArrayDotProduct"
    Just (ArrayType headArrayValues) ->
      case readEnv env tailArray of
```

```
Nothing -> error "tailArray not in env in ArrayDotProduct"
Just (ArrayType tailArrayValues) ->
  do
    if length headArrayValues == length tailArrayValues
    then do
      let values = [x * y | (x,y) <- (zip headArrayValues tailArrayValues)]
      return values
    else error "length mismatch between headArray and tailArray"
Just _ -> error "type mismatch for tailArray"
Just _ -> error "type mismatch for headArray"
```

## 4.4 Program Execution

We have given the interpreter all of the tools needed to manipulate the state and to evaluate all kinds of expressions. We now need to define the heart of the program, which is the capability to execute commands that were present in the source code and then parsed by the parser.

This is done by the function *executeProgram*, which takes the Environment and a list of commands. It extracts the first command and executes it: this may result in updating the state (e.g. an assignment), updating the rest of the commands (e.g. a loop command), or in an error (e.g. the current state is incompatible with the desired command).

```
executeProgram :: Env -> [Command] -> Env

-- Execute nothing, env unaltered
executeProgram env [] = env

-- Execute skip --> execute rest of commands
executeProgram env (Skip: restOfCommands) = executeProgram env restOfCommands

executeProgram env ((IfElse predicate ifBranch elseBranch) : restOfCommands) =
    case boolExprEval env predicate of
        Just True -> executeProgram env (ifBranch ++ restOfCommands)
        Just False -> executeProgram env (elseBranch ++ restOfCommands)
        Nothing -> error "Error on IfElse evaluation"

executeProgram env ((While predicate whileBody) : restOfCommands) =
    case boolExprEval env predicate of
        Just True -> executeProgram env (whileBody ++ [(While predicate whileBody)]
                                                    ++ restOfCommands)
        Just False -> executeProgram env restOfCommands
        Nothing -> error "Error while"

executeProgram env ((ArithAssign identifier aExp) : restOfCommands) =
    case readEnv env identifier of
        Just (IntType _) -> executeProgram (writeEnv env var) restOfCommands
                                where var = Variable identifier (IntType evaluated)
                                where Just evaluated = arithExprEval env aExp
        Just _ -> error "Type mismatch in ArithAssign"
        Nothing -> error "Identifier not in env for ArithAssign"

executeProgram env ((BoolAssign identifier bExp) : restOfCommands) =
    case readEnv env identifier of
        Just (BoolType _) -> executeProgram (writeEnv env var) restOfCommands
                                where var = Variable identifier (BoolType evaluated)
```

```

                                where Just evaluated = boolExprEval env bExp
Just _ -> error "Type mismatch in BoolAssign"
Nothing -> error "Identifier not in env for BoolAssign"

executeProgram env (( ArithDeclare identifier aExp ) : restOfCommands ) =
  case arithExprEval env aExp of
    Just exp -> case readEnv env identifier of
      Just (IntType _) -> error "Double ArithDeclare"
      Just _ -> error "Type mismatch in ArithDeclare"
      Nothing -> executeProgram (writeEnv env var) restOfCommands
        where var = Variable identifier (IntType evaluated)
        where Just evaluated = arithExprEval env aExp
    Nothing -> error "Error in ArithDeclare"

executeProgram env (( BoolDeclare identifier bExp ) : restOfCommands ) =
  case boolExprEval env bExp of
    Just exp -> case readEnv env identifier of
      Just (BoolType _) -> error "Double BoolDeclare"
      Just _ -> error "Type mismatch in BoolDeclare"
      Nothing -> executeProgram (writeEnv env var) restOfCommands
        where var = Variable identifier (BoolType evaluated)
        where Just evaluated = boolExprEval env bExp
    Nothing -> error "Error in BoolDeclare"

-- array a[5];
executeProgram env (( ArrayDeclare identifier arrayExp ) : restOfCommands ) =
  case readEnv env identifier of
    Just (ArrayType array) -> error "double ArrayDeclare"
    Just _ -> error "Type mismatch for ArrayDeclare"
    Nothing -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable identifier (ArrayType values)
      where Just values = arrayExprEval env arrayExp

-- a[1] = 2;
executeProgram env (( ArrayAssign identifier indexaExp valueaExp ) : restOfCommands ) =
  case readEnv env identifier of
    Just (ArrayType array) -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable identifier (ArrayType (replaceElemAt array index value))
      where
        Just index = arithExprEval env indexaExp
        Just value = arithExprEval env valueaExp
    Just _ -> error "Type mismatch in ArrayAssign"
    Nothing -> error "Trying to assign to an array that has not been declared"

```

```

executeProgram env (( ArrayFullAssign identifier arrayExp ) : restOfCommands ) =
  case readEnv env identifier of
    Just (ArrayType array) -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable identifier (ArrayType values)
      where Just values = arrayExprEval env arrayExp

-- stack myStack;
executeProgram env (( StackDeclare stackName ) : restOfCommands ) =
  case readEnv env stackName of
    Just _ -> error "double StackDeclare"
    Nothing -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable stackName (StackType [])

-- push myStack 3;
executeProgram env (( StackPush stackName valueaExp ) : restOfCommands ) =
  case readEnv env stackName of
    Nothing -> error "stackName not in env"
    Just (StackType stackValues) -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable stackName (StackType (value : stackValues))
      where Just value = arithExprEval env valueaExp
    Just _ -> error "type mismatch for stackName in StackPush"

-- pop myStack;
executeProgram env (( StackPop stackName ) : restOfCommands ) =
  case readEnv env stackName of
    Nothing -> error "stackName not in env"
    Just (StackType (v:vs)) -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable stackName (StackType vs)
    Just (StackType []) -> error "trying to pop from an empty stack"
    Just _ -> error "type mismatch for stackName in StackPop"

-- queue myQueue;
executeProgram env (( QueueDeclare queueName ) : restOfCommands ) =
  case readEnv env queueName of
    Just _ -> error "double QueueDeclare"
    Nothing -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable queueName (QueueType [])

-- enqueue myQueue 3;
executeProgram env (( QueueEnqueue queueName valueaExp ) : restOfCommands ) =
  case readEnv env queueName of
    Nothing -> error "queueName not in env in QueueEnqueue"
    Just (QueueType queueValues) -> executeProgram (writeEnv env var) restOfCommands
      where var = Variable queueName (QueueType (queueValues ++ [value]))
      where Just value = arithExprEval env valueaExp
    Just _ -> error "type mismatch for queueName in QueueEnqueue"

```



```
executeProgram env (( QueueDequeue queueName ) : restOfCommands ) =  
  case readEnv env queueName of  
    Nothing -> error "queueName not in env"  
    Just (QueueType (v : vs)) -> executeProgram (writeEnv env var) restOfCommands  
      where var = Variable queueName (QueueType (tail (v : vs)))  
    Just (QueueType []) -> error "trying to dequeue from an empty queue"  
    Just _ -> error "Type mismatch for queueName in QueueDequeue"
```

# Chapter 5

## Using the interpreter

To use the LIMP interpreter, an Haskell interpreter is needed on the machine. To start the program, load the module Main.hs; this will compile and load the file modules that compose the program:

```
C:\Users\39347\Desktop\limp>ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> :l main
[1 of 5] Compiling Grammar          ( Grammar.hs, interpreted )
[2 of 5] Compiling Array              ( Array.hs, interpreted )
[3 of 5] Compiling Interpreter        ( Interpreter.hs, interpreted )
[4 of 5] Compiling Parser            ( Parser.hs, interpreted )
[5 of 5] Compiling Main              ( main.hs, interpreted )
Ok, five modules loaded.
*Main> 
```

Now, to run the program type "main". The menu shows up with 3 choices:

```
*Main> main
1) Run a program from a file
2) Run the interactive shell
3) Exit
```

The third is used to exit the program. The other two choices correspond to the two execution mode of LIMP, which are inspired from Python. If an incorrect choice is made, the menu is presented again.

## 5.1 Interactive shell

The interactive shell mode allows the user to write some code and immediately check the results. The user can type:

- one command
- many commands in one line

Each time a command or a sequence of command is typed, if the parsing is successful, the program prints:

1. the input program
2. the internal tree representing the program
3. the environment

During a session of the interactive shell, the Environment is retained even between sequences of commands: this is very useful for testing and prototyping code.

```
*Main> main
1) Run a program from a file
2) Run the interactive shell
3) Exit

2
Interactive shell
limp> array x[5];

Parsing success!

Input Program
array x[5];
Representation of the program:
[ArrayDeclare "x" (ArrayInit (Constant 5))]

State of the memory:
[
x = ArrayType [0,0,0,0,0]]
```

**Figure 5.1:** Execution of a single command in the LIMP interactive shell

```
limp> x[1]=1;x[2]=2;/* this is a comment between commands */ x[3]=3; x[4]=4;

Parsing success!

Input Program
x[1]=1;x[2]=2;/* this is a comment between commands */ x[3]=3; x[4]=4;
Representation of the program:
[ArrayAssign "x" (Constant 1) (Constant 1),ArrayAssign "x" (Constant 2) (Constant 2),ArrayAssign "x" (Constant 3) (Constant 3),ArrayAssign "x" (Constant 4) (Constant 4)]

State of the memory:
[
x = ArrayType [0,1,2,3,4]]
```

**Figure 5.2:** *Execution of multiple command in the LIMP interactive shell. The Environment from previous command is retained. A comment is present between the commands*

Comments are also allowed inside the shell, making it easy to copy and paste code. (Try copying the content of the file `tests/bubbleSort_online.txt` and pasting it in the interactive shell).

To exit the shell, type "quit".

## 5.2 Running files

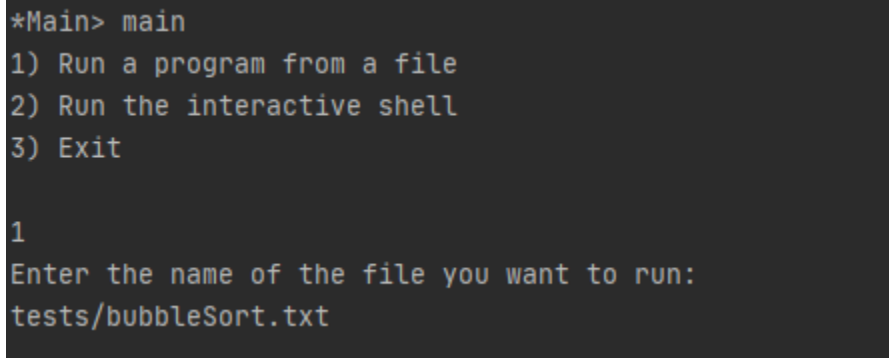
Alternative, by choosing the first option in the menu, files can be directly executed. To do so, the path to the source code must be typed in the shell.

The source code ships with many test files, to test the commands and the data structures implemented in LIMP.

When the execution of the program in the file terminates, the interactive shell is enabled with the Environment in the state left by the program. This is very useful for testing, debugging and prototyping.

### 5.2.1 Bubble Sort

The program in the file `tests/bubbleSort.txt` is the most complete test program in the suite. It shows how arrays can be declared and fully assigned in LIMP, also shows the use of the function *length* on arrays; it shows how to declare and assign integers and booleans, how to use while statements and booleans as conditions; moreover it demonstrates how to access specific elements of arrays and how to use their values in boolean expressions. Last but not least, it shows the usage of comments.



```
*Main> main
1) Run a program from a file
2) Run the interactive shell
3) Exit

1
Enter the name of the file you want to run:
tests/bubbleSort.txt
```

**Figure 5.3:** *Execution of a program from source file*

```

Representation of the program:
[ArrayDeclare "x" (ArrayFull [Constant 10,Constant 123,Constant 23,Constant 32,Constant 345,Constant 7,Constant 78,Constant 354,Constant 1234,Constant 465]),ArithDe
variable "swapped") [Skip,ArithAssign "i" (Constant 0),BoolAssign "swapped" (Boolean
hVariable "i") (Constant 1)))] [Skip,ArithAssign "tmp" (ArrayPos "x" (Add (ArithVar
") (ArithVariable "tmp"),BoolAssign "swapped" (Boolean True),Skip] [Skip],ArithAssi

State of the memory:
[
x = ArrayType [4,7,10,12,23,32,34,53,78,78,123,123,213,234,234,345,354,465,676,867,
i = IntType 21,
n = IntType 22,
tmp = IntType 7,
swapped = BoolType False]

```

**Figure 5.4:** *Result of the bubble sort program*

### 5.2.2 Stack

The file `tests/testStack.txt` contains source code for testing stacks. The program declares a stack and pushes some values. Then, using the functions *top*, *empty* and *pop*, fills an array with the values from the stack.

```

Enter the name of the file you want to run:
tests/testStack.txt

Parsing success!

Input Program
stack a;
push a 3;
push a 4;
push a 45;
push a (3+1);
array b[4];
int i = 0;
while(not empty a){
    b[i] = top a;
    i = i+1;
    pop a;
}

Representation of the program:
[StackDeclare "a",StackPush "a" (Constant 3),StackPush "a" (Constant 4),StackPush "a" (Constant 45),StackPush "a" (Constant 4),StackEmpty "a") [ArrayAssign "b" (ArithVariable "i") (StackTop "a"),ArithAssign "i" (Add (ArithVariable "i") (Constant 1))]

State of the memory:
[
a = StackType [],
b = ArrayType [4,45,4,3],
i = IntType 4]
limp>

```

**Figure 5.5:** *Result of the test program for stacks*

### 5.2.3 Queue

The file `tests/testQueue.txt` contains source code for testing queues. The program declares a queue and enqueues some values. Then, using the functions *first*, *qempty* and *dequeue*, fills an array with the values from the queue.

```
Representation of the program:
[ArrayDeclare "x" (ArrayFull [Constant 10,Constant 123,Constant 23,Constant 32,Constant 345,Constant 7,Constant 78,Constant 354,Constant 1234,Constant 465]),ArithDe
variable "swapped") [Skip,ArithAssign "i" (Constant 0),BoolAssign "swapped" (Boolean
hVariable "i") (Constant 1)))] [Skip,ArithAssign "tmp" (ArrayPos "x" (Add (ArithVar
") (ArithVariable "tmp"),BoolAssign "swapped" (Boolean True),Skip] [Skip],ArithAssi

State of the memory:
[
x = ArrayType [4,7,10,12,23,32,34,53,78,78,123,123,213,234,234,345,354,465,676,867,
i = IntType 21,
n = IntType 22,
tmp = IntType 7,
swapped = BoolType False]
```

Figure 5.6: Result of the test program for queues



### **5.2.4 Other test programs**

Most of the other functions of LIMP can be tested using the other source files in the folder tests.

# Chapter 6

## Conclusions

This project had the academic purpose of building an Interpreter for a light-weight imperative program and explore all of the needed components. However, the modularity of the implementation make it easy to add new features, commands and data structures, and this makes LIMP a possible building block to evolve a more complex and complete language.