University of Bari "Aldo Moro"
Master Degree in Artificial Intelligence

IMP Interpreter Documentation

KLIMP INTERPRETER

Francesco Ranieri

Academic Year 2021 - 2022

Contents

Chapter 1

Introduction

1.1 Interpreter

An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.

The interpreter transforms the high-level program into an intermediate language that it then executes, or it could parse the high-level source code and then performs the commands directly, which is done line by line or statement by statement.

1.2 IMP

IMP was a systems programming language developed by E. T. Irons in the late 1960s through early 1970s. Unlike most other systems programming languages, IMP was an extensible syntax programming language. IMP is a simple imperative language which gives us the possibilities to use the basic constructs used in the most common imperative languages. These are:

- **assignment**: assigns a value to a variable;
- **skip**: just does nothing;
- **if then else**: if the boolean expression after the if it's true, then executes some commands otherwise(else) executes some other ones;
- **while**: executes and continue executing the commands while the boolean condition remains true.

Chapter 2

Backus-Naur Form Grammar

BNF (Backus–Naur Form) is a context-free grammar commonly used by developers of programming languages to specify the syntax rules of a language.

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

command ::= <variableDeclaration>
  | <skip>
  | <assignment>
  | <ifThenElse>
  | <while>
  | <forParser>
  | <commentParser>

variableDeclaration ::= <AExpDeclaration>
  | <BExpDeclaration>
  | <ArrayDeclaration>
  | <ArrayFullDeclaration>
  | <SetDeclaration>
  | <SetFullDeclaration>
  | <StackDeclaration>


assignment ::= <AExpAssignment>
  | <BExpAssignment>
  | <ArrayAssignmentSingleValue>
  | <ArrayAssignmentValues>
  | <ArrayAssignmentValues>
  | <SetAssignmentSingleValue>
  | <SetAssignmentValues>
  | <StackPushValue>
  | <StackPopValue>

AExpDeclaration ::= "int" <identifier> "=" <aexp> ";"
BExpDeclaration ::= "bool" <identifier> "<-" <bexp> ";"
ArrayDeclaration ::= "array" <identifier> "[" <aexp> "]" ";"
ArrayFullDeclaration ::= "array" <identifier> "[" <arrayexp> "]" ";"
SetDeclaration ::= "set" <identifier> "{" "}" ";"
SetFullDeclaration ::= "set" <identifier> "{" <setexp> "}" ";"
StackDeclaration ::= "stack" <identifier> ";"
```

```
37
38    AExpAssignment ::= <identifier> "=" <aexp> ";"
39    BExpAssignment ::= <identifier> "<-" <bexp> ";"
40
41    ArrayAssignmentSingleValue ::= <identifier> "[" <aexp> "]" ":=" <aexp> ";"
42    ArrayAssignmentValues ::= <identifier> ":=" "[" <aexp> ["," <aexp>]* "]" ";"
43     | <identifier> ":=" <identifier> ";"
44
45    SetAssignmentSingleValue ::= <identifier> "add" <aexp> ";"
46    SetAssignmentValues ::= <identifier> ":=" "{" <aexp> ["," <aexp>]* "}" ";"
47     | <identifier> ":=" "(set)" <identifier> ";"
48
49    StackPushValue ::= <identifier> "push" <aexp> ";"
50    StackPushValue ::= <identifier> "pop" ";"
51
52    ifThenElse ::= "if" "(" <bexp> ")" "{" <program> "}"
53     | "if" "(" <bexp> ")" "{" <program> "}" "else" <program> "}"
54
55    while ::= "while" "(" <bexp> ")" "{" <program> "}"
56
57    for ::= "for" "(" <variableDeclaration> ";" <bexp> ";" <identifier> "++"  ")" "{" <program> "}"
58     | "for" "(" <variableDeclaration> ";" <bexp> ";" <identifier> "++"  ")" "{" <program> "}"
59    skip ::= "skip" ";"
60
61    comment ::= "{-" <comment> "-}"
62
63    aexp ::= <aterm> ["+" <aterm>]*
64     | <aterm> ["-" <aterm>]*
65
66    aterm ::= <afact> ["*" <afact>]*
```

Chapter 3

Environment

The program consists of two parts:

- the first part takes one input file as a String and creates an internal representation of the program (**the parser**)
- the second one which takes the output of the first part and evaluates the program updating the state of the memory (**the interpreter**).

To keep track of the variables and their values during the program execution, it has been created a Dictionary data structure.

3.1 Dictionary

A dictionary is an abstract data type that defines an unordered collection of data as a set of key-value pairs. Each key, which must be unique, has an associated value. Typically, the keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
Using a key allows direct access to items stored within the collection. This provides much quicker access to unordered data than searching through the data set using a linear search and will also outperform a binary search.

In order to have this direct access to items we define a type State as follow:

```
type State = Dictionary String Type
```

In this way we can represent the state of the memory by using a dictionary where every variable is kept stored with its corresponding value. In this case we are able to store values of type Integer, Boolean and Array, Set and stack structures. These six kind of types are grouped into an Haskell data type and defined as Type.

```
data Type
    --
    = IntegerType Int
    --
    | BooleanType Bool
    --
    | ArrayType (Array Int)
    --
    | SetType (Array Int)
    --
    | StackType (Array Int)
    deriving Show
```

The dictionary data structure is implemented as it follows:

```haskell
-- Declare a Dictionary k v type
newtype Dictionary k v = Dictionary [(k, v)]
  deriving (Show)


-- Empty dictionary
empty :: (Eq k) => Dictionary k v
empty = Dictionary []


-- Check empty dictionary
isempty :: (Eq k) => Dictionary k v -> Bool
isempty (Dictionary []) = True
isempty _ = False


-- Lookup from dictionary
get :: (Eq k) => Dictionary k v -> k -> Maybe v
get (Dictionary []) _ = Nothing
get (Dictionary ((h, v) : ps)) k =
  if k == h
    then Just v
    else get (Dictionary ps) k
```

```haskell
-- Insert into dictionary
insert :: (Eq k) => Dictionary k v -> k -> v -> Dictionary k v
insert (Dictionary []) k v = Dictionary [(k, v)]
insert (Dictionary ((h, u) : ps)) k v =
  if k == h
    then Dictionary ((h, v) : ps)
    else Dictionary ((h, u) : ds)
  where
    (Dictionary ds) = insert (Dictionary ps) k v



-- Delete from dictionary
delete :: (Eq k) => Dictionary k v -> k -> Maybe [(k,v)]
delete (Dictionary []) _ = Nothing
delete (Dictionary ((h, v) : ps)) k =
  if k == h
    then return ps
    else delete (Dictionary ps) k
```

In the project the types of k and v are respectively String and Type:

- String indicates instead the name of the variable;
- Type stays for the value contained into the variable (Integer, Boolean or Array as described previously)

If we have this program

```
---------------------------------
Insert the program or ESC to exit
int number = 10; bool boolean <- True;
```

at the end of a program execution, the environment contains only two variables, one named 'number' which contains an integer value 10 and another one named 'boolean' containing a Boolean value True. In this case the representation will be:

```
 -- INPUT PARSED --

AExpDeclaration "number" (AExpConstant 10)
BExpDeclaration "boolean" (BExpConstant True)

 -- INPUT NOT PARSED --


 -- STATE --

Dictionary [("number",IntegerType 10),("boolean",BooleanType True)]
```

We can see that the parser read the the program as two instruction and classified classifies them as

- **AExpDeclaration** ”number” (**AExpConstant** 10)
- **BExpDeclaration** ”boolean” (**BExpConstant** True)

which means that we declare the variable ”number” and ”boolean” with two constants of type arithmetic(**aexp**) and Boolean(**bexp**)
The final state is represented as a dictionary or more simple as an array of tuple (**k**,**v**) where **k** is the identifier of the variable and **v** is the type composed of type of variable and actual value.

Chapter 4

Internal Representations of Commands: Grammar

4.1 Program

As said before, the parser takes a string as input and creates an internal representation of the program commands. The previous statement is represented in the interpreter as:

```
-- Program declaration
type Program = [Command]
```

4.2 Command

In this way we define a **Program** as a list of commands.

```
-- Command declaration
data Command
    --
    = AExpDeclaration String AExp
    --
    | BExpDeclaration String BExp
    -- Array
    | ArrayDeclaration String AExp
    --
    | ArrayFullDeclaration String ArrayExp
    -- Set
    | SetDeclaration String
    --
    | SetFullDeclaration String SetExp
    -- Stack
    | StackDeclaration String
    --
    | Skip
    --
    | AExpAssignment String AExp
    --
    | BExpAssignment String BExp
    --
    | ArrayAssignmentSingleValue String AExp AExp
    --
    | ArrayAssignmentValues String ArrayExp
    --
    | SetAssignmentSingleValue String AExp
    --
    | SetAssignmentValues String SetExp
    --
    | StackPushValue String AExp
    --
    | StackPopValue String
    --
    | IfThenElse BExp [Command] [Command]
    --
    | While BExp [Command]
    --
    | ForIncrement Command BExp String [Command]
    --
    | ForDecrement Command BExp String [Command]
    deriving Show
```

A single command can be:

- a declaration of a variable which will contain the result of an arithmetic expression. In this step it's mandatory to assign a value to the variable;
- a declaration of a variable which will contain the result of a boolean expression. In this step it's mandatory to assign a value to the variable;
- a declaration of a variable which will contain an array:
  - we also have to specify the size n of the array and the program will produce and array of n zero. (array a[2] = [0,0])
  - we can specify the values that the array contains (array a = [1,2,3])
- a declaration of a variable which will contain a set:
  - we can specify only the identifier for empty set (set a;)
  - we can specify the values that the set contains (set a = 1,2,3). If the declaration of set contains duplicate values, they will be removed.

```
Insert the program or ESC to exit
set a = {1,2,3,4,5,5};

 -- INPUT PARSED --

SetFullDeclaration "a" (SetValues [AExpConstan

 -- INPUT NOT PARSED --


 -- STATE --

Dictionary [("a",SetType [1,2,3,4,5])]
```

- a declaration of a variable which will contain a stack:
  - we also have to specify the identifier (stack a). The only way to populate the stack is by using push and pop functions.
- skip, which simply does nothing;
- an assignment of an arithmetic expression (more properly we assign its result) to a variable previously

- declared (which has been declared in the proper way, that is, to store arithmetic expressions);
- an assignment of a boolean expression (more properly we assign its result) to a variable previously declared (which has been declared in the proper way, that is, to store boolean expressions);
- an assignment of an arithmetic expression (more properly we assign its result) to a specific position of the array variable previously declared;
- an assignment to an array variable of:
  - a list of arithmetic expressions (E.g. [1, 2, 3] or [1, 2*3, 4-1]);
  - an array variable previously declared.
- an assignment to a set variable which always **remove duplicate values** of:
  - an array expression in square brackets (E.g. [1, 2, 3] or [1, 2*3, 4-1]);
  - a list of arithmetic expressions in curly brackets (E.g. 1, 2, 3);
  - a set previously declared.
- a pop operation on stack which removes the head of the stack
- a push operation on stack which adds an item at the top of the stack
- if-then-else construct, which evaluates the boolean expression and if it's true then executes the subprogram after the then keyword, otherwise exectues the one after the else keyword;
- while contruct, which evaluates the boolean condition and executes the sub-program in the parenthesis until it remains true
- for contruct, which take three expressions:
  - a variable declaration (index)
  - a condition on the variable declared
  - an increment or decrement of the index (++ or –)

  So the for construct evaluates the Boolean condition on the declared variable and executes the sub-program in the parenthesis until it remains true. At the end of each iterate there is an increment or decrement of the declared variable in order to avoid the loop status.

4.3 Arithmetic Expression (AExp)

```haskell
data AExp
    -- Constant integer
    = AExpConstant Int
    -- Identifier string
    | AExpVariable String
    -- value contained in a specific position of an array variable
    | ValueFromArray String AExp
    -- Addition between sub-expressions
    | Add AExp AExp
    -- Subtraction between sub-expressions
    | Sub AExp AExp
    -- Multiplication between sub-expressions
    | Mul AExp AExp
    --
    | Length String
    deriving Show
```

An Arithmetic Expression (AExp) could be:

- a constant;
- a variable which is able to store an arithmetic expression;
- a value contained in a specific position of an array variable;
- an operation between two arithmetic expressions (in this case Add, Sub and Mul that indicate respectively the sum, the subtraction and the multiplication)
- a value indicated the length of a structure (array, set, stack)

4.4 Boolean Expression (BExp)

```
data BExp
    -- Ground True and False
    = BExpConstant Bool
    -- Identifier string
    | BExpVariable String
    -- Not unary operator
    | Not BExp
    -- Or binary operator
    | Or BExp BExp
    -- And binary operator
    | And BExp BExp
    -- < binary operator between arithmetical expressions
    | Less AExp AExp
    -- > binary operator between arithmetical expressions
    | Greater AExp AExp
    -- >= binary operator between arithmetical expressions
    | LessEqual AExp AExp
    -- <= binary operator between arithmetical expressions
    | GreaterEqual AExp AExp
    -- == binary operator between arithmetical expressions
    | Equal AExp AExp
    -- != binary operator between arithmetical expressions
    | NotEqual AExp AExp
    -- is_empty operator --> True is structure is empty
    | IsEmpty String
    deriving Show
```

A Boolean Expression (BExp) could be:

- a Boolean value;
- a variable which is able to store a Boolean expression;
- **Not** operator defined on one Boolean expression
- **Or** and **And** operators define on two Boolean expression
- **Less or Equal** operator defined on two arthritic expressions
- **Greater or Equal** operator defined on two arthritic expressions

- **Greater** operator defined on two arthritic expressions
- **Equal** operator defined on two arthritic expressions
- **Not Equal** operator defined on two arthritic expressions
- **Empty** operator defined on a structure type (array, set, stack) which return True if the structure is empty, False otherwise.
- an operator between two arithmetic expressions (less or equal, less, greater or equal, greater, equal or not equal)

4.5 Array Expression

```
data ArrayExp
    --  x = [1,2,3,3]
    = ArrayValues [AExp]
    -- x = y       y = [1,2,3,3]
    | ArrayExpVariable String
    deriving Show
```

An Array Expression could be:

- a list of arithmetic expressions (E.g. [1, 2, 3] or [1, 2*3, 4-1]);
- an array variable already declared previously

4.6 Set Expression

```
data SetExp
    --  x = [1,2,3]
    = SetValues [AExp]
    -- x = y       y = [1,2,3]
    | SetExpVariable String
    deriving Show
```

A Set Expression could be:

- a list of arithmetic expressions (E.g. 1, 2, 3);
- a set variable already declared previously

4.7 Stack Expression

```
data StackExp
    --  x = [1,2,3]
    = StackValues [AExp]
    -- x = y       y = [1,2,3]
    | StackExpVariable String
    deriving Show
```

A Stack Expression could be:

- a list of arithmetic expressions (E.g. 1, 2, 3);
- a stack variable already declared previously

Chapter 5

Array

The arrays are defined as follow in the module Array.hs:

```
type Array a = [a]
```

with the functions:

- declare an array of fixed size with all zeroes;
- read a value from a given position of the array;
- write a value in the given position of the array.

```
declare :: Int -> Array Int
declare n = replicate n 0

read :: Int -> Array a -> Maybe a
read _ [] = error "Index out of bound"
read i (v : vs)
  | i == 0 = Just v
  | i < 0 = error "Index out of bound"
  | otherwise = Array.read (i - 1) vs

write :: Int -> a -> Array a -> Maybe (Array a)
write _ _ [] = error "Index out of bound"
write i v' (v : vs)
  | i == 0 = Just (v' : vs)
  | i < 0 = error "Index out of bound"
  | otherwise = case write (i - 1) v' vs of
                  Just vs' -> Just (v : vs')
                  Nothing -> error "Out of bound"
```

Chapter 6

Set

The set are defined as follow in the module Set.hs:

```
type Set a = [a]
```

with the functions:

- declare an empty set;
- full declare a set with an array of integer value and remove duplicate values
- read a value from a given position of the set;
- add a value to the set if and only if the value is not already in the set otherwise the value will not be added.
- check if the set contains duplicate elements
- convert an array to a set by removing duplicate values

```haskell
--
fullDeclareSet :: [Int] -> Set Int
fullDeclareSet [] = []
fullDeclareSet (v:vs)
    | isSet (v:vs) == False = arrayToSet (v:vs)
    | otherwise = (v:vs)


--
readSet :: Int -> Set a -> Maybe a
readSet _ [] = error "Index out of bound :: SET"
readSet i (v : vs)
  | i == 0 = Just v
  | i < 0 = error "Index out of bound :: SET"
  | otherwise = Set.readSet (i - 1) vs


--
insertSet :: Int -> Set Int -> Set Int
insertSet elem [] = [elem]
insertSet elem (s : ss) = do
                            case posElem of
                                  -1 -> (s : ss) ++ [elem]
                                  _ -> (s : ss)
                              where
                                   posElem = position elem (s : ss)


--
isSet :: Set Int -> Bool
isSet [] = True
isSet (s : ss) = case has of
                 True -> False
                 False -> isSet ss
                 where
                  has = contain ss s


--
arrayToSet :: [Int] -> [Int]
arrayToSet [] = []
arrayToSet (x:xs) = unique (x:xs)
```

Chapter 7

Stack

The set are defined as follow in the module Stack.hs:

```
type Stack a = [a]
```

with the functions:

- declare an empty stack;
- add a value to the top of the stack
- remove a value from the top of the stack

```
declareStack ::  Stack Int
declareStack = []


pushValue :: Stack Int -> Int -> Stack Int
pushValue [] elem = [elem]
pushValue stack elem = [elem] ++ stack


popValue :: Stack Int -> Stack Int
popValue [] = []
popValue (s:ss) = ss
```

Chapter 8

Program Structure

8.1 Parser Implementation

A parser can be seen as a function which takes a string as input and produces a couple as output:

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

where **a** is the parametrized type and indicates the type of the parser; the string returned with **a** represents instead the part of the input which has still not be parsed or that failed the parsing procedure. This means that if at the end, the empty string is returned, the parsing has been completed successfully.

8.1.1 Functor, Applicative, Monad and Alternative

In order to combine parsers together we need to create instances of Functor, Applicative, Monad and Alternative. The functor allows to apply a function g to a value wrapped in a parser p. This way we define the fmap function for parser which takes as input a function and an object wrapped inside a functor. The result is then the application of the function to the unwrapped object inside the functor.

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

The applicative, which can be used only if Functor has been already defined, let us to apply a function wrapped in a parser pg to another parser px. It defines the pure and the <*> operators. The first one just takes as input an object and wraps it into an applicative. The second one instead, takes as input a function and an object both wrapped inside the applicative. The result is the application of the unwrapped function to the unwrapped object.

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

The monad, which can be used only if the Applicative has been already defined, is used to apply a function which returns a wrapped parser to a wrapped parser. It defines the return and the bind operators. The first one returns an object wrapped inside a Monad. The second one is denoted with »== and takes as input an object wrapped inside a Monad and a function. The result of the bind operator is the application of the function to the unwrapped object. Since Parser is a monadic type, the do notation can be used to have a sequence of parsers and process their resulting values.

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

In order to explore all the possible ways when trying to match for an expression of the grammar we use the <|> operator. By using it, supposing we have two parsers like P and Q, if the first one fails then we will have only the output of the second parser, otherwise just the output of the first. By using this we can use some useful functions like some and many.

```
instance Alternative Parser where
  -- empty :: Parser a
  empty = P (\input -> Nothing)
  -- <|> :: Parser a -> Parser a -> Parser b
  (P p) <|> (P q) = P (\input -> case p input of
                                   Nothing -> q input
                                   Just (v, out) -> Just (v, out))
```

With Alernative we are able to define and use some combinators like some, many and chain in the do construct which can be used thanks to the implementation of the monads.

- The some construct let us to use many parsers together. In this way at least the first parser has to succed;
- The many it's like the previous one with the difference that every parser could fail;
- The chain implements the left-associative order in the evaluation of arithmetic and boolean expressions

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

8.1.2 Arithmetic Expression Parsing

Arithmetic Expression Parsing This parser is built in such a way to ensure that for each arithmetic expression will be a single derivation tree. The parser function aExpParser is supported by aTermParser and aFactorParser. These parsers basically are needed to create an order of execution of the arithmetic expressions: given, for example, more priority to multiplication rather than addition and subtraction, and also to execute first the expressions in the brackets. We can also see the use of the chain for the sum and the subtraction.

```
aExpParser :: Parser AExp
aExpParser = do chain aTermParser op
   where op = (do keywordParser "+"; return Add)
              <|> do keywordParser "-"; return Sub

--
aTermParser :: Parser AExp
aTermParser = do chain aFactParser op
              where op = do keywordParser "*"; return Mul

--
aFactParser :: Parser AExp
aFactParser = (do a <- integerNumberParser; return (AExpConstant a))
               <|> do
                      keywordParser "len"
                      a <- identifierParser
                      return (Length a)
               <|> do
                  a <- identifierParser
                  do
                     keywordParser "["
                     n <- aExpParser
                     keywordParser "]"
                     return (ValueFromArray a n)
                     <|>
                     return (AExpVariable a)
               <|> do
                    keywordParser "("
                    a <- aExpParser
                    keywordParser ")"
                    return a
```

8.1.3 Boolean Expression Parsing

A parser for evaluate boolean expressions has been created and the functioning is similar to the one of the arithmetic case: for each boolean expression exists a unique derivation tree. In this case, the and operator has an higher priority with respect to or operation. Moreover, bExpParser could need to use aExpParser, for example when a comparison between two numbers is required.

```
bFactParser :: Parser BExp
bFactParser = do keywordParser "True"
                 return (BExpConstant True)
              <|>
              do keywordParser "False"
                 return (BExpConstant False)
              <|>
              do keywordParser "not"
                 a <- bExpParser
                 return (Not a)
              <|>
              do keywordParser "("
                 b <- bExpParser
                 keywordParser ")"
                 return b
              <|>
              do keywordParser "empty"
                 i <- identifierParser
                 return (IsEmpty i)
              <|>
              do {
                 a1 <- aExpParser;
                 do keywordParser "<"
                    a2 <- aExpParser
                    return (Less a1 a2)
                  <|>
                  do keywordParser "<="
                     a2 <- aExpParser
                     return (LessEqual a1 a2)
```

```
            <|>
            do keywordParser ">"
               a2 <- aExpParser
               return (Greater a1 a2)
            <|>
            do keywordParser ">="
               a2 <- aExpParser
               return (GreaterEqual a1 a2)
               <|>
            do keywordParser "=="
               a2 <- aExpParser
               return (Equal a1 a2)
               <|>
            do keywordParser "!="
               a2 <- aExpParser
               return (NotEqual a1 a2)
      }
      <|>
      do a <- identifierParser
         return (BExpVariable a)
```

8.1.4 Command Parser

The parser for commands is defined as:

```
commandParser :: Parser Command
commandParser = variableDeclParser
              <|> skipParser
              <|> assignmentParser
              <|> ifThenElseParser
              <|> whileParser
              <|> forParser
              <|> commentParser
```

where variableDeclParser can be of 5 different types:

- The first one let us to declare a variable for arithmetic expressions with the keyword int (Example: int a;);
- The second one let us to declare a variable for boolean expressions with the keyword bool (E.g.: bool a;);

- The third one let us to declare an array of integers with the keyword array (E.g.: array a[10]; is an array of integers of size 10).
- The fourth one let us to declare a set with the keyword set (E.g.: set a = 1,2,3,4,4; is an set with values 1,2,3,4).
- The fifth one let us to declare an empty stack with the keyword stack (E.g : stack a; is an empty stack).

As decribed previously, it's also mandatory to assign a value for the first two cases. The implementation is the following:

```
variableDeclParser :: Parser Command
variableDeclParser = do {
   keywordParser "int";
   i <- identifierParser;
   keywordParser "=";
   a <- aExpParser;
   keywordParser ";";
   return (AExpDeclaration i a)
   }
   <|> do {
       keywordParser "bool";
       i <- identifierParser;
       keywordParser "<-";
       b <- bExpParser;
       keywordParser ";";
       return (BExpDeclaration i b)
   }
   <|> do {
       keywordParser "array";
       i <- identifierParser;
       keywordParser "[";
       n <- aExpParser;
       keywordParser "]";
       keywordParser ";";
       return (ArrayDeclaration i n)
   }
   <|> do {
       keywordParser "array";
       i <- identifierParser;
       keywordParser "=";
       keywordParser "[";
       i' <- aExpParser;
       i'' <- many (do keywordParser ","; aExpParser);
       keywordParser "]";
       keywordParser ";";
       return (ArrayFullDeclaration i (ArrayValues (i':i'')))
    }
```

```
<|> do {
    keywordParser "set";
    i <- identifierParser;
    keywordParser "{";
    keywordParser "}";
    keywordParser ";";
    return (SetDeclaration i)
 }
<|> do {
    keywordParser "set";
    i <- identifierParser;
    keywordParser "=";
    keywordParser "{";
    i' <- aExpParser;
    i'' <- many (do keywordParser ","; aExpParser);
    keywordParser "}";
    keywordParser ";";
    return (SetFullDeclaration i (SetValues (i':i'')))
    }
<|> do {
    keywordParser "stack";
    i <- identifierParser;
    keywordParser ";";
    return (StackDeclaration i )
}
```

The assignment parser is the same like the declaration one with the difference that we are not declaring a new variable but we are using an existing one to assign it a new value.

```
assignmentParser :: Parser Command
assignmentParser = do i <- identifierParser
                                        -- ARITHMETIC EXPRESSION
                      do keywordParser "="
                         a <- aExpParser
                         keywordParser ";"
                         return (AExpAssignment i a)
                      <|>
                                        -- BOOL
                      do keywordParser "<-"
                         b <- bExpParser
                         keywordParser ";"
                         return (BExpAssignment i b)
                      <|>
                                        -- ARRAY
                      -- a[0] = 1;
                      do keywordParser "["
                         i' <- aExpParser
                         keywordParser "]"
                         keywordParser ":="
                         a <- aExpParser
                         keywordParser ";"
                         return (ArrayAssignmentSingleValue i i' a)
                      <|>
                      -- a:= [1,2,3];
                      do keywordParser ":="
                         keywordParser "["
                         i' <- aExpParser
                         i'' <- many (do keywordParser ","; aExpParser)
                         keywordParser "]"
                         keywordParser ";"
                         return (ArrayAssignmentValues i (ArrayValues (i':i'')))
                      <|>
                      -- a := b where a,b -> array
                      do keywordParser ":="
                         x <- identifierParser
                         keywordParser ";"
                         return (ArrayAssignmentValues i (ArrayExpVariable x))
                      <|>
```

```
        <|>
                            -- SET
        -- a add 1
        do keywordParser "add"
           a <- aExpParser
           keywordParser ";"
           return (SetAssignmentSingleValue i a)
        <|>
        -- a := {1,2,3}
        do keywordParser ":="
           keywordParser "{"
           i' <- aExpParser
           i'' <- many (do keywordParser ","; aExpParser)
           keywordParser "}"
           keywordParser ";"
           return (SetAssignmentValues i (SetValues (i':i'')))
        <|>
        -- a := (set) b
        do keywordParser ":= (set)"
           x <- identifierParser
           keywordParser ";"
           return (SetAssignmentValues i (SetExpVariable x))
                            -- STACK
        <|>
        do keywordParser "push"
           x <- aExpParser
           keywordParser ";"
           return (StackPushValue i x)
        <|>
        do keywordParser "pop"
           keywordParser ";"
           return (StackPopValue i)
--
```

The following is the skip parser:

```
skipParser :: Parser Command
skipParser = do keywordParser "skip"
                keywordParser ";"
                return Skip
```

Here we have the if-then-else parser:

```
ifThenElseParser :: Parser Command
ifThenElseParser = do keywordParser "if"
                      keywordParser "("
                      b <- bExpParser
                      keywordParser ")"
                      keywordParser "{"
                      thenCase <- programParser
                      do keywordParser "}"
                         keywordParser "else"
                         keywordParser "{"
                         elseCase <- programParser
                         keywordParser "}"
                         return (IfThenElse b thenCase elseCase)
                       <|>
                       do keywordParser "}"
                          return (IfThenElse b thenCase [Skip])
```

Here the while parser:

```
whileParser :: Parser Command
whileParser = do keywordParser "while"
                 keywordParser "("
                 b <- bExpParser
                 keywordParser ")"
                 keywordParser "{"
                 p <- programParser
                 keywordParser "}"
                 return (While b p)
```

Then the for parser:

```
forParser :: Parser Command
forParser = do keywordParser "for"
               keywordParser "("
               counter <- variableDeclParser
               booleanExp <- bExpParser
               keywordParser ";"
               identifier <- identifierParser;
               keywordParser "++";
               keywordParser ")"
               keywordParser "{"
               body <- programParser
               keywordParser "}"
               return (ForIncrement counter booleanExp identifier body)
               <|>
               do keywordParser "for"
                  keywordParser "("
                  counter <- variableDeclParser
                  booleanExp <- bExpParser
                  keywordParser ";"
                  identifier <- identifierParser;
                  keywordParser "--";
                  keywordParser ")"
                  keywordParser "{"
                  body <- programParser
                  keywordParser "}"
                  return (ForDecrement counter booleanExp identifier body)
```

And the comment parser:

```
commentParser :: Parser Command
commentParser = do keywordParser "{-"
                   whileCommand1

whileCommand1 = do
                   whileCommand
                   do keywordParser "}"
                      return Skip;
                   <|>
                   do next' <- readNext
                      whileCommand1
whileCommand = do
                   next' <- readNext
                   unless (next' == '-') $ do
                   whileCommand
```

8.1.5 Program Parser

```
programParser :: Parser [Command]
programParser = do many commandParser
```

In this way, I can have a program which is empty or a program with a potentially infinite number of commands. To execute the parser the executeParser function has been defined:

```
executeParser :: String -> ([Command], String)
executeParser s = case result of
                Nothing -> error "Parsed not executed"
                Just (cs, str) -> (cs, str)
              where (P p) = programParser
                    result = p s
```

so that the result of the parser is a list of commands and a string. If the parsing fails, in the sense that there are some errors in the execution of the parser module,

then an error is raised; if instead the parsing fails, in the sense that the structure of the program given as input doesn't fit the grammar of IMP, the list of commands contains the part of the program successfully parsed and, the string, the rest of the program which has not been parsed; if lastly it succeds, then the string will be empty and the list of commands will contain all the commands to give as input to the interpreter which is going to evaluate the final result (if it there exists).

8.2 Interpreter Implementation

The main function of the interpreter is to take the output of the parser and work on it. When indeed the parser computes correctly the program, as we said before it returns and internal representation of it and the interpreter has to compute the final result. For example we have a program to execute defined as follow:

```
int b = 0;
int numIter = 0;
for (int i = 0; i<6; i++){
    b = b + i * 2;
    numIter = numIter + 1;
}

int d = 0;
int numIter2 = 0;

for (int y = 5; y>=0; y--){
    d = d + y * 2;
    numIter2 = numIter2 + 1;
}
```

then the internal representation of the program which is given as output from the parser and given as input to the interpreter is:

```
 -- INPUT PARSED --

AExpDeclaration "b" (AExpConstant 0)
AExpDeclaration "numIter" (AExpConstant 0)
ForIncrement (AExpDeclaration "i" (AExpConstant 0)) (Less (AExpVariable "i") (AExpConstant 6)) "i" [AExpAssignment "b" (Add (AExpVar
iable "b") (Mul (AExpVariable "i") (AExpConstant 2))),AExpAssignment "numIter" (Add (AExpVariable "numIter") (AExpConstant 1))]
AExpDeclaration "d" (AExpConstant 0)
AExpDeclaration "numIter2" (AExpConstant 0)
ForDecrement (AExpDeclaration "y" (AExpConstant 5)) (GreaterEqual (AExpVariable "y") (AExpConstant 0)) "y" [AExpAssignment "d" (Add
(AExpVariable "d") (Mul (AExpVariable "y") (AExpConstant 2))),AExpAssignment "numIter2" (Add (AExpVariable "numIter2") (AExpConstant
 1))]

 -- INPUT NOT PARSED --


 -- STATE --

Dictionary [("b",IntegerType 30),("numIter",IntegerType 6),("i",IntegerType 6),("d",IntegerType 30),("numIter2",IntegerType 6),("y",
IntegerType (-1))]
```

To make this possible we will need to evaluate arithmetic expressions, boolean expressions and commands. The results of every single evaluation is a Maybe type so that we can distinguish easily a failure (and raise an error with a custom message to help the user in understanding) from a correct execution.

8.2.1 Arithmetic Expressions Evaluation

The evaluation of an arithmetic expression takes a state as input (which represents an internal representation of the memory) and an arithmetic expression (coded in the internal reprentation way) and gives Just Int or Nothing depending of the success or failure of the computation.

```
aExpEval :: State -> AExp -> Maybe Int
aExpEval _ (AExpConstant c) = Just c
aExpEval s (AExpVariable v) =
  case get s v of
    Just (IntegerType r) -> Just r
    Just (BooleanType _) -> error "Variable of type boolean!"
    Just (ArrayType _) -> error "Variable of type array!"
    Just (SetType _) -> error "Variable of type set!"
    Nothing -> error "Variable not found!"
aExpEval s (ValueFromArray v i) =
  case get s v of
    Just (IntegerType _) -> error "Variable of type integer!"
    Just (BooleanType _) -> error "Variable of type boolean!"
    Just (SetType _) -> error "Variable of type set!"
    Just (ArrayType r) -> Array.read i' r
                          where Just i' = aExpEval s i
    Nothing -> error "Variable not found!"
aExpEval s (Add a b) = (+) <$> aExpEval s a <*> aExpEval s b
aExpEval s (Sub a b) = (-) <$> aExpEval s a <*> aExpEval s b
aExpEval s (Mul a b) = (*) <$> aExpEval s a <*> aExpEval s b
aExpEval s (Length i) =
    case get s i of
        Just (ArrayType a) -> Just (Utils.len a)
        Just (SetType a) -> Just (Utils.len a)
        Just (StackType a) -> Just (Utils.len a)
        Just _ -> error "Function LEN not defined for this typ
        Nothing -> error "Variable not found"
```

8.2.2 Boolean Expression Evaluation

The evaluation of a boolean expression takes a state as input and gives Just Bool or Nothing depending of the success or failure of the computation.

```
bExpEval :: State -> BExp -> Maybe Bool
bExpEval _ (BExpConstant b) = Just b
bExpEval s (BExpVariable v) =
  case get s v of
    Just (IntegerType _) -> error "Variable of type integer!"
    Just (BooleanType r) -> Just r
    Just (ArrayType _) -> error "Variable of type array!"
    Just (SetType _) -> error "Variable of type set!"
    Just (StackType _) -> error "Variable of type stack!"
    Nothing -> error "Variable not found"
bExpEval s (Not b) = not <$> bExpEval s b
bExpEval s (Or a b) = (||) <$> bExpEval s a <*> bExpEval s b
bExpEval s (And a b) = (&&) <$> bExpEval s a <*> bExpEval s b
bExpEval s (Less a b) = (<) <$> aExpEval s a <*> aExpEval s b
bExpEval s (LessEqual a b) = (<=) <$> aExpEval s a <*> aExpEval s b
bExpEval s (Greater a b) = (>) <$> aExpEval s a <*> aExpEval s b
bExpEval s (GreaterEqual a b) = (>=) <$> aExpEval s a <*> aExpEval s b
bExpEval s (Equal a b) = (==) <$> aExpEval s a <*> aExpEval s b
bExpEval s (NotEqual a b) = (/=) <$> aExpEval s a <*> aExpEval s b
bExpEval s (IsEmpty i) =
    case get s i of
      Just (SetType a) -> Just (Utils.isEmpty a)
      Just (ArrayType a) -> Just (Utils.isEmpty a)
      Just (StackType a) -> Just (Utils.isEmpty a)
      Just _ -> error "Cannot apply function to variable"
      Nothing -> Nothing
```

8.2.3 Array Expression Evaluation

```
-- ARRAY EVAL
arrayExpEval :: State -> ArrayExp -> Maybe (Array Int)
arrayExpEval s (ArrayValues a) = if hasFailed
                                 then Nothing
                                 else Just $ map (\v -> case v of Just x -> x) r
                                   where hasFailed = or $ map (\v -> case v of
                                           Nothing -> True
                                           Just x -> False) r
                                         r = map (\exp -> aExpEval s exp) a


arrayExpEval s (ArrayExpVariable v) =
  case get s v of
    Just (IntegerType _) -> error "Assignment of an integer value to an array one not a
    Just (BooleanType _) -> error "Assignment of an boolean value to an array one not a
    Just (SetType _) -> error "Assignment of a set value to an array one not allowed!"
    Just (ArrayType a) -> Just a
    Nothing -> error "Variable to assign not found"
```

8.2.4 Set Expression Evaluation

```
--SET EVAL
setExpEval :: State -> SetExp -> Maybe (Set Int)
setExpEval s (SetValues a) = if hasFailed
                                    then Nothing
                                    else Just $ map (\v -> case v of Just x -> x) r
                                      where hasFailed = or $ map (\v -> case v of
                                              Nothing -> True
                                              Just x -> False) r
                                            r = map (\exp -> aExpEval s exp) a


setExpEval s (SetExpVariable v) =
  case get s v of
    Just (IntegerType _) -> error "Assignment of an integer value to a set one not allo
    Just (BooleanType _) -> error "Assignment of an boolean value to a set one not allo
    Just (ArrayType a) -> Just (Set.fullDeclareSet a)
    Just (SetType b) -> Just b
    Nothing -> error "Variable to assign not found"
```

8.2.5 Set Expression Evaluation

```
-- STACK EVAL
stackExpEval :: State -> StackExp -> Maybe (Stack Int)
stackExpEval s (StackValues a) = if hasFailed
                                 then Nothing
                                 else Just $ map (\v -> case v of Just x -> x) r
                                   where hasFailed = or $ map (\v -> case v of
                                           Nothing -> True
                                           Just x -> False) r
                                         r = map (\exp -> aExpEval s exp) a


stackExpEval s (StackExpVariable v) =
  case get s v of
    Just (ArrayType a) -> Just (Set.fullDeclareSet a)
    Just (SetType b) -> Just b
    Just _ -> error "Assignment not allowed!"
    Nothing -> error "Variable to assign not found"
```

8.2.6 Commands Evaluation

The possible commands are the one listed above at the beginning of the documentation:

- Skip;
- AExpDeclaration;
- BExpDeclaration;
- ArrayDeclaration;
- AExpAssignment;
- BExpAssignment;
- ArrayAssignmentSingleValue;
- ArrayAssignmentValues;
- SetAssignmentSingleValue

- SetAssignmentValues
- StackPushValue
- StackPopValue
- IfThenElse;
- While
- ForIncrement
- ForDecrement

Declarations Eval

In this section of our interpreter we define the declaration evaluations

```
                                                -- START DECLARATION AREA --

-- EXECUTE INTEGER DECLARATION --
executeCommands s ((AExpDeclaration v exp) : cs) =
  case aExpEval s exp of
    Just ex' -> case get s v of
                  Just _ -> error "Variable already declared!"
                  Nothing -> executeCommands (insert s v (IntegerType ex')) cs
    Nothing -> error "Invalid aExp"


-- EXECUTE BOOLEAN DECLARATION --
executeCommands s ((BExpDeclaration v exp) : cs) =
  case bExpEval s exp of
    Just ex' -> case get s v of
                  Just _ -> error "Variable already declared!"
                  Nothing -> executeCommands (insert s v (BooleanType ex')) cs
    Nothing -> error "Invalid bExp"


-- EXECUTE ARRAY DECLARATION --
executeCommands s ((ArrayDeclaration v exp) : cs) =
  case aExpEval s exp of
    Just ex' -> case get s v of
                  Just _ -> error "Variable already declared!"
                  Nothing -> executeCommands (insert s v (ArrayType a)) cs
                             where a = declare ex'
    Nothing -> error "Invalid size!"
```

```
-- EXECUTE ARRAY DECLARATION --
executeCommands s ((ArrayDeclaration v exp) : cs) =
  case aExpEval s exp of
    Just ex' -> case get s v of
                  Just _ -> error "Variable already declared!"
                  Nothing -> executeCommands (insert s v (ArrayType a)) cs
                             where a = declare ex'
    Nothing -> error "Invalid size!"


executeCommands s ((ArrayFullDeclaration v exp) : cs) =
    case get s v of
     Just _ -> error "Variable already declared!"
     Nothing -> case arrayExpEval s exp of
                  Just b -> executeCommands (insert s v (ArrayType b)) cs
```

```
-- EXECUTE SET DECLARATION --
executeCommands s ((SetDeclaration v) : cs) =
    case get s v of
     Just _ -> error "Variable already declared!"
     Nothing -> executeCommands (insert s v (SetType []) ) cs


executeCommands s ((SetFullDeclaration v exp) : cs) =
    case get s v of
     Just _ -> error "Variable already declared!"
     Nothing -> case setExpEval s exp of
                  Just b -> executeCommands (insert s v (SetType c)) cs
                    where
                     c = Set.arrayToSet b
```

Assignments Eval

In this section of our interpreter we define the assignments evaluations

```
                                        -- START ASSIGMENT AREA --

-- EXECUTE INTEGER ASSIGMENT --
executeCommands s ((AExpAssignment v exp) : cs) =
  case get s v of
    Just (IntegerType _) -> executeCommands (insert s v (IntegerType exp')) cs
                            where Just exp' = aExpEval s exp
    Just (BooleanType _) -> error "Assignment of a boolean value to an aExp variable not allowed!"
    Just (ArrayType _) -> error "Assignment of an array value to an aExp variable not allowed!"
    Just (SetType _) -> error "Assignment of a set value to an aExp variable not allowed!"
    Nothing -> error "Undeclared variable!"


-- EXECUTE BOOLEAN ASSIGMENT --
executeCommands s ((BExpAssignment v exp) : cs) =
  case get s v of
    Just (BooleanType _) -> executeCommands (insert s v (BooleanType exp')) cs
                            where Just exp' = bExpEval s exp
    Just (IntegerType _) -> error "Assignment of an integer value to a bExp variable not allowed!"
    Just (ArrayType _) -> error "Assignment of an array value to an bExp variable not allowed!"
    Just (SetType _) -> error "Assignment of a set value to an bExp variable not allowed!"
    Nothing -> error "Undeclared variable!"
```

```
-- EXECUTE ARRAY ASSIGMENT SINGLE VALUE --
executeCommands s ((ArrayAssignmentSingleValue v i exp) : cs) =
  case get s v of
    Just (ArrayType a) -> case aExpEval s exp of
                            Just r -> executeCommands (insert s v (ArrayType exp')) cs
                              where Just exp' = Array.write i' r a
                                                  where Just i' = aExpEval s i
                            Nothing -> error "The expression you want to assign is not valid!"
    Just (SetType a) -> error "Cannot access a specific position of a set!"
    Just (IntegerType _) -> error "Assignment of an integer value to an array variable not allowed!"
    Just (BooleanType _) -> error "Assignment of an boolean value to an array variable not allowed!"
    Nothing -> error "Undeclared variable!"


-- EXECUTE ARRAY ASSIGMENT MULTI VALUE --
executeCommands s ((ArrayAssignmentValues v exp) : cs) =
  case get s v of
    Just (ArrayType a) -> case arrayExpEval s exp of
                            Just b -> if length a == length b
                                      then executeCommands (insert s v (ArrayType b)) cs
                                      else error "Length not valid!"
                            Nothing -> error "One of the aExp evaluation of the array you want to assign failed"
    Just (SetType _) -> case arrayExpEval s exp of
                            Just b -> executeCommands (insert s v (SetType c)) cs
                                where
                                  c = Set.arrayToSet b
                            Nothing -> error "One of the aExp evaluation of the array you want to assign failed"
    Just (IntegerType _) -> error "Assignment of an aExp value to an array variable not allowed!"
    Just (BooleanType _) -> error "Assignment of an bExp value to an array variable not allowed!"
    Nothing -> error "Undeclared variable!"
```

```
-- EXECUTE SET ASSIGMENT SINGLE VALUE --
executeCommands s ((SetAssignmentSingleValue set exp) : cs) =
  case get s set of
    Just (SetType a) -> case aExpEval s exp of
                          Just r -> executeCommands (insert s set (SetType exp1)) cs
                            where exp1 = Set.insertSet r a
                          Nothing -> error "The expression you want to assign is not valid!"
    Just _ -> error "Cannot Add!"
    Nothing -> error "Undeclared variable!"


-- EXECUTE SET ASSIGMENT MULTI VALUE --
executeCommands s ((SetAssignmentValues v exp) : cs) =
  case get s v of
    Just (SetType a) -> case setExpEval s exp of
                          Just b -> executeCommands (insert s v (SetType c)) cs
                              where
                               c = Set.arrayToSet b
                          Nothing -> error "The expression you want to assign is not valid!"
    Just (ArrayType a) -> case setExpEval s exp of
                          Just b -> executeCommands (insert s v (ArrayType c)) cs
                              where
                               c = Set.arrayToSet b
                          Nothing -> error "The expression you want to assign is not valid!"
    Just _ -> error "Assignment not allowed!"
    Nothing -> error "Undeclared variable!"
```

We predicted that the only way to populate or declare a stack is to use the push and pop functions:

```
executeCommands s ((StackPushValue v exp) : cs) =
  case get s v of
    Just (StackType a) -> case aExpEval s exp of
                            Just b -> executeCommands (insert s v (StackType c)) cs
                              where
                               c = Stack.pushValue a b
                            Nothing -> error "The expression you want to assign is not valid!"
    Just _ -> error "Assignment not allowed!"
    Nothing -> error "Undeclared variable!"


executeCommands s ((StackPopValue v) : cs) =
  case get s v of
    Just (StackType a) -> executeCommands (insert s v (StackType c)) cs
                          where
                           c = Stack.popValue a
    Just _ -> error "Assignment not allowed!"
    Nothing -> error "Undeclared variable!"
```

Chapter 9

Running Interpreter

In order to run the interpreter, you need to open the command line console and reach the path **../src**.

```
C:\Users\ranie\Desktop\IMP-Interpreter\src>
```

Then you insert the command "**gchi**"

```
C:\Users\ranie\Desktop\IMP-Interpreter\src>ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

Then the command "**:l main**"

```
C:\Users\ranie\Desktop\IMP-Interpreter\src>ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> :l main
[1 of 9] Compiling Array              ( Array.hs, interpreted )
[2 of 9] Compiling Grammar            ( Grammar.hs, interpreted )
[3 of 9] Compiling Dictionary         ( Dictionary.hs, interpreted )
[4 of 9] Compiling Parser             ( Parser.hs, interpreted )
[5 of 9] Compiling Stack              ( Stack.hs, interpreted )
[6 of 9] Compiling Utils              ( Utils.hs, interpreted )
[7 of 9] Compiling Set                ( Set.hs, interpreted )
[8 of 9] Compiling Interpreter        ( Interpreter.hs, interpreted )
[9 of 9] Compiling Main               ( main.hs, interpreted )
Ok, 9 modules loaded.
*Main>
```

And finally run the function "**main**"

Now you can either run a program inline using the option **1)** or run a script by specifying the path of the script. You can use the script in the "**../source_file**" root.