Formal Methods

Donato Francioso

Contents

Chapter 1

Introduction

The aim of this project is to build an interpreter for a simple imperative language which is able to parse its instructions and execute them. It is always able to keep track of an environment in which are saved all the variables involved in the process. It involves the basic structures and elements of common imperative language. The types involved in this languare are:

- **Int:** represents Integer numbers, so they can have positive or negative value;
- **Array:** represents the Array, a data structures that contains a group of finite value. In this case the value are only Int value.

The control structures and operation that can be used are the **assignment**, **skip**, **IfElse** and **while**.

Chapter 2

Grammar

In this chapter will be defined the grammar of the language using the BNF.

2.1 Preliminary definitions

- <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- <natural> ::= <digit> | <digit> <natural>
- <integer> ::= -<natural> | <natural>
- <lower> ::= a-z
- <upper> ::= A-Z
- <alphaNum> ::= <lower><alphaNum> | <upper><alphaNum> | <lower> | <upper>
- <ident> ::= <lower> <alphaNum>

2.2 Arithmetical Expression

- <aExp> ::= <aTerm> + <aExp> | <aTerm> - <Aexp> | <aTerm>
- <aTerm> ::= <aFactor>*< aTerm> | <aFactor> / <aTerm> | <aFactor
- <aFactor> ::= (<aExp>) | - <aFactor> | <ident> | <integer>

2.3 Boolean Expression

- <bExp> ::= <bTerm> 'oo' <bExp> | <bTerm>
- <bTerm> ::= <bFactor> 'aa' <bTerm> | <bFactor>

- <bFactor> ::= (<bExp>) | 'V' | 'F' | 'nn' <bFactor> | <bComparison>
- <bComparison> ::= <aExp> '=' <aExp> | <aExp> '<=' <aExp> | <aExp> '>=' <aExp> | <aExp> '<' <aExp> | <aExp> '>' <aExp>

2.4 Commands

- <program> ::= <command> <program> | <program>
- <command> ::= <assignment> | <ifElse> | <whileST> | 'skip' '.'
- <assignment>
- <ifElse>
- <whileST>

2.4.1 Array

The interpreter can manage array of Integers.

- <array> ::= '#' <aItem> '#' | <ident>
- <aItem> ::= <integer> ',' <aItem> | <integer>

An array is defined as:
x := #1,2,3#.

We can use only one item of the array using the position: y := x#1#. In memory y is equal to 2.

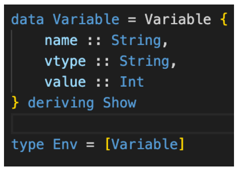
Chapter 3

Implementation

The following pages descrives the technical details and choices adopted for the implementation of the interpreter.

3.1 Environment

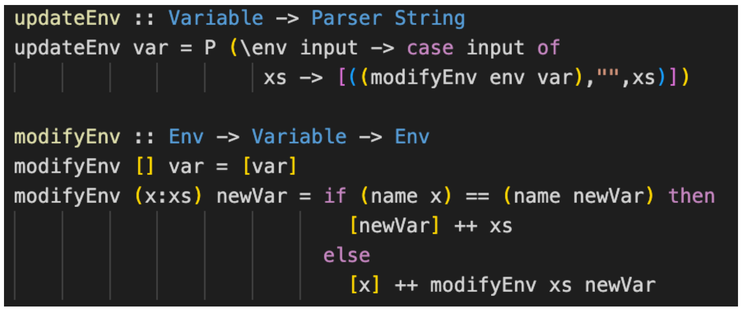
The interpreter need something that siulates memory. For this reason is created the environment.



```haskell
data Variable = Variable {
    name :: String,
    vtype :: String,
    value :: Int
} deriving Show

type Env = [Variable]
```

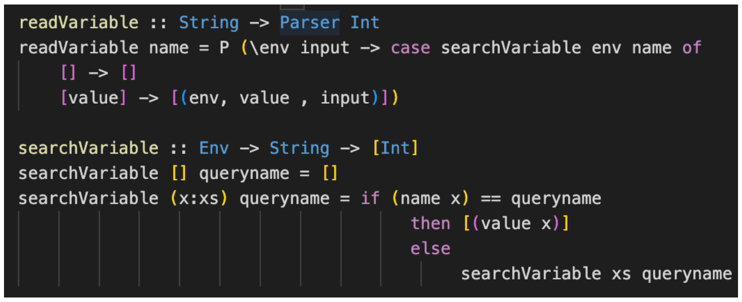
The type Variable contain the name of the *variable*, the *type* (int or bool) and is *real value*. Additional functions are also necessary for read and write on the environment.



```haskell
updateEnv :: Variable -> Parser String
updateEnv var = P (\env input -> case input of
                     xs -> [((modifyEnv env var),"",xs)])

modifyEnv :: Env -> Variable -> Env
modifyEnv [] var = [var]
modifyEnv (x:xs) newVar = if (name x) == (name newVar) then
                            [newVar] ++ xs
                          else
                            [x] ++ modifyEnv xs newVar
```

The function modifyEnv takes as input an environment and a variable, if it is alredy in the environment then overwrite it, otherwise the variable is added to the environment.


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

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

The functions searchVariable and readVariable are necessary to search for a variable in the environment given the name, and to return its value if exist.

3.2 Parser

A parser can be seen as a function that takes a string in input and partially consume the input, providing as output a generic type and the remaining string to process. During the execution of the program the Environment could change, so it is important to keep track of these changes. For do that the parser is defined as:

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

Parser type is then defined as an instance of Functor, Applicative, Monad and Alternative.

- **Functor:** It apply a fmap to the parser, applying the function to the feneric type the Parser holds keepin the Parser structur
- **Applicative:** it can be applied sequentially ignoring the results of the previously applied parsers.
- **Monad:** it can be applied sequentially using the results of the previously applied parsers.
- **Alternative:** a parser may fail and if so another parser might be executed instead.

Following are showed Parser for Arithmetic and Boolean expression and for commands. I use removeSpaces for consider the possible space inserted between commands.

Arithmetic Parser: The Parser for Arithmetic expression is divided in three sub-parsers:

```
aExp :: Parser Int
aExp = do{
          t <- aTerm;
```

```
        symbol "+";
        a <- aExp;
        return (t + a);
} <|>
do {
        t <- aTerm;
        symbol "-";
        a <- aExp;
        return (t - a);
} <|>
do {aTerm}

aTerm :: Parser Int
aTerm =
do{
        f <- aFactor;
        symbol "*";
        a <- aTerm;
        return (f * a);
} <|>
do{
        f <- aFactor;
        symbol "/";
        a <- aTerm;
        return (f `div` a);
} <|>
do{aFactor}


aFactor :: Parser Int
aFactor =
do {
        symbol "(";
        a <- aExp;
        symbol ")";
        return a
} <|>
do{
        i <- ident;
        readVariable i;
}<|>
do{
        x <- ident;
        symbol "#";
        i <- aExp;
        symbol "#";
        readVariable ( x ++ "#" ++ (show i) ++ "#" );
}<|>
do{
        integer;
}
```

Boolean Parser: The same is created for Boolean expression but int his case we have four sub-parsers.

```
bExp :: Parser Bool
bExp = do{
        b1 <- bTerm;
        symbol "oo";
        b2 <- bExp;
        return (b1 || b2);
}<|>
do{bTerm;}

bTerm :: Parser Bool
bTerm =
do {
        b1 <- bFactor;
        symbol "aa";
        b2 <- bTerm;
        return (b1 && b2)
}<|>
do{bFactor;}

bFactor :: Parser Bool
bFactor =
do{
        symbol "(";
        b <- bExp;
        symbol ")";
        return b;
}<|>
do{
        symbol "V";
        return True;
}<|>
do{
        symbol "F";
        return False;
}<|>
do{
        symbol "nn";
        b <- bFactor;
        return (not b);
} <|>
do{bComparison;}


bComparison :: Parser Bool
bComparison =
do {
        a <- aExp;
        symbol "=";
        b <- aExp;
        return (a == b);
}<|>
do{
        a <- aExp;
        symbol "<=";
        b <- aExp;
        return (a <= b);
}<|>
do{
        a <- aExp;
        symbol ">=";
        b <- aExp;
        return (a >= b);
}<|>
do{
        a <- aExp;
        symbol ">";
        b <- aExp;
        return (a > b);
}<|>
do{
        a <- aExp;
        symbol "<";
        b <- aExp;
        return (a < b);
}<|>
do{
        a <- aExp;
        symbol "!!";
        b <- aExp;
        return (a /= b)
}
```

Command Parser: The program is defined recursively as a sequence of commands which are definded as follow.

```
program :: Parser String
program = do{
        do {
                command;
                program;
        }<|>
        do {
                command;
        }
}

command :: Parser String
command = do{
        removeSpaces;
        assignment;}
<|>
do{
        removeSpaces;
        ifElse;}
<|>do{
        removeSpaces;
        whileST;}
<|>
do{
        removeSpaces;
        symbol "skip";
        symbol ".";}

assignment :: Parser String
assignment =
do{
        x <- ident;
        symbol ":=";
        v <- aExp;
        symbol ".";
        updateEnv Variable{name = x, vtype = "Integer", value = v};}
<|> do{ -- x = array
        x <- ident;
        symbol ":=";
        a <- array;
        symbol ".";
        saveArray x a;
}<|> --x = y#1# x = elemento in posizione 1
do{
        i1 <- ident;
        symbol ":=";
        i2 <- ident;
        symbol "#";
        a <- aExp;
        symbol "#";
        symbol ".";
        val <- readVariable (i2 ++ "#" ++ (show a) ++ "#" );
        updateEnv Variable {name = i1, vtype ="Integer", value = val};
}<|> -- y#1# = x elemento in posizione 1 uguale a x
do{
        i1 <- ident;
        symbol "#";
        a <- aExp;
        symbol "#";
        symbol ":=";
        val <- aExp;
        symbol ".";
        array <- readArray i1;
        (if length array <= a then empty else updateEnv Variable{name =
}
<|> -- array concatenated to another array
do{
        i <- ident;
        symbol ":=";
        a1 <- array;
        removeSpaces;
        symbol "conc";
        removeSpaces;
        a2 <- array;
        symbol ".";
        saveArray i (a1 ++ a2);
}

--Consider If without Else
ifElse :: Parser String
ifElse = do{
        symbol "if";
        removeSpaces; symbol "(";
        removeSpaces; b <- bExp;
        removeSpaces; symbol ")";
        if b then
        do{
                removeSpaces; symbol "{";
                        removeSpaces; program;
                        removeSpaces; symbol "}else{";
                        removeSpaces; consumeProgram;
                        removeSpaces; symbol "}";
                removeSpaces; return ""
            }
            else
            do{
                symbol "{";
                    removeSpaces; consumeProgram;
                    symbol "}";
                removeSpaces; symbol "else{";
                    removeSpaces; program;
                    removeSpaces; symbol "}";
                removeSpaces; return "";
            }
    }<|>
    do {
            symbol "if";
            removeSpaces; symbol "(";
            removeSpaces; b <- bExp;
            removeSpaces; symbol ")";
            if b then
            do{
                removeSpaces; symbol "{";
                    removeSpaces; program;
                    removeSpaces; symbol "}";
                removeSpaces; return "";
            }
            else
            do{
                symbol "{";
                    removeSpaces; consumeProgram;
                    symbol "}";
                return "";
            }
    }

--Consider while and do-while
    whileST :: Parser String
    whileST = do{
            removeSpaces;
            w <- consumeWhileST;
            executeWhile w;
            symbol "while";
            removeSpaces; symbol "(";
            removeSpaces; b <- bExp;
            removeSpaces; symbol ")";
            removeSpaces; symbol "{";
                if b then
```

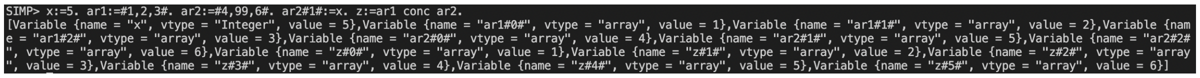
```
                    do {
                              program;
                              symbol "}";
                    executeWhile w;
                    whileST;}
          else
          do{
                    consumeProgram;
                    symbol "}";
          return "";}
}<|>
do{
w <- consumeWhileST;
executeWhile w;
symbol "do";
removeSpaces; symbol "{";
          program;
          removeSpaces; symbol "}";
removeSpaces; symbol "while";
removeSpaces; symbol "(";
removeSpaces; b <- bExp;
removeSpaces; symbol ").";
if b then
do{
          executeWhile w;
          whileST;
}
else
do{
          return "";
}
}
```

Chapter 4

Test

Now pass to show some example:

First we have simple assignment with a concatenation of two array. We have:

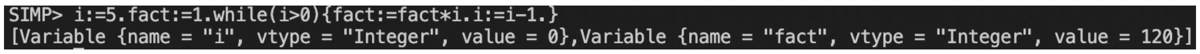


```
SIMP> x:=5. ar1:=#1,2,3#. ar2:=#4,99,6#. ar2#1#:=x. z:=ar1 conc ar2.
[Variable {name = "x", vtype = "Integer", value = 5},Variable {name = "ar1#0#", vtype = "array", value = 1},Variable {name = "ar1#1#", vtype = "array", value = 2},Variable {nam
e = "ar1#2#", vtype = "array", value = 3},Variable {name = "ar2#0#", vtype = "array", value = 4},Variable {name = "ar2#1#", vtype = "array", value = 5},Variable {name = "ar2#2#
", vtype = "array", value = 6},Variable {name = "z#0#", vtype = "array", value = 1},Variable {name = "z#1#", vtype = "array", value = 2},Variable {name = "z#2#", vtype = "array
", value = 3},Variable {name = "z#3#", vtype = "array", value = 4},Variable {name = "z#4#", vtype = "array", value = 5},Variable {name = "z#5#", vtype = "array", value = 6}]
```

```
x:=5.
ar1:=#1,2,3#. ar2:=#4,99,6#.
ar2#1#:=x.
z:=ar1 conc ar2.

At the end of the execution in memory we have modified the variables:
i = 0, ar2 = #3,4,5# and z = #1,2,3,4,5,6#
```

This example show how compute factorial of a number. We have:

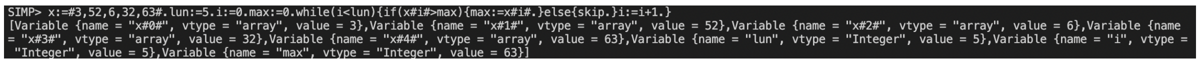


```
SIMP> i:=5.fact:=1.while(i>0){fact:=fact*i.i:=i-1.}
[Variable {name = "i", vtype = "Integer", value = 0},Variable {name = "fact", vtype = "Integer", value = 120}]
```

```
i:=5.
fact:=1.
while(i > 0){
        fact:=fact*i.i:=i-1.}

At the end of the execution in memory we have modified the variables:
fact = 120 and i = 0.
```

This example show how find a max value in an array. We have:



```
SIMP> x:=#3,52,6,32,63#.lun:=5.i:=0.max:=0.while(i<lun){if(x#i#>max){max:=x#i#.}else{skip.}i:=i+1.}
[Variable {name = "x#0#", vtype = "array", value = 3},Variable {name = "x#1#", vtype = "array", value = 52},Variable {name = "x#2#", vtype = "array", value = 6},Variable {name
= "x#3#", vtype = "array", value = 32},Variable {name = "x#4#", vtype = "array", value = 63},Variable {name = "lun", vtype = "Integer", value = 5},Variable {name = "i", vtype =
 "Integer", value = 5},Variable {name = "max", vtype = "Integer", value = 63}]
```

```
x:=#3,52,6,32,63#.
lun:=5.
i:=0.
max:=0.
while(i<lun){
        if(x#i# > max)
                {max:=x#i#.}
        else
                {skip.}
        i:=i+1.}
```

At the end of the execution in memory we have modified the variables: max = 63 and i = 5.