# Università degli Studi di Bari “Aldo Moro”

“Dipartimento di Informatica”


Case Study in Formal Methods for Computer Science

“LCParser an imperative language interpreter”

A.Y. 2020/2021


Student:

Lorenzo Capocchiano

Professor:

Giovanni Pani

# Index

# 1. Introduction

This document describes the implementation of a simple imperative language interpreter and how it works.

An interpreter translates code into machine code, instruction by instruction – the CPU executes each instruction before the interpreter moves on to translate the next instruction.

For the scope of this project, it was decided to implement only few basis commands, described as follows:

assignment: Istruction to perform the assignment of a value to a variable, identified by an alphabetic character (for simplicity it was decided that character was sufficient to identify enough variables)

ifThenElse: instruction that implements the conditional construct

while: instruction for loop

LCParser uses the eager strategy for evaluating programs. It means that evaluation occurs as soon as possible and then the value of the evaluation is passed to function.

# 2. Grammar

The following rules are ordered from a higher level of abstraction to lower one.

(Elements in apices are the keywords used to parse the program correctly)

program ::= <cmd> <program> | <cmd>
The program can be composed of a lot of commands

cmd ::= <ifThenElse> | <assignment>|<while>|<comment>
As mentioned above, the commands implemented are the conditional and iterative constructs and the assignment of variables.

ifThenElse ::= '*if* ' <BexprAND> '*then*' <Program> '*endif*' | '*if*' <BexprAND> '*then*' <Program> '*else*' <Program> '*endif*'
The conditional construct can have or not the else branch, moreover, in order to avoid any ambiguity it is necessary to use a keyword for the closure of the instruction (endif).

assignment ::= variable '=' <expr> ';' | variable '=' <bexprAND> ';'
It is possible to assign to a variable both an arithmetical or Boolean expression.

bexprOR ::= <bexprAND> '*OR*' <bexprOR> | <bexprAND>
This instruction will eventually evaluate AND expression first and only then the OR one. In this way we have a priority for AND and OR evaluations.

bexprAND ::= <bexpr> '*AND*' <bexprAND> | <bexpr>
This instruction will evaluate AND expression if present or directly the pure Boolean expression

bexpr ::= '*True*'| '*False*'| <bexprAnd> | <compareTo> | variable | '*NOT*' <bexprOR>
Pure Boolean expression can be represented by the truth values, eventually their negation or the confrontation between two expressions. A Boolean value can also be represented by a Boolean variable.
compareTo ::= <expr> '<' <expr> | <expr> '>' <expr> | <expr> '==' <expr> | <bexprOR> '==' <bexprOR> | <expr> '<>' <expr>

The compareTo instruction enables to compare two expression or to verify if two expressions have the same arithmentic or Boolean value.

expr ::= <term>+<expr>|<term>-<expr>|<term>

As we have seen for Boolean expressions, also for the arithmetical we have a priority in the evaluation: we will evaluate * and / first and then + and -.

term ::= <factor>*<term>|<factor>/<term>|<factor>

factor ::= int|(<expr>)|intVariable

it represents the pure int value or the value of an Integer variable or an arithmetical expression within parenthesis

int ::= <nat>|-<nat>

the integer value is represented by a natural number or a natural number preceded by a minus sign

nat ::= <digit>|<digit> <nat>

the natural numbers are described as sequence of digits

digit ::= ‘0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’

space ::= ‘ ‘

# 3. Implementation

In order to keep track of the value of the variables we need an environment that will be keep fresh everytime we do an assignment. The environment has been defined in the interpreter as follows:

type Environment = [(Char, String, String)]

where the first element of the couple is the identifier of the variable (for simplicity an alphabetic char), the second element represent the type of the variable that can be "Integer" or "Boolean" and the last is the value of the variable.

The implemented parser is then defined as

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

The way with which the environment is updated is defined by these two methods

```
changeEnvironment :: Char -> String -> String -> Parser String
```

```
changeEnvironment varName varType varValue =
        P(\env inp -> case inp of
              xs -> [(modifyEnvironment env varName varType varValue,"",xs)])

modifyEnvironment :: Environment -> Char -> String -> String -> Environment
modifyEnvironment [] varName varType varValue = [(varName,varType,varValue)]
modifyEnvironment xs varName varType varValue =
                        if getName (head xs) == varName
                                then [(varName,varType,varValue)] ++ tail xs
                        else [head xs] ++ modifyEnvironment (tail xs) varName varType varValue
```

As you can see, 'modifyEnvironment' looks for the variable defined with the name given in input and, if the variable does not exist, it adds a new tuple to the environment, otherwise it will change the value of the existent one.

## 3.1. Instances of Functor, Applicative, Monad and Alternative for type Parser

It was created Instances of the Functor, Applicative and Monad classes for the parser type created before in order to combine parsers in a sequential way and allow them to work together.

This was made as follows:

```
    instance Functor Parser where
```

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

        instance Applicative Parser where
        -- pure :: a -> Parser a
              pure v = P (\env inp -> [(env,v,inp)])
              -- <*> :: Parser (a -> b) -> Parser a -> Parser b
              pg <*> px = P (\env inp -> case parse pg env inp of
                                        []          -> []
                                        [(env, g,out)] -> parse (fmap g px) env out)

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

Then we can instance Alternative too

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

The function ‘fmap’ is a generalization of map, and it applies the function g to the result value v of a parser p if the parser succeeds, and it propagates the failure otherwise.

‘Pure’ is a function that transforms a value into a parser that always succeeds with the value v as its result without consuming the input string.

The instance of Monad parser enables the use of do notation and we can thus perform sequential parsers that will fail if one of the parsers in the ‘chain’ fails.
The p >> q notation stands for: apply parser p, then give the output of p to q and apply q.

Finally, the alternative instance (<|>) allows to use parser in conditional way. That is, in the notation p <|> q, if p succeeds then return result of p, else apply q.

## 3.2. Basic parser functions

After the Parser type definition is finished, all functions that are useful for the interpreter have been implemented. A list of these functions follows with a brief description:

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

The simplest function is the item parser that parse the first character in a string and return it as output. The rest of the string Is also given as output in order to be used from the next parsers

```
        failure :: Parser a
        failure  = P(\env inp -> [])
```

Failure is the parser that fails everytime regardless of the input.

```
        sat  :: (Char -> Bool) -> Parser Char
        sat p = item >>= \x -> if p x then return x else failure
```

'sat' is useful to verify if an item satisfies a property p (given in input). If yes, sat function will give the char in output, else it will fail.

Example of usage:

```
ghci>parse (sat isDigit) [] "16"
[([],'1',"6")]
```

```
        identifier :: Parser Char
        identifier = sat isLetter
```

The identifier parser will verify the syntax for variables, as it was mentioned above, variables are for simplicity an alphabetical char.

## 3.3. Arithmetic expression

The arithmetic expression parser is divided in 3 parsers.

First of all we check if the element in the string is a term (see below) then, if the term is in addiction or subtraction with other expressions we will parse those expression, else we will return the term itself.

```
        expr :: Parser Int
        expr = do
                t <- term
                do
                symbol "+"
                ;e <- expr
                ;return (t + e)
                <|> do
                        symbol "-"
                        ;e <- expr
                        ;return (t - e)
                        <|> return t
```

The term parser is responsible to parse the multiplication and divisions (note that for simplicity divisions are integer divisions.

```
term :: Parser Int
        term = do
                factor >>= \f ->
                        do
                        symbol "*" >>= \c -> term >>= \t -> return (f * t)
                        <|> do
                                symbol "/" >>= \c -> term >>= \t -
> return (f `div`  t)
                                <|> return f
```

Finally we have the factor function that parse all integers, and also expressions that are in brackets and also take the value of an integer variable that is stored in the memory (if the variable is not present in the environment parser will fail).

```
        factor :: Parser Int
        factor =
                do
                        int
                        <|>
                        do
                                symbol "("
                                e <- expr
                                symbol ")"
                                return e
                                <|>
                                do
                                space
                                id <- identifier
```

```
                                space
                                vartype <- getVariableType id
                                if vartype == intType then
                                        do
                                        value <- getVariableValue id
                                        return (read value)
                                else failure
```

An example of usage of this parser (it will considered expr as main parser for arithmetical expressions):

```
ghci> parse expr [] "((55-10)*4)+2-(9*3+10)"
[([],145,"")]
```

## 3.4. Boolean expressions parser

Similar to arithmetic expression, Boolean expressions parser is divided into 3 parser (the main of them is bexprOR, that is the one that parse the OR operator (that can be seen as the sum for arithmetic)

```
        bexprOR :: Parser Bool
        bexprOR =do
                        b1 <- bexprAND
                        symbol "OR"
                        b2 <- bexprOR
                        return (b1 || b2)
                        <|>
                        bexprAND
```

The OR and AND parsers will recurse on the left, while on the right will try to change parser. In this way we will avoid infinite loop but at the same time we are capable to parse all type of expressions.

```
        bexprAND :: Parser Bool
        bexprAND =do
                        b1 <- bexpr
                        symbol "AND"
                        b2 <- bexprAND
                        return (b1 && b2)
                        <|>
                        bexpr
```

Finally the bexpr parser parse True and False strings to their Boolean respective, or it parse a Boolean expression in brackets, or compare 2 arithmetical expressions, or parse the negation of a Boolean expression or, at the end, read a variable from the memory.

```
        bexpr :: Parser Bool
        bexpr = do
                symbol "True"
                return True
                <|>
                do
                symbol "False"
```

```
                return False
                <|>
                do
                        symbol "("
                        b <-bexprOR
                        symbol ")"
                        return b
                        <|>
                        compareTo
                        <|>
                        do
                        symbol "NOT"
                        b<-bexprOR
                        return (not b)
                        <|>
                                do
                                space
                                id <- identifier
                                space
                                value <- getVariableValue id
                                if value == "" then failure else
                                        if (value == "True" || (value/="False" && v
alue /= "0")) then return True else return False
```

example of usage of this parser:

```
ghci> parse bexprOR [('b',"Boolean","False")] "90>10 OR False OR False AND (20==3 A
ND b)"

[([('b',"Boolean","False")],True,"")]
```

## 3.5. Assignment

The assignment function takes a variable (identifier) and then assign the evaluation of the expression on the right of the '=' symbol. It can assign both Arithmetic or Boolean values.

The assignment come off with the update of the environment. So if a variable already exists its value will be overwritten (also if the type is different).

```
assignment :: Parser String
        assignment = do
                id <- identifier
                symbol "="
                e <- expr
                symbol ";"
                changeEnvironment id intType (show e)
                return (show e)
                <|>
                do
                        id <- identifier
                        symbol "="
                        b <- bexprAND
```

```
                        symbol ";"
                        changeEnvironment id boolType (show b)
                        return (show b)
```

Example of usage:

```
ghci> parse assignment [] "a=True;"
[([('a',"Boolean","True")],"True","")]
ghci> parse assignment [('a',"Boolean","True")] "a=50;"
[([('a',"Integer","50")],"50","")]
```

## 3.6. Comment parser

Just for fun and for the scope of this project a parser for comments was implemented

```
comment :: Parser String
comment = string "--" >>= \c -> many (sat isComment) >>= \i -> string "/--
" >>= \end -> return ""
```

It take the "--" keyword and all is between it and "\n" and simply ignore it. Then returns the rest of the string program as unused input.

Example of usage:

```
ghci> parse comment [] "--commentline/--Rest of the program"
[([],"","Rest of the program")]
```

## 3.7. If then else and While commands

IfThenElse and While commands are the most difficult to implement because it is necessary to keep part of the program in memory until the command is not finished.

In order to do so all the parsers had to be duplicated and then copy is identical except for one thing: they do not consume the input string. In this way we can parse the else branch (necessary for the syntax check at least) without affects the rest of the program.

All the parsers that do not consume the input are called as the name of main parser preceded by 'parse' word.

Moreover, in order to emulate the loop iteration, a 'duplicateWhile' function has been implemented, which duplicate the piece of while program until the condition is no longer satisfied.

```
duplicateWhile c = P(\env inp -> [(env,"",c ++ " " ++ inp)])
```

example

```
ghci> parse (duplicateWhile  "while x<3 do x=x+1; endWhile") [] "while x<3 do x=x+1
; endWhile"
[([],"","while x<3 do x=x+1; endWhile while x<3 do x=x+1; endWhile")]
ghci>
```

```
ifThenElse :: Parser String
```

```
        ifThenElse = do
                        symbol "if"
                        condition <- bexprOR
                        symbol "then"
                        if condition then do
                                a<-program
                                symbol "else"
                                b<-parseProgram
                                symbol "endif"
                                return a
                                <|>
                                do
                                a<-program
                                symbol "endif"
                                return a
                        else
                                do
                                a<-parseProgram
                                symbol "else"
                                b<-program
                                symbol "endif"
                                return b
                                <|>
                                do
                                a<-parseProgram
                                symbol "endif"
                                return ""
```

Example of ifThenElse usage:

```
ghci> parse ifThenElse [] "if 3<5 then a=3; else b=4; endif"
[([('a',"Integer","3")],"3","")]
```

```
while :: Parser String
        while = do
                whileString <- parseWhile
                duplicateWhile whileString
                symbol "while"
                condition <- bexprOR
                symbol "do"
                if condition then do
                        program
                        symbol "endWhile"
                        duplicateWhile whileString
                        while
                else do
                        parseProgram
                        symbol "endWhile"
```

```
                        return ""
```

example of while usage:

```
ghci> parse while [('a',"Integer","3")] "while a<5 do a=a+1;x=a*5; endWhile"
[([('a',"Integer","5"),('x',"Integer","25")],"","")]
```

## 3.8. Program

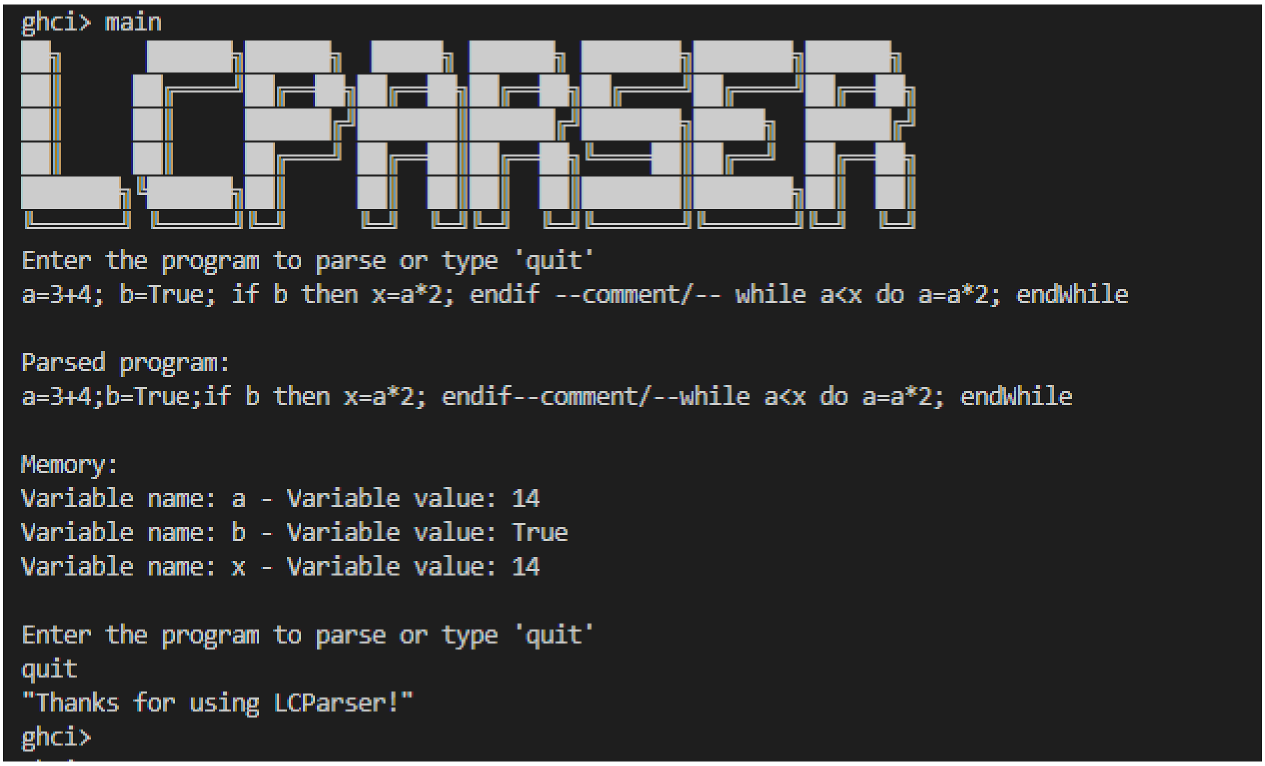
Finally the program is the parser of the commands seen above:

```
        cmd :: Parser String
        cmd = assignment
                <|>
                ifThenElse
                <|>
                while
                <|>
                comment

        program :: Parser String
        program = do
                cmd;
                program
                <|>
                cmd
```

# 4. Interpreter

The interpreter itself is a program that take inputs from user and parse them as a program. Every program that is parsed will give as output the parsed string and the updated memory after the execution of the program parsed. If an error occurs during parse of the program, the interpreter will display a message and will indicates where the error is.

```
executeInterpreter =do
                putStrLn "Enter the program to parse or type 'quit'"
                input <- getLine
                if input == "quit" then return "Thanks for using LCParser!" else do
                        putStrLn (executeProgram (parse parseProgram [] input))
                        executeInterpreter


        main = do
                showHeader
                executeInterpreter
```



```
ghci> main
LCPARSER
Enter the program to parse or type 'quit'
a=3+4; b=True; if b then x=a*2; endif --comment/-- while a<x do a=a*2; endWhile

Parsed program:
a=3+4;b=True;if b then x=a*2; endif--comment/--while a<x do a=a*2; endWhile

Memory:
Variable name: a - Variable value: 14
Variable name: b - Variable value: True
Variable name: x - Variable value: 14

Enter the program to parse or type 'quit'
quit
"Thanks for using LCParser!"
ghci>
```

# 5. Code snippet:

Example in screen above:

a=3+4; b=True; if b then x=a*2; endif --comment/-- while a<x do a=a*2; endWhile

Fibonacci sequence:

--input/--n=12; --program/--if n<2 then f=n; else f=1;e=1;i=2;while i<n do t=f;f=f+e;e=t;i=i+1; endWhile  endif

expected result →144