Formal Methods in Computer Science

Haskell Interpreter for an Imperative Language

"A parser for things is a function from strings to lists of pairs of things and strings"

Student

Rocco Caliandro
Mat: 728931

Professor

Giovanni Pani

Sommario

Introduction

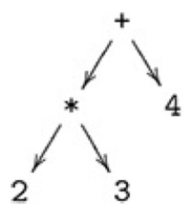
The purpose of this document is to define the Backus-Naur Form (BNF) grammar of the IMP language and the implementation of an interpreter written in Haskell. The programming language Haskell (https://www.haskell.org/) takes the name from the logician Haskell Curry (September 12, 1900 – September 1, 1982). Haskell is a purely functional programming language that use the call by name evaluation strategy in which an expression is evaluated only if it is necessary. It is also called “lazy” evaluation because the compiler will procrastinate. Unlike, the call by value strategy first compute and then substitute. The benefits of the lazy evaluation are:

- The ability to define control flow (structures) as abstraction instead of primitives
- The ability to define potentially infinite data structures
- Avoid in some case the non-termination programming
- Clean the code of unnecessary constructs

Haskell is an excellent language for all the parsing needs, the functional nature of the language makes it easy to compose different building block together without worrying about nasty side effects common in imperative languages.

Parser

A parser is a program that takes a string of characters as input and produces some form of tree that makes the syntactic structure of the string explicit. For example, given the string 2*3+4, a parser for arithmetic expression might produce a tree of the following form, in which the numbers appear at the leaves of the tree, and the operators appears at the nodes:

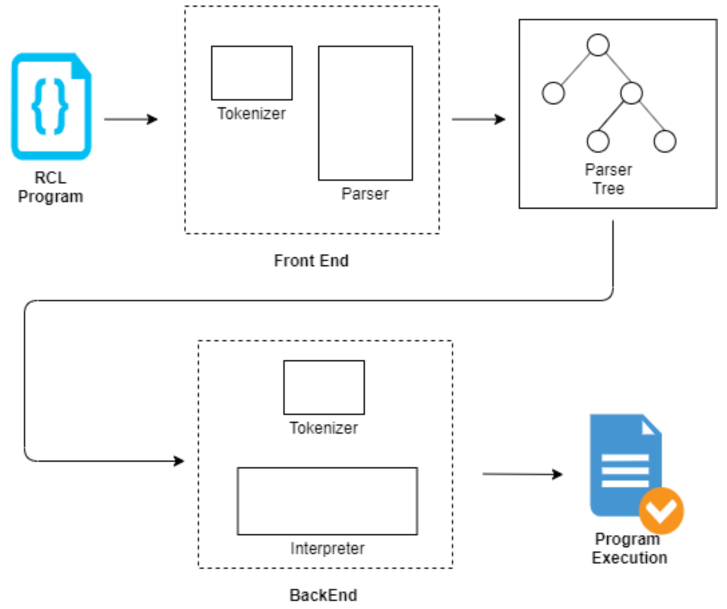


The structure of this tree makes explicit that + and * are operators with two arguments, and that * has higher priority than +

Interpreter

An interpreter is a program that parse and execute instructions written in a programming language, for example the one defined in the next section. The interpreter directly executes the instructions of the language without requiring them previously to have been compiled into a machine language program. An interpreter generally executes programs using the following steps:

1. Parse the source code
2. Translation of the source code into some efficient intermediate representation
3. Execution of the optimized code



Backus-Naur Form grammar of IMP

<table>
<tr>
<td><digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

<integer> ::= [-] <nat>

<identifier> ::= <lower> | <lower> <alphanum>

<alphanum> ::= <upper> <alphanum> | <lower> <alphanum> | <nat> <alphanum> | <upper> | <lower> | <nat>

<lower> ::= a-z

<upper> ::= A-Z

<aexp> ::= <aterm> '+' <aexp> | <aterm> '-' <aexp> | <aterm>

<aterm> ::= <afactor> '*' <aterm> | <afactor> '/' <aterm> | <afactor>

<afactor> ::= '('<aexp>')' | <integer> | <identifier>

<bexp> ::= <bterm> 'OR' <bexp> | <bterm>

<bterm> ::= <bfactor> 'AND' <bterm> | <bfactor></td>
<td><bfactor> ::= 'True' | 'False' | '!'<bfactor> | '('<bexp>')' | <bcomparison>

<bcomparison> ::= <aexp> '==' <aexp> | <aexp> '<=' <aexp> | <aexp> '<' <aexp> | <aexp> '>=' <aexp>
<aexp> '>' <aexp> | <aexp> '!=' <aexp>

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

<command> ::= <assignment> | <ifThenElse> | <while> | <forLoop> | skip';'

<assignment> ::= <identifier> ':=' <aexp> ';' | <identifier> ':=' <bexp> ';'

<ifThenElse> ::= 'if' '('<bexp>')' '{' <program> '}' | 'if' '('<bexp>')' '{' <program> '}' 'else' '{' <program> '}'

<while> ::= 'while(' <bexp> ') {' <program> '}'

<forLoop> ::= 'for(' <assignment> <bexp> ';' <identifier> '++) { ' <program> '}' |'for(' <assignment> <bexp> ';' <identifier> '--) { ' <program> '}' | 'for(' <assignment> <bexp> ';' <assignment> ') { ' <program> '}'</td>
</tr>
</table>

The Rocco Caliandro Language (**RCL**) supports the following constructs of typical imperative programming:

- *skip*: do nothing, skip to next instruction
- *assignment*: instruction that assign a value to a variable
- *ifThenElse*: conditional statement
- *while*: execute a cycle repeatedly based on a Boolean condition
- *forLoop*: like a while, it executes a set of instructions repeatedly

RCL is a dynamically typed language that allows <u>Integer</u> and <u>Boolean</u> as data types.

Environment

RC Int (Rocco Caliandro Interpreter) executes a series of explicit commands in order to change the state of the program. We need to define an environment as a set of variables in order to keep track of the changes after the execution of each command. Basically, the environment can be seen as a *memory*, and therefore must be kept up to date: the instruction/command that modify the state of the environment are the assignment.

```haskell

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

type Env = [Variable]

getVarType :: Variable -> String
getVarType = vtype

getVarName :: Variable -> String
getVarName = name

getVarValue :: Variable -> Int
getVarValue = value

```

The type Env (environment) is a list of Variables: a variable is composed by:

- **1st string**: Represents the name of the variable
- **2nd string**: Represents the type of the variable (such as Integer, Boolean etc.)
- **Integer Value**: Contains the value of the variable. In particular, for a Boolean variable 0 is the value that encode False and 1 is the value that encode True

Let suppose that at the end of an execution of a program the environment contains the variable *a* with the value 3; the corresponding environment representation will be:

```
[([Variable {name = "a", vtype = "Integer", value = 3}],"","")]
```

Environment Management

We have seen that the assignment instruction changes the state of the memory. So, in these cases we need to update the environment. The list of variables (composed by name, type, and value) is scanned and when a variable matches the name of the variable involved in the assignment, the associate value is changed. To do that the functions **updateEnv** and **modifyEnv** are fired. Finally, using the function **readVariable** and specifying the name and the type of the required variable, the associated value is returned.

```haskell
-- Update the environment with a variable
-- If the variable is new (not declared before), it will be
-- added in the environment
-- If the variable exstits, its value will be overwritten in.
modifyEnv :: Env -> Variable -> Env
modifyEnv [] var = [var]
modifyEnv (x:xs) newVar = if (name x) == (name newVar) then [newVar] ++ xs
                          else [x] ++ modifyEnv xs newVar

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


-- Return the value of a variable given the name
readVariable :: String -> Parser Int
readVariable name = P (\env input -> case searchVariable env name of
                                        [] -> []
                                        [value] -> [(env, value, input)])

-- Search the value of a variable stored in the Env, given the name
searchVariable :: Env -> String -> [Int]
searchVariable [] queryname = []
searchVariable (x:xs) queryname =
                                    if (name x) == queryname then [(value x)]
                                    else searchVariable xs queryname

```

Parser implementation

Remember that a parser is a program that takes a string and produces a syntactic tree. In the current implementation of the parser the result of the parsing process is a generic type as output. A parser could not always consume its entire input string, for this reason, we generalize the parser type to return also any unconsumed part of the argument string.

```haskell

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

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

To allow the Parser type to be made into instances of classes (Functor, Applicative, Monad and Alternative), it is first redefined using *newtype*, with a dummy constructor called P. Parser of this type can then be applied to an input string using a function that simply removes the dummy constructor.

The function parse has the following input:

- **Env**: the environment (usually we pass the first-time empty list)
- **String**: which represents the program that will be parsed

The function parse has the following output:

- **Env**: the status of the environment after the execution of the program
- **a**: the correctly parsed code
- **String**: the part of input string not parsed by errors or ignored

Parser as Functor, Applicative and Monad

In order to combine parsers in sequence and allow them to work together, we need to create an instance of the Functor, Applicative and Monad classes for the parser type. 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 alternatives isn't specific to parsers but can generalised to a range of applicative types. This concept is captured by the class Alternative present in the library Control.Applicative of the Prelude.

```haskell

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

```

A parser is an instance of Alternative class and so support empty and <|> primitives of the specified types. We can use this construct for define the alternative symbol "|" used in the BNF.   The empty is the parser that always fails regardless of the input string, and <|> is a choice operator that returns of the first parser if it succeeds on the input and applies the second parser to the same input otherwise. For example:

Functor

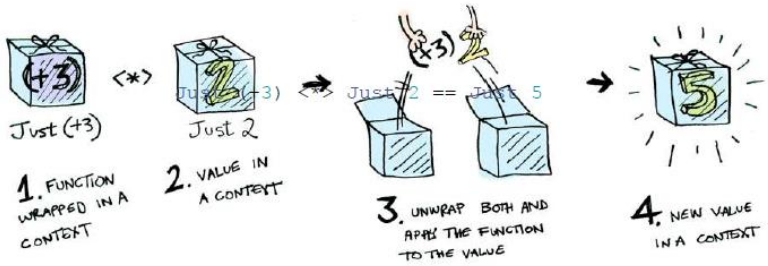
A Functor is a way to apply a function to a box or container. The type-class Functor has the function **fmap** that takes as input a function and a box and returns another box with inside the element after the function f.

The parser type is an instance of Functor and we need to define the function fmap. The function fmap applies the function g to the result value v of a parser p if the parser succeeds and it propagates the failure otherwise:

```haskell

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

```

Applicative

We can generalize Functors using Applicative. We can think to also put the input function in a context and to do this we need to define a new operator **<*>**. The Applicative needs also of another function **pure** that in Haskell is called '*return*' for historical reasons. The pure needs only one element as input and put it in a context.



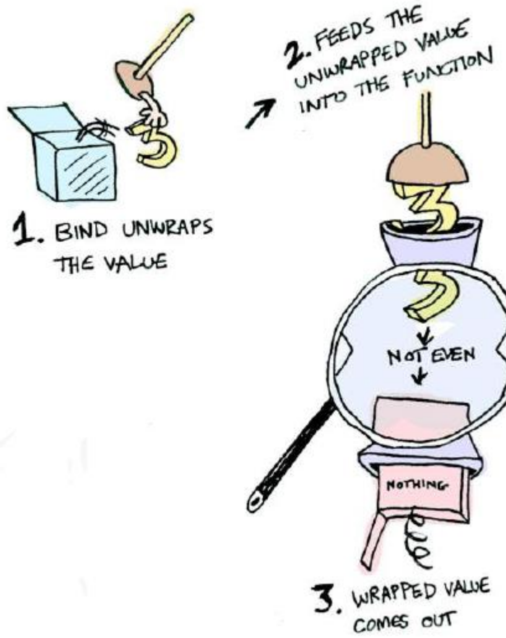
Just (+3) <*> Just 2 == Just 5

The parser type is an instance of Applicative and we need to define <*> and pure. The function pure transforms a value into a parser that always succeeds with this value v as its result, without consuming any of the input string. The function <*> applies a parser that returns a function to a parser that returns an argument to five a parser that return the result of applying the function to the argument, and only succeeds if all the components succeed.

```haskell

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

```

Monad

A monad is a function that take a box as input and returns another box using the bind operator **>>=**. Like Applicative and Functor, the Monad is a type-class. Bind is a function from Functor ma, then extracts the value a and finally applies the function to a that returns another functor mb.



The parser p >>= 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. Because parser is a monadic type, the do notation can now be used to sequence parsers and process their result values.

```haskell

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


```

Item Parser

The first parsing primitive is called *item*, which fails if the input string is empty, and succeeds with the first character as the result value otherwise. The *item* parser is the basic building block from which all other parsers that consume characters from the input will ultimately constructed.

```haskell

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

```

Example

```
print(parse item [] "")
print(parse item [] "abc")        You, seconds ago • Uncommitted changes
```

```
TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[]
[([],'a',"bc")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

We can see from the example, that when we call the parse item on the empty list we have empty as result (failure) and when the input is represented by the string "abc", we have as output: the empty environment, the parsed char 'a' and the not parsed string "bc".

Aexp: Arithmetic Expression Parser

Aexp is the name of the parser that evaluate arithmetic expressions built up from natural numbers using addition, multiplication, division, subtraction, and parentheses. In arithmetic expressions division and multiplication have higher priority than addition and subtraction and the operators associate to the right.

```haskell
aexp :: Parser Int
aexp = (do
    t <- aterm
    symbol "+"
    a <- aexp
    return (t+a))
    <|>
    (do
    t <- aterm
    symbol "-"
    a <- aexp
    return (t-a))
    <|>
    aterm

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

```haskell

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

Example

```
print(parse aexp [] "3 + (8/4) * 6 NOT PARSED AEXP")        You, seconds ago • Uncommitted changes
```

```
TERMINAL    PROBLEMS    OUTPUT    DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([],15,"NOT PARSED AEXP")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

We can see from the example, that when we call the parse aexp on the empty list and we obtain as result 15 that is the computation of the expression 3+(8/4)*6 and the string error “NOT PARSED AEXP” as the result of the string that is not parsed.

Bexp: Arithmetic Expression Parser

Similar to aexp, a parser that evaluate Boolean expression is called bexp. The functioning is analogous to aexp: for each Boolean expression exists a unique derivation Tree. In this case the AND operator (&&) has a higher priority with respect to OR operator (||). Moreover, bexp could need to use aexp parser, for example when a comparison between two numbers is required: this kind of comparison is managed by <u>bcomparison</u> parser.

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

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

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

Skip Parser

The skip parser is the simplest one, it recognizes the <u>skip</u> key-word and it parse without evaluate the next command.

```haskell

skip :: Parser String
skip = do {
    symbol "skip";
    symbol ";";
    parseCommand;
}

```

Example

```
print(parse skip [] "skip; a:=3;")        You, a minute ago • Uncommitted changes

TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([],"a:=3;","")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

Assignment Parser

The assignment parser recognizes the identifier of a variable and evaluates an arithmetic or Boolean expression and modifies the environment using the function **updateEnv**.

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

Example

```haskell
    print(parse assignment [] "a:=3;")
```

```
TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([Variable {name = "a", vtype = "Integer", value = 3}],"","")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> 
```

In the above example, we can note that the output of the program has a change on the environment. So, now, the environment contains a variable a of the type Integer with the value 3.

IfThenElse Parser

This parser evaluates the if-then-else command. It recognizes the if keyword and evaluates the Boolean expression included between parentheses. If the if-condition is satisfied, the program in the true-branch is evaluated and eventually the program in the false-branch is parsed without being evaluated (if a false-branch exists). In case the if-condition is false, the program in the true-branch is parsed without being evaluated and in case the false-branch exists it will be evaluated.

```haskell

ifThenElse :: Parser String
ifThenElse = (do
    symbol "if"
    symbol "("
    b <- bexp
    symbol ")"
    symbol "{"
    if (b) then
        (do
            program
            symbol "}"
            (do
                symbol "else"
                symbol "{"
                parseProgram;
                symbol "}"
                return "")
            <|>
            (return ""))
    else
        (do
            parseProgram        You, 5 days ago • Implementation of assignment and ifThenElse
            symbol "}"
            (do
                symbol "else"
                symbol "{"
                program
                symbol "}"
                return "")
            <|>
            return "")
        )
```

Example

```haskell

    print(parse ifThenElse [] "if (False) {a:=4;} else {b:=5;}")        You, seconds ago • Uncommitted changes

```

```
TERMINAL    PROBLEMS    OUTPUT    DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([Variable {name = "b", vtype = "Integer", value = 5}],"","")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

In the example, the Boolean condition is False, so the program will compute the else-branch with the update of the environment inserting the value 5 to the variable b.

While Parser

To repeat a while-command in case the while-condition is True, we insert the code of the while command at the head of the input string using the function **repeatWhile**. The code of the while command is returned by the parse function **parseWhile**. When the while-condition is False, we stop to insert the while-code at the head of the input string and just parse the code contained in the while and continue the evaluation.

```haskell

while :: Parser String
while = do
    w <- consumeWhile
    repeatWhile w
    symbol "while"
    symbol "("
    b <- bexp
    symbol ")"
    symbol "{"

    if (b) then (
        do
            program
            symbol "}"
            repeatWhile w
            while)
    else (
        do
            parseProgram
            symbol "}"
            return "")
```

Example

```haskell

    print(parse parseWhile [] "while(a<=x) {i:=3;}")

```

```
TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([],"while(a<=x){i:=3;}","")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

ForLoop Parser

The forLoop is similar to the while loop, in fact is another way to execute a program until a condition is True. The syntax of the forLoop needs:

For(i:=0; i<=9;i++) { program }

- the 'for (' keyword,
- An assignment operation
- A condition that repeats the program inside the forLoop until such condition is False
- A program or identifier++ or identifier-- that allow us to avoid non-ending computation
- The ') {' keyword
- The program p to repeat
- The '}' keyword

The forLoop parse all the grammar and then transform a forLoop in a while statement!

interpreter.hs

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

Example

```haskell
    print(parse parseForLoop [] "for (i:=0; i<9; i:=i-1;)  {a:=a+1;}")        You, seconds ago • Uncommitted changes
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

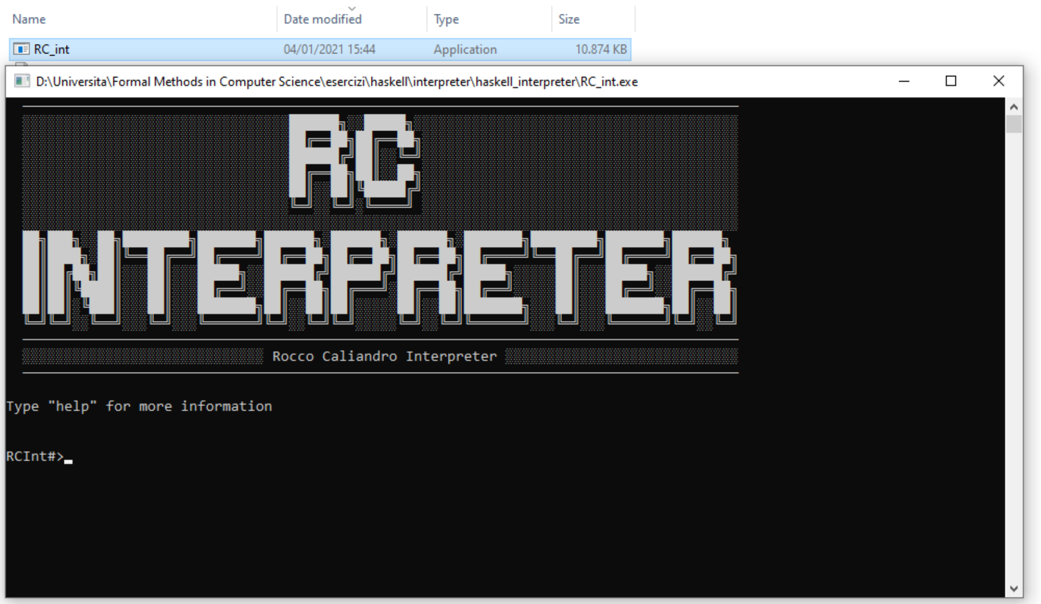
```
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter> runhaskell .\interpreter.hs
[([],"i:=0; while(i<9) {a:=a+1;i:=i-1;}","")]
PS D:\Universita\Formal Methods in Computer Science\esercizi\haskell\interpreter\haskell_interpreter>
```

We can see from the example that the forLoop is first parsed into a while command. After this transformation, the interpreter will compute the result using the created while.

RC Int – Usage example

To use the interpreter, we have two possibilities:

1. Directly launch the file RC_int.exe with a double click.



2. Use the source code with .hs extension and run from the shell the command "runhaskell .\RC_interpreter.hs"

Supported commands

The <u>help</u> command gives to the user all the possible thinks that she/he can do with a brief description

```
RCInt#>help
***** RC-Interpreter Help *****

 printmem        => Print the parsed code and the status of the memory

 syntax          => Show the BNF grammar for the RC-Interpreter

 examples        => Examples of programs written in the grammar of the RC-Interpreter

 help            => Print this help

 quit exit bye   => Stops RC-Interpreter

RCInt#>
```

The syntax command prints the BNF grammar of RCL

```
RCInt#>syntax
***** RC Interpreter - Syntax *****

 <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

 <integer> ::= [-] <nat>

 <identifier> ::= <lower> | <lower> <alphanum>

 <alphanum> ::= <upper> <alphanum> | <lower> <alphanum> | <nat> <alphanum> |
               <upper> | <lower> | <nat>

 <lower> ::= a-z

 <upper> ::= A-Z

 <aexp> ::= <aterm> '+' <aexp> | <aterm> '-' <aexp> | <aterm>

 <aterm> ::= <afactor> '*' <aterm> | <afactor> '/' <aterm> | <afactor>

 <afactor> ::= '('<aexp>')' | <integer> | <identifier>

 <bexp> ::= <bterm> 'OR' <bexp> | <bterm>

 <bterm> ::= <bfactor> 'AND' <bterm> | <bfactor>

 <bfactor> ::= 'True' | 'False' | '!'<bfactor> | '('<bexp>')' | <bcomparison>

 <bcomparison> ::= <aexp> '==' <aexp> | <aexp> '<=' <aexp> | <aexp> '<' <aexp> |
                   <aexp> '>=' <aexp> | <aexp> '>' <aexp> | <aexp> '!=' <aexp>
```

(and so on…)

The examples command contains an example of program written in RCL for each construct and the definition of the factorial computation as final example

```
RCInt#>examples
***** RC-Interpreter Program examples *****
 Assignment :
 bool := False; num := 7;

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

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

 for loop :
 a:=0; for (i:=10; i>0; i--)  {a:=a+1;}

 Factorial of 3:
 n := 3; i := 0; fact := 1; while (i < n) {fact := fact * (i+1); i := i+1;}

RCInt#>
```

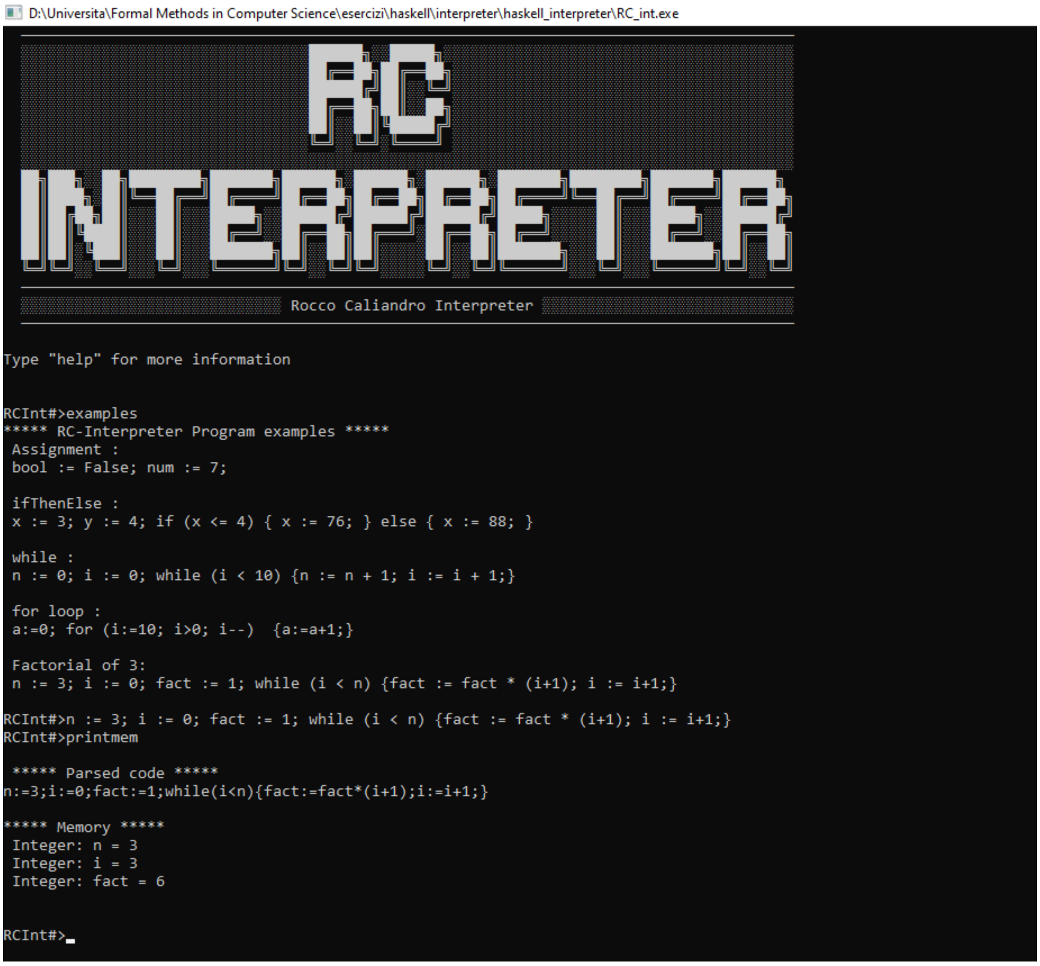
The printmem command print the parsed code and the status of the memory

```
RCInt#>printmem

 ***** Parsed code *****
a:=3;b:=4;

***** Memory *****
 Integer: a = 3
 Integer: b = 4


RCInt#>
```

The exit, bye or quit command allow us to close the interpreter

Example of usage:

- Open the interpreter
- Write “examples” on the shell
- Copy the last program of the Factorial of 3 (or invent a program using RCL)
- Paste the code and press enter
- Write “printmem”



References

- Book “Programming in Haskell”, Graham Hutton, Second Edition
- Material of the course *Formal Methods in Computer Science* of the master’s degree in Computer Science
- Haskell (programming language), Wikipedia: https://en.wikipedia.org/wiki/Haskell_(programming_language)
- Parsing with Haskell: https://mmhaskell.com/parsing
- RC Interpreter program