Part II

Chapter 6
Interpreter for a functional programming language

In this chapter we describe an interpreter for a (Clean like) functional programming language (without type checking). The language used in the interpreter is a simple pure functional programming language. In the exercises the language can be extended with constructors and pattern matching (enabling abstract data types) and list comprehensions (zf-expressions).

The interpreter will be used in the spreadsheet of the next chapter.

In the next section we will first give a description of the language we will make an interpreter for.

The remainder sections describe the several steps needed for interpretation: parsing, graph transformation and evaluation. For the parser we will not use the parsing techniques from chapter II.5, but instead use a dedicated infix expression parser. Parsing results in a parse tree for every function. Graph transformation transforms a parse tree into a tree which can directly be evaluated. The evaluator takes care of this evaluation.

In section 4 we will describe a user interface for the interpreter.

6.1 The interpreted programming language

The interpreted language we use is a simple pure functional language. We will not give a detailed syntax description of it, but instead introduce it by means of a number of examples. These examples can be found in the file `test.fp`.

Definition of a number of functions:

```
fac n = if (n=0) 1 (n*fac (n-1))

take n xs = if (xs=[]) [] (if (n=0) [] (hd xs : take (n-1) (tl xs))

filter p xs = if (xs=[]) [] (if (p (hd xs)) (hd xs : filter p (tl xs))
                                             (filter p (tl xs)))

notmodzero x y = y % x ~= 0 || x modulo y, % is the pre-defined infix operator mod.

from x = x : from (x+1)

sieve xs = hd xs : sieve (filter (notmodzero (hd xs) (tl xs)))

primes = sieve (from 2)
```

Syntax: We see that there is no special syntax for conditional statements, but instead we use a ternary function `if`. In this way the body of a function can be considered as a pure infix expression, which simplifies parsing. Notice the difference in the use of the `:` (cons) operator between this language and Clean. Here `:` is a pure infix operator. So `[a:b]` in Clean is equivalent with `a:b` in our language. This also simplifies parsing. Patterns and local definitions are not supported.

Data types: The only pre-defined data types are `num` (int), `bool` and `lists` (`char` and `string` can be added by the reader). Tuples are not supported yet, but can also be added by the reader. In the non-typed version of the language you can use lists to mimic tuples.

Pre-defined functions: `if`, `hd` and `tl` are the only (hard coded) pre-defined functions.

Infix operators: Our language supports the following infix operations: `+, -, *, /, % (mod), ^, =, ~=, : (cons), <, >, <=, >=`. Currently, it is not possible to define infix operators yourself (see exercises).

Currying is allowed. We use the infix operator @ to indicate the application of a function to one argument.

Lay-out rule: The lay-out rules are simple, a function definition starts at the beginning of a line, so lines that do not start at the beginning of a line are considered to be continuations of the previous line.

Use of the interpreter

The interpreter has a command-line interface. After the (`eval>`) prompt expressions can be typed in. During the start-up a system file (`sys.fp`) is loaded. This system file contains standard list functions like, `take`, `drop`, `filter` and `map`. The system function file can be edited by the user. At the prompt it is also possible to load a user defined file with function definitions. This is done with the command: `load filename`. Here `filename` is the name of the file (including its path, if it is not in same directory as the interpreter). The name of the file should not be "quoted".

After loading a user defined file, expressions using functions defined in this file can be typed in after the prompt. For example: `take 30 primes`, after loading the example file `test.fp`.

6.2 Parser

Before parsing a function or an expression we first turn the input string into a list of tokens. Tokenizing before parsing frees the parser from the task of recognizing infix operators like +, -, *, ^, etc, forming identifiers from groups of characters. The function tokenize take care of this.

```
::Token = IdNum [Char] | Op [Char] | Lpar | Rpar | TokError [Char] Char

tokenize :: [Char] -> [Token]
tokenize []        = []
tokenize ['[]':xs] = [IdNum ['nil'] : tokenize xs]
tokenize ['<=':xs] = [Op ['<='] : tokenize xs]
tokenize ['>=':xs] = [Op ['>='] : tokenize xs]
tokenize ['<':xs]  = [Op ['<'] : tokenize xs]
tokenize ['>':xs]  = [Op ['>'] : tokenize xs]
tokenize ['~=':xs] = [Op ['~='] : tokenize xs]
tokenize [' ':xs]  = tokenize xs
tokenize ['\t':xs] = tokenize xs
tokenize ['(':xs]  = [Lpar : tokenize xs]
tokenize [')':xs]  = [Rpar : tokenize xs]
tokenize ['[':xs]  = [Lpar : tokenize xs]
tokenize [']':xs]  = [Op [':'], IdNum ['nil'], Rpar :tokenize xs]
tokenize [',':xs]  = [Op [':'] : tokenize xs]
tokenize ['\n':xs] = []
tokenize l=:[x:xs] | oper [x]     = [Op [x] : tokenize xs]
                   | isCharNum x = [idnum : tokenize r]
                                 = [TokError ['Unexpected token'] x]
                           where (idnum, r) = readIdNum l
```

The definition of tokenize is straightforward. Identifiers and numbers are considered to be the same token (`IdNum`). The empty list `[]` is already recognized by the tokenizer. Also, the shorthand notation for lists, `[1,2,3]`, is turned into `1 : 2 : 3 : nil`. This can be done because the `','` is only used in lists.

The parser consists of only one function `parse`. The function `parse` can parse arbitrary infix expressions given a list of infix operators together with their priorities and the way they associate (left or right). These are supplied by the (constant) function `operators`. Operators consists of a list of triples, in which the first field contains the operator itself, the second field the priority of it (only for comparison) and the third field the association (left or right) of the operator:

```
operators =  [(['+'],10,'l'),(['*'],20,'l'),(['%'],5,'l'),(['/'],20,'l'),
              (['^'],30,'r'),(['='],3,'r'),([':'],0,'r'),(['@'],100,'l'),
              (['-'],10,'l'),(['<'],4,'r'), (['>'],4,'r'),(['~='],3,'r'),
              (['<='],4,'r'),(['>='],4,'r'),(['<='],4,'r'),(['>='],4,'r')]
```

The result of parse is an expression tree. The data type for expressions is:

```
:: Expr
 = Func [Char] Int Expr    // An occurance of a function: its name, arity and body.
 | Var [Char] Int          // A variable: its name and argument number (first variable is 0)
 | BOOL Bool               // Boolean value
 | Num Int                 // Integer value
 | Null                    // The empty list
 | Inf Expr [Char] Expr    // An infix expression: left argument, operator, right argument
 | PreError [Char]         // An erroneous expression. The argument is a message.
 | Empty                   // Undefined expression. e.g. the function body before its is filled
 | SysFunc [Char]          // A build-in function (if, hd or tl)
```

One problem we have is that the expressions we use are not pure infix expressions. They also contain function applications. To deal with this we assume the presence of the invisible application operator `@`. `@` is present between every function name and argument. If there are more arguments, `@` must also put between the arguments (this facilitates currying).

Example: `if (n=0) 1 (n * fac (n-1))` should be read as:


```
if @ (n=0) @ 1 @ (n * fac @ (n-1))
```

For this expression the parser generates the following parse tree:

```
Inf (Inf (Inf SysFunc ['if']
              ['@']
              (Inf (Var ['n'] 0)
                   ['=']
                   (Num 0)))
         ['@']
         (Num 1))
    ['@']
    (Inf (Var ['n'] 0)
         ['*']
         (Inf (Func ['Fac'] 1 Empty)
              ['@']
              (Inf (Var ['n'] 0)
                   ['-']
                   (Num 1)))))
```

This parse tree can be depicted as:

```
                          @
                         / \
           --------------   --------------
         @                                *
        / \                              / \
  ------   ------                  ------   ------
 @               1                n               @
/ \                                              / \
--   --                                        --   --
if     =                                      fac     -
      / \                                            / \
     n   0                                          n   1
```

Actually the `parser` generates the subexpression `Func` … for each function, variable and constant in the expression. The system functions and constants are transformed to the correct subexpressions by `graphTrans`. The expressions for variables are fixed by `placevars` in `parsefunc`. Finally, the function `fillin` is used to replace the `Empty` function body of `fac` by a copy of the expression shown.

6.2.1 The function parse

The function parse is a recursive descent parser. In contrast with the parsers form chapter II.4 this parser is deterministic. Parsing starts at the top level of the syntax tree. The parser tries to recognize the grammar rule at the top level (start symol for parsing) using the the grammar rules of lower levels. This is descending the grammar rules. For recursive rules the same parser is used to recognize the recursive occurence. Moreover we do not use parser combinators, but tailor made functions to construct the parser.

Parsing expressions containing operators of various priority is done by a kind of shift reduce parsing. We postpone the decission what has to be done with the current oparator and srguments until the next operator is found. By comparing the priority of the operators we can decide how the syntax tree should be constructed.

The data type `Context` is used to keep track of the context the parser is in. Currently, this is only necessary to keep track of bracketed expressions (to balance the brackets). If one wants to parse more complicated expressions `Context` can be extended.

```
::Context = Bexp | Exp
```

Type of parse:

```
parse :: [Context] [Token] [Expr] [Token] -> (Bool,Expr,[Token],String)
```

Arguments:

1. a stack of contexts, currently only used for balancing parenthesis.
2. the input token list
3. the expressions that are build up to now (if you start there is no expression yet, therefore we use a list with at most one element)
4. last infix operator we have met. This must be maintained to compare priorities. If we have not met an operator yet the list is empty, otherwise it contains one element.

Result, a four tuple with the following fields:

1. a boolean telling if the parse was succesful.
2. the result of the `parse`.
3. the remainder of the input string. A recursive call of `parse` only parses a sub-expressions and then returns to the calling function.
4. if the parse was not succesful, result 4 contains an appropriate error message.

The functions `parseExp` and `parse`:

```
parseExp :: [Token] -> Expr
parseExp xs | ok = r
= PreError (fromString error)
where (ok,r,rem,error) = parse [Exp] xs [] []

parse cs        []          [e] op   = (True,e,[],"")
```

1. No input left, `e` is the result.

```
parse [Bexp:cs] [Rpar:xs]   [e] op   = (True,e,[Rpar:xs],"")
parse [Bexp:cs] []           e  op   = (False,PreError [],[],"Missing )")
parse cs        [Rpar:xs]    e  op   = (False,PreError [],
                                               [Rpar:xs], "unexpected )")
```

2. Right parenthesis on input and context is bracketed expression, so return to the calling function (leave the parenthesis on the input).
3. Input empty, but still in bracket exp. context, so error message.
4. Right parenthesis, but not in bracket context, so error message.

```
parse cs        [Lpar:xs]   [ ] op
|  ok &&  okRpar                   = parse cs (tl r) [f] op
                                   = (False,PreError [],[],
                                         if ok "missing )" er)
  where (ok,f,r,er) = parse [Bexp] xs [] []
        okRpar = r <> [] && hd r == Rpar
```

5. `Lpar` on input, do first recursive call for bracketed exp, if this turns out to be ok, continue with a resursive call with this `Bexp` as a possible left hand side.

```
parse cs        [Lpar:xs]   [e] op
     | ok && okRpar                  = parse cs (tl r) [Inf e ['@'] f] op
                                     = (False,PreError [],[],
                                          if ok "missing )" er)
  where (ok,f,r,er) = parse [Bexp] xs [] []
        okRpar = r <> [] && hd r == Rpar
```

6. `Lpar` on input with a left hand side already present. Recursively parse a bracket expression, this becomes the right hand side of a function application. Now recursively call parse with the function application as the left hand side.


```
parse cs        [IdNum i:xs][ ] op   = parse cs xs [Func i 0 Empty] op
```

7. Identifier on input, recursive call of parse with this identifier as candidate left side of infix expression.

```
parse cs        [IdNum i:xs][e] op   = parse cs xs
                                          [Inf e ['@'] (Func i 0 Empty)] op
```

8. Identifier on input with already a left hand side parsed, thus function application, continue with this function application as candidate left hand side.

```
parse cs        [Op o:xs]   []  []   = (False,PreError [],[],
                                          "term expected at:" +++ toString o)
```

9. Infix operator on input and no left hand side present, so error.

```
parse cs        [Op o:xs]   [e] []
 | ok                                = parse cs r [Inf e o f] []
                                     = (False,f,r,er)
  where (ok,f,r,er) = parse cs xs [] [Op o]
```

10. Infix operator on input, left hand side present, no previous operator with which priorities should be compared present, so do a recursive call for the right hand side. Here the operator is given as a parameter to compare its priority with the next operator. Form the corresponding infix expression tree, and do a recursive call with this tree as left hand side.

```
parse cs        [Op o1:xs]  [e] [Op o2]
 | prio o1 < prio o2 ||
   prio o1 == prio o2 &&
   assocl o2                         = (True,e,[Op o1:xs],"")
 | ok                                = parse cs r [Inf e o1 f] [Op o2]
                                     = (False,f,r,er)
  where (ok,f,r,er) = parse cs xs [] [Op o1]
```

11. Compare priority of the two infix operators involved. See examples below.

```
parse cs [TokError er x:xs] es  os     = (False, PreError [], [], toString er)
```

12. If a token error is encountered this is turned into a parse error.

```
parse cs [x:xs]             es  os     = (False,PreError "",[],
                                            "Syntax error near: " +++
                                             toString (printToken x))
```

13. All correct cases are already handled so this should be an error.

Example of cases 10 and 11:

Consider the input string: `1 * 2 + 3`

When `*` is parsed a recursive call is done for the remainder of the input string with `*` as operator argument. When the parser now meets `+` it compares its priority with `*`, `*` has greater priority so parse returns `2`, which becomes the right hand side for `*`. `Parse` is now recursively called with `1 * 2` as a right hand side.

Consider the input string: `1 + 2 * 3`

When `+` is parsed a recursive call is done for the remainder of the input string with `+` as an argument. When the parser meets `*` it compares its priority with `+`, `*` has greater priority so parse continues with parsing the remainder of the input string until the input is empty or it meets an operator with lower priority than `+` returns `2`. In this case the remainder of the input string is parsed and `2 * 3` becomes the right hand side of `+`.

Remarks on this parsing scheme

Without modification of parse this parser can easily extended to more complex expression as long as they can be modelled as infix expression. Even zf-expressions can be parsed in this way (see exercise 6.1).

Exercise 6.1

Extend the parser in such a way that it is able to parse zf-expressions. For examples consider the (Clean) zf-expression:

```
[x\\ x <- [1..10] | x mod 2 = 0]
```

This can be considered as the following infix expression

```
Inf x "\\" (Inf (Inf x "<-" [1..10]) "|"  (x % 2 = 0))
```

Here `[1..10]` and `x % 2 = 0` are left unparsed for readability.

Add the infix operators for `\\`, `|`, `<-` to the parser together with appropriate priorities and association.

Think of a way to cope with the brackets (`[` and `]`) surrounding a zf-expression expression.

Parsing of function definitions

Parsing a script file with `parsefile` consists of parsing the function definitions in the file. This is done by reading the lines in the file, skipping the empty lines and merging the lines that make up one function definition (indented lines following a non indented line).

```
parsefile :: [([Char],[Expr],Int,Expr)] String *Files
             -> *(Bool,*Files,[([Char],[Expr],Int,Expr)])
parsefile sysfuncs fn files = (okparse && okfillin,files2,resfuncs)
where
  (input,files2) = ReadFileStrings fn files
  funcs          = (map parsefunc o mergeLines o remEmpty o map fromString) input
  parseErrors    = [b \\ (_,_,_,b) <- funcs | isParseError b]
  okparse        = parseErrors == []
  okfillin       = fillinErrors == []
  fillinErrors   = [b \\ (_,_,_,b) <- recfuncs | isParseError b]
  recfuncs        = map (fillin (sysfuncs ++ recfuncs)) funcs // recursive definition!!!
 resfuncs | okparse &&
            okfillin     = recfuncs
          | not okparse  = [([],[],0,PreError (flatten [print b ++ ['\n']
                             \\ b <- parseErrors]))]
          | not okfillin = [([],[],0,PreError (flatten [print b ++ ['\n']
                             \\ b <- fillinErrors]))]

remEmpty xss = [xs \\ xs <- xss| xs <> []]

mergeLines = foldr f []
            where f a [[' ':b]:bs]   = [(a ++ [' ':b]) : bs]
                  f a [['\t':b]:bs]  = [(a ++ [' ':b]) : bs]
                  f a bs             = [a : bs]
```

A function definition can also be considered as an infix expression with `'='` as the top operator (with lowest priority) and on the left side an application of the function name to the variables and on the right hand side the defining body expressions of the function.

`parsefunc` takes care of filtering out the variable names and substituting them in the body of the function (`placevars` does the substitution).

```
parsefunc :: [Char] -> ([Char],[Expr],Int,Expr)
parsefunc inv    = pfunc (Inf func ['='] body)
where func = (parseExp o tokenize) (takeWhile (noteq '=') inv)
      body = (parseExp o tokenize) (tl (dropWhile (noteq '=') inv))
```



```
pfunc (Inf l ['='] body)    = (name,vars,#vars,substbody)
where [Var name _ :vars] = findVars l
      substbody | isParseError body = PreError (['Error in function '] ++ name
                                         ++ [': ']   ++ error)
                                    = placevars vars (graphTrans body)
      (PreError error) = body

findVars (Func n a b)       =  [Var n 0]
findVars (Inf l ['@'] r)    =  findVars l ++ findVars r

placevars vs (Func f nv bf) | vn == (-1) = Func f nv bf
                                         = Var f vn
where vn = hd ([i \\ (i,Var v n) <- zip([0..],vs) | v == f] ++[-1])
placevars vs    (Inf a o b) = (Inf (placevars vs    a) o (placevars vs    b))
placevars vs     x = x
```

The parsed functions are maintained in a list of tuples: `(name, variables, nr_of_vars, body_expression)`.

In expressions calls to other functions occur. These function calls are initially parsed as `(Func name 0 Empty)`, where `name` is the name of the function (see the definition of `Expr`). To speed up evaluation the expressions corresponding to the functions are already substituted before evaluation (avoiding run-time look-up). This is done with the function `fillin`. Note that in `fillin` the result of the function is used as an argument of it (so the substituted bodies are substituted themselves already)!

```
recfuncs       = map (fillin (sysfuncs ++ recfuncs)) funcs
   ^                                     ^
   |_____________________________________|
```

In a non-lazy (functional) language this would lead to an infinite recursion for recursive function definitions. But in lazy functional programming language this is an (equal efficient) alternative for pointers.

In `fillin` it is also checked whether functions occuring in expressions indeed exist.

```
fillin :: [([Char],[Expr],Int,Expr)] ([Char],[Expr],Int,Expr) ->
          ([Char],[Expr],Int,Expr)
fillin pfs (n,vs,nvs,bf) = (n,vs,nvs,fbody)
where fbody | okfill = fillinbody
                     = PreError (['Error in '] ++ n ++ [': '] ++ error)
      (okfill,fillinbody,error) = fillbody pfs bf

fillbody pfs (Func name n b)  | isEmpty fs = (False,PreError [],
                                              ['function not found: '] ++ name)
                                           = (True,Func name nvs bf, [])
where [(_,_,nvs,bf):_] = fs
      fs = findfuncs name pfs

fillbody pfs (Inf e1 o e2)                 = (okleft && okright,
                                              Inf fillinleft o fillinright,
                                              errorleft ++ errorright)
where (okleft,fillinleft,errorleft)        = fillbody pfs e1
      (okright,fillinright,errorright)     = fillbody pfs e2

fillbody pfs x                             = (True,x,[])

findfuncs name fss = [(n,vs,nvs,b) \\ (n,vs,nvs,b) <- fss| name == n]
```

Exercise 6.2

Extend parsefile in such a way that it is possible to parse and use user defined infix operators. The name, priority and association of such an infix operators should be added in a similar way as in Clean.

Example: `infixl 3 ++ = concat` for adding the left associatie operator ++ with priority 3 which is defined by the function `concat`.

At first glance we have a problem. You have to find all infix operators before you can start parsing functions, making it necessary to go two times through the file. But this is not true. Exploiting lazy evaluation in a manner similar as used in the use of `fillin` makes it only necessary to go only ones through the file.

Transformation of parsed expressions

`Parse` returns a parsetree in which no distinction is made between function names, numbers pre-defined (system) function like `if`, `hd` and `tl`. `GraphTrans` takes care of this.

```
graphTrans :: Expr -> Expr
graphTrans (Inf e1 oper e2)            = Inf (graphTrans e1) oper
                                               (graphTrans e2)
graphTrans (Func ['True'] n bf)        = BOOL True
graphTrans (Func ['False'] n bf)       = BOOL False
graphTrans (Func ['hd'] n bf)          = (SysFunc ['hd'])
graphTrans (Func ['tl'] n bf)          = (SysFunc ['tl'])
graphTrans (Func ['if'] n bf)          = (SysFunc ['if'])
graphTrans (Func ['nil'] n bf)         = Null
graphTrans (Func f n bf) | isNum (hd f) = Num (toInt (toString f))
                                       = Func f n bf
graphTrans x                           = x
```

6.3 Evaluation

`parseEval` takes care of parsing and evaluating an input string. Before an expression is evaluated first the bodies of the functions occuring in this expression are substituted. This is done by the function `fillbody` (see also `parsefile`).

```
parseEval :: [([Char],[Expr],Int,Expr)] [Char] -> [Char]
parseEval fs xs | okparse && okfillin = printeval (eval fillexp [])
                | not okparse = perror
                | not okfillin = fillinerror
where okparse                          = not (isParseError exp)
      (PreError perror)                = exp
      (okfillin,fillexp,fillinerror) = (fillbody fs o graphTrans) exp
      exp                              = (parseExp o tokenize) xs

eval :: Expr [Expr] -> Expr
```

`eval` evaluates an expression taking the expression and an operand stack as input.

Below we discuss the several cases of `eval`:

```
eval (Num n) es = (Num n)
eval (BOOL b) es = (BOOL b)
eval (PreError s) es = (PreError s)
```

These are the easy cases because nothing happens.

```
eval (SysFunc ['if']) [BOOL t,e2,e3:es] | t = e2
                                            = e3
```

Evaluation of `if` needs 3 operands on the stack. If the evaluation of the first operand leads to `True` the second operand is returned, otherwise the third.

```
eval (SysFunc ['hd']) [Inf h [':'] t:es] = h
eval (SysFunc ['tl']) [Inf h [':'] t:es] = t
```

Evaluation of `hd` or `tl` needs 1 operands on the stack. This should be an infix expression representing the cons of a head and a tail. In case of `hd` the head is returned, otherwise, the tail.

```
eval (Func f n bf) es | #es < n = partapp (Func f n bf) es
```



```
                                = eval (substvar es bf) (drop n es)
where partapp e [] = e
      partapp e [ex:es] = partapp (Inf e '@' ex) es

      substvar es (Var x n) = es ! n
      substvar es (Inf a o b) = Inf (substvar es a) o (substvar es b)
      substvar es x = x
```

Evaluation of a function call: First it is checked if there are enough arguments on the stack. If not, (curried use of a function) the original expression consisting of application of the function to its arguments is returned (`partapp`). Otherwise, the arguments are substituted in the function body by `substvar`, and the result is evaluated after popping the arguments off the stack. Notice that the place of an argument on the stack exactly corresponds to its variable number.

```
eval (Inf e1 ['@'] e2) es = eval e1 [eval e2 [] : es]
```

Evaluation of an infix expression representing a function application leads to putting the right side (argument) on the stack and calling `eval` for the left side. Note that the argument is put evaluated on the stack, but due to the laziness of Clean the actual evaluation is postponed to the moment it is really needed, resulting in a lazy interpreter.

```
eval (Inf h [':'] t) es = Inf (eval h es) [':']  (eval t es)
```

Evaluation of an infix expression representing a cons of a head and a tail returns the same expression with the and the tail (lazy) evaluated.

```
eval (Inf e1 ['+'] e2) es   = Num (getNum (eval e1 []) + getNum (eval e2 []))
eval (Inf e1 ['-'] e2) es   = Num (getNum (eval e1 []) - getNum (eval e2 []))
eval (Inf e1 ['*'] e2) es   = Num (getNum (eval e1 []) * getNum (eval e2 []))
eval (Inf e1 ['/'] e2) es   = Num (getNum (eval e1 []) / getNum (eval e2 []))
eval (Inf e1 ['%'] e2) es   = Num (getNum (eval e1 []) mod getNum (eval e2 []))
eval (Inf e1 ['^'] e2) es   = Num (getNum (eval e1 []) ^ getNum (eval e2 []))
eval (Inf e1 ['<'] e2) es   = BOOL (getNum (eval e1 []) < getNum (eval e2 []))
eval (Inf e1 ['>'] e2) es   = BOOL (getNum (eval e1 []) > getNum (eval e2 []))
eval (Inf e1 ['<='] e2) es  = BOOL (getNum (eval e1 []) <= getNum (eval e2 []))
eval (Inf e1 ['>='] e2) es  = BOOL (getNum (eval e1 []) >= getNum (eval e2 []))
eval (Inf e1 ['~='] e2) es  = BOOL (getNum (eval e1 []) <> getNum (eval e2 []))
eval (Inf e1 ['='] e2) es   = BOOL (eval e1 [] == eval e2 [])
```

Evaluation of the remaining infix expressions is straightforward. `getNum` retracts the num out of `(Num num)` expression. Note that sections are not supported.

```
eval x es = (PreError (['Cannot be evaluated near: '] ++ print x))
```

If eval is not applicable an appropriate error message is generated.

Exercise 6.4

Currently, the interpreter can only deal with integers, booleans and lists of them. Extend the interpreter with character and string handling. In order to do this, the following additions have to be made:

- the tokenizer has to be extended to handle chars `'a'` and strings `"this is a string"` (take care about the special notations for newline, `'\n'` etc.).
- `Expr` has to be extended to handle these new datatypes (a string can be represented by a list of characters).
- `print` should be adapted for printing lists of characters as a string.
- `eval` should be adapted to handle the evaluation of a character (trivial, like `num` and `bool`)

Exercise 6.5 (hard)

In exercise 6.1 we added parsing of zf-expressions to the parse, now add evaluation of zf-expressions to the evaluator. In order to do this the parse tree of a zf-expression should first be rewritten in a form that allows evalution. This can be done in `GraphTrans`.

Exercise 6.6 (project)

One of the major drawbacks of the interpreter from the previous sections is its inability to deal with algebraic data types. For example, it is hard to define a tree like data with operations on it. In Clean such a date type would be defined like:

```
:: Tree = Empty | Node int Tree Tree
```

One can define functions on trees with the use of pattern matching:

```
insert :: int Tree -> Tree
insert n Empty = Node n Empty Empty
insert n (Node k l r) | n <= k = Node (insert n l) r
                               = Node l (insert n r)
```

Especially, this use of pattern matching makes algebraic data types so powerful.

It is possible to add abstract data types to the interpreter without adding types. Only constructors and pattern matching are added. A constructor can be viewed as a tag to a type, only added to make pattern matching possible. To distinguish constructors from function names, constructors should start with an uppercase character. Function names should not be allowed to start with a capital. The function insert in the formalism of the interpreter now becomes:

```
insert n Empty = Node n Empty Empty
insert n (Node k l r) = if (n <= k) (Node (insert n l) r)
                                    (Node l (insert n r))
```

In order to add constructors and pattern matching to the interpreter, the following additions should be made to the interpreter.

- Parsefunc should be adapted to make it possible to parse patterns on the left hand side of a function definition and to extract the variables that occur in the patterns.
- GraphTransform should be adapted to recognise constructors.
- Pattern matching should be added to eval in order to select the right body in case of a function call.
- The definition of Func should be extended to include functions with several bodies, corresponding to different patterns.

To realise this the type Expr is extended with the following cases:

```
Cstr [Char] | Funcp [Char] Int [([Expr],Expr)]
```

`Cstr [Char]`: A constructor has a name.

`Funcp [Char] Int [([Expr],Expr)]`: A function with patterns has a name, a number of patterns, a list of two tuples consisting of: a list of patterns (patterns are also expressions) with which the argument have to be matched and the body expression that has to be executed in case the match succeeds.

Inplement the above mentioned addition to the interpreter.

6.4 User Interface

The interpreter has a command line interface, so the user can type in an expression after a prompt, hit return and examine the result.

`Start` does the preparing work for starting such an interface. It opens `files` and `sio` and reads in the system functions from `sys.fp` and then calls `ProcessLines`, which handles the remainder of the io.

To avoid buffering of output results are not written as entire strings to the output but character by character. This is done by the utility function `fwriteclist`.

```
Start :: *World -> *File
Start world = finalsio
where
  (files,worldwithoutfiles) = openfiles world
  (sio,files2)              = stdio files
  hallosio                  = fwrites hallo sio
  syssio | ok               = fwriteclist ['Reading system files\n'] hallosio
                            = fwriteclist (firstError sysfuncs) hallosio
  (finalsio,_)              = ProcessLines sysfuncs [] syssio files3
  (ok,files3,sysfuncs)      = parsefile [] libfile files2

hallo = "Interpreter for simple functional programming language (in Clean) V1.0\n"
endstring = "quit\n"
prompt    = "\neval> "
libfile   = "sys.fp"
```

`ProcessLines` recursively takes an input string from the input device (terminal), examines whether it is `quit`, `load filename` or an expression to be evaluated.

In case `quit` is encountered `ProcessLines` returns.

In case of `load filename`, the filename is extracted, the file is parsed and ProcessLines is recursively called with the new functions (found in the file) as an argument.

If the input string is empty (the user hits return) `ProcessLines` is called recursively.

In all other cases the input string is assumed to be an expression, so parseEval is called for this string, the result is written to the output and `ProcessLines` is called recursively.

```
ProcessLines :: [([Char],[Expr],Int,Expr)]
                [([Char],[Expr],Int,Expr)] *File *Files
                -> *(.File,*Files);

ProcessLines  sysfuncs funcs sio files
 | string == endstring  = (readsio,files)
 | isload               = ProcessLines sysfuncs newfuncs loadsio files2
                          // loading a file
 | string == "\n"       = ProcessLines sysfuncs funcs readsio files
                          // skip empty lines
                        = ProcessLines sysfuncs funcs resultsio files
                          // evaluate expression
where
  promptsio             = fwrites prompt sio
  input                 = fromString string
  (string, readsio)     = freadline promptsio
  resultsio             = fwriteclist (parseEval (sysfuncs ++ funcs) input) readsio
  (isload,fn)           = isloadstring input
  loadsio      | ok     = fwriteclist ['No errors in file'] readsio
                        = fwriteclist (firstError newfuncs) readsio
  (ok,files2,newfuncs)  = parsefile sysfuncs filename files
  filename              = toString (takeWhile (noteq '\n') fn)

isloadstring ['load ': fn] = (True,fn)
isloadstring x             = (False,[])
```

6.5 Examples programs for the interpreter

Sys.fp (system file)

```
from x = x : from (x+1)
fromto x y = if (x=y) [x] (x : fromto (x+1) y)
downto x y = if (x=y) [x] (x : downto (x-1) y)
```

```
take n xs = if (n=0) nil (if (xs = nil) nil ((hd xs):take (n-1) (tl xs)))

drop n xs = if (xs=[]) [] (if (n=0) xs (drop (n-1) (tl xs)))

map f xs = if (xs = []) [] (f ( (hd xs)): (map f (tl xs)))

insert x xs = if (xs=[]) [x] (if (x < (hd xs))
                                 (x:xs)
                                 (hd xs: insert x (tl xs)))

sort xs = if (xs=[]) [] (insert (hd xs) (sort (tl xs)))

add x xs  = if (xs=[]) [x] (hd xs : add x (tl xs))

append xs ys = if (xs=[]) ys (hd xs : append (tl xs) ys)

concat xss = if (xss=[]) [] (if (hd xss = [])
                                (concat (tl xss))
                                (hd (hd xss) : concat (tl (hd xss): tl xss)))

filter p xs = if (xs=nil) nil (if (p (hd xs))
                                  (hd xs:filter p (tl xs))
                                  (filter p (tl xs)))

takewhile f xs = if (xs = []) [] (if (f (hd xs))
                                     ((hd xs) : takewhile f (tl xs))
                                     [])

foldr f a xs = if (xs = []) a (f (hd xs) (foldr f a (tl xs)))
foldl f a xs = if (xs = []) a (foldl f (f a (hd xs)) (tl xs))
```

Test.fp (example file)

```
fac n = if (n=0) 1 (n*fac (n-1))

notmodzero x y = y % x ~= 0

mod x y = x % y

sieve xs = (hd xs) : sieve (filter (notmodzero (hd xs)) (tl xs))

primes = sieve (from 2)
```

6.6 Adding type checking to the interpreter

The next step is to extend the interpreter with a type checker. Type checking eliminates may run-time errors because they are already detected at compile time. For examples adding an integer to a string, or having lists with elements of different type. The type checker we use for our interpreter is capable of deriving the types of functions without having the user to supply the type of it.

6.6.1 Restriction on the interpreted language due to the type checker

The type checker imposes certain restrictions on the interpreted language. These restrictions are included to simplify the type checking. In the exercises suggestions are done to remove these restrcitions.

The first restriction is that functions in a definition file should be ordered by dependency. This means that if a function f depends on a function g (g appears in the definition of f) g should be defined before g in the definition file.

As a consequence mutual depended functions are not allowed. For example:


```
f = 1 : g
g = 2 : f,
```

is a (currently) not allowed definition.

Recursive function defintions are allowed.

Definition of type

```
::Type = VarT Int | Arrow Type Type | NumT | BoolT |
         ListT Type | TypeError String | CharT
```

Example: After typing `info` after the `eval>` prompt the types of the currently loaded functions are displayed.

```
from :: num -> [num]
fromto :: num -> num -> [num]
downto :: num -> num -> [num]
take :: num -> [t0] -> [t0]
drop :: num -> [t0] -> [t0]
map :: (t0 -> t1) -> [t0] -> [t1]
insert :: num -> [num] -> [num]
sort :: [num] -> [num]
add :: t0 -> [t0] -> [t0]
append :: [t0] -> [t0] -> [t0]
concat :: [[t0]] -> [t0]
filter :: (t0 -> bool) -> [t0] -> [t0]
takewhile :: (t0 -> bool) -> [t0] -> [t0]
foldr :: (t0 -> t1 -> t1) -> t1 -> [t0] -> t1
foldl :: (t1 -> t0 -> t1) -> t1 -> [t0] -> t1
fac :: num -> num
notmodzero :: num -> num -> bool
sieve :: [num] -> [num]
primes :: [num]
compose :: (t0 -> t2) -> (t1 -> t0) -> t1 -> t2
som :: num -> num
plus :: num -> num -> num
```

Supporting functions

```
fillintypes types (VarT n) = deref types (types ! n)
fillintypes types (Arrow l r) = Arrow (fillintypes types l) (fillintypes types r)
fillintypes types  (ListT l) = ListT (fillintypes types  l)
fillintypes types x = x

substtypes [] t = t
substtypes [s:ss] t = substtypes ss (substtype s t)

substtype (VarT m,r) (VarT n) | n == m = r
                                       = VarT n
substtype t (Arrow l r) = Arrow (substtype t l) (substtype t r)
substtype t (ListT l) = ListT (substtype t l)
substtype t tt = tt

deref types (VarT n)
 = dref (types!n)
 where dref (VarT m) | n == m = VarT n
                             = deref types (VarT m)
       dref (Arrow l r) = Arrow (deref types l) (deref types r)
       dref (ListT l) = ListT (deref types l)
       dref x = x

deref types (Arrow l r) = Arrow (deref types l) (deref types r)
deref types (ListT a) = ListT (deref types a)
deref types x = x

occur (VarT n) (VarT m) = n == m
```

```
occur t (Arrow l r) = occur t l || occur t r
occur t (ListT l) = occur t l
occur t tt = False

//              unify types l r

unify types l r
 | l == r = (True,types,[])
          = (ok,restypes,error)
   where (ok,restypes) = unif types (deref types l) (deref types r)
         error = unerror ok (deref types l) (deref types r)

unif types (VarT n) (VarT m) | n == m = (True,types)
                                      = (True,update n (VarT m) types)
unif types (VarT n) b = (not (occur (VarT n) b), update n b types)
unif types a (VarT n) = (not (occur (VarT n) a),update n a types)
unif types (ListT a) (ListT b) = unif types a b
unif types (Arrow a b) (Arrow c d)
 = (rl && rr,if rl typesr typesl)
 where
  (rl,typesl) = unif types a c
  (rr,typesr) = unif typesl (deref typesl b) (deref typesl d)
unif types a b | a == b = (True,types)
                        = (False,types)

update n a xs = [y\\ (x,i) <- zip(xs,[0..]), y <- [if (i==n) a x]]
```

The functions derivetype and buildType

```
derivetype functypes (func,vars,nvar,body)
 | not okBuild = (okBuild,TypeError (toString(['Error in '] ++ func ++ [', ']
++error)),0)
               = (okBuild,renamedtype ,# diftypes)
 where

  types = [VarT n\\ n <- [0..]]

  bodytype = buildArrows nvar
  restype =  (fillintypes dertypes bodytype)
  diftypes = (removeDup o typeVars) restype
  renamedtype = substtypes [(t,VarT n)\\ (t,n) <- zip(diftypes,[0..])] restype

  (okBuild,dertypes,nvartypes,error) = buildType body types (VarT nvar) nvar

  buildType (PreError m) types restype ntype
   = (False,[],0,fromString m)

  buildType (Var x n) types restype ntype
   = (okun,newtypes,ntype,error)
     where (okun,newtypes,error) = unify types (types ! n) restype

  buildType Null types restype ntype
   =  (ok,newtypes,ntype+1,error)
   where (ok,newtypes,error) = unify types restype (ListT (VarT  (ntype + 1)))

  buildType (SysFunc ['if']) types restype ntype
   =  (ok,newtypes,ntype+1,error)
   where (ok,newtypes,error) = unify types restype (renType (ntype+ 1) typeIf)

  buildType (SysFunc ['hd']) types restype ntype
   =  (ok,newtypes,ntype+1,error)
   where (ok,newtypes,error) = unify types restype (renType (ntype+1) (Arrow
(ListT (VarT 0)) (VarT 0)))

  buildType (SysFunc ['tl']) types restype ntype
```



```
   =  (ok,newtypes,ntype+1,error)
   where (ok,newtypes,error) = unify types restype (renType (ntype+1) (Arrow
(ListT (VarT 0)) (ListT (VarT 0))))

  buildType (Func name n b) types restype ntype
   | name == func =  (okself,newselftypes,ntype,erself)
                  =  (okfound && ok,newtypes,ntype+nvartype,error)
   where (ok,newtypes,ermatch) = unify types restype (renType (ntype+1) typefunc)
         (okfound,typefunc,nvartype) = findtype name functypes
         (okself,newselftypes,erself) = unify types restype (fillintypes types
bodytype)
         error = if (not okfound) (['function not found: '] ++ name ++ ['\n'])
                                  ermatch

  buildType (Num n) types restype ntype
   =  (ok,newtypes,ntype,error)
   where (ok,newtypes,error) = unify types restype NumT

  buildType (CHAR n) types restype ntype
   =  (ok,newtypes,ntype,error)
   where (ok,newtypes,error) = unify types restype CharT

  buildType (Inf l ['@'] r) types restype ntype
   = (okr && okl,newtypesl,ntypel,errorl++errorr)
   where
    (okr,newtypesr, ntyper,errorr) = buildType r types (VarT (ntype + 1)) (ntype +
1)
    (okl,newtypesl, ntypel,errorl) = buildType l newtypesr
                                     (Arrow (VarT (ntype + 1)) restype)
                                     ntyper

  buildType (Inf l [':'] r) types restype ntype
   = (ok && okl && okr,newtypesr,ntyper,errorr++errorl++error)
   where
    (ok,newtypes,error) = unify types restype (ListT elemtype)
    (okl,newtypesl, ntypel,errorl) = buildType l newtypes elemtype (ntype + 1)
    (okr,newtypesr, ntyper,errorr) = buildType r newtypesl (ListT elemtype) ntypel
    elemtype = VarT (ntype+1)

  buildType (Inf l o r) types restype ntype
   | isMember o [['='],['~=']]= (ok && okl &&
okr,newtypesr,ntyper,errorr++errorl++errorr)
   where
    (ok,newtypes,error) = unify types restype BoolT
    (okl,newtypesl, ntypel,errorl) = buildType l newtypes elemtype (ntype + 1)
    (okr,newtypesr, ntyper,errorr) = buildType r newtypesl elemtype ntypel
    elemtype = VarT (ntype+1)

  buildType (Inf l o r) types restype ntype
   | isMember o [['+'],['-'],['*'],['/'],['%'],['^']] = (ok && okl &&
okr,newtypesl,ntypel,error++errorr++errorl)
   where
    (ok,newtypes,error) = unify types restype NumT
    (okr,newtypesr, ntyper,errorr) = buildType r newtypes NumT ntype
    (okl,newtypesl, ntypel,errorl) = buildType l newtypesr NumT ntyper

  buildType (Inf l o r) types restype ntype
   | isMember o [['<'],['>'],['<='],['>=']] = (ok && okl &&
                                    okr,newtypesl,ntypel,error++errorr++errorl)
   where
    (ok,newtypes,error) = unify types restype BoolT
    (okr,newtypesr, ntyper,errorr) = buildType r newtypes NumT ntype
    (okl,newtypesl, ntypel,errorl) = buildType l newtypesr NumT ntyper
```