How to build a monadic interpreter in one day
(based on papers provided by the Haskell community and some other resources)

by Dan Popa
Dept. Comp.Sci. Univ. Bacau, Romania
popavdan@yahoo.com

Abstract: In this paper the author is building a monadic language processor for a small while language, step by step, in one day, and is explaining how can we do it using Haskell 98, based on principles from [Hut-98] , [TZE-*] and some other papers. Both the front-end and the back-end are using monads.

I. The plan

The process of building a new while language is always split in parts. The task to be accomplished in one day will be split in two parts, according to the structure of a traditional compiler or interpreter. I mean the front-end and the back-end. Because the data processing made by the front-end is always a simple and well known part of the process (it includes usually a lexer and a parser or a scanner-less parser) we have chosen to begin with the back-end. There is an other reason for that: The first part of a language – in the designer's order - (in our opinion) is or should be always the abstract syntax tree (AST). It is enough in order to implement the semantics and the syntax is still free. Even after the construction of the AST and the back-end, the language designer is still free to chose a syntax which is able to fit the needs. And even more than one syntax cam be defined, for example in order to use keywords from some different languages. So, the plan is to build the back-end in the morning and the front-end in the afternoon.

II. The Morning; Building The Back-End, the detailed plan.

There are some small tasks needed to be accomplished in the morning. To build the AST data structure and the rest of the back-end means to pass through the followings:
1)Declare the AST (expressions, commands). The language will include both kind: expressions and commands, so we need to declare two data structures.
2)Prepare a small program including commands (for testing purpose) referred as “the s1 program”. In order to run a program (even in abstract syntax form), we have to have one already prepared.
3)Declare the data structures of the environment and some functions used to manipulate it A large set of imperative languages (and even some languages like LISP) are using an environment in order to store things like the values of the variables and more.
4)Replace the virtual machine with a careful selected monad. The existence of the monadic semantics allows us to use the do-notation in Haskell and also to take the data structures of the environment and the monad separated from the description of the semantics. It is a well known property of the monadic semantics.
5)Define some basic operations with monadic values. If we had decided to deal with semantics expressed by chains of binds (i.e. the monadic operator >>= is repeatedly used) we have to start from somewhere. The first monadic values produced, for example by accessing to the environment will have to come as results of some basic functions.
6)Write the expression-interpreter in do-notation. The monadic semantic of the

expressions should be implemented.
7)Write the command-interpreter in do-notation. The monadic semantic of the commands will be implemented.
After the first seven steps the back-end is ready to run programs, like the s1 program.

III. The afternoon; Building The Front-End, the detailed plan.

1)Prepare the grammar of the language. Of course, we have to have in mind a grammar of the future language. The use of programs expressed as ASTs is not enough practically.
2)Use a parser combinators library(or write a new one). The parsers from a “parser combinators” library are the bricks of the future parser of the language.
3)Combine simple parsers until you get a parser of the language. That is why we are using parser combinators. The allow us to incrementally build the parser.
4)Parse the source of the “the s1 program” . You should get the same tree which had been hand-written in the morning.
5)Combine: Front-end, Back-end and an eventual tree rewriter (if needed) to get the interpreter.

IV. The morning, step by step

IV.1 How to declare the AST which are implementing the expressions ? You may want to follow an example like this. Of course, this is a minimal one. It shows the implementation for constants, variables, differences and products of expressions. Other operators like:+, / etc should be added in the same way. This task is left as exercise for the reader. The “deriving show” clause instructs the Haskell system to build a “show” function for the Exp data-type. This makes the Exp-s printable. So when something evaluates to Exp and an Exp is returned by a function that value can be printed by the Haskell system.

```
data Exp =  Constant Int
           | Variable String
           | Minus Exp Exp
           | Greater Exp Exp
           | Times Exp Exp
           deriving Show
```

Next step is to declare the AST used for commands (also called statements).
The following statements will be implemented: = , sequencing, the conditional (usual called “if”, the while loop and a local declaration). Being composed statements we have to declare the type of components for each of them. The sequence are limited to two commands but this is no limitation at all, the parser can be still made to parse long sequences and generate something like:

Seq com1 (seq com2 (Seq com3 .... (Seq com_{n-1}, com_n)...))

```
data Com =  Assign String Exp
           | Seq Com Com
           | Cond Exp Com Com
           | While Exp Com
           | Declare String Exp Com
           | Print Exp
          deriving Show
```

In both situations, the user data-type constructors like: Assign, Seq , Cond, While, Declare, Print can be named as you wish. The first symbol should be capital letter. Haskell uses such identifiers are used as user defined data constructors.

IV.2 To prepare a small program including commands (for testing purpose) referred as “the s1 program” is the next step. It will be used to test the back-end. An usual program have to use all (or almost all) kind of commands and expressions. Exercise for the reader: Find a better one.

```
declare x = 150 in
     declare y = 200 in
      {while x > 0 do { x:=x-1; y:=y-1 };
        print y
      }
```

As an alternative solution you may prepare a bigger set of (small) programs. This is usual when prototyping or building languages or compilers. Also, the results of that programs should already be known.

Prepare the AST – abstract syntax tree- of “the s1 program”. Here it is:

```
s1 = Declare "x" (Constant 150)
       (Declare "y" (Constant 200)
          (Seq (While (Greater (Variable "x" ) (Constant 0)
                      )
                      (Seq (Assign "x" (Minus (Variable "x")
                                              (Constant 1)
                                       )
                           )
                           (Assign "y" (Minus (Variable "y")
                                              (Constant 1)
                                       )
                           )
                      )
               )
               (Print (Variable "y"))
          )
       )
```

It can be included as a constant declaration in the Haskell code.

IV.3 Next step is to declare the data-structures of the environment
There are only integers in this small implementation. The set of values of the user variables will be kept as a list of integers. The location of a variable on that list will be traced by it's position in the list. That's why location was defined a s a synonym for the Int. The dictionary (usually called “symbols table”) is called Index and is declared as a list of strings.

```
type Location = Int
type Index = [String]
type Stack = [Int]
```

Remark: Other kind of environments may and was used to store the relation between identifiers and values. For ex: list of pairs which are declared in Haskell as:

[(String, Value)]

Also some functions are needed in order to manipulate such an environment:

```
position            :: String  -> Index ->  Location
-- Remember: the location is just an Int
position name index = let
                       pos n (nm:nms) = if name == nm
                                        then n
                                        else pos (n+1) nms
                                        --n is growing,
                                         --the list is shorter
                      in pos 1 index
```

To get the n-th value, the value from a specified Location, we may use this fetch:

```
fetch             :: Location -> Stack -> Int
fetch n (v:vs)   =  if n == 1 then v else fetch (n-1) vs
```

An other function is used in order to compute an updated environment. A new stack is computed based on the number of the updated location, the “stored” value and the previous content of the stack.

```
put             :: Location -> Int -> Stack -> Stack
put n x (v:vs) = if n==1
                 -- if the replacement of the head is required
                 then x:vs
                 -- the old stack with a replaced head is returned
                 else v:(put (n-1) x vs)
                 -- otherwise we must put the value in the list's tail
```

Step four is to select a monad to be used as a model of calculus instead of a virtual machine. This example is using a “State and Output” Monad. That is why the type constructor is called StOut. Such a monadic case is containing a function which map a `Stack` to a triple `(a, Stack, String)`.

```
newtype M a = StOut (Stack -> (a, Stack, String))

instance Monad M where
  return x = StOut (\n -> (x,n, ""))
  e >>=  f = StOut (\n -> let (a,n1,s1) = (unStOut e) n
                              (b,n2,s2) = unStOut (f a) n1
                          in (b,n2,s1++s2) )
--unStOut is used to extract the embeded function from a monadic capsule
unStOut (StOut f) = f
```

Remark: the value returned by a commutation is in fact the first part of the tuple `(a, Stack, String)`. The updated value of the stack is the second and the output – a string is the third.

All the computations will take place in the universe of the monad. So, the basic operations with the environment should be projected in some monadic actions / values. That is why we have to define some basic operations with monadic values.

-- the monadic action capable of returning a value from the environment as
-- the main result of a computation

```
getfrom   :: Location -> M Int
getfrom i = StOut (\ns -> (fetch i ns, ns, ""))
```

-- the monadic action capable of modifying the stack is:

```
write     :: Location -> Int -> M ()
write i v = StOut (\ns -> ( (), put i v ns, "") )
```

-- this action is modifying the stack (but not the Index dictionary) by pushing a value as top / head of the stack. It will be useful when the creation of some new variables is demanded. Must be accompanied by the insertion of the new identifier into the Index.

```
push :: Int -> M ()
push x = StOut(\ns -> ((), x:ns, "") )
```

If we are using a stack, a sort of pop should be defined too. Due to the needs of this specific implementation a monadic action which is simultaneously “return () and pop” is defined:

```
-- Please note the simultaneous effects of the function from the
-- monadic  capsule: the returned value is the  0-tuple () and the
-- stack have just lost it's head.

pop :: M ()
pop = StOut (\m -> let  (n:ns) = m
                   in   ( () , ns ,"" )
            )
```

-- this kind of pop may be useful to finish some semantic definitions which will be written using the do-notation. Usually, such definitions are finished by a return. The problem with the standard return is that it could not have the effect of removing a user defined variable from the stack (for example one which is defined by a let statement)

IV.6 An important goal: To write the expression-interpreter in do-notation

-- This is the expressions evaluator
-- It takes an expression, the dictionary of variables called Index and returns a
--monadic value containing an Int

```
eval1           :: Exp -> Index -> M Int
eval1 exp index = case exp of
                    Constant n   -> return n
                    Variable x   -> let loc = position x index
                                    in getfrom loc
                    Minus x y    -> do { a <- eval1 x index ;
                                         b <- eval1 y index ;
                                         return (a-b) }
                    Greater x y  -> do { a <- eval1 x index ;
                                         b <- eval1 y index ;
                                         return (if a > b
                                                 then 1
                                                 else 0) }
                    Times x y    -> do { a <- eval1 x index ;
                                         b <- eval1 y index ;
                                         return ( a * b )  }
```

Other sequences of actions can also be written using the do notation, in order to deal with the evaluation of expressions involving other operators like: +, / or relational operators like < , >=, ==, /= etc. To complete the evaluator is left as exercise for the reader.

IV. 7 An other important step is to Write the command-interpreter in do-notation:

```
interpret1 :: Com -> Index -> M ()
interpret1 stmt index = case stmt of
        Assign name e-> let loc = position name index
                        in  do { v <- eval1 e index ;
                                 write loc v }
        Seq s1 s2    -> do { x <- interpret1 s1 index ;
                             y <- interpret1 s2 index ;
                            return () }
        Cond e s1 s2 -> do { x <- eval1 e index ;
                             if x == 1
                             then interpret1 s1 index
                             else interpret1 s2 index }

        While e b    -> let loop () = do { v <- eval1 e index ;
                                           if v==0 then return ()
                                          else  do {interpret1 b index ;
                                                    loop () } }
                        in loop ()
        Declare nm e stmt ->  do { v <- eval1 e index ;
                                   push v ;
                                   interpret1 stmt (nm:index) ;
                                   pop }
        Print e -> do { v <- eval1 e index ;
                        output v}
```

A special monadic action called output is used during the interpretation of the Print. The value – called v – is converted by show in a string which will be delivered as output (as the third element).

```
output :: Show a => a -> M ()
output v = StOut (\n -> ((),n,show v))
```

Having both eval1 and interpret1 implemented means we may test the interpreter now, using, for example the s1 program previously defined.

V. Now, the expression evaluator and the command interpreter can be tested

In order to test the expression evaluator, a simplest user interface may be defined. Here, eval1 is invoked for the expression 'a', using an empty dictionary as a starting point. The function extracted from the monadic capsule by unStOut is applied on the empty list [].

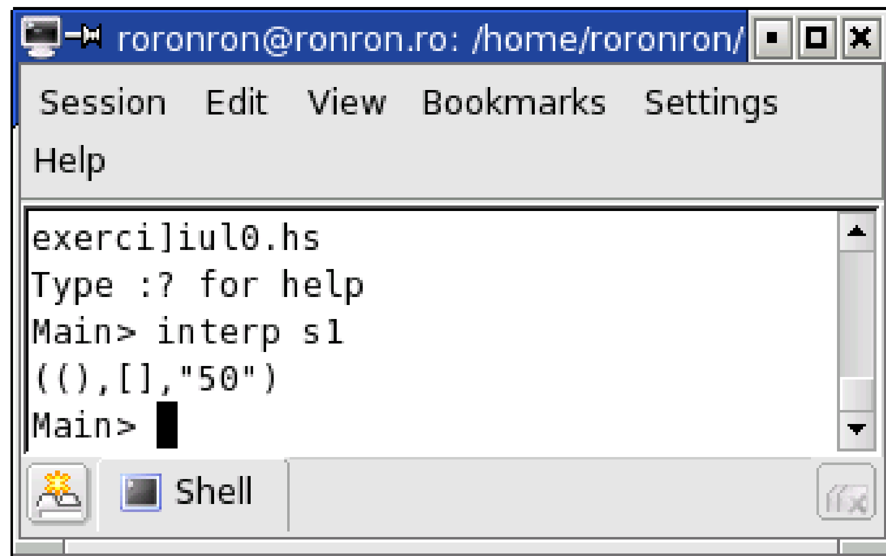
```
test a = unStOut (eval1 a [])  []
```

A similar function is used to simplify the use of the interpreter.

interp a = unStOut (interpret1 a []) []

For example, the s1 program – represented as an AST will run as can be seen in the picture bellow:

```
declare x = 150 in
    declare y = 200 in
     {while x > 0 do { x:=x-1; y:=y-1 };
       print y
     }
```



End of part one.

VI. The afternoon: the front-end will be developed

VI.1 The first step, in the afternoon: Prepare the grammar of the language beginning with the expressions.
The grammar used to specify expressions is a variant of a well known grammar. Here <rexp> means expressions which includes relations and <expr> means expressions which do not include relations. <digiti> means numbers (one or more digits).

```
<rexp>  ::= <rexp>  <relop> <expr> | <expr>
<expr> ::= <expr> <addop> <term> | <term>
<term> ::= <term> <mulop> <factor>  |< factor>
<factor> ::= <var> | <digiti>  | ( <expr> )
<var> ::= <Identifier>
<digiti> ::= <digit> | <digit> <digiti>
<digit> ::= 0 | 1 | ... | 9
<addop>  ::= + | -
<mulop>  ::= * | /
<relop> ::= > | <  | =
```

The preparation of the grammar of the language is continuing with the syntax of the future commands of our language:

<com> ::= <assign> | <seqv> | <cond> | <while> | <declare> | <printe>
<assign> ::= <identif> ":=" <rexp>
<seqv> ::= "{" <com> ";" <com> "}"
<cond> ::= "if" <rexp> "then" <com> "else" <com>
<while> ::= "while" <rexp> "do" <com>
<declare> ::= "declare" <identif> "=" <rexp> "in" <com>
<printe> ::= "print" <rexp>

VI.2 The next step is to select an existing parser combinator library (common examples: Parselib or Parsec, but don't forget to check for another one on http://www.haskell.org/) or to write a new one if needed. The following lines are explaining and using ParseLib, a library of parser combinators. The goal of the paragraph is to explain ParseLib step by step.

The ParseLib library and some other similar libraries are defining a monad of parsers. The elements of the monads belongs to the newtype Parser.

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

Remark: Every Parser is in fact a function from String to a list of pairs [(a,String)] hidden in a (monadic) capsule.

If you want to apply the hidden function to some arguments, you have to extract it from the monadic capsule, first of all. This can be done using a sort of “de-constructor” like this:

```
parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p
```

Our new data-type is declared as an instance of the predefined class Monad. The declaration should express how the return and >>= (bind) will work in the new declared monad.

```
instance Monad Parser where
return a = Parser (\cs -> [(a,cs)] )
p >>= f  = Parser (\cs ->  concat [ parse (f a) cs'
                                  | (a,cs') <- parse p cs] )
```

A parser combinators library usually includes :

```
----------------------- Simple parsers----------------
-- The parser for a single character. The first available character (c)
-- is  passed and accepted.
-- As a result, a pair meaning “c was parsed, the rest is cs” will be
-- returned by the function hidden in the monadic capsule.

            item :: Parser Char
            item = Parser (\xs -> case xs of
                                  ""     -> []
                                  (c:cs) -> [ (c,cs) ]       )
```

```
----An operator needed to deal with multiple parsing possibilities  ----
----It is simulating a paralel use of two parsers which will return a
----list containing both set of results. Algebrists had noticed that
----Parser is a actually a MonadPlus, a monad which have a “zero” and a
----“plus”.”mzero” and “plus” are respecting the appropriate algebraic
----rules.

instance MonadPlus Parser where
  mzero =  Parser (\cs -> [] )
  p `mplus` q = Parser (\cs -> parse p cs ++ parse q cs )

-- the given string of chars , cs is parsed by both
-- parsers and the final list is computed by concatenation.
```

Such a library can also include Parser Combinators which in fact are functions (or templates) able to produce other parsers, if one (or more) argument is given. Sometimes more than one arguments are parsers which means that parsers are effectively combined to create new parsers.

------------------------Parser Combinators --------------------
--1) Deterministic choice of the first posibility of parsing. Note the discarded
--   tail xs of the list (x:xs).

```
(+++)   :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p `mplus` q) cs of
                           [] -> []
                           (x:xs) -> [x]  )
```

-- 2) using a predicate we can create a parser which is able to respect a specific rule

```
sat :: (Char -> Bool) -> Parser Char
sat p = do { c <- item ; if p c then return c else mzero }
```

-- 2') A more general combinator may be defined:

```
infix  7 ?

(?) :: Parser a -> ( a-> Bool) -> Parser a
p ? test = do { b <- p ; if test b then return b else mzero }
```

-- 3) This function is able to create the parser for a specific Char.

```
char :: Char -> Parser Char
char c = sat (c ==)
```

-- 4) This function is able to create the parser for a specific String.

```
string :: String -> Parser String
string ""        = return ""
string (c:cs)    = do { char c ; string cs; return (c:cs)}
```

2''') Special combinators are usually included in the library:

-- For example, “many”. Many was explained and can be find in [Hut-98] - Monadic Parser Combinators by Graham Hutton si Erik Meijer

```
many :: Parser a -> Parser [a]
```

```
many p = many1 p +++ return []

many1 :: Parser a -> Parser [a]
many1 p = do { a<-p ; as <- many p ;
                              return (a:as) }
```

Some lexical combinators are used to provide services usually offered by lexical analyzer (lexer). Here they are:

-- L1) A parser for spaces, blanks and tabs:

```
space :: Parser String
space = many (sat isSpace)
```

-- L2) Tokens followed by spaces (usual situation) can be parsed using “token”:

```
token :: Parser a -> Parser a
token p = do { a<- p ; space ; return a}
```

-- L3) Symbol is used to parse a specific keyword (or a specific string) followed by spaces. In fact it is the previous parser “string' modified by “token”.
Different authors used different names for this parser: “symbol” and “symb” appears to be common names.

```
symbol :: String -> Parser String
symbol cs = token (string cs)
```

-- L4) In order to use a parser and simultaneously discard spaces from the beginning of the text, “apply” was defined as it follows:

```
apply :: Parser a -> String -> [(a,String)]
apply p = parse (do {space ; p } )
```

... Parsers for variables can be written

```
------------------- Parsers for variables ----------------------------

-- Parser for identifiers

ident :: Parser [Char]
ident = do { l   <- sat isAlpha ;
             lsc <- many (sat (\a -> isAlpha a || isDigit a)) ;
             return (l:lsc) }

-- ... and may be followed by spaces
identif :: Parser [Char]
identif = token ident

--  Parser for variables
var :: Parser Exp
var = do { v <-  identif ; return (Variable v) }

-- Rules of the grammar like    a::= a op b | b  may request some
specific (well known) parser combinators, quoted from [Hut-98].

chainl :: Parser a -> Parser (a->a->a) -> a -> Parser a
chainl p op a = (p `chainl1` op) +++ return a
```

```
chainl1 :: Parser a -> Parser (a->a->a) -> Parser a
p `chainl1` op = do { a <- p; rest a }
                 where
                   rest a = (do  f <- op
                                 b <-p
                                 rest (f a b) )
                              +++ return a
```

3)Combine simple parsers until you get a parser of the whole language. (Parsers are producing ASTs !)... Parsers for digit and digits can be immediately written ... Parser for expressions (and its parts) follows... Parsers for operators are simple... Parsers for commands according to the grammar
... Parsers for digit and digits can be immediately written :

```
---------- Parsers for one and more than one digits-----------------

-- 3') The digit parser is used for one single digit

digit  :: Parser Exp
digit  = do { x <- token (sat isDigit) ;
              return  (Constant ( ord x - ord '0' ) ) }

-- 3'') Parser for an unsigned number

digiti :: Parser Exp
digiti = do{ p <- digit;
             l <- many digit;
             return( foldl (\a b -> let Constant nra = a
                                        Constant nrb = b
                                    in Constant  (10*nra + nrb)  )
                           (Constant 0)
                           (p:l)            ) }

... The parser for expresions (and its parts) follows:
----------Parsers for expressions ---------------------
rexp  :: Parser Exp
rexp  = expr `chainl1` relop

expr :: Parser Exp
expr = term `chainl1` addop

term :: Parser Exp
term = factor `chainl1` mulop

-- Parser for factor, where   factor ::= var |digiti  | (expr)
factor :: Parser Exp
factor = var +++
         digiti +++
         do {  symbol "(" ; n <- rexp; symbol ")" ; return n }
```

They are based on parsers for operators. Parsers for operators are simple.

```
-- The parser for operators are using the data-constructors ---
addop :: Parser (Exp -> Exp -> Exp)
addop = do { symbol "-" ; return (Minus) }
        +++
        do { symbol "+" ; return (Plus) }
```

```
instance Fractional Int where

mulop :: Parser (Exp -> Exp -> Exp)
mulop = do { symbol "*" ; return (Times) }
       +++
       do { symbol "/" ; return (Div) }

relop :: Parser (Exp -> Exp -> Exp)
relop = do { symbol ">" ; return (Greater) }
        +++
        do { symbol "<" ; return (Less) }
        +++
        do { symbol "=" ; return (Equal) }

------------------------------------------------------------------
-- Parsers for commands (1)
-- Never forget the followings: This kind parsers did not consume the
-- blanks from the begining of the text.
-- A special function which is applied before parsing should deal with
this kind of blanks or tabs.


-- 1) print:  print rexp  (there exists a 'print'  defined in Prelude)

printe :: Parser Com
printe = do { symbol "print" ; x <- rexp ; return (Print x) }

-- 2) assign.: identif ::= rexp
-- The identifier is used here, nor the variable.

assign :: Parser Com
assign = do{x <- identif; symbol ":="; e <- rexp; return ( Assign x e)}

-- Parsers for commands (2)
-- 3)seqv ::= "{" com ";" com "}"

seqv  :: Parser Com
seqv = do { symbol "{" ; c <- com ; symbol ";"  ; d <- com ; symbol "}"
           ; return (Seq c d) }

-- 4) if-then-else: cond ::=  "if"   rexp  "then"  com  "else" com
cond  :: Parser Com
cond  = do {  symbol "if"   ; e <- rexp ;
              symbol "then" ; c <- com ;
              symbol "else" ; d <- com ;
              return (Cond e c d) }

-- 5) while, bucla: while ::=  "while" rexp  "do" com
while  :: Parser Com
while = do { symbol "while" ;
             e <- rexp ;
             symbol "do" ;
             c <- com ;
             return (While e c)  }

-- Parsers for commands (3)
-- 6) declaratia:
```

```
-- declare ::= "declare" identif  "=" rexp  "in" com
declare :: Parser Com
declare = do { symbol "declare" ;
               x <- identif ;
               symbol "=" ;
               e <- rexp  ;
               symbol "in" ;
               c <- com ;
               return  (Declare x e c ) }

-----------------------------------------------------------------
-- A command can be either recognized by one ofthe folowings parsers:
-- assign , seqv , cond , while , declare , printe
-- (Remark:  +++ is used instead of | )
-- com ::= assign | seqv | cond | while | declare | printe

com :: Parser Com
com = assign +++ seqv +++ cond +++ while +++ declare +++ printe
```

Next step: Parse the source of the “the s1 program” . You should get the same tree which was hand-written in the morning.

```
   Declare "x" (Constant 150)
       (Declare "y" (Constant 200)
          (Seq (While (Greater (Variable "x" ) (Constant 0)
                      )
                      (Seq (Assign "x" (Minus (Variable "x")
                                              (Constant 1)
                                       )
                           )
                           (Assign "y" (Minus (Variable "y")
                                              (Constant 1)
                                       )
                           )
                      )
               )
               (Print (Variable "y"))
          )
       )
```

Last step: Combine the Front-end, Back-end and an eventual tree rewriter or type-checker (if needed) to get the interpreter. Because the Front-end and the Back-end are sharing the same declarations of the ASTs, they can be combined without any problems.

REFERENCES

[Aab-96] Aaby, A. Anthony; Haskell Tutorial, http://www.cs.wwc.edu/~cs_dept/KU/PR/Haskell.html

[Aab-**] Aaby, A. Anthony; Introduction to Programming Languages , http://cs.wwc.edu/~aabyan/221_2/PLBOOK/

[Aab-05] Aaby,Anthony; Popa,Dan; Construcția Compilatoarelor folosind Flex și Bison, ISBN 973-0-04013-3, EduSoft 2005

[Aho-86] A.V. Aho, R.Sethi, and J.D. Ullman; Compilers: Principles, techniques, and Tools, Addison-Wesley, 1986

[And-05] Anderson, Lennart; Parsing with Haskell; Comp. Sci. Lund. Univ., 28 oct. 2001 – www resource

[Bar-98] Barros, José Bernardo; Almeida, José João; Haskell Tutorial,Dept. of Info., Univ. Minho, Braga, Portugal, Sept.1998

[Ben-**] Benton, Nick; Hughes, John; Moggi, Eugenio; Monads and Effects – www resource

[Cio-96] Ciobanu, Gabriel; Semantica limbajelor de programare,Editura Universității, IAȘI, 1996 ;

[Cre-01] Crenshaw, Jack W; Let's Build a Compiler - Compiler Building Tutorial, v.1.8 , 11 april. 2001 www resource

[Dau-04] Daume,Hal; Yet Another Haskell Tutorial; Copyright (c) Hal Daume III ,2004 – preprint version

[Erk-00] Erkok, Levent; Launchbury, John; Recursive Monadic Bindings: Technical Development and Details, Oregon Graduate Institute of Science and Technology, June 20,2000

[Esp-95] Espinosa. David; Semantic Lego, Ph. D. Thesis, Columbia University, 1995

[Dup-95] Duponcheel, Luc; Using catamorphism, subtypes and monad transformers for writing modular functional interpreters, Utrecht University, 16 Nov. 1995

[Dýe-**] Dýez, M.C. Luego; Gonzalez, B.M. Rodrýguez; A Language Prototyping Tool based on Semantic Building Blocks, www resource

[Erk-02] Erkok, Levent; Value Recursion in Monadic Computations,(Ph.D thesis), OGI School of Science and Engineering at Oregon Health and Science University , Oct. 2002

[Har-01] Harrison Lawrence William – *Modular compilers and their correctness proofs*, University of Illinois, 2001

[Hud-99] Hudak Paul, Peterson John, Fasel Joseph – *A Gentle Introduction to Haskell 98*, Yale University, Los Alamos Laboratory, 1999

[Hut-98] Hutton, Graham ; Meijer, Erik - *Monadic Parser Combinators* - Technical report NOTTCS-TR-96-4,- Department of Computer Science, University of Nottingham, 1996. http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps

[TZE-*] Tim Sheard, Zine-el-abidine Benaisa, Emir Pasalic; DSL Implementation using staging and monads, Pacific Software Research Center, www resource, undated

... and more ...including (last but not least) papers by E.Moggi, W.Harrison, P.Wadler, J.Peterson, G.Hutton, L.Duponcheel, S.Peyton Jones