How to build an adaptable interpreter in one day

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

based on papers provided by the Haskell community and some other resources

Morning: The Back -End

1)Declare the AST (expressions, commands)
2)Prepare a small program including commands (for testing purpose) refered as “the s1 program”
3)Declare the datastructures of the environment and some functions to manipulate it
4)Replace the virtual machine with a monad
5)Define some basic operations with monadic values
6)Write the expression-interpreter in do-notation
7)Write the command-interpreter in do-notation

You may test the back-end now !

Afternoon: The Front -End

1) Prepare the grammare of the language
2) Use a parser combinators library(or write a new one)
3) Combine simple parsers until you get a parser of the entier language. (Parsers are producing AST)
4) Parse the source of the “the s1 program” . You should get the same tree which was hand-written in the morning
5) Combine: Front-end, Back-end and an eventual tree rewriter (if needed) to get the interpreter

1) Declare the AST (expressions)

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

1') Declare the AST (commands)

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

2)Prepare a small program including commands (for testing purpose) refered as “the s1 program”

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

2')Prepare a small program including commands (for testing purpose) refered as “the s1 program”. Here: the AST of s1

- 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"))
-       )
-     )
-

3)Declare the datastructures of the environment

- **type Location = Int**
- **type Index = [String]**
- **type Stack = [Int]**

Note: We can use other kind of environments :
Ex: list of pairs:

[(String, Value)]

3')...some functions to manipulate it

```
position          :: String  -> Index ->  Location
-- locatia era doar un numar
position name index = let
                      pos n (nm:nms) = if name == nm
                                       then n
                                       else pos (n+1) nms
                                       -- param n creste , lista scade
                    in pos 1 index
-- de la nr 1  plecam cu cautarea
```

3'')...some functions to manipulate it

- -- functia care extrage de pe stiva intregul din pozitia indicata
- -- acest intreg este valoarea variabilei
- fetch :: Location -> Stack -> Int
- fetch n (v:vs) = if n == 1 then v else fetch (n-1) vs

3''')...some functions to manipulate it

- functia care pune ... in stiva, opera^nd ca intr-un vector !!
- put   :: Location -> Int -> Stack -> Stack
- put n x (v:vs) = if n==1
-                          -- daca se cerea punere in pozitia n==1
-                          then x:vs
-                          -- rezulta aceeasi stiva dar cu un nou cap/varf
-                         else v:(put (n-1) x vs)

4)Replace the virtual machine with a monad

```
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) )
-- unde
unStOut (StOut f) = f
```

5)Define some basic operations with monadic values

- getfrom :: Location -> M Int
- getfrom i = StOut (\ns -> (fetch i ns, ns, ""))
-
- -- modifica stiva punind in ea o valoare, intr-o anume pozitie (ca la vector)
- write :: Location -> Int -> M ()
- write i v = StOut (\ns -> ((), put i v ns, ""))
-
- -- modifica stiva punind in ea o valoare, in cap / top
- push :: Int -> M ()
- push x = StOut(\ns -> ((), x:ns, ""))

5')Define some basic operations with monadic values

- -- acest pop este un fel de "return_pop " si este folosit ca atare
- pop :: M ()
- pop = StOut (\m -> let
-   (n:ns) = m
-                                   in
-   (() , ns ,"")
-                        )
- -- e necesar deoarece nu puteam folosi la finalul unor definitii in do-notatie return (). Prin felul cum e definit return incapsuleaza o functie care nu modifica stiva, deci nu are efectul lui pop

6)Write the expression-interpreter in do-notation

- -- 1) - Evaluatorul de expresii aritmetice
- -- evalueaza o expresie intr-un context si rezulta un intreg "incapsulat"

```
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) }
```

6')Write the expression-interpreter in do-notation

```
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 )  }
```

7)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 }
```

7')Write the command-interpreter in do-notation

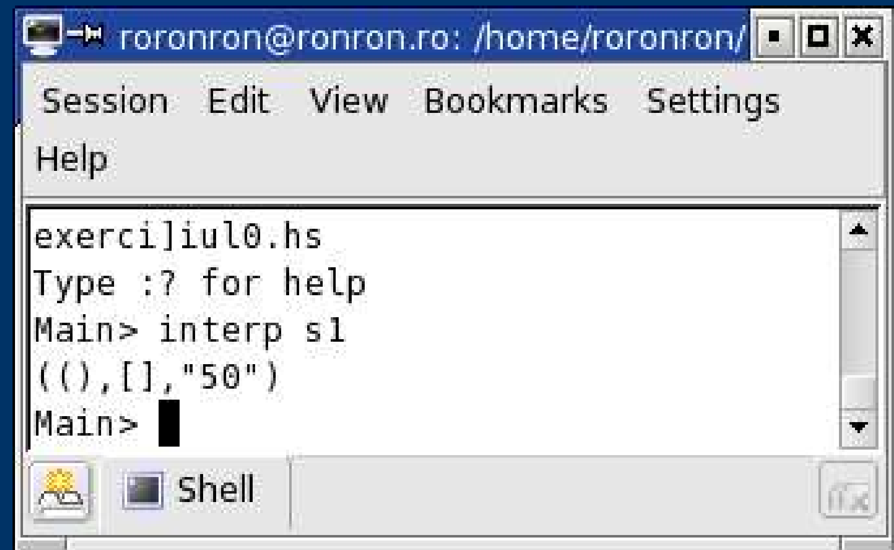
```
While e b    -> let loop () = do { v <- eval1 e index ;
                                     if v==0 then return ()
                                     else  do {interpret1 b index ;
                                               loop () } }
                in loop ()
Declare nm e stmt ->  do {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

You may test the interpreter now !

- -- Testarea interpretarii expresiilor
- test a = unStOut (eval1 a []) []
-
- -- Testarea interpretarii comenzilor
- interp a = unStOut (interpret1 a []) []
-
- output :: Show a => a -> M ()
- output v = StOut (\n -> ((),n,show v))
-

TAKE A BREAK – TAKE LUNCH !

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



1)Prepare the grammare of the language beginning with the expressions

Gramatica expresiilor este cea cunoascuta:
(rexp - expresiile cu relatii, expr - expresiile fara comparatii)

```
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 ::= > | <  | =
```

1')Prepare the grammare of the language; continuing with the commands

```
com ::= assign | seqv |  cond |while | declare | printe

    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
```

2)Use a parser combinators library (or write a new one)

Tipul parser

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

Functia deconstructor, scoate functia cs -> ... de sub constructorul de tip.
Ca sa pot folosi valoarea monadica intr-o alta prelucrare trebuie sa scot functia din capsula

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

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

2') Such a library can includes :

```
--------------------- Parsere simple -----------------
      parserul pt. caractere, primul caracter e parsat si
      dat ca valoare
      rezultata iar restul stringului e in asteptare. o
      singura parsare posibila.

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

-------------------- Parsere nedeterministe -------------

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

      Aplica ambii "parseri" pe string-ul de intrare si
      returneaza lista rezultatelor.
```

2") Such a library can includes :

```
-------------------------Combinatori de parsere ---------------------
--- Pentru a putea scrie parsere complicate ne trebuie combinatori

-- 1) alegerea determinista, doar a primei variante de analiza
(+++)    :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p `mplus` q) cs of
                           [] -> []
                           (x:xs) -> [x]  )

-- 2) predicatul, conditia impusa unui caracter
sat :: (Char -> Bool) -> Parser Char
sat p = do { c <- item ; if p c then return c else mzero }

-- 2') un combinator mult mai general, introdus de noi asa
infix  7 ?
(?) :: Parser a -> ( a-> Bool) -> Parser a
p ? test = do { b <- p ; if test b then return b else mzero }


-- 3) parserul pentru un caracter
char :: Char -> Parser Char
char c = sat (c ==)

-- 4) Parserul recursiv pentru un anumit string (parser cu "exemplu")
string :: String -> Parser String
string ""          = return ""
string (c:cs)      = do { char c ; string cs; return (c:cs)}
```

2''') Such a library can includes :

```
--Alti combinatori,printre care many

-- Extras din lucrarea [Hut-98] - Monadic Parsser
Combinators
-- de 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) }
```

... Lexical combinators to provide services usually ofered by lexical analyzer (lexer)

```
----------- Combinatorii lexicali pentru analiza lexicala
----------
-- L1) Parseaza un SIR de spatii, tab-uri si enter-uri

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

-- L2) Parseaza un atom URMAT de spatii (modificator)
token :: Parser a -> Parser a
token p = do { a<- p ; space ; return a}

-- L3) parseaza un anume atom lexical urmat de spatii
-- e parserul string modificat de modificatorul token, se mai nu-
mea symbol
-- la alti autori iar la altii symb
symbol :: String -> Parser String
symbol cs = token (string cs)

-- L4) aplicarea unui parser ignorandu-se spatiile dinaintea tex-
tului
apply :: Parser a -> String -> [(a,String)]
apply p = parse (do {space ; p } )
```

... Parsers for variables can be written

```
-------------------------Parserele pentru variabile
----------------------------
-- Intai definim notiunea de identificator pornind de la
-- caracter si alfanum
-- El e format din o litera alfabetica si litera sau cifra
-- repetata de mai multe ori

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

-- poate avea si spatii
identif :: Parser [Char]
identif = token ident

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

... Tipical rules of a grammar may request specific (well known) parser combinators

```
-------------- Combinatorul pentru expresii date prin reguli -----
-- ale  gramaticii de forma             a::= a op b | b

-- Se presupune ca parserul care recunoaste operatorul returneaza
-- chiar
-- functia de 2 argumente care face operatia, iar in plus aceasta
-- este asociativa la stanga.



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 entier language. (Parsers are producing ASTs !)

*... Parsers for **digit** and **digits** can be imediately written*
*... Parser for **expresions** (and its parts) follows*
*... Parsers for **operators** are simple*
*... Parsers for **commands** according to the grammar*

... Parsers for *digit* and *digits* can be imediately written :

```
----------Parserele pentru digiti se modifica -------------------

-- 3') parserul pentru un digit

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

-- 3'') Parserul pentru un numar intreg fara semn - introdus  aici

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)             ) }
```

... Parser for *expresions* (and its parts) follows:

```
----------Trecem la construirea expresiilor ---------------------
rexp  :: Parser Exp
rexp  = expr `chainl1` relop

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

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

-- factorul --   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

```
---------------- Operatorii folosesc constructorii tipului -------

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)

```
-- Nota: Nu uitati ca aceste parsere NU elimina spatiile de la
-- inceputul programului, ci doar cele de dupa atomii lexicali
-- Eliminarea spatiilor de la inceput va fi o functie care
-- include parserul limbajului.


-- 1) print-ul:  print rexp  (nu uitati ca print e definit in Pre-
lude)

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

-- 2) assignarea: identif ::= rexp
-- atentie, aici se foloseste identificatorul variabilei nu vari-
abila

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 ; sym-
bol "}" ; return (Seq c d) }

-- 4) if-ul: 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 ) }

--------------------------------------------------------------------
-- Putem acum defini  comanda, vom adauga aici alte alternative
-- (Nota bene:  +++ separa alternativele )

-- com ::= assign | seqv | cond | while | declare | printe

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

4)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"))
        )
      )
```

5) Combine: Front-end, Back-end and an eventual tree rewriter (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 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, 2005

[Aab-05] Aaby,Anthony; The Lambda calculus; Walla Walla College, June 29, 2005, (draft copy – www resource)

[Adá-**] Adámek, Jirí; Herrlich, Horst; Strecker, George E.; *Abstract and Concrete Categories*; http://katmat.math.uni-bremen.de/acc/

[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-02] Barr, Michael; Wells, Charles; *Toposes Triples and Theories*, ver 1.1 , 7 nov 2002, http://www.cwru.edu/artsci/math/wells/pub/ttt.html

[Adá-**] Adámek, Jirí; Herrlich, Horst; Strecker, George E.; *Abstract and Concrete Categories*; http://katmat.math.uni-bremen.de/acc/

[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-02] Barr, Michael; Wells, Charles; *Toposes Triples and Theories*, ver 1.1 , 7 nov 2002, http://www.cwru.edu/artsci/math/wells/pub/ttt.html

[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

[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

[Dav-86] Davidoviciu, Adrian; Bărbat, Boldur; *Limbaje de programare pentru sisteme în timp real*; Editura Tehnică,
București,1986, p.185.

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

[Ear-70] Earley,Jay; Sturgis Howard; *A Formalism for Translator Interactions*, Communications of the ACM, Volume 13, Number 10, October 1970, p.607-617;

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

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

[Ear-70] Earley,Jay; Sturgis Howard; *A Formalism for Translator Interactions*, Communications of the ACM, Volume 13, Number 10, October 1970, p.607-617;

[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

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

The complete A-Z reference will probably be included in the printed paper.

That's all ! Thank you very much !

Questions may be sent to:
Dan Popa popavdan@yahoo.com