Thesis Implementation
Document 1
Parser

Due on Monday, May 19, 2014

Dr. David Casperson 13:30

Mehul Solanki
solanki@unbc.ca
230108015

Last modified on May 21, 2015

Contents

Floating Point Parser

Phase 1 : Problems, Librabries and Minimalistic Examples

The first issue is that the library **prolog-0.2.0.1** does not support floating point numbers. Hence, the requirement is for a simple floating point parser which could use the available HASKELL packages like **parsec** for it to fit into the existing implementation from the library.

This code has its roots in a tutorial from the company FP Complete which recently released a web based IDE for HASKELL. As an exercise this code parses signed floating point numbers. The code will be described function by function.

Beginning with a few basic operators from the imported modules,

1. Control.Applicative

   - (<$>) :- Represents an infix synonym for *fmap*.

     ```
     (<$>) :: Functor f => (a -> b) -> f a -> f b
     ```

   - (<⋆>), Sequential application.

     ```
     <*> :: f (a -> b) -> f a -> f b
     ```

   - (⋆>), Same as the above but discarding the value of the first argument.

     ```
     (*>) :: f a -> f b -> f b
     ```

   - (<|>) Binary Operator.

     ```
     (<|>) :: f a -> f a -> f a
     ```

2. Parsec

   - Text.ParserCombinators.Parsec.Prim.**parseTest**, for testing / running parsers.

     ```
     parseTest :: (Stream s Identity t, Show a) => Parsec s () a
                                                   -> s -> IO ()
     ```

   - Text.Parsec.Combinator.**many1**, applying a parser one or more times.

     ```
     many1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
     ```

   - Text.ParserCombinators.Parsec.Char.**digit**, parsing a digit to a single character.

     ```
     digit :: Stream s m Char => ParsecT s u m Char
     ```

Some auxiliary functions for working with lists for appending lists and elements.

```
(<++>) :: Applicative f => f [a] -> f [a] -> f [a]
(<++>) a b = (++) <$> a <*> b

(<:>) :: Applicative f => f a -> f [a] -> f [a]
(<:>) a b = (:) <$> a <*> b
```

Each function below parses a specific part of a floating point number, hence each function is a parser in itself. Beginning with parsing numbers, the funcion used here is *digit* along with *many1* for parsing a number with multiple digits.

```
number :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
number = many1 digit
```

To parse signed numbers the sign leading a number which can either be positive or negative needs to be parsed. In the *plus* parser the sign is discarded as *+3* can be written as *3*. But for the negative sign it must be appended to the number itself, hence <:>is used.

```
plus :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
plus = char '+' *> number

minus :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
minus = char '-' <:> number
```

Putting the three parser together we get a function for parsing signed integers.

```
integer :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
integer = plus <|> minus <|> number
```

The parser for **prolog-0.2.0.1** treats numbers are strings but the example treats them as floats and hence some modifications are needed in an attempt to fit the code for parsing floats to extend the library. The original code in the example is,

```
float :: Stream s m Char => ParsecT s u m Float
float :: ParsecT [Char] u Data.Functor.Identity.Identity Float
float =  fmap rd $ integer <++> decimal <++> exponent
    where rd       = read :: String -> Float
          decimal  = option "" $ char '.' <:> number
          exponent = option "" $ oneOf "eE" <:> integer
```

while the library parser for Prolog *atoms* has the following type signature,

```
atom :: ParsecT String u Identity [Char]
```

The modification is not to read the number as a string, hence removing the use of *rd* function.

```
float :: ParsecT [Char] u Identity [Char]
float = integer <++> decimal <++> exponent
    where rd       = read :: String -> Float -- not used
          decimal  = option "" $ char '.' <:> number
          exponent = option "" $ oneOf "eE" <:> integer
```

Lastly putting it all together and testing out the parser using the function *parseTest*.

```
main :: IO ()
main = forever $ do putStrLn "Enter a float: "
                    input <- getLine
                    parseTest float input
{--
OUTPUT :-
Enter a float:
-12.3
"-12.3"
--}
```

The complete program with pragmas and imports is as follows,

```
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE FlexibleContexts #-}

module Experiment(float
        )where

-- Mehul Solanki.

import Control.Monad
import Text.Parsec
import Control.Applicative hiding ((<|>))
import Data.Functor.Identity
import Text.Parsec.Prim

(<++>) :: Applicative f => f [a] -> f [a] -> f [a]
(<++>) a b = (++) <$> a <*> b

(<:>) :: Applicative f => f a -> f [a] -> f [a]
(<:>) a b = (:) <$> a <*> b

number :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
number = many1 digit

plus :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
plus = char '+' *> number

minus :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
```



```haskell
minus = char '-' <:> number

integer :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
integer = plus <|> minus <|> number

--float :: Stream s m Char => ParsecT s u m Float
--float :: ParsecT [Char] u Data.Functor.Identity.Identity Float
--float =  fmap rd $ integer <++> decimal <++> exponent
--    where rd       = read :: String -> Float
--          decimal  = option "" $ char '.' <:> number
--          exponent = option "" $ oneOf "eE" <:> integer

float :: ParsecT [Char] u Identity [Char]
float = integer <++> decimal <++> exponent
    where rd       = read :: String -> Float
          decimal  = option "" $ char '.' <:> number
          exponent = option "" $ oneOf "eE" <:> integer

main :: IO ()
main = forever $ do putStrLn "Enter a float: "
                    input <- getLine
                    parseTest float input
```

The shortcomings of the above program are the type mismatch and floating point numbers starting with a decimal point throw an error. To solve the issue we need to look at other options. Either create some sort of support for the same or look into some other libraries. Turn out that there are a number of variations that have sprung out of **parsec** namely, **parsec-number**, **parsec1**, **parsec2**, **parsec3**, **parsec-number** which give a variety of improvements over the parent. The intrest is in the *floating* parsers that the library provides.

```haskell
floatParser x = Primitive.parseTest (Numbers.floating3 True) x
```

The above example uses the *floating3* function, which accepts a *boolean* value for puting a condition on whether or not there should be a number after the decimal dot. In the above *3.* will throw an error.

```haskell
import Text.ParserCombinators.Parsec.Number as Numbers

import Text.ParserCombinators.Parsec.Prim as Primitive

floatParser :: [Char] -> IO ()
floatParser x = Primitive.parseTest (Numbers.floating3 True) x
```

```

{--
Output :-

floatParser "12.3"
12.3

floatParser ".3"
0.3

floatParser "3."
parse error at (line 1, column 3):
unexpected end of input
expecting fraction

--}
```

But along with the ease of use comes a few other issues, namely the type incompatibilities and that signed numbers are not recognized.

```
floatParser "+3.1"
parse error at (line 1, column 1):
unexpected "+"
expecting fraction or digit
```

The way around this would be add something to recognise positive or negative signs just like the example code mentioned before.

```
sign :: Num a => CharParser st (a -> a)
```

The function above parses an optional plus or minus sign, returning negate or id. The work for parsing signed numbers is in progress. Mean while talking about the type mismatch, in the **prolog-0.2.0.1** the parsers have the following general type signature,

```
parser :: ParsecT s u m a
```

while the one in **parsec** has,

```
parser :: Floating f => CharParser st f
```

The easy solution to this issue is to look at the variations of the parent library. The **parsec3** and **parsec3-numbers** libraries gives us parsers with the following type signatures,

```
parser :: (Floating f, Stream s m Char) => ParsecT s u m f
```

which is very similar to the ones in the PROLOG library. As with the parent library, **parsec3-number** provides *sign* function for dealing with positive or negative numbers. The only thing needed is to add this functionality tot the parser(They seem to have given how to use it in the hackage description but I am not able to get it).
To recap,

1. **prolog-0.2.0.1** does not support floating point numbers.
2. The example supports signed floating point numbers but there exists a type mismatch.
3. The **parsec** library along with its variations does come close but again not completely.
4. The libraries above provide a function to recognise signs but they need to be integrated with the parser.
5. Changes need to be made in the *Interpreter, Parser, Syntax* among others of the PROLOG library in order to accomodate the above.

Phase 2 : Moving towards a partial solution

This section is concerned with the points in the recap listed above. Mainly points 2, 3 and 4 which are in the direction of adding parsing capabilities to the the existing floating parsers. Going back to the libraries **parsec** and **parsec-number** which provides functions like *parsetest, floating, sign* and so on. The addition is from the *Control.Monad* module namely *ap*,

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

which is *liftM* but it promotes function application. The result is a parser which can deal with signed floating point number. Consider the following,

```
floatParser :: [Char] -> IO ()
floatParser s = parseTest (ap sign floating) s

{--
OUTPUT :-

floatParser "12.3"
12.3

floatParser "-12.3"
-12.3
```

```
floatParser "+12.3"
12.3

--}
```

will result in a parsed floating point number. Improving the code would be to allow signed floating point numbers to begin with a decimal point and requiring a number after the decimal point. The **parsec-numbers** library provides a function *floating3* with accepts a boolean value for restriction on the decimal parts of a number.

```
floating3 :: Floating f => Bool -> CharParser st f
```

Consider the following function,

```
floatParser :: [Char] -> IO ()
floatParser x = Primitive.parseTest (ap sign
         (Numbers.floating3 True)) x

{--
OUTPUT :-

floatParser "-.3"
-0.3

floatParser "-3."
parse error at (line 1, column 4):
unexpected end of input
expecting fraction

--}
```

So summing up, the function *floatParser* parses signed and unsigned floating point numbers which may begin with a decimal point but cannot be ending with one. But one mojor issue remains, the incompatibility of the types between the program above and how things are done in the library.

The **parsec** library has given rise to a number of modified libraries which add functionality and even claim to be ready for the industry. Coming back to **parsec3** and **parsec3-numbers** which which provide the correct type signature with respect to the Prolog library. From the example above parsing the sign involves the same procedure as the derived libraries have the same functions as the parent.

So we have looked at a number of options that can be used to extend just the parser of the PROLOG library. A last detail is that floating point numbers have a decimal point contained in them. The issue here is that PROLOG clauses written in a file are terminated using a period, the same as floating point numbers. So for now (till we depend upon PROLOG syntax) a decimal point will be replaced by a question mark. Another change is that since the library parser does sign checking for us, the program below does not require the functions *plus* and *minus*. The modified example from earlier is below,

```
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE FlexibleContexts #-}


module Experiment(float
        )where

-- Mehul Solanki.

import Control.Monad
import Text.Parsec
import Control.Applicative hiding ((<|>))
import Data.Functor.Identity
import Text.Parsec.Prim

(<++>) :: Applicative f => f [a] -> f [a] -> f [a]
(<++>) a b = (++) <$> a <*> b

(<:>) :: Applicative f => f a -> f [a] -> f [a]
(<:>) a b = (:) <$> a <*> b

number :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
number = many1 digit

integer :: ParsecT [Char] u Data.Functor.Identity.Identity [Char]
integer = number

--float :: Stream s m Char => ParsecT s u m Float
--float :: ParsecT [Char] u Data.Functor.Identity.Identity Float
--float =  fmap rd $ integer <++> decimal <++> exponent
--    where rd       = read :: String -> Float
--          decimal  = option "" $ char '.' <:> number
--          exponent = option "" $ oneOf "eE" <:> integer
```

Floating Point Parser continued on next page...

```

float :: ParsecT String u Identity [Char]
float = integer <++> decimal <++> exponent
    where rd       = read :: String -> Float
          decimal  = option "" $ char '?' <:> number
          exponent = option "" $ oneOf "eE" <:> integer

main :: IO ()
main = forever $ do putStrLn "Enter a float: "
                    input <- getLine
                    parseTest float input

{--
OUTPUT :-

parseTest float "12?3"
"12?3"
--}
```

along with the syntax file,

```
{-# LANGUAGE DeriveDataTypeable, ViewPatterns, ScopedTypeVariables #-}
module Syntax
   ( Term(..), var, cut
   , Clause(..), rhs
   , VariableName(..), Atom, Goal, Program
   , cons, nil, foldr_pl
   , arguments -- FIXME Should not be exposed
   , hierarchy
   , Operator(..), Assoc(..)
   )
where

import Data.Generics (Data(..), Typeable(..))
import Data.List (intercalate)
import Data.Char (isLetter)


{--
Intercalate Examples

```

```
:t intercalate
intercalate :: [a] -> [[a]] -> [a]

intercalate [1..5] [[6..10], [11..15], [16..20]]
[6,7,8,9,10,1,2,3,4,5,11,12,13,14,15,1,2,3,4,5,16,17,18,19,20]

intercalate [1..5] [[6..10], [11..15]]
[6,7,8,9,10,1,2,3,4,5,11,12,13,14,15]
--}

{--
A Prolog Term can be an

Atom
Struct "hello" []
hello
Struct "hello" [Struct "a" []]
hello(a)

Variable
Var (VariableName 125 "X")
X#125

Wildcard (Don't Care)
_

Cut
Cut 0
!
Cut 4
!

--}

data Term = Struct Atom [Term]
          | Var VariableName
          | Wildcard -- Don't cares
          | Cut Int
      deriving (Eq, Data, Typeable)

var :: String -> Term
```

```haskell
var = Var . VariableName 0

cut :: Term
cut = Cut 0

{--

Clause
Clause (Struct "hello" [Struct "a" []]) ([Struct "world" []])
"hello(a) :- world"


Clausefn
????

--}

data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
            | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
      deriving (Data, Typeable)

rhs :: Clause -> [Term] -> [Goal]
rhs (Clause   _ rhs) = const rhs
rhs (ClauseFn _ fn ) = fn

data VariableName = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)

type Atom         = String
type Goal         = Term
type Program      = [Clause]

--Precedence, less than or equal to.
instance Ord Term where
   (<=) = wildcards <=! variables <=! atoms <=! compound_terms
                               <=! error "incomparable"

-- Uses the auxiliary functions below

(<=!) :: Ord a => (t -> Maybe a) -> (t -> t -> Bool) -> t -> t ->
                                                   Bool
```

```
infixr 4 <=!
(q <=! _) (q->Just l) (q->Just r) = l <= r
(q <=! _) (q->Just _) _ = True
(q <=! _) _ (q->Just _) = False
(_ <=! c) x y = c x y

{--
The following functions take terms and convert them into Maybes
--}

wildcards :: Term -> Maybe ()
wildcards Wildcard = Just ()
wildcards _        = Nothing

variables :: Term -> Maybe VariableName
variables (Var v) = Just v
variables _       = Nothing

numbers :: Term -> Maybe Double
numbers (Struct (reads->[(n :: Double,"")]) []) = Just n
numbers _                                       = Nothing

atoms :: Term -> Maybe [Atom]
atoms (Struct a []) = Just [a]
atoms _             = Nothing

compound_terms :: Term -> Maybe (Int, Atom, [Term])
compound_terms (Struct a ts) = Just (length ts, a, ts)
compound_terms _             = Nothing

-- Printing stuff
instance Show Term where
   show = prettyPrint False 0

prettyPrint :: Bool -> Int -> Term -> [Char]
prettyPrint True _ t@(Struct "," [_,_]) = "(" ++
                        prettyPrint False 0 t ++  ")"

prettyPrint f n (Struct (flip lookup operatorTable->
                        Just (p,InfixOp assoc name)) [l,r]) =
   parensIf (n >= p) $ prettyPrint f n_l l ++ spaced name ++
```

```
                                                prettyPrint f n_r r
      where (n_l,n_r) = case assoc of
                              AssocLeft  -> (p-1, p)
                              AssocRight -> (p, p-1)

prettyPrint f n (Struct (flip lookup operatorTable->
                        Just (p,PrefixOp name)) [r]) =
   parensIf (n >= p) $ name ++
                                    prettyPrint f (p {- Non-associative -})

prettyPrint _ _ t@(Struct "." [_,_]) =
   let (ts,rest) = g [] t in
      --case guard (isNil rest) >> sequence (map toChar ts) of
      --   Just str -> prettyPrint str
      --   Nothing  ->
            "[" ++ intercalate "," (map (prettyPrint True 0) ts) ++ (if isN
                    ++ (prettyPrint True 0) rest) ++  "]"
   where g ts (Struct "." [h,t]) = g (h:ts) t
         g ts t = (reverse ts, t)
         isNil (Struct "[]" []) = True
         isNil _                = False

prettyPrint _ _ (Struct a [])   = a
prettyPrint _ _ (Struct a ts)   = a ++ "(" ++ intercalate ", " (map (pretty
prettyPrint _ _ (Var v)         = show v
prettyPrint _ _ Wildcard        = "_"
prettyPrint _ _ (Cut _)         = "!"
--prettyPrint _ _ ((==cut)->True)  = "!"
--prettyPrint _ _ (Cut n)          = "!^" ++ show n


spaced s = let h = head s
               l = last s
           in spaceIf (isLetter h) ++ s ++ spaceIf (isLetter l || ',' == l)

spaceIf True  = " "
spaceIf False = ""

parensIf :: Bool -> String -> String
parensIf True  s = "(" ++ s ++")"
parensIf False s = s
```

Floating Point Parser continued on next page...

```
operatorTable :: [(String, (Int,Operator))]
operatorTable = concat $ zipWith (map . g) [1..] $ hierarchy False
 where g p op@(InfixOp _ name) = (name,(p,op))
       g p op@(PrefixOp name)  = (name,(p,op))

instance Show VariableName where
   show (VariableName 0 v) = v
   show (VariableName i v) = v ++ "#" ++  show i

instance Show Clause where
   show (Clause   lhs [] ) = show $ show lhs
   show (Clause   lhs rhs) = show $ show lhs ++ " :- " ++ intercalate ", "
   show (ClauseFn lhs _  ) = show $ show lhs ++ " :- " ++ "<Haskell functio



foldr_pl :: (Term -> t -> t) -> t -> Term -> t
foldr_pl f k (Struct "." [h,t]) = f h (foldr_pl f k t)
foldr_pl _ k (Struct "[]" [])   = k

cons t1 t2 = Struct "."  [t1,t2]
nil        = Struct "[]" []

data Operator = PrefixOp String
              | InfixOp Assoc String
data Assoc = AssocLeft
           | AssocRight

hierarchy :: Bool -> [[Operator]]
hierarchy ignoreConjunction =
   --[ [ InfixOp NonAssoc "-->", InfixOp NonAssoc ":-" ]
   [ [ infixR ";" ] ] ++
   (if ignoreConjunction then [] else [ [ infixR "," ] ])  ++
   [ [ prefix "\\+" ]
   , map infixL ["<", "=..", "=:=", "=<", "=", ">=", ">", "\\=", "is",
                 "==", "@<", "@=<", "@>=", "@>"]
   , map infixL ["+", "-", "\\"]
   , [ infixL "*"]
   , [ infixL "mod" ]
```

```
    , [ prefix "-" ]
    , [ prefix "$" ] -- used for quasi quotation
    ]
  where
    prefix = PrefixOp
    infixL = InfixOp AssocLeft
    infixR = InfixOp AssocRight


--infix 6 \\
--x \\ y = Struct "\\" [x,y]

arguments ts xs ds = ts ++ [ xs, ds ]
-- arguments ts xs ds = [ xs \\ ds ] ++ ts


{--
Data Type Genric Programming / Generic Programming is a way of defining
functions to work on Structures of Data Types rather than Data Types
themselves.

Thus a single function can be designed to work on a number of Data Types

--}
```

the Prolog example to be parsed,

```
% Mehul Solanki.

% Shoe Problem.

/*

Harriet, upon returning from the mall, is happily describing her four shoe p
to her friend Aurora. Aurora just loves the four different kinds of shoes th
bought (ecru espadrilles, fuchsia flats, purple pumps, and suede sandals),
but Harriet can't recall at which different store  (Foot Farm, Heels in a Ha
The Shoe Palace, or Tootsies) she got each pair. Can you help these two figu
order in which Harriet bought each pair of shoes, and where she bought each

1. Harriet bought fuchsia flats at Heels in a Handcart.
2. The store she visited just after buying her purple pumps was not Tootsies
```

```
3. The Foot Farm was Harriet's second stop.
4. Two stops after leaving The Shoe Place, Harriet bought her suede sandals

Determine: Order - Shoes - Store
*/

myList(['a', 'A', -12?3]).

append([],X,X).
append([H1| T1], L2,[H1| T2]) :- append(T1, L2, T2).

right(X,Y,L):-         append(_,[X,Y|_],L).

len([],0).
len([_|T],N) :- len(T,X), N is X+1.

start(S) :-
len(S,4),
S = [[Shoe1,Store1],[Shoe2,Store2],[Shoe3,Store3],[Shoe4,Store4]],
member(Store1,[ffs,hhs,tsps,ts]),
member(Store2,[ffs,hhs,tsps,ts]),
member(Store3,[ffs,hhs,tsps,ts]),
member(Store4,[ffs,hhs,tsps,ts]),
member(Shoe1,[ee,ff,pp,ss]),
member(Shoe2,[ee,ff,pp,ss]),
member(Shoe3,[ee,ff,pp,ss]),
member(Shoe4,[ee,ff,pp,ss]),
not(Store1 = Store2),
not(Store1 = Store3),
not(Store1 = Store4),
not(Store2 = Store3),
not(Store2 = Store4),
not(Store3 = Store4),
not(Shoe1 = Shoe2),
not(Shoe1 = Shoe3),
not(Shoe1 = Shoe4),
not(Shoe2 = Shoe3),
not(Shoe2 = Shoe4),
not(Shoe3 = Shoe4),
member([ff,hhs],S),
not(right([pp,_],[_,ts],S)),
```

```
S = [_,[_,ffs],_,_],
S = [[_,tsps],_,_,[ss,_]].


/*
start(S).
S = [[ee, tsps], [pp, ffs], [ff, hhs], [ss, ts]] ;
S = [[pp, tsps], [ee, ffs], [ff, hhs], [ss, ts]] ;
false.
*/
```

```

```

and finally the parser,

```
-- Mehul Solanki.

-- A simple parser for Prolog in Haskell derived and modified
-- from prolog-0.2.0.1.

module Parser
   ( consult, consultString, parseQuery
   , program, whitespace, comment, clause, terms, term, bottom, vname
   ) where


import Text.Parsec
import Text.Parsec.Expr hiding (Assoc(..))
import qualified Text.Parsec.Expr as Parsec
import qualified Text.Parsec.Token as P
import Text.Parsec.Language (emptyDef)
import Control.Applicative ((<$>),(<*>),(<$),(<*))
import Data.Functor.Identity  -- Added later was not there as a result of w



import Text.ParserCombinators.Parsec.Number as Numbers


import Experiment

import Syntax
```

```
{--
Like consult in Prolog.
If the program is parsed correctly, then each predicate
is added to the list of results.
--}
consult :: FilePath -> IO (Either ParseError Program)
consult = fmap consultString . readFile
{--
consult "/home/mehul/Dropbox/PrologPrograms/shoeStore.pl"
Right ["myList([a,A,-(12?3)])",
"append([], X, X)",
"append([H1|T1], L2, [H1|T2]) :- append(T1, L2, T2)",
"right(X, Y, L) :- append(_, [X,Y|_], L)",
"len([], 0)",
"len([_|T], N) :- len(T, X),
                  N is X+1",
"start(S) :- len(S, 4),
            S=[[Shoe1,Store1],[Shoe2,Store2],[Shoe3,Store3],
            [Shoe4,Store4]],
            member(Store1, [ffs,hhs,tsps,ts]),
            member(Store2, [ffs,hhs,tsps,ts]),
            member(Store3, [ffs,hhs,tsps,ts]),
            member(Store4, [ffs,hhs,tsps,ts]),
            member(Shoe1, [ee,ff,pp,ss]),
            member(Shoe2, [ee,ff,pp,ss]),
            member(Shoe3, [ee,ff,pp,ss]),
            member(Shoe4, [ee,ff,pp,ss]),
            not(Store1=Store2),
            not(Store1=Store3),
            not(Store1=Store4),
            not(Store2=Store3),
            not(Store2=Store4),
            not(Store3=Store4),
            not(Shoe1=Shoe2),
            not(Shoe1=Shoe3),
            not(Shoe1=Shoe4),
            not(Shoe2=Shoe3),
            not(Shoe2=Shoe4),
            not(Shoe3=Shoe4),
            member([ff,hhs], S),
```

```
              not(right([pp,_], [_,ts], S)),
              S=[_,[_,ffs],_,_], S=[[_,tsps],_,_,[ss,_]]"]
--}


consultString :: String -> Either ParseError Program
consultString = parse (whitespace >> program <* eof) "(input)"
{--
consultString "hello."
Right ["hello"]
--}

parseQuery :: String -> Either ParseError [Term]
parseQuery = parse (whitespace >> terms <* eof) "(query)"
{--
parseQuery "hello(X)"
Right [hello(X)]
--}

program :: ParsecT String () Data.Functor.Identity.Identity [Clause]
program = many (clause <* char '.' <* whitespace)

whitespace :: ParsecT String () Data.Functor.Identity.Identity ()
whitespace = skipMany (comment <|> skip space <?> "")

comment :: ParsecT String () Data.Functor.Identity.Identity ()
comment = skip $ choice
   [ string "/*" >> (manyTill anyChar $ try $ string "*/")
   , char '%' >> (manyTill anyChar $ try $ skip newline <|> eof)
   ]

skip :: ParsecT String () Data.Functor.Identity.Identity a ->
                        ParsecT String () Data.Functor.Identity.Identity ()
skip = (>> return ())

clause :: ParsecT String () Data.Functor.Identity.Identity Clause
clause = do t <- struct <* whitespace
            dcg t <|> normal t
   where
      normal t = do
            ts <- option [] $ do string ":-" <* whitespace
```

```
                                 terms
            return (Clause t ts)

      dcg t = do
            string "-->" <* whitespace
            ts <- terms
            return (translate (t,ts))

      translate ((Struct a ts), rhs) =
         let lhs' = Struct a (arguments ts (head vars) (last vars))
             vars = map (var.("d_"++).(a++).show) [0..length rhs]
                                     -- We explicitly choose otherwise inva
             rhs' = zipWith3 translate' rhs vars (tail vars)
         in Clause lhs' rhs'

      translate' t s s0 | isList t   = Struct "="
                      [ s, foldr_pl cons s0 t ]     -- Terminal
      translate' t@(Struct "{}" ts) s s0 =
                      foldr and (Struct "=" [ s, s0 ]) ts  -- Braced terms
      translate' (Struct a ts)  s s0 =
                       Struct a (arguments ts s s0)             -- Non-Term

      and x y = Struct "," [x,y]


isList :: Term -> Bool
isList (Struct "." [_,_]) = True
isList (Struct "[]" [])   = True
isList _                  = False

terms :: ParsecT String () Identity [Term]
terms = sepBy1 termWithoutConjunction (charWs ',')

term :: ParsecT String () Identity Term
term = term' False

termWithoutConjunction :: ParsecT String () Identity Term
termWithoutConjunction = term' True

term' :: Bool -> ParsecT String () Identity Term
term' ignoreConjunction = buildExpressionParser (reverse $ map (map toParse
```

```
                         hierarchy ignoreConjunction) (bottom <* whitespace)

bottom :: ParsecT String () Identity Term
bottom = variable
      <|> struct
      <|> list
      <|> stringLiteral
      <|> cut <$ char '!'
      <|> Struct "{}" <$> between (charWs '{') (char '}') terms
      <|> between (charWs '(') (char ')') term

toParser :: Syntax.Operator -> Parsec.Operator String u Identity Term
toParser (PrefixOp name)      = Prefix (reservedOp name >>
                        return (\t -> Struct name [t]))
toParser (InfixOp assoc name) = Infix  (reservedOp name >>
                                        return (\t1 t2 -> Struct name [t1, t
                                       (case assoc of AssocLeft  -> Parsec.A
                                                      AssocRight -> Parsec.A

reservedOp :: String -> ParsecT String u Identity ()
reservedOp = P.reservedOp $ P.makeTokenParser $ emptyDef
   { P.opStart = oneOf ";,<=>\\i*+m@"
   , P.opLetter = oneOf "=.:<+"
   , P.reservedOpNames = operatorNames
   , P.caseSensitive = True
   }

charWs :: Char -> ParsecT String () Identity Char
charWs c = char c <* whitespace

operatorNames :: [[Char]]
operatorNames = [ ";", ",", "<", "=..", "=:=", "=<", "=", ">=", ">", "\\=",
                                "is", "*", "+", "-", "\\", "mod", "div", "\

variable :: ParsecT String u Identity Term
variable = (Wildcard <$ try (char '_' <*
                notFollowedBy (alphaNum <|> char '_')))
       <|> Var <$> vname
       <?> "variable"

vname :: ParsecT String u Identity VariableName
```

Floating Point Parser continued on next page...

```haskell
vname = VariableName 0 <$> ((:) <$> upper    <*> many
                        (alphaNum <|> char '_') <|>
                            (:) <$> char '_' <*> many1 (alphaNum <|> char '_

atom :: ParsecT String u Identity [Char]
atom = (:) <$> lower <*> many (alphaNum <|> char '_')
   <|> Experiment.float
   <|> many1 digit
   <|> choice (map string operatorNames)
   <|> many1 (oneOf "#$&*+/.<=>\\^~")
   <|> between (char '\'') (char '\'') (many (noneOf "'"))
   <?> "atom"

struct :: ParsecT String () Identity Term
struct = do a <- atom
            ts <- option [] $ between (charWs '(') (char ')') terms
            return (Struct a ts)

list :: ParsecT String () Identity Term
list = between (charWs '[') (char ']') $
         flip (foldr cons) <$> option []  terms
                            <*> option nil (charWs '|' >> term)

stringLiteral :: ParsecT String u Identity Term
stringLiteral = foldr cons nil . map representChar <$> between (char '"') (c
                        (try (many (noneOf "\"")))

representChar :: Enum a => a -> Term
representChar c = Struct (show (fromEnum c)) []
                -- This is the classical Prolog representation of chars as c
--representChar c = Struct [c] []
                -- This is the more natural representation as one-character
--representChar c = Struct "char" [Struct (show (fromEnum c)) []]
                -- This is a representation as tagged code points.
--toChar :: Term -> Maybe Char
--toChar (Struct "char" [Struct (toEnum . read->c) []]) = Just c
--toChar _                                               = Nothing
```

So that is it for now.

Unification Monad

Coming soon . . . . . .

Narrowing in Curry

This document has been written from various sources such as reports, manuals and research papers from the Curry programming language and implementing small working examples.

1. Functional Programming == Equational Definition of Functions.
2. Logic Programming == First Order Predicate Logic.
3. Therefore,Functional Logic Programming == Equational Logic Programming (resolution + replacement of subterms).
4. Narrowing is a feature found in high level declarative languages like CURRY and TOY.
5. Enables function like evaluation of expressions containing containing uninstantiated variables.
   Narrowing == Glue.
6. Residuation is somewhat like an **alternative** for Narrowing, provided by languages such as ESCHER and LIFE. Both features can co-exist.
7. If Narrowing is not executed it means that the computation is functional and almost as efficient as a functional interpreter.
8. Narrowing is used for solving sets of equations possibly involving user-defined data types. ??????????????
9.

Rewriting

1. It is a special case of Narrowing.
2. Rewriting rules transform expressions / terms.
3. A set of such rules is called a Rewrite System.
4. If the no.of expressions are infinite then it contains variables. If there is a single occurrence then the variable is just an anonymous place holder and is represented by "_".
5. An expression has to match with the L.H.S of the rule for finding out what each variable stands for them to be equal.

6. For example,
   $top(S) \rightarrow E$
   $empty \rightarrow EmptyStack$
   $push(E, S) \rightarrow S$
   $pop(S) \rightarrow S$

An expression,
top(push(1,empty)) $\rightarrow$ 1
$pop(push(1, empty)) \rightarrow empty$

7. *push, empty* construct stack instances hence they are "data constructors" while *pop, top* operate on stack instances hence they are "defined operations". A rule defining an operation shows how to rewrite an expression with constructors and variables. This is called **Constructor Discipline**.

8. Expressions with constructors only abstract data and evaluate to themselves.

9. Rewrite system specifies what are the steps but not when and where to perform them, hence a strategy is required for the same.

Narrowing

1. Problems arise when expressions to evaluate contain variables, this is when Narrowing comes into play.

2. A variable in a rule stands for **any** value while a variable in an expression stands for some value. ???????????????????

3. Narrowing guesses some value for some variable then does the replacing in the context of t and rewrites the instantiation of t as usual.

4. An instantiation of an expression t is found by "unifying" t with the left hand side of a rule. Unification is the process of finding what each variable in t and l stands for them to be equal. The substitution is called a unifier.

5. The narrowing space of an expression t is the set of all expressions obtained in zero or more narrowing steps from t.

6. Finding the values of the variables for both sides to be equal is known as equation solving which can result in no solutions or many solutions.

7. **Soundness** means that any instantiation of the variables of an equation computed while narrowing the sides of the equation to a same datum is a solution of the equation.

8. **Completeness** means that if an equation has a solution, narrowing will find that solution, or a more general one, while narrowing the sides of the equation to a same datum.

9. Operations that are allowed to narrow their arguments are called **flexible**.

10. Operations that are not allowed to narrow their arguments are called **rigid**.

11. The expressions used in conditions are called **constraints**.

12. **Left Linear** each variable in L.H.S occurs only once. The following is not left-linear,

```
    member x (x:_) = True
```

13. **Overlapping** means that more than one rule may reqwrite the same expression, helps with non-determinism.

```
    insert x y = x:y
    insert x (y:ys) = y:insert x ys
{--
insert x [1..5] where x free
[1 of 3] Skipping  Prelude ( /home/mehul/kics2-0.3.1/lib/Prelude.curry,
/home/mehul/kics2-0.3.1/lib/.curry/Prelude.fcy )
[2 of 3] Skipping  Program1 ( Program1.curry, .curry/Program1.fcy )
[3 of 3] Compiling Curry_Main_Goal
(Curry_Main_Goal.curry, .curry/Curry_Main_Goal.fcy )
Evaluating expression: insert x [1..5] where x free
 {x = _x5} [_x5,1,2,3,4,5]
{x = _x5} [1,_x5,2,3,4,5]
{x = _x5} [1,2,_x5,3,4,5]
{x = _x5} [1,2,3,_x5,4,5]
{x = _x5} [1,2,3,4,_x5,5]
{x = _x5} [1,2,3,4,5,_x5]
--}
```

14. **Call time choice semantics / sharing** the operands of an operator are the same for example,

```
double x = x + x
```

15.

Curry

1. Curry Program == Rewrite System with Constructor Discipline.
2. Curry is lazy like Haskell.
3. Two compilers for Curry.
   KiCS == Curry to Haskell.
   PAKCS == Curry to Prolog.
4. Two types of equality, **Constrained equality (=:=)** and **Boolean equality(==)**.
   The test is an expression of type **Success**.
5. **Constrained equality (=:=)** is used to solve an equation and is flexible.
6. **Boolean equality(==)** is used to check whether an equation holds and is rigid.
7. A boolean test t is a short hand for t =:= True.
8. **Extra variables** are the variables that occur on the R.H.S and/or in the condition of a rule but not on the L.H.S.
9. Curry supports higher order functions but **without** higher order narrowing.

Narrowing in Curry continued on next page...

david-0.2.0.1

Syntax.hs
PrettyPrint.hs
Variables.hs
Unflatten.hs
Parser.hs
Database.hs

Tutorial

Adding variable search strategies to solve a query in PROLOG for HUGS 98.

The following are the files in the library categorized according to their purpose,

- Engines
  1. Andorra Engine
  2. Pure Prolog Engine
  3. Stack Engine
- Functions for Interactive programs
  1. AnsiInteract
  2. AnsiScreen
  3. Interact
- Data Types and Utilities
  1. Subst (Also has unification)
  2. Number
  3. Prolog
  4. ListUtils
- Parsing and Hugs libraries
  1. ParseLib
  2. CombParse
  3. StdLibs
  4. HugLibs
- Main

Talking about the three engines in the library, each of them exports a function

```
prove    :: Database -> [Term] -> [Subst]
```

Each engine has its own set of data structures and helper functions.

1. Pure Engine

   This is how the Prolog works by default. It works using Proof Trees,

```
data Prooftree = Done Subst   |   Choice [Prooftree]
```

A tree can have two types of nodes,

(a) `Done s`

which represents a solution to the current goal.

(b) `Choice [Prooftree]`

which represnts a list of subtrees for proof of each subgoals

Problem 6

Things to do

- Prolog for Hugs 98 (Variable search strategy)
    1. Embed a search strategy.
    2. Try and print `Subst` datatype to get an idea about the `Prooftree`.
    3. Implement a program to distinguish between the search strategies.
- unification-fd
    1. Replace unificaion in `prolog-0.2.0.1` using the library.
- Do something about IO
- Do something about Quasi quotation.