EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programing

Sommersemester 2014, Wintersemester 2015

Author:
Philipp Moers,
Update by: Denis Hirn

Dozent:
Torsten Grust,
Alexander Ulrich

Last updated: Thursday 28th January, 2016, 11:59

Abstract

This is just the product of me taking notes on the lecture. Nothing official. If you find mistakes or have got any questions, please feel free to contact me. Cheers!

Contents

»A programming language is a medium for expressing ideas (not to get a computer to perform operations) and only incidentally for machines to execute.«

Harold Abelson and Gerald Jay Sussman

Links

Site 2014: `http://db.inf.uni-tuebingen.de/teaching/FunctionalProgrammingSS2014.html`
Site 2015: `http://db.inf.uni-tuebingen.de/teaching/FunctionalProgrammingWS2015-2016.html` Ilias: `http://goo.gl/rlqbkK`

Literature

- Bird:
  Thinking Functionally with Haskell, Cambridge University Press 2014
  `http://www.cs.ox.ac.uk/publications/books/functional/`
- Keller, Chakravarthy: "Learning Haskell", online course in development
  `http://learn.hfm.io/`
- Lipovača:
  Learn You a Haskell for Great Good
  No Starch Press 2011,
  `http://learnyouahaskell.com`
- O'Sullivan, Steward, Goerzen:
  Real World Haskell
  O'Reilly 2010
  `http://book.realworldhaskell.org`
- Haskell 2010 Report,
  `http://www.haskell.org/onlinereport/haskell2010`

1 Introduction

Computational model in Functional Programming: **reduction** (replace expression to values)
In Functional Programming, expressions are formed by applying functions to values.

1. Functions as in math: $x = y \Rightarrow f(x) = f(y)$
2. Functions are values (just like numbers, text ...)

	Functional	Imperative
program construction	function application and composition	statement sequencing
execution	reduction (expression evaluation)	state changes
semantics	lambda calculus	complex (denotational)

Example

$n \in \mathbb{N}, n \geq 2$ is a prime number *if* the set of non-trivial factors is empty:

$$n \text{ is prime} \Leftrightarrow \{\ m \mid m \in \{2, \ldots, n-1\},\ n \bmod m = 0\} = \emptyset$$

```haskell
-- Is n a prime number?
isPrime :: Integer -> Bool
isPrime n = factors n == []
  where
    factors :: Integer -> [Integer]
    factors n = [ m | m <- [2..n-1], mod n m == 0 ]


main :: IO ()
main = do
  let n = 43
  print (isPrime n)
```

2 Haskell Ramp-Up

(Read ≡ as ”denotes the same value as”)

- Apply f to value e: `f e` (juxtaposition, ”apply”, binary operator ␣, Haskell speak: infixL 10 ␣)
- ␣ has max precedence (10): `f` e_1 `+` e_2 ≡ `(f` e_1`)` `+` e_2
- ␣ associates to the left: `g f e` ≡ `(g f) e` *--(g f) is a function)*
- Function composition:
  - `(g . f) e` ≡ `g (f e)` *--(. is something like mathematical |∘| ''after'')*
  - Alternative ”apply”-operator `$` (lowest precedence, associates to the right, infixR 0 $):
    `g $ f $ e` ≡ `g $ (f $ e)` ≡ `g (f e)`
  - Prefix application of binary infix operator ⊗: (⊗) e_1 e_2 ≡ e_1 ⊗ e_2
  - Infix application of binary function f: e_1 `` `f` `` e_2 ≡ `f` e_1 e_2:
    - `` 1 `elem` [1,2,3] `` --$(1 \in \{1, 2, 3\})$
    - `` n `mod` m ``
    - ...
  - User defined operators, built from symbols
    ! # $ % & * + / ¡ = ¿ ? \ˆ |∼:.

2.1 Function Application

Any series of identifiers is a function call or, as we often call it, a function application.
`a b c d`
This is an application of a function a to three arguments b, c and d.
You may parenthesize function application if you need to.
`f a b` ≡ `(f a b)` $\not\equiv$ `f (a, b)`
The last one is valid Haskell, but f is a function that takes a pair, `(a, b)` as an argument.

3 Values and Types

Any Haskell expression e has a type t (`e :: t`) that is determined at compile time. The **type assignment ::** is either given explicitly or inferred by the compiler.

3.1 Base Types

Type	Description	values
Int	fixed-prec. integer	0, 1, (-42)
Integer	arbitrary prec. integer	10ˆ100
Float, Double	single/double floating point (IEEE)	0.1, 1e02
Char	Unicode character	“x”, “\t”, “△”, “\8710”
Bool	Boolean	True, False
()	Unit	()

3.2 Type Constructors

- Build new types from existing types
- Let a, b . . . denote arbitrary types (**type variables**)

Type	Description	values
`(a, b)`	pairs of values of type a, b	`(1, True) :: (Int, Bool)`
`(`a_1`, `a_2`, ... `a_n`)`	n-tuples	
`[a]`	list of values of type a	`[True, False] :: [Bool], []::[a]`

`Maybe a`	optional value of type a	`Just 42 :: Maybe Int`
		 `Nothing :: Maybe a`

`Either a b`	choice	`Left 'x' :: Either Char b`
		 `Right pi :: Either a Double`

`IO a`	I/O actions that return a value of type a	`print 42 :: IO ()`
`a -> b`	functions from a to b	`isLetter :: Char -> Bool`

3.3 Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

- *Recall*: e_1 `++` e_2 $\equiv$ `(++)` e_1 e_2
- `(++)` e_1 e_2 $\equiv$ `((++)` e_1`)` e_2
- Function application happens one argument at a time.

(**Currying**, Haskell B. Curry)

- Type of n-ary function is a_1 `->` a_2 `->` `...` a_n `->` `b`
- Type fun `->` associates to the right,

read above type as
a_1 `->` (a_2 `->` (... (a_n `->` b)))

- Enables **Partial Application**

3.4 Defining Values (and thus functions)

- `=` binds names to values. Names must not start with A-Z (Haskell style: camelCase)
- Define constant (0-ary function) c. Value of c is value of expression e.
  `c = e`
- Define n-ary function f with arguments x_i. f may occur in e.
  `f` x_1 x_2 ... x_n `= e`
- A Haskell program is a set of bindings.
- Good style: give type assignments for top-level (global) bindings:

```
f :: a₁ -> a₂ -> b
f x₁ x₂ = e
```

3.4.1 Guards

Guards are conditional expressions (something like "switch" in Java). They are a lot more readable and more powerful than `if ... then ... else ...` .

Guards are introduced by `|`:

```
f x₁ x₂ ... xₙ
  | q₁      = e₁
  | q₂      = e₂
  ...
  | qₘ       = eₘ
[ | otherwise    = eₘ₊₁ ]
```

Guards (q_i) are expressions of type Bool, evaluated top to bottom.

```
-- Compute n!
fac :: Integer -> Integer
fac n | n <= 1     = 1
      | otherwise = n * fac (n - 1)

main :: IO ()
main = print $ fac 10
```

3.4.2 Local Definitions

1. **Where bindings**: local definitions visible in the entire rhs of a definition.

```
f₁ x₁ x₂ ... xₙ | q₁ = e₁
                 | q₂ = e₂
                 ...
                 | qₘ = eₘ
    where
        g₁ = ...
        g₂ = ...
        ...
        gₒ
```

```
-- Efficient power computation, basic idea: x^2k = (x^2)^k

power :: Double -> Integer -> Double
power x k | k == 1     = x
          | even k     = power (x * x) (halve k)
          | otherwise = x * power (x * x) (halve k)
  where
    even n  = n `mod` 2 == 0
    halve n = n `div` 2

main :: IO ()
main = print $ power 2 16
```

2. **Let expressions**: local definitions visible inside one expression.

```
let g₁ = ...
    g₂ = ...
    ...
    gₒ
in e
```

3.4.3 Lists

- Recursive definitions:
  1. `[]` is a list (nil), type `[] :: [a]`
  2. `x:xs` is a list, if `x :: a, xs :: [a]`
     (x is head, xs is tail)
- Notation: `3:(2:(1:[]))` ≡ `3:2:1:[]` ≡ `[3,2,1]` ≡ `3:[2,1]`
- Law: $\forall$ xs :: [a] : (xs $\neq$ [])
  `head xs : tail xs` == xs

3.4.4 Pattern Matching

- *The* idiomatic Haskell way to define a function by cases:

```
f :: a₁ -> ... aₙ -> b
f p₁1 ... p₁k = e₁
f p₂1 ... p₂k = e₂
...
f pₙ1 ... pₙk = eₖ
```

Pattern	Matches If	Bindings in e_r
constant c	x_i == c	
variable v	always	$v \equiv x_i$
wildcard _	always	
tuple $(p_1, \ldots p_m)$	components of x_i match patterns p	
[]	x_i == []	
$(p_1 : p_2)$	head x_i matches p_1, tail x_i matches p_2	

```haskell
-- Equivalent definitions of sum (over lists of integers)

-- (1) Conditional expression
sum' :: [Integer] -> Integer
sum' xs = if xs == [] then 0 else head xs + sum' (tail xs)

-- (2) Guards
sum'' :: [Integer] -> Integer
sum'' xs | xs == []   = 0
         | otherwise = head xs + sum'' (tail xs)

-- (3) Pattern matching
sum''' :: [Integer] -> Integer
sum''' []     = 0
sum''' (x:xs) = x + sum''' xs

main :: IO ()
main = print $ (sum' [1..100], sum'' [1..100], sum''' [1..100])
```

```haskell
-- Finite prefix of a list
take' :: Integer -> [a] -> [a]
take' 0 _      = []
take' _ []     = []
take' n (x:xs) = x:take' (n-1) xs

main :: IO ()
main = print $ take' 20 [1,3..]
```

```haskell
-- Mergesort list xs, respecting ordering (<<<)

mergeSort :: (a -> a -> Bool) -> [a] -> [a]
mergeSort _     []   = []
mergeSort _     [x]  = [x]
mergeSort (<<<) xs   = merge (<<<) (mergeSort (<<<) ls)
                                   (mergeSort (<<<) rs)
  where
    (ls, rs) = splitAt (length xs `div` 2) xs

    merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
    merge (<<<) []     ys         = ys
    merge (<<<) xs     []         = xs
    merge (<<<) l1(x:xs) l2(y:ys)
      | x <<< y    = x:merge (<<<) xs l2
      | otherwise = y:merge (<<<) l1 ys

main :: IO ()
main = print $ mergeSort (>) [1,3..19]
```

3.5 Algebraic Data Types

(also known as **Sum-of-Product-Types**)

- *Recall*: `[]` and `(:)` are the **values constructors** for **type constructor** [a].
- Can define entirely new type T and its constructors K_i:

  $$
  \begin{array}{l}
  \texttt{data T } a_1\ a_2\ \ldots\ a_n = K_1\ b_{11}\ \ldots\ b_{1n_1} \\
  \qquad\qquad\qquad\qquad\quad K_2\ b_{21}\ \ldots\ b_{2n_2} \\
  \qquad\qquad\qquad\qquad\quad \ldots \\
  \qquad\qquad\qquad\qquad\quad K_r\ b_{r1}\ \ldots\ b_{rn_r}
  \end{array}
  $$

  b_{ij} types mentioning the type vars $a_1 \ldots a_n$

- Defines type constructor T and r value constructors:
  $K_i :: b_{i_1} \to b_{i_2} \to \ldots\ b_{i_n} \to T\ a_1 \ldots\ a_n$
- Compare `[] :: [a]` and `(:) :: a -> [a] -> [a]`
- **Sum Type** (n=0, $n_i = 0$)

```haskell
-- Demonstrate algebraic data types: sum type

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Show, Ord, Enum, Bounded)

-- Is this day on a weekend?
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False


-- The classic rock/paper/scissor game

data Move = Rock | Paper | Scissor
  deriving (Eq)

data Outcome = Lose | Tie | Win
  deriving (Show)

-- Outcome of a game round (us vs. them)
outcome :: Move -> Move -> Outcome
outcome Rock    Scissor = Win
outcome Paper   Rock    = Win
outcome Scissor Paper   = Win
outcome us      them
  | us == them = Tie
  | otherwise  = Lose


main :: IO ()
main = do
  print $ Mon == Sun
  print $ Thu < Sat
  print [Mon .. Fri]
  print (minBound :: Weekday)
  print $ outcome Rock Rock
```

- Add `deriving (c, c, ... c)` to data declaration to define canonical operations:

c	operations
Eq	equality (==, /=)
Show	printing (show)
Ord	ordering (<, <=, max)
Enum	enumeration
Bounded	minBound, maxBound

- **Product Types** (r=1)

```haskell
-- Demonstrate algebraic data types: product type

data Sequence a = S Int [a]
  deriving (Eq, Show)


fromList :: [a] -> Sequence a
fromList xs = S (length xs) xs

(+++) :: Sequence a -> Sequence a -> Sequence a
S lx xs +++ S ly ys = S (lx + ly) (xs ++ ys)

len :: Sequence a -> Int
len (S lx _) = lx


main :: IO ()
main = do
  print $ fromList [0..9]
  print $ len (fromList ['a'..'m'] +++ fromList ['n'..'z'])
```

- **Sum-of-Product-Types**

```haskell
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
data List a = Nil | Cons a (List a)
```

```haskell
-- Our own formulation of cons lists
data List a = Nil
            | Cons a (List a)
  deriving (Show)

-- Haskell's builtin type [a] and List a are isomorphic:
--         toList . fromList = id
--   and   fromList . toList = id
toList :: [a] -> List a
toList []     = Nil
toList (x:xs) = Cons x (toList xs)

fromList :: List a -> [a]
fromList Nil         = []
fromList (Cons x xs) = x:fromList xs

-- The family of well-known list functions (combinators) can be
-- reformulated for List a
mapList :: (a -> b) -> List a -> List b
mapList f Nil         = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)

filterList :: (a -> Bool) -> List a -> List a
filterList p Nil                     = Nil
filterList p (Cons x xs) | p x       = Cons x (filterList p xs)
                         | otherwise = filterList p xs

liftList :: ([a] -> [b]) -> List a -> List b
liftList f = toList . f . fromList

mapList' :: (a -> b) -> List a -> List b
mapList' f = liftList (map f)

filterList' :: (a -> Bool) -> List a -> List a
filterList' p = liftList (filter p)

main :: IO ()
main = print $ fromList $ filterList' (> 3) $ mapList' (+1) $ toList
 ↪  [1..5]
```

```haskell
-- Use the isomorphism between [a] and List a
-- to save work when defining functions over List a:
--
--                    fromList
--         List a -------------→ [a]
--           |                    |
--         g |                    | f
--           ↓                    ↓
--         List b ←------------- [b]
```

```haskell
-- Abstract syntax tree for arithmetic expressions of literals
data Exp a = Lit a
           | Add (Exp a) (Exp a)
           | Sub (Exp a) (Exp a)
           | Mul (Exp a) (Exp a)
  deriving (Show)

ex1 :: Exp Integer
ex1 = Add (Mul (Lit 5) (Lit 8)) (Lit 2)


evaluate :: Num a => Exp a -> a
evaluate (Lit n)      = n
evaluate (Add e1 e2) = evaluate e1 + evaluate e2
evaluate (Mul e1 e2) = evaluate e1 * evaluate e2
evaluate (Sub e1 e2) = evaluate e1 - evaluate e2


main :: IO ()
main = print $ evaluate ex1
```

4 Type Classes

A **type class** C defines a family of type signatures ("methods") which all **instances** of C must implement.

```
class C a where
    f₁ :: t₁
    ...
    fₙ :: tₙ
```

The t_i must mention a.
For any f_i the class may provide default implementations.
We have `fᵢ :: C a => tᵢ`
(read "if a is instance C then f_i has type t_i").
`C a` is called **class constraint**.

Example:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

(These are default implementations. To redefine one of them is sufficient.)

4.1 Class Inheritance

- Defining `class (c₁ a, c₂ a, ... ) => C a where ...` makes type class C a subclass of the C_i.
- `C a => t` implies `C₁ a, C₂ a ...` .

4.2 Class Instances

If type t implements the methods of class C, t becomes an **instance of** C:

```
instance C t where
    f₁ = <def of f₁>
    ...
    fₙ = <def of fₙ>
```

(All defs of f_i may be provided, minimal complete definition must be provided.) Class constraint C t is satisfied from now on.

Example:

```
instance Eq Bool where
    x == y = x && y || (not x && not y)
```

An instance definition for type constructor t may formulate class constraints for its argument types a, b, . . .: `instance (C₁ a, C₂ a, ... ) => C t where`

```haskell
import Data.Maybe
import Data.Tuple
-- The classic rock/paper/scissor game
data Outcome = Lose | Tie | Win

instance Eq Outcome where
  Lose == Lose = True
  Tie  == Tie  = True
  Win  == Win  = True
  _    == _    = False

instance Enum Outcome where
  fromEnum Lose = 0
  fromEnum Tie  = 1
  fromEnum Win  = 2

  toEnum 0 = Lose
  toEnum 1 = Tie
  toEnum 2 = Win

instance Show Outcome where
  show Lose = "Lose"
  show Tie  = "Tie"
  show Win  = "Win"

instance Ord Outcome where
  Lose <= Lose = True
  Lose <= Tie  = True
  Lose <= Win  = True
  Tie  <= Tie  = True
  Tie  <= Win  = True
  Win  <= Win  = True
  _    <= _    = False

instance Bounded Outcome where
  minBound = Lose
  maxBound = Win
----------------------------------------------------------------
data Move = Rock | Paper | Scissor

instance Eq Move where
  Rock    == Rock    = True
  Paper    == Paper    = True
  Scissor  == Scissor  = True
  _        == _        = False

-- Lookup table defining a consistent mapping between Move and Int
table :: [(Move, Int)]
table = [(Rock, 0), (Paper, 1), (Scissor, 2)]

instance Enum Move where
  fromEnum o = fromJust $ lookup o table
  toEnum n   = fromJust $ lookup n $ map swap table

instance Show Move where
  show Rock    = "Rock"
  show Paper   = "Paper"
  show Scissor = "Scissor"

instance Ord Move where
  Rock     <= Rock     = True
  Rock     <= Paper    = True
  Rock     <= Scissor  = True
  Paper    <= Paper    = True
  Paper    <= Scissor  = True
  Scissor  <= Scissor  = True
  _        <= _        = False

instance Bounded Move where
  minBound = Rock
  maxBound = Scissor
--------------------------------------------------------------
outcome :: Move -> Move -> Outcome
outcome Rock    Scissor = Win
outcome Paper   Rock    = Win
outcome Scissor Paper   = Win
outcome us      them
  | us == them = Tie
  | otherwise  = Lose

main :: IO ()
main = print $ outcome Paper Paper
```

4.2.1 Deriving Class Instances

Automatically make user-defined data types (`data ...`) instances of classes $C_i \in \{$ Eq, Ord, Enum, Bounded, Show, Read $\}$:

```
data T a₁ a₁ ... aₙ = ...
                    | ...
    deriving (C₁, C₂, ...)
```

```
-- Use deriving the obtain the standard interpretation for the type
↪  classes
-- Eq, Ord, Enum, Bounded, Show, Read
-- The classic rock/paper/scissor game

data Outcome = Lose | Tie | Win
  deriving (Eq, Ord, Enum, Bounded, Show, Read)

data Move = Rock | Paper | Scissor
  deriving (Eq, Enum, Read)    -- Ord, Show defined below;
                               -- Bounded makes no sense
instance Show Move where
  show Rock    = "rock"
  show Paper   = "paper"
  show Scissor = "scissor"

instance Ord Move where           -- NB: non-conventional,
  Rock    <= Rock    = True       -- encodes game rules
  Rock    <= Paper   = True
  Paper   <= Paper   = True
  Paper   <= Scissor = True
  Scissor <= Scissor = True
  Scissor <= Rock    = True
  _       <= _       = False

outcome :: Move -> Move -> Outcome
outcome m1 m2 | m1 == m2   = Tie
              | m1 <  m2   = Lose
              | otherwise = Win

main :: IO ()
main = do
  print $ outcome Paper Scissor
  print $ [Rock, Paper, Scissor]
  print $ (read "Scissor" :: Move)
```

5 Domain-Specific Languages

a.k.a. **DSLs**

- ”small” languages designed to easily and directly express the concepts/idioms of a specific domain. Not Turing-complete in general.
- Examples:

Domain	DSLs
OS automation	shell scripts, OSX Automater
Typesetting	LaTeX
Queries	SQL
Game Scripting	Unreal Script, Lua
Parsing	Yacc, Bison, ANTLR

- Functional Languages make good hosts for **embedded DSLs**:
  - algebraic data types (e.g. to model ASTs)
  - higher-order functions (abstraction, control constructs)
  - lightweight syntax (layout / whitespace, non-alphabetic ids)

Example: An embedded DSL for integer sets:

```
type IntegerSet =
    -- constructors:
    empty :: IntegerSet
    insert :: Integer -> IntegerSet -> IntegerSet
    delete :: Integer -> IntegerSet -> IntegerSet
    -- observer:
    member :: Integer -> IntegerSet -> Bool

member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
    ≡ False
```

(1) DSL as library of functions, implementation details exposed.

```haskell
import Data.List (nub)
-- A library of functions on integer sets,
-- implementation fully exposed

type IntegerSet = [Integer] -- unsorted, duplicates allowed

empty :: IntegerSet
empty = []

insert :: Integer -> IntegerSet -> IntegerSet
insert x xs = x:xs

delete :: Integer -> IntegerSet -> IntegerSet
delete x xs = filter (/= x) xs

(∈) :: Integer -> IntegerSet -> Bool
x ∈ xs = elem x xs

-------------------------------------------------------------------------

-- "Extending" the library, accessing the exposed
-- implementation.  Now we're doomed to stick the
-- list-based representation...

(⊆) :: IntegerSet -> IntegerSet -> Bool
xs ⊆ ys = all (\x -> x ∈ ys) xs

card :: IntegerSet -> Int
card xs = length (nub xs)

-------------------------------------------------------------------------

s1, s2 :: IntegerSet
s1 = insert 1 (insert 2 (insert 3 empty))
s2 = foldr insert empty [1..10]

prog :: Bool
prog = s1 ⊆ s2

main :: IO ()
main = print $ prog
```

5.1 Modules

- Group of related definitions (values, types) in a single file
  (named "M.hs" / "M.lhs"):

```haskell
module M where
    type Predicate a = a -> Bool
    id :: a -> a
    id x = x
```

- Hierarchy: module A.B.C.M in file A/B/C/M.hs
- Access definitions in other module M: `import M`

- Explicit export lists hide all other definitions:

```haskell
module M (id) where
    ...
    -- type Predicate a not exported
```

- **Abstract data types**:
  export algebraic data types, but <u>not</u> its constructors:

```haskell
module M (Rose, leaf) where
    data Rose a = Node a [Rose a]
    leaf :: a -> Rose a [Rose a]
    leaf x = Node x []
```

  – Export constructors:

```haskell
module M (Rose(Node), leaf) where
    ...
-- or export all constructors:
module M (Rose(..), leaf) where
```

  – Instance definitions (including deriving) are exported with their type.

- Qualified imports to partition name space:

```haskell
import qualified M
    ...
    -- use M.foobar syntax
    t :: M.Rose Char
    t = M.leaf 'x'
```

- Partially import module:

```haskell
-- only import nub and reverse
import Data.List (nub, reverse)

-- import whole module but without otherwise
import Prelude hiding (otherwise)
otherwise :: Bool
otherwise = False -- harhar
```

- Pass imported modules to importer of own module:

```haskell
module M (..., module Data.List, ...) where
    import Data.List (nub)
```

```haskell
import qualified M
    M.nub
```

```haskell
module SetLanguageShallowCard (IntegerSet,
                               empty,
                               insert,
                               delete,
                               member,
                               card) where

data IntegerSet = IS (Integer -> Bool) Int -- characteristic function +
 ↪ cardinality

-- constructors
empty :: IntegerSet
empty = IS (\_ -> False)
           0
-- empty = IS (const False) 0
{-
insert :: IntegerSet -> Integer -> IntegerSet
insert xs@(IS f c) x = IS (\y -> x == y || f y)
                           (if member xs x then c else c + 1)
-}
delete :: IntegerSet -> Integer -> IntegerSet
delete xs@(IS f c) x = IS (\y -> y /= x && f y)
                          (if member xs x then c - 1 else c)

-- observer
member :: IntegerSet -> Integer -> Bool
member (IS f _) x = f x
-- member (IS f _) = f

card :: IntegerSet -> Int
card (IS _ c) = c
```

```haskell
--import SetLanguage
import SetLanguageShallow
--import SetLanguageShallowCard

-- impossible
-- (SetLanguage/SetLanguageShallow do not export
--  data constructor IS)
{-

union :: IntegerSet -> IntegerSet -> IntegerSet
union (IS xs) (IS ys) = IS (xs ++ ys)

union :: IntegerSet -> IntegerSet -> IntegerSet
union (IS f) (IS g) = IS (\y -> f y || g y)

-}

set12 :: IntegerSet
set12 = (((empty `insert` 3) `insert` 2) `delete` 3) `insert` 1

main :: IO ()
main = print $ set12 `member` 3
```

(2) DSL as library of functions, abstract data type (module).

- **Shallow DSL embedding**:
  Semantics of DSL operations directly expressed in terms of host language value (e.g. list or characteristic function)
  - constructors (empty, insert, delete) perform the work, harder to add
  - observers (member) trivial
- **Deep DSL embedding**:
  DSL operations build an abstract syntax tree (AST) that represents applications and arguments
  - constructors merely build the AST, very easy to add
  - observers interpret (traverse) the AST and perform the work

```haskell
module SetLanguageDeepCard (IntegerSet(Empty, Insert, Delete),
                            member,
                            card) where

-- constructors

data IntegerSet =
    Empty
  | Insert IntegerSet Integer
  | Delete IntegerSet Integer
  deriving (Show)

-- TODO: build a suitable Show instance for IntegerSet

-- observers

member :: IntegerSet -> Integer -> Bool
member Empty          _ = False
member (Insert xs x) y = x == y || member xs y
member (Delete xs x) y = x /= y && member xs y

card :: IntegerSet -> Int
card Empty                          = 0
card (Insert xs x) | member xs x = card xs
                   | otherwise   = card xs + 1
card (Delete xs x) | member xs x = card xs - 1
                   | otherwise   = card xs
```

```haskell
import SetLanguageDeepCard

set12 :: IntegerSet
set12 = (((Empty `Insert` 3) `Insert` 2) `Delete` 3) `Insert` 1

main :: IO ()
main = do
  print $ set12
  print $ set12 `member` 3
  print $ card set12
```

```haskell
module ExprDeepNum (Expr(..),
                    eval) where

-- constructors

data Expr =
    Val Integer
  | Add Expr Expr
  | Mul Expr Expr
  | Sub Expr Expr
  deriving (Show)

instance Num Expr where
  fromInteger n = Val n
  e1 + e2 = Add e1 e2
  e1 - e2 = Sub e1 e2
  e1 * e2 = Mul e1 e2

  abs e = undefined
  signum e = undefined


-- observer

eval :: Expr -> Integer
eval (Val n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Sub e1 e2) = eval e1 - eval e2
```

```haskell
import ExprDeepNum

-- e1 = 8 * 7 - 14
e1 :: Expr
e1 = Sub (Mul (Val 8) (Val 7)) (Val 14)

e2 :: Expr
e2 = 8 * 7 - 14

main :: IO ()
main = do
  print $ eval e1
  print $ negate e2
  print $ eval e2
```

5.2 Generalized Algebraic Data Types (GADTs)

Idea:

- Encode the type of a DSL expression (here: Integer or Bool) in its **Haskell representation type**.
- Use Haskell's type checker to ensure at compile time that only well-typed DSL expressions are built.

Language extensions:

```haskell
\{-\# LANGUAGE GADTs \#-\}
```

- Define new parameterized type T, its constructors k_i and their type signatures:

```
data T a_1 a_2 ... a_n where
    k_1 :: b_11 -> ... -> b_1n_1 -> T t_11 t_1n
    ...
    k_r :: b_r1 -> ... -> b_rn_r -> T t_r1 t_rn
```

```haskell
{-# LANGUAGE GADTs #-}

module ExprDeepGADTTyped (Expr(..),
                          eval) where

-- Expr a: an expression that, if evaluated, will yield a value of type a

data Expr a where
  ValI   :: Integer                      -> Expr Integer
  ValB   :: Bool                         -> Expr Bool
  Add    :: Expr Integer -> Expr Integer  -> Expr Integer
  And    :: Expr Bool -> Expr Bool        -> Expr Bool
  EqZero :: Expr Integer                 -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a


instance Show (Expr a) where
  show (ValI n)      = show n
  show (ValB b)      = show b
  show (Add e1 e2)   = show e1 ++ " + " ++ show e2
  show (And e1 e2)   = show e1 ++ " ∧ " ++ show e2
  show (EqZero e)    = show e  ++ " == 0"
  show (If p e1 e2) = "if " ++ show p ++ " then " ++ show e1 ++ " else " ++ show e2


-- NB: this is *typed* evaluation of expressions:
eval :: Expr a -> a
eval (ValI n)     = n
eval (ValB b)     = b
eval (Add e1 e2)  = eval e1 + eval e2
eval (And e1 e2)  = eval e1 && eval e2
eval (EqZero e)   = eval e == 0
eval (If p e1 e2) = if eval p then eval e1 else eval e2
```

```haskell
import ExprDeepGADTTyped

e1 :: Expr Integer
e1 = If (EqZero (Add (ValI 0) (ValI 0))) (ValI 42) (ValI 43)

-- e2 yields compile-time type error

e2 :: Expr Bool
e2 = EqZero (ValB True)

main :: IO ()
main = do
  print $ e1
  print $ eval e1
```

5.3 Shallow embedding of a String Matching DSL

- **Pattern**:
  1. Given a String, a pattern returns the list of matches. Match failure? Returns the empty list.
  2. A match consists of a value (e.g. the matched characters, tokens, parse trees) and the residual String left to match.

  Thus: `type Pattern a = String -> [(a, String)]`

5.3.1 DSL design

Pattern		DSL function
match lit. char	”x”	`lit :: Char -> Pattern Char`
match empty string	ϵ	`empty :: a -> Pattern a`
fail always	$\emptyset$	`fail :: Pattern a`
alternative	—	`alt :: Pattern a -> Pattern a -> Pattern a`
sequence	.	`seq :: (a -> b -> c) -> Pattern a -> Pattern b -> Pattern c`
repetition	*	`rep :: Pattern a -> Pattern [a]`

```haskell
module PatternMatching (Pattern,
                        module Prelude,
                        lit, empty, fail,
                        alt, seq, rep, rep1,
                          alts, seqs, lits, app) where

import Prelude hiding (seq, fail)

-- Given a string, a pattern returns the (possibly empty) list of
-- possible matches.  A match consists of a match value (e.g., matched
-- the matched character(s), token, or parse tree) and the residual
↪  string
-- left to match:

type Pattern a = String -> [(a, String)]

-- BASIC PATTERNS

-- match character c, returning the matched character
lit ::  Char -> Pattern Char
lit _c []                  = []
lit c  (x:xs) | c == x     = [(c, xs)]
              | otherwise = []

-- match the empty string, return v
empty :: a -> Pattern a
empty v xs = [(v, xs)]

-- fail always (yields empty list of matches)
fail :: Pattern a
fail _ = []

-- COMBINE PATTERNS

-- match p or q
alt :: Pattern a -> Pattern a -> Pattern a
alt p q xs = p xs ++ q xs

-- match p and q in sequence (use f to combine match values)
seq :: (a -> b -> c) -> Pattern a -> Pattern b -> Pattern c
seq f p q xs = concat (map (\(v1, xs1) ->
                          map (\(v2, xs2) -> (f v1 v2, xs2))
                              (q xs1))
                          (p xs))

-- An alternative (more consise and readable) implementation of seq
-- based on list comprehension syntax:
--
-- seq f p q xs = [ (f v1 v2, xs2) | (v1, xs1) <- p xs, (v2, xs2) <- q
↪  xs1 ]

-- match p repeatedly (including 0 times)
rep :: Pattern a -> Pattern [a]
rep p = alt (seq (:) p (rep p)) (empty [])

-- match p repeatedly, but at least once
rep1 :: Pattern a -> Pattern [a]
rep1 p = seq (:) p (rep p)

-- CONVENIENCE

-- build "or" choice pattern from a list of alternatives
alts :: [Pattern a] -> Pattern a
alts = foldr alt fail

-- build "and" sequence pattern from a list of patterns
seqs :: [Pattern a] -> Pattern [a]
seqs = foldr (seq (:)) (empty [])

-- match a string (= sequence of characters)
lits :: String -> Pattern String
lits cs = seqs [ lit c | c <- cs ]

-- apply function f to match value (for match post-processing)
app :: (a -> b) -> Pattern a -> Pattern b
app f p xs = [ (f v1, xs1) | (v1, xs1) <- p xs ]

{-

-- Using rep leads to longest match first:

rep p xs
  = alt (seq (:) p (rep p)) (empty []) xs
  = seq (:) p (rep p) xs ++ empty [] xs
  = seq (:) p (rep p) xs ++ [([], xs)]
  = [ (v1:v2, xs2) | (v1,xs1) <- p xs, (v2,xs2) <- rep p xs1 ] ++ [([],
↪  xs)]
rep P'a' "aab"
  = [ (v1:v2, xs2) | (v1,xs1) <- P'a' "aab", (v2,xs2) <- rep P'a' xs1 ]
↪  ++ [([], "aab")]
  = [ (v1:v2, xs2) | (v1,xs1) <- [('a',"ab")], (v2,xs2) <- rep P'a' xs1
↪  ] ++ [([], "aab")]
  = [ ('a':v2,xs2) | (v2,xs2) <- rep P'a' "ab" ] ++ [([], "aab")]

-}
```

```haskell
import Prelude ()

import PatternMatching

fortytwo :: Pattern (Char,Char)
fortytwo = seq (,) (lit '4') (lit '2')

digit :: Pattern Char
digit = lit '0' `alt` lit '1' `alt` lit '2' `alt` lit '3' `alt`
        lit '4' `alt` lit '5' `alt` lit '6' `alt` lit '7' `alt`
        lit '8' `alt` lit '9'

number :: Pattern String
number = rep digit

smiley :: Pattern String
smiley = seq (:) (alt (lit ':') (lit ';'))
                 (seq (:) (lit '-')
                          (seq (:) (alt (lit '(') (lit ')'))
                                   (empty [])))

main :: IO ()
main = do
  print $ fortytwo "42 rules"
  print $ rep digit "42 rules"
  print $ smiley ";-)"
```

```haskell
import Prelude ()
import PatternMatching
-- Make use of the fact that the pattern matching DSL is *embedded*
-- into Haskell: define new functions (abstractions) that combine
-- simple patterns
-- Example:
-- Match a fully parenthesized arithmetic expression over integers,
-- e.g. ((4*10)+2)

-- Variant 1: return list of matched characters
digit :: Pattern Char
digit = alts [ lit d | d <- ['0'..'9'] ]

number :: Pattern String
number = rep1 digit

op :: Pattern String
op = alts [ lits o | o <- ["+", "-", "*", "/"] ]

expr :: Pattern String
expr = alts [ number, app concat (seqs [lits "(", expr, op, expr, lits
 ↪  ")"]) ]

-- Variant 2: return a simple AST for the matched expression
data Expr a =
    Num a
  | Op (Expr a) String (Expr a)
  deriving (Show)

number' :: Pattern (Expr Integer)
number' = app (Num . read) (rep1 digit)

expr' :: Pattern (Expr Integer)
expr' = alts [ number', seq (\_ (e1,(o,(e2,_))) -> Op e1 o e2)
                            (lit '(') (seq (,)
                                      expr' (seq (,)
                                            op (seq (,)
                                               expr' (lit ')'))))
             ]

main :: IO ()
main = do
  print $ rep1 digit "1234.56"
  print $ lits "abc" "abcdef"
  print $ expr "((4*10)+2)"
  print $ expr' "((4*10)+2)"
```

6 Lazy Evaluation

To execute a program, Haskell reduces expressions to values. Haskell uses normal order reduction to select the next expression to reduce:

1. The outermost reducible expression (redex) is reduced first
2. ⇒ Function applications are reduced first before their arguments.
3. If no further redex is found, the expression is in normal form and reduction terminates.

```
fst :: (a, b) -> a
fst (x, y) = x
sqr :: Num a => a -> a
sqr x = x * x

-- ->: reduces to
fst (sqr (1+3), sqr 2) -> sqr (1 + 3) [fst]
                       -> (1+3) * (1+3) [sqr]
                       -> 4 * 4 [+/+]
                       -> 16 [*]
```

Haskell avoids the duplication of work through graph reduction. Expression are shared (referenced more than once) instead of duplicated.
Lazy evaluation: normal order reduction and sharing.

6.1 WHNF

An expression e is in weak head normal form (WHNF) if it is of the following form

1. v (where v is an atomic value

   `Integer, Bool, Char, ...`

)
2. `c e1 e2 ... en`

   (where c is an n-ary constructor, like

   `(:)`

)
3. `f e1 e2 ... em`

   (where f is an n-ary function, $m < n$)

Haskell reduces values to WHNF only (stop criterium for reduction) unless we request reduction to normal form (e.g. when printing results).

Example expressions in WHNF:

- `42 --1.`
- `(sqr 2, sqr 4) --2. (,)`
- `f x : map f xs --2. (:)`
- `Just (40+2) --2. Just`
- `(* (40+2)) --3. * binär`
- `(\x -> 40+2) --3. unary function w/o args`

6.2 Lazy Evaluation and Bottom

Some Haskell expressions have the value bottom. Examples:

`error, undefined, bomb`

. Lazy evaluation admits functions that return a non-bottom value even if they receive bottom as argument (also: non-strict functions). N-ary function f is strict in its i-th argument, if

`f x1 .. xi-1 bottom xi+1 ... xn = bottom`

Examples:

- `const :: a -> b -> a --strict in first, non-strict in second argument`
- `(&&) :: Bool -> Bool -> Bool -- dito`

If a function pattern matches on an argument, Haskell semantics define it to be strict in that argument.
Example:

```haskell
data T = T Int
f :: T -> Int
f (T x) = 42

f undefined -> undefined
f (T undefined) -> 42
```

7 Infinite Lists (Data Structures)

One consequence of lazy evaluation: programs can handle infinite Lists as long as any run will inspect only a finite prefix of such a list. Enables a modular programming style:

1. generator functions produce an infinite number of solutions / approximations / ...
2. test functions select one (or finite number of) solutions from this infinite list.

7.1 Example: Newton-Raphson square root approximation

Iteratively approximate the square root of x:

1. $a_0 = \frac{x}{2}$
2. $a_{i+1} = \frac{a_i + \frac{x}{a_i}}{2}$

```haskell
-- Demonstrate modular program construction through laziness:
-- value generation (here: iterate) and consume/test (here: within)
-- can be implemented separately.
-- Can replace test (within → relative) without modifying the generator.
-- See John Hughes, "Why Functional Programming Matters", Section 4.1
import Prelude hiding (iterate)
-- [x, f x, f (f x), f (f (f x)), ...]
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
-- Consume list until two adjacent elements are
-- 1. within eps of each other
-- 2. differ by a factor less than eps
within :: (Ord a, Num a) => a -> [a] -> a
within eps (x1:x2:xs) | abs (x1 - x2) <= eps = x2
                      | otherwise            = within eps (x2:xs)

relative :: (Ord a, Fractional a) => a -> [a] -> a
relative eps (x1:x2:xs) | abs (x1/x2 - 1) <= eps = x2
                        | otherwise              = relative eps (x2:xs)
-- Square root of x using the Newton-Raphson algorithm:
--   a0   = x / 2
--   ai+1 = (ai + x / ai) / 2
-- Why does this work?  If the approximations ai converge to some
-- limit a, then:
--   a = (a + x / a) / 2
--  2a = a + x / a
--   a = x / a
--  a^2 = x
--   a = √x
sqroot :: Double -> Double -> Double
sqroot eps x = within eps (iterate next a0)
--             relative
  where
    -- initial approximation
    a0 :: Double
    a0 = x / 2
    -- find next ai+1, given ai
    next :: Double -> Double
    next a = (a + x / a) / 2

main :: IO ()
main = print $ sqroot 0.001 81
```

7.2 Example: Tic-Tac-Toe game tree

Build the (potentially huge) tree of possible moves for the Tic-Tac-Toe board game. Evaluate promise of game position. Plan:

1. Find representation of game position (board + player next up)

   ```
   |1|2|3| next: x
   |4|5|6| square #6: open
   |0|x|0| spare #9: occupied by player 0
   ```

2. Provide pretty-printing for game positions.

3. Define initial position and possible moves:

   ```
   moves :: Position -> [Position]
   ```

4. Evaluate a given position:

   ```
   static :: Position -> Int
   ```

   (1 / -1: x/0 won the game, 0 draw)

5. Build a game tree of positions:

   ```
   gameTree :: Position -> Tree Position
   ```

6. Rather than simple static evaluation, now evaluate positions based on possible game futures. ⇒ in game tree, perform evaluation bottom up.

7. Optimization ($\alpha - \beta$ algorithm)

Code: tic-tac-toe.hs

8 Functors

Type class Functor embodies the application of a function to the elements (or: inside) of a structure, which leaving structure (or: outside) alone.

8.1 Examples

```haskell
map :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b

--Note: f is a type constructor that receives exactly one argument
--(Functor is also called a constructor class).
class Functor f where
   fmap :: (a -> b) -> f a -> f b

--Examples:
instance Functor [] where
   fmap = map

instance Functor Tree where
   fmap = mapTree

instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f Nothing  = Nothing
```

Type constructors can be partially applied. Uses prefix notation:

```
a -> b ≡ (->) a b
(a, b) ≡ (,) a b
[a] ≡ [] a

--Examples (defines type constructors):
type Flagged = (,) Bool
type Indexed (->) Int
--MayFail e a: computation may yield value a or fail with error e
type MayFail e = Either e

instance Functor (Either e) where
   fmap f (Left e)  = Left e
   fmap f (Right x) = Right (f x)

instance Functor Flagged where
--fmap :: (a -> b) -> (Bool, a) -> (Bool, b)
   fmap f (b, x) = (b, f x)

instance Functor Indexed where
   fmap f g = f . g

--Functor Laws
fmap id ≡ id
fmap (f . g) = fmap f . fmap g
```

8.2 Kinds - Types for Types

kind	describes...	example
`*`	types	`Int, Bool, (Bool, Int), [Char], ...`
`* -> *`	unary type constructor	`Maybe, []`
`* -> * -> *`	binary type constructor	`Either, (,), (->)`

9 Applicative

```
--Compare:
($) :: (a -> b) -> a -> b
fmap :: Functor f => (a -> b) -> f a -> f b --<$>
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Read <*> as (ap)ply, "tiefighter".

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b

--Make any Applicative f a Functor:
fmap g x = pure g <*> x
```

Applicative embodies

1. function application on the level of (contained) values.
2. combination on the level of structures.

9.1 Applicative instances

```
instance Applicative Maybe where
   pure x = Just x
   Just f <*> Just x = Just (f x)
   _ <*> _ = Nothing

instance Monoid c => Applicative ((,) c) where
   pure x = (mempty, x)
   (c₁, f) <*> (c₂, x) = (c₁ `mappend`c₂, f x)

instance Applicative [] where
   pure x = [x]
   fs <*> xs = [f x | f <- fs, x <- xs]
```

10 Interlude: Monoid

Type class Monoid a represents combinable values of type a:

```
class Monoid a where
   mempty :: a -- empty, neutral element for mappend
   mappend :: a -> a -> a -- combination
   mconcat :: [a] -> a
```

Examples: $(0, +), (1, \cdot), (\text{true}, \wedge), (\text{false}, \vee), (\{\}, \cup), ([], (++))$

10.1 Monoid Laws

```
mempty `mappend`xs ≡ xs
xs `mappend`(ya `mappend`zs) ≡ (xs `mappend`ys) `mappend`zs``
mconcat xs ≡ foldr mempty mappend xs
```