

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



Functional Programing

SOMMERSEMESTER 2014, WINTERSEMESTER 2015

Author:

Philipp Moers,
Update by: Denis Hirn

Dozent:

TORSTEN GRUST,
ALEXANDER ULRICH

Last updated: Tuesday 22nd December, 2015, 01:01

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

1	Introduction	4
2	Haskell Ramp-Up	5
2.1	Function Application	5
3	Values and Types	6
3.1	Base Types	6
3.2	Type Constructors	6
3.3	Currying	6
3.4	Defining Values (and thus functions)	7
3.4.1	Guards	7
3.4.2	Local Definitions	8
3.4.3	Lists	8
3.4.4	Pattern Matching	9
3.5	Algebraic Data Types	10
4	Type Classes	15
4.1	Class Inheritance	15
4.2	Class Instances	15
4.2.1	Deriving Class Instances	18
5	Domain-Specific Languages	19
5.1	Modules	21
5.2	Generalized Algebraic Data Types (GADTs)	27
5.3	Shallow embedding of a String Matching DSL	29
5.3.1	DSL design	29

»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:

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

```
1  -- Is n a prime number?
2  isPrime :: Integer -> Bool
3  isPrime n = factors n == []
4      where
5          factors :: Integer -> [Integer]
6          factors n = [ m | m <- [2..n-1], mod n m == 0 ]
7
8
9  main :: IO ()
10 main = do
11     let n = 43
12     print (isPrime n)
```

2 Haskell Ramp-Up

(Read \equiv 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): $f\ e_1 + e_2 \equiv (f\ e_1) + e_2$
- $_$ associates to the left: $g\ f\ e \equiv (g\ f)\ e$ *--(g f) is a function*)
- Function composition:
 - $(g . f)\ e \equiv g\ (f\ e)$ *--(. is something like mathematical /o/ 'after')*
 - Alternative "apply"-operator $\$$ (lowest precedence, associates to the right, `infixR 0 \$`):
 $g\ \$\ f\ \$\ e \equiv g\ \$\ (f\ \$\ e) \equiv g\ (f\ e)$
 - Prefix application of binary infix operator \otimes : $(\otimes)\ e_1\ e_2 \equiv e_1\ \otimes\ e_2$
 - Infix application of binary function f : $e_1\ 'f'\ e_2 \equiv f\ e_1\ e_2$:
 - * `1 'elem' [1,2,3]` *--(1 ∈ {1,2,3})*
 - * `n 'mod' m`
 - * ...
 - User defined operators, built from symbols
`! # $ % & * + / | = < > ? \ ^ | ~ .`

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.

$f\ a\ b \equiv (f\ a\ b) \not\equiv f\ (a,\ b)$

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	10^{100}
Float, Double	single/double floating point (IEEE)	0.1, 1e02
Char	Unicode character	"x", "\t", "\u0394", "\8710"
Bool	Boolean	True, False
()	Unit	()

3.2 Type Constructors

- Build new types from existing types
- Let $a, b \dots$ 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: $e_1 ++ e_2 \equiv (++) e_1 e_2$ (Currying, Haskell B. Curry)
- $(++) e_1 e_2 \equiv ((++) e_1) e_2$
- Function application happens one argument at a time.
- Type of n-ary function is $a_1 \rightarrow a_2 \rightarrow \dots a_n \rightarrow b$
- Type fun \rightarrow associates to the right,

read above type as

$a_1 \rightarrow (a_2 \rightarrow (\dots (a_n \rightarrow b)))$

- 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 `xi`. `f` may occur in `e`.

`f x1 x2 ... xn = e`

- A Haskell program is a set of bindings.
- Good style: give type assignments for top-level (global) bindings:

```
1 | f :: a1 -> a2 -> b
2 | 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 `|`:

```
1 | f x1 x2 ... xn
2 |   | q1          = e1
3 |   | q2          = e2
4 |   ...
5 |   | qm          = em
6 | [ | otherwise    = em+1 ]
```

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

```
1 | -- Compute n!
2 | fac :: Integer -> Integer
3 | fac n | n <= 1      = 1
4 |       | otherwise  = n * fac (n - 1)
5 |
6 | main :: IO ()
7 | main = print $ fac 10
```


3.4.2 Local Definitions

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

```

1  f1 x1 x2 ... xn | q1 = e1
2                        | q2 = e2
3                        ...
4                        | qm = em
5      where
6          g1 = ...
7          g2 = ...
8          ...
9          go

1  -- Efficient power computation, basic idea: x2k = (x2)k
2
3  power :: Double -> Integer -> Double
4  power x k | k == 1      = x
5             | even k     = power (x * x) (halve k)
6             | otherwise = x * power (x * x) (halve k)
7      where
8          even n = n `mod` 2 == 0
9          halve n = n `div` 2
10
11  main :: IO ()
12  main = print $ power 2 16

```

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

```

1  let g1 = ...
2      g2 = ...
3      ...
4      go
5  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:

```

1 f :: a1 -> ... an -> b
2 f p11 ... p1k = e1
3 f p21 ... p2k = e2
4 ...
5 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 , ..., 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	

```

1 -- Equivalent definitions of sum (over lists of integers)
2
3 -- (1) Conditional expression
4 sum' :: [Integer] -> Integer
5 sum' xs = if xs == [] then 0 else head xs + sum' (tail xs)
6
7 -- (2) Guards
8 sum'' :: [Integer] -> Integer
9 sum'' xs | xs == [] = 0
10          | otherwise = head xs + sum'' (tail xs)
11
12 -- (3) Pattern matching
13 sum''' :: [Integer] -> Integer
14 sum''' [] = 0
15 sum''' (x:xs) = x + sum''' xs
16
17 main :: IO ()
18 main = print $ (sum' [1..100], sum'' [1..100], sum''' [1..100])
19
20 -- Finite prefix of a list
21 take' :: Integer -> [a] -> [a]
22 take' 0 _ = []
23 take' _ [] = []
24 take' n (x:xs) = x : take' (n-1) xs
25
26
27 main :: IO ()
28 main = print $ take' 20 [1,3..]

```

```

1  -- Mergesort list xs, respecting ordering (<<<)
2
3  mergeSort :: (a -> a -> Bool) -> [a] -> [a]
4  mergeSort _ [] = []
5  mergeSort _ [x] = [x]
6  mergeSort (<<<) xs = merge (<<<) (mergeSort (<<<) ls)
7                               (mergeSort (<<<) rs)
8
9  where
10     (ls, rs) = splitAt (length xs `div` 2) xs
11
12     merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
13     merge (<<<) [] ys = ys
14     merge (<<<) xs [] = xs
15     merge (<<<) l1(x:xs) l2(y:ys)
16       | x <<< y = x:merge (<<<) xs l2
17       | otherwise = y:merge (<<<) l1 ys
18
19  main :: IO ()
20  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 `Ki`:

```

1  data T a1 a2 ... an = K1 b11 ... b1n1
2                        K2 b21 ... b2n2
3                        ...
4                        Kr br1 ... brnr

```

`bij` types mentioning the type vars `a1 ... an`

- Defines type constructor `T` and `r` value constructors:
`Ki :: bi1 -> bi2 -> ... bin -> T a1 ... an`
- Compare `[] :: [a]` and `(:) :: a -> [a] -> [a]`
- **Sum Type** (`n=0`, `ni = 0`)

```

1  -- Demonstrate algebraic data types: sum type
2
3  data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
4    deriving (Eq, Show, Ord, Enum, Bounded)
5
6  -- Is this day on a weekend?
7  weekend :: Weekday -> Bool
8  weekend Sat = True
9  weekend Sun = True
10 weekend _   = False
11
12
13 -- The classic rock/paper/scissor game
14
15 data Move = Rock | Paper | Scissor
16   deriving (Eq)
17
18 data Outcome = Lose | Tie | Win
19   deriving (Show)
20
21 -- Outcome of a game round (us vs. them)
22 outcome :: Move -> Move -> Outcome
23 outcome Rock    Scissor = Win
24 outcome Paper   Rock    = Win
25 outcome Scissor Paper   = Win
26 outcome us      them
27   | us == them = Tie
28   | otherwise  = Lose
29
30
31 main :: IO ()
32 main = do
33   print $ Mon == Sun
34   print $ Thu < Sat
35   print [Mon .. Fri]
36   print (minBound :: Weekday)
37   print $ outcome Rock Rock

```

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

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

- Product Types (r=1)

```

1  -- Demonstrate algebraic data types: product type
2
3  data Sequence a = S Int [a]
4      deriving (Eq, Show)
5
6
7  fromList :: [a] -> Sequence a
8  fromList xs = S (length xs) xs
9
10 (+++) :: Sequence a -> Sequence a -> Sequence a
11 S lx xs +++ S ly ys = S (lx + ly) (xs ++ ys)
12
13 len :: Sequence a -> Int
14 len (S lx _) = lx
15
16
17 main :: IO ()
18 main = do
19     print $ fromList [0..9]
20     print $ len (fromList ['a'..'m'] +++ fromList ['n'..'z'])

```

- Sum-of-Product-Types

```

1  data Maybe a = Just a | Nothing
2  data Either a b = Left a | Right b
3  data List a = Nil | Cons a (List a)

```

```

1  -- Our own formulation of cons lists
2  data List a = Nil
3              | Cons a (List a)
4      deriving (Show)
5
6  -- Haskell's builtin type [a] and List a are isomorphic:
7  --      toList . fromList = id
8  --      and   fromList . toList = id
9  toList :: [a] -> List a
10 toList []      = Nil
11 toList (x:xs) = Cons x (toList xs)
12
13 fromList :: List a -> [a]
14 fromList Nil      = []
15 fromList (Cons x xs) = x:fromList xs
16
17 -- The family of well-known list functions (combinators) can be
18 -- reformulated for List a
19 mapList :: (a -> b) -> List a -> List b
20 mapList f Nil      = Nil
21 mapList f (Cons x xs) = Cons (f x) (mapList f xs)
22
23 filterList :: (a -> Bool) -> List a -> List a
24 filterList p Nil      = Nil
25 filterList p (Cons x xs) | p x      = Cons x (filterList p xs)
26                          | otherwise = filterList p xs
27
28 liftList :: ([a] -> [b]) -> List a -> List b
29 liftList f = toList . f . fromList
30
31 mapList' :: (a -> b) -> List a -> List b
32 mapList' f = liftList (map f)
33
34 filterList' :: (a -> Bool) -> List a -> List a
35 filterList' p = liftList (filter p)
36
37 main :: IO ()
38 main = print $ fromList $ filterList' (> 3) $ mapList' (+1) $ toList
    ↪ [1..5]

```

```

1  -- Use the isomorphism between [a] and List a
2  -- to save work when defining functions over List a:
3
4  --
5  --           fromList
6  --   List a  -----> [a]
7  --   |
8  --   g |
9  --   ↓
10 --   List b <----- [b]
11
12 -- Abstract syntax tree for arithmetic expressions of literals
13 data Exp a = Lit a
14             | Add (Exp a) (Exp a)
15             | Sub (Exp a) (Exp a)
16             | Mul (Exp a) (Exp a)
17
18 deriving (Show)
19
20 ex1 :: Exp Integer
21 ex1 = Add (Mul (Lit 5) (Lit 8)) (Lit 2)
22
23
24 evaluate :: Num a => Exp a -> a
25 evaluate (Lit n)      = n
26 evaluate (Add e1 e2) = evaluate e1 + evaluate e2
27 evaluate (Mul e1 e2) = evaluate e1 * evaluate e2
28 evaluate (Sub e1 e2) = evaluate e1 - evaluate e2
29
30
31 main :: IO ()
32 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.

```
1 class C a where
2   f1 :: t1
3   ...
4   fn :: tn
```

The t_i must mention a .

For any f_i the class may provide default implementations.

We have $f_i :: C\ a \Rightarrow t_i$

(read "if a is instance C then f_i has type t_i ").

$C\ a$ is called **class constraint**.

Example:

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4   x == y = not (x /= y)
5   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 .
- $C\ a \Rightarrow t$ implies $C_1\ a, C_2\ a \dots$

4.2 Class Instances

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

```
1 instance C t where
2   f1 = <def of f1>
3   ...
4   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:

```
1 instance Eq Bool where
2   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, \dots : `instance (C1 a, C2 a, ...) => C t where`


```
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 $C_i \in \{ \text{Eq}, \text{Ord}, \text{Enum}, \text{Bounded}, \text{Show}, \text{Read} \}$:

```

1  data T a1 a1 ... an = ...
2      | ...
3  deriving (C1, C2, ...)

1  -- Use deriving to obtain the standard interpretation for the type
   -- ↪ classes
2  -- Eq, Ord, Enum, Bounded, Show, Read
3  -- The classic rock/paper/scissor game
4
5  data Outcome = Lose | Tie | Win
6      deriving (Eq, Ord, Enum, Bounded, Show, Read)
7
8  data Move = Rock | Paper | Scissor
9      deriving (Eq, Enum, Read)    -- Ord, Show defined below;
10                                     -- Bounded makes no sense
11 instance Show Move where
12     show Rock    = "rock"
13     show Paper   = "paper"
14     show Scissor = "scissor"
15
16 instance Ord Move where           -- NB: non-conventional,
17     Rock    <= Rock    = True      -- encodes game rules
18     Rock    <= Paper   = True
19     Paper    <= Paper   = True
20     Paper    <= Scissor = True
21     Scissor <= Scissor = True
22     Scissor <= Rock    = True
23     _        <= _      = False
24
25 outcome :: Move -> Move -> Outcome
26 outcome m1 m2 | m1 == m2 = Tie
27               | m1 < m2  = Lose
28               | otherwise = Win
29
30 main :: IO ()
31 main = do
32     print $ outcome Paper Scissor
33     print $ [Rock, Paper, Scissor]
34     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	L ^A T _E X
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:

```
1 type IntegerSet =
2   -- constructors:
3   empty :: IntegerSet
4   insert :: Integer -> IntegerSet -> IntegerSet
5   delete :: Integer -> IntegerSet -> IntegerSet
6   -- observer:
7   member :: Integer -> IntegerSet -> Bool
8
9 member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
10    ≡ False
```

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

```

1 import Data.List (nub)
2 -- A library of functions on integer sets,
3 -- implementation fully exposed
4
5 type IntegerSet = [Integer] -- unsorted, duplicates allowed
6
7 empty :: IntegerSet
8 empty = []
9
10 insert :: Integer -> IntegerSet -> IntegerSet
11 insert x xs = x:xs
12
13 delete :: Integer -> IntegerSet -> IntegerSet
14 delete x xs = filter (/= x) xs
15
16 (∈) :: Integer -> IntegerSet -> Bool
17 x ∈ xs = elem x xs
18
19 -----
20
21 -- "Extending" the library, accessing the exposed
22 -- implementation. Now we're doomed to stick the
23 -- list-based representation...
24
25 (⊆) :: IntegerSet -> IntegerSet -> Bool
26 xs ⊆ ys = all (\x -> x ∈ ys) xs
27
28 card :: IntegerSet -> Int
29 card xs = length (nub xs)
30
31 -----
32
33 s1, s2 :: IntegerSet
34 s1 = insert 1 (insert 2 (insert 3 empty))
35 s2 = foldr insert empty [1..10]
36
37 prog :: Bool
38 prog = s1 ⊆ s2
39
40 main :: IO ()
41 main = print $ prog

```

5.1 Modules

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

```
1 module M where
2     type Predicate a = a -> Bool
3     id :: a -> a
4     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:

```

1 | module M (id) where
2 |     ...
3 |     -- type Predicate a not exported

```

- **Abstract data types:**

export algebraic data types, but not its constructors:

```

1 | module M (Rose, leaf) where
2 |     data Rose a = Node a [Rose a]
3 |     leaf :: a -> Rose a [Rose a]
4 |     leaf x = Node x []

```

– Export constructors:

```

1 | module M (Rose(Node), leaf) where
2 |     ...
3 |     -- or export all constructors:
4 | module M (Rose(..), leaf) where

```

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

- Qualified imports to partition name space:

```

1 | import qualified M
2 |     ...
3 |     -- use M.foo syntax
4 | t :: M.Rose Char
5 | t = M.leaf 'x'

```

- Partially import module:

```

1 | -- only import nub and reverse
2 | import Data.List (nub, reverse)
3 |
4 | -- import whole module but without otherwise
5 | import Prelude hiding (otherwise)
6 | otherwise :: Bool
7 | otherwise = False -- harhar

```

- Pass imported modules to importer of own module:

```

1 | module M (... , module Data.List, ...) where
2 |     import Data.List (nub)

```

```

1 | import qualified M
2 |     M.nub

```

```

1 module SetLanguageShallowCard (IntegerSet,
2                               empty,
3                               insert,
4                               delete,
5                               member,
6                               card) where
7
8 data IntegerSet = IS (Integer -> Bool) Int -- characteristic function +
9      ↪ cardinality
10
11 -- constructors
12 empty :: IntegerSet
13 empty = IS (\_ -> False)
14         0
15 -- empty = IS (const False) 0
16 {-
17 insert :: IntegerSet -> Integer -> IntegerSet
18 insert xs@(IS f c) x = IS (\y -> x == y || f y)
19                        (if member xs x then c else c + 1)
20 -}
21 delete :: IntegerSet -> Integer -> IntegerSet
22 delete xs@(IS f c) x = IS (\y -> y /= x && f y)
23                        (if member xs x then c - 1 else c)
24
25 -- observer
26 member :: IntegerSet -> Integer -> Bool
27 member (IS f _) x = f x
28 -- member (IS f _) = f
29
30 card :: IntegerSet -> Int
31 card (IS _ c) = c

```



```

1  --import SetLanguage
2  import SetLanguageShallow
3  --import SetLanguageShallowCard
4
5  -- impossible
6  -- (SetLanguage/SetLanguageShallow do not export
7  --   data constructor IS)
8  {-
9
10 union :: IntegerSet -> IntegerSet -> IntegerSet
11 union (IS xs) (IS ys) = IS (xs ++ ys)
12
13 union :: IntegerSet -> IntegerSet -> IntegerSet
14 union (IS f) (IS g) = IS (\y -> f y || g y)
15
16 -}
17
18 set12 :: IntegerSet
19 set12 = (((empty 'insert' 3) 'insert' 2) 'delete' 3) 'insert' 1
20
21 main :: IO ()
22 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

```

1 module SetLanguageDeepCard (IntegerSet (Empty, Insert, Delete),
2                               member,
3                               card) where
4
5   -- constructors
6
7   data IntegerSet =
8       Empty
9       | Insert IntegerSet Integer
10      | Delete IntegerSet Integer
11      deriving (Show)
12
13   -- TODO: build a suitable Show instance for IntegerSet
14
15   -- observers
16
17   member :: IntegerSet -> Integer -> Bool
18   member Empty _ = False
19   member (Insert xs x) y = x == y || member xs y
20   member (Delete xs x) y = x /= y && member xs y
21
22   card :: IntegerSet -> Int
23   card Empty = 0
24   card (Insert xs x) | member xs x = card xs
25                      | otherwise = card xs + 1
26   card (Delete xs x) | member xs x = card xs - 1
27                      | otherwise = card xs
28
29 import SetLanguageDeepCard
30
31 set12 :: IntegerSet
32 set12 = (((Empty 'Insert' 3) 'Insert' 2) 'Delete' 3) 'Insert' 1
33
34 main :: IO ()
35 main = do
36     print $ set12
37     print $ set12 'member' 3
38     print $ card set12

```

```
1 module ExprDeepNum (Expr(..),
2                       eval) where
3
4   -- constructors
5
6   data Expr =
7       Val Integer
8     | Add Expr Expr
9     | Mul Expr Expr
10    | Sub Expr Expr
11    deriving (Show)
12
13   instance Num Expr where
14       fromInteger n = Val n
15       e1 + e2 = Add e1 e2
16       e1 - e2 = Sub e1 e2
17       e1 * e2 = Mul e1 e2
18
19       abs e = undefined
20       signum e = undefined
21
22
23   -- observer
24
25   eval :: Expr -> Integer
26   eval (Val n) = n
27   eval (Add e1 e2) = eval e1 + eval e2
28   eval (Mul e1 e2) = eval e1 * eval e2
29   eval (Sub e1 e2) = eval e1 - eval e2
```

```

1  import ExprDeepNum
2
3  -- e1 = 8 * 7 - 14
4  e1 :: Expr
5  e1 = Sub (Mul (Val 8) (Val 7)) (Val 14)
6
7  e2 :: Expr
8  e2 = 8 * 7 - 14
9
10 main :: IO ()
11 main = do
12     print $ eval e1
13     print $ negate e2
14     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:

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

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

```

1  data T a1 a2 ... an where
2      k1 :: b11 -> ... -> b1n1 -> T t11 t1n
3      ...
4      kr :: br1 -> ... -> brnr -> T tr1 trn

```

```

1  {-# LANGUAGE GADTs #-}
2
3  module ExprDeepGADTTyped (Expr(..),
4                           eval) where
5
6  -- Expr a: an expression that, if evaluated, will yield a value of type
7  --      a
8
9  data Expr a where
10     ValI    :: Integer          -> Expr Integer
11     ValB    :: Bool             -> Expr Bool
12     Add     :: Expr Integer -> Expr Integer -> Expr Integer
13     And     :: Expr Bool  -> Expr Bool      -> Expr Bool
14     EqZero  :: Expr Integer          -> Expr Bool
15     If      :: Expr Bool  -> Expr a -> Expr a -> Expr a
16
17  instance Show (Expr a) where
18     show (ValI n)    = show n
19     show (ValB b)    = show b
20     show (Add e1 e2) = show e1 ++ " + " ++ show e2
21     show (And e1 e2) = show e1 ++ " ^ " ++ show e2
22     show (EqZero e)  = show e  ++ " == 0"
23     show (If p e1 e2) = "if " ++ show p ++ " then " ++ show e1 ++ " else "
24     --      ++ show e2
25
26  -- NB: this is *typed* evaluation of expressions:
27  eval :: Expr a -> a
28  eval (ValI n)    = n
29  eval (ValB b)    = b
30  eval (Add e1 e2) = eval e1 + eval e2
31  eval (And e1 e2) = eval e1 && eval e2
32  eval (EqZero e)  = eval e == 0
33  eval (If p e1 e2) = if eval p then eval e1 else eval e2

```

```

1  import ExprDeepGADTTyped
2
3  e1 :: Expr Integer
4  e1 = If (EqZero (Add (ValI 0) (ValI 0))) (ValI 42) (ValI 43)
5
6  -- e2 yields compile-time type error
7
8  e2 :: Expr Bool
9  e2 = EqZero (ValB True)
10
11 main :: IO ()
12 main = do
13   print $ e1
14   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"	<code>lit :: Char -> Pattern Char</code>
match empty string	ϵ	<code>empty :: a -> Pattern a</code>
fail always	\emptyset	<code>fail :: Pattern a</code>
alternative	—	<code>alt :: Pattern a -> Pattern a -> Pattern a</code>
sequence	.	<code>seq :: (a -> b -> c) -> Pattern a -> Pattern b -> Pattern c</code>
repetition	*	<code>rep :: Pattern a -> Pattern [a]</code>

```

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")]

-}

```

```

1 import Prelude ()
2
3 import PatternMatching
4
5 fortytwo :: Pattern (Char,Char)
6 fortytwo = seq (,) (lit '4') (lit '2')
7
8 digit :: Pattern Char
9 digit = lit '0' 'alt' lit '1' 'alt' lit '2' 'alt' lit '3' 'alt'
10         lit '4' 'alt' lit '5' 'alt' lit '6' 'alt' lit '7' 'alt'
11         lit '8' 'alt' lit '9'
12
13 number :: Pattern String
14 number = rep digit
15
16 smiley :: Pattern String
17 smiley = seq (:) (alt (lit ':' (lit ';'))
18                     (seq (:) (lit '-')
19                           (seq (:) (alt (lit '(' (lit ')'))
20                                     (empty []))))))
21
22 main :: IO ()
23 main = do
24   print $ fortytwo "42 rules"
25   print $ rep digit "42 rules"
26   print $ smiley ";-)"

```

```

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)"

```