Part I, Chapter 1 ‘Why Functional Programming Matters?’

1. What does John Hughes consider glue of a programming language? What is glue good for?

glueing solutions of subproblems together; support modular program design

2. What kinds of new glue do functional programming languages offer compared to other, especially the imperative programming paradigm?

- High-order functions (glueing functions together / Functional compositions)
- lazy evaluation (glueing programs together)

5. How does folk knowledge wisdom argue a possible superiority of functional programming compared to programming in other, especially the imperative programming paradigm?

- programs free of assignment & side-effects
- calls have no effect except of computing their result
- less bugs (free of major source bugs)
- evaluation order is irrelevant, expressions can be evaluated whenever
- no control flow needs to be specified
- proving correctness is easier
- programs are smaller than imperative programs (more productive)

6. Why does John Hughes consider the reasoning along the lines of folk knowledge wisdom to not stand the test?

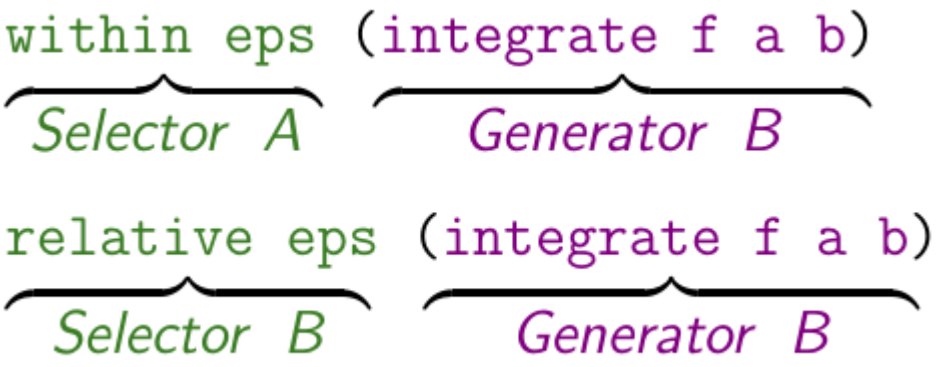
do not really explain the power of functional programming
in particular, they do not provide:

- explanation of functional programming strengths
- and no definition of what makes a functional program good

8. Shall functional programming languages be lazy or eager? The eternal question of functional programming. How does John Hughes answer it? What is the reasoning underlying his answer?

- Lazy - as it is so important as a glue

9. How does the generator/prune pattern support the reasoning of John Hughes?



10. What are characteristics of a 'good' functional program? What shall a functional programmer strive for when programming? What shall (s)he expect to be the relevant 'tools?'

- modularization
- use generator selector
- lazy evaluation
- function composition

Interesting stuff:

- decomposition (modularization) + simple function
  - with higher-order function (like reduce) and specific simple function as argument
  - = easy program frame that allows code reusage and reduces programming effort
- whenever a new data type is defined (lists, trees,...) implement first a higher-order function allowing to process values of this type

Part II, Chapter 2 'Programming with Streams'

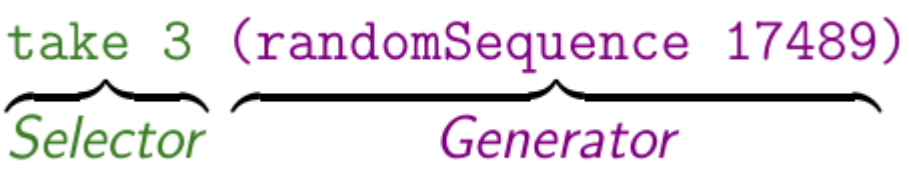
1. What is the generator/prune paradigm, pattern, or principle? What are instances of this paradigm?

- Termination of a generate-select program depends crucially on evaluating the program in normal order reduction (typically implemented in terms of efficient lazy order reduction) to avoid non-terminating infinite sequence of reductions of applicative order
- Applicative order

```
head twos
  ->> head (2 : twos)
  ->> head (2 : 2 : twos)
  ->> head (2 : 2 : 2 : twos)
  ->> ...
```

- normal order reduction will always terminate if there is a terminating reduction (Church/Rosser Theorem)

- Generate-select



- Generate-filter

```
primes :: [Int]
primes  =  sieve  [2..]
```
Generator *Filter* *Generator*

- Generate-transform

```
map (\x -> x-1)  primes ->> [1,2,4,6,10,12,16,...
```
Transformer *Generator*

- Patterns can be combined

```
((take 5) . (drop 5))  primes  ->>  [13,17,19,23,29]
```
Selector *Filter* *Generator*

3. How can we achieve performance gains using stream programming? Illustrate your answer by means of appropriate examples.
Reusing already calculated date (think of fibonacci numbers)
= avoiding recomputations and re-cursions

4. What is memoization? How is it related to stream programming?

- Replace, where possible, the (costly) computation of a function according to its body by looking up its value in a table, a so-called memo table.
- have a stream of all previous fibonacci numbers which is the memo table in this case

5. Illustrate memoization by means of a concrete example.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)      -- Not reached by memo!

memo_list = [ memo fib n | n <- [0..] ]        -- Generator
memo :: (Int -> Int) -> Int -> Int
memo fib 0 = fib 0                          -- Basis ('tuft')
memo fib 1 = fib 1                          -- Basis ('tuft')
memo fib n = memolist !! (n-1) + memolist !! (n-2) -- Trigger

memo_fib n = memo fib n          -- memo_fib replacing fib
```

6. What is the difference between recursion and corecursion? Illustrate your answer also in terms of appropriate examples.
recursion - has a base case; does terminate eventually
corecursion - no base case; does never terminate

8. From a pragmatical point of view, what are possible problems a programmer will face when implementing a memoization-inspired approach?

- running out of memory
- usage of a refutable pattern may lead to a livelock as pattern matching does not work (too eager)

```
client :: [Response] -> [Request]
client ys = 1 : ys
```

by:

```
client' :: [Response] -> [Request]
client' (y:ys) = if ok y then 1 : (y:ys)
                         else error "Faulty Server"
  where ok y = True
```

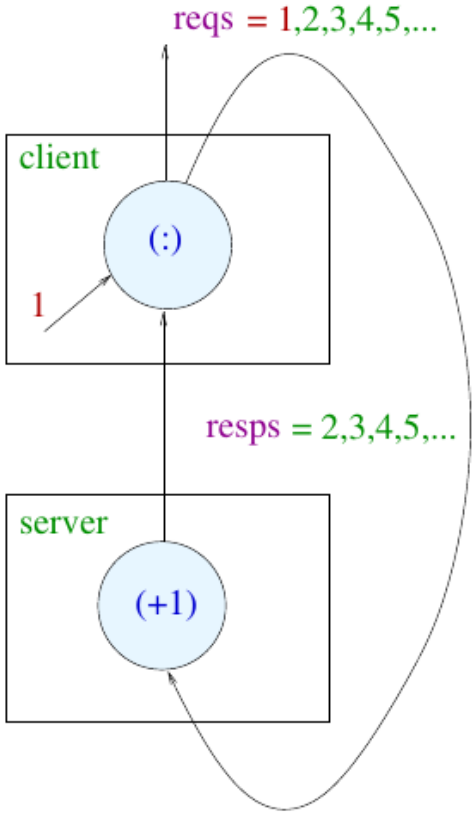
-
- client' = refutable; client = irrefutable

9. What is a stream diagram? What does it serve for? Illustrate your answer by means of an example.

- means of considering and visualizing problems on streams as processes

```
reqs  = client resps
resps = server reqs
```

as a stream diagram:



10. Stream programming and eager evaluation. Do they go well along with each other? What is the reasoning leading to your answer?

- no, they do not go well together as streams are infinite and therefore if we want to evaluate eager we will never get a result

Interesting stuff

- streams (infinite by definition) vs. lists (finite by definition)
    - however model streams as ordinary list types - so one can reuse predefined functions on lists
- corecursive streams - recursive definitions that lack a base case, yield infinite objects
    - schopfe und sümpfe
    - 
    - e.g.: ones = 1 : ones
- selector (termination ensured - e.g.: take 5) - filter (termination not-ensured - e.g. filter is_palindrome)
- memo function memo :: (a -> b) -> (a -> b)
    - identity on functions
    - keeps track on the arguments it has been applied to and their results
    - memo functions not part of Haskell'98 but supported by some non-standard libs

- Haskell distinguishes between
    - irrefutable (e.g. variable names, wild card _)
    - refutalbe (all other types e.g. [], (x:xs),...
    - patterns
- Any value passed to an irrefutable pattern always matches it e.g. [] matches xs
- Values only match refutable patterns if they fit e.g. [] matches [] but not (y:ys)
- Irrefutable patterns can be made refutable by using ~ e.g. ~(x:xs) = lazy pattern
    - pattern matching is postponed until needed

Part IV, Chapter 7 'Functional Arrays'

1. Functional programming languages offers lists for storing data. The name of the functional language 'Lisp' is actually an acronym standing for 'List Processing.' Why should there be something like 'arrays' in functional languages? Why is adding arrays to a functional language a challenge?

- Why use arrays?
    - can be accessed or updated in constant time
    - update does not need extra space
    - can be stored in contiguous memory
    - HOWEVER: size is fixed
- Lists have the advantage that size is NOT fixed (e.g. accessing list element using (!!) takes proportional time)
- functional arrays not supported by standard prelude but there are specific libraries
    - Static arrays (immutable): without destructive update
    - Dynamic arrays (mutable): with destructive update

2. Can a functional programming language offering dynamic arrays be pure?
no as dynamic arrays are mutable and hence come with side effects

3. Referential transparency, static arrays, dynamic arrays. Which of these terms go well along with each other in functional programming languages? Which ones do not? Why?
Referential transparency is a concept in functional programming where a function, when given the same input, always produces the same output, and it has no side effects. So this goes together with static arrays as dynamic ones are mutable

4. What are appealing features of functional lists? What are appealing features of imperative arrays?

- functional lists
    - non-fixed - potential infinite - size
- imperative arrays
    - can be accessed or updated in constant time
    - update does not need extra space
    - can be stored in contiguous memory

5. What are distracting features of functional lists? What are distracting features of imperative arrays?

- functional lists
    - proportional runtime for access and update
- imperative arrays
    - fixed size

6. What means 'updating an entry of a static array' from an implementation point of view? What 'updating an entry of a dynamic array?' — Consider the non-standard library Array and the static arrays and supporting functions it provides.

- static: creating a new array and copying everything but the new value
- dynamic: overwriting stuff in the array

7. Array offers various means to create and initialize an array value. Which ones? How do they differ from each other?

- array

```
array :: Ix a => (a,a) -> [(a,b)] -> Array a b
array bounds list_of_associations
```

- listArray

```
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
listArray bounds list_of_values
```

- accumArray

```
accumArray :: (Ix a) => (b -> c -> b) -> b
                     -> (a,a) -> [(a,c)] -> Array a b
accumArray f init bounds list_of_associations
```

9. The implementation of:
fibs n = a where
a = array (1,n) ([(1,0),(2,1)] ++ [(i,a!(i-1) + a!(i-2)) | i <- [3..n]])
shows very poor performance. Why? How could the implementation be modified to improve
performance (while still using arrays)? (Question wrong??)

```
fibs n = a where
            a = array (1,n)
             ([(1,0),(2,1)] ++ [(i, a!(i-1) +,a!(i-2))
                                  | i <- [3..n]])

xfibs n = a n
a n     = array (1,n) ([(1,0),(2,1)] ++
                       [(i,a n!(i-1) + a n!(i-2))
                         | i <- [3..n] ])
```

xfibs is slow; as the array always has to be newly created during computation; while fibs contains a local array definition

10. What is type class Ix good for? What are its member functions?
Ix is the type class for index types; member functions are: range, index, inRange, rangeSize

Interessant:

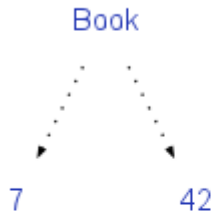
- pre-defined array operations:
  - (!) - access
  - bounds - smallest and largest index
  - indices - list of indices
  - elems - list of the elemtents/values of the arrays
  - associates - index/value pair list
  - (//) - array update (new array no destructive update)
  - amap: wie map nur für arrays

Part IV, Chapter 8 'Abstract Data Types'

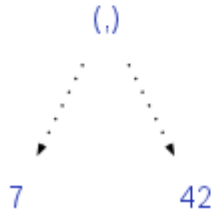
Interesting:

- Concrete data type description (trees)
  - everything about data type, about what they look like
  - but nothing about functions allowing us to create, access and manipulate values
- Abstract data type description (like stacks)
  - everything about functions
  - nothing about the kind of values of the data type
- ADTs are NOT first-class citizens in Haskell
- Type - newtype - data
  - type - type alias

- `data Book = Book Int Int`



- `newtype Book = Book (Int, Int)`



  -
  - data – works with algebraic data types, multiple different options can be given for creating a data
  - newtype - does not work with algebraic data types, newtype has only one constructor

1. Abstract data types are appealing. Why? Why conceptually? Why pragmatically? Why from a software-engineering point of view?

- separation of concerns - separate specification and implementation
- information hiding - no disclosure of internal structure of concrete data type (CDT)
- safety and security - CDT values implementing their only implicitly defined abstract data type (ADT) can exclusively be created, accessed and manipulated using the ADT operations
- support programming in the large
  - modular program development
- reusability and maintainability
  - CDT implementation can easily be replaced

2. The definition of an abstract data type consists of several parts. Which ones? What is their role?

- Specification (user-visible)
  - interface specification: signatures of ADT operations
  - behaviour specification: laws for ADT operations
- implementation (user-invisible)
  - implementing the ADT values and operations as CDT or CDT operations
- Verification
  - is the specification ADT correct (consistent and complete)

    - is the implementation of the CDT correct (does it comply with the rules of the ADT)

3. Programming languages supporting abstract data types do not need to additionally support concrete data types. Right or wrong? Why?
No, this is false as ADT cannot be used for anything. There is no implementation for the abstract types

4. What are challenges of defining abstract data types in general? What are particular challenges of doing so in Haskell?
- ADTs are not first-class citizens in Haskell
- pragmatically use of Haskell features allowing us to achieve the constituting properties
- behaviour specification of ADTs can only be given as comments
- if values need to be displayed use explicit instance declaration

5. What are the proof obligations when defining an abstract data type? Who is in charge for accomplishing them?
- specification - ADT specificator
- implementation - CDT implementor

6. The lecture notes provide some examples of abstract data types. What could be other examples?
- Stacks, Queues, Priority Queues, Tables
- Ring buffer, linked lists

8. Considering stacks and queues as abstract data type examples. What is the conceptual difference between stacks and queues? How is this difference taken care of by the definitions of stacks and queues as abstract data types?

- Stack: LIFO
- Queue: FIFO
- different rules

10. If there were need to display set values, how could this be done in Haskell? What should be taken care of? Why?
- could be done by using automatic deriving-clause for Show (making the CDT an instance of the type class Show) - this however reveals the internal structure -> unsafe and does violate information hiding
- safe and secure way: explicit instance-declaration

```
instance (Show a) => Show (Stack a) where
 showsPrec _ Empty str = showChar '-' str
 showsPrec _ (Stk x s) str
   = shows x (showChar '|' (shows s str))
```

Part II, Chapter 3 'Algorithm Patterns'

1. Algorithm patterns and glue in the sense of John Hughes. What is the link between the two?
The algorithmic patterns should be implemented as a form of higher-order-functions so the patterns can be reused instead of being writing again and again

2. The search space for backtracking, priority-first or greedy search can be (conceptually) infinite. Why isn't this a problem for the Haskell implementations provided for these algorithm patterns?

- the graph representations for e.g. backtracking should be generated on-the-fly as computation proceeds (implicitly represented graphs)
    - paired with lazy evaluation

3. Conceptually, the algorithm patterns considered in Chapter 3 fall into two groups. Which ones?

- top-down: starting form the initial problem; algorithm works down to the solution by considering sub-problems
    - e.g. divide-and-conquer
- bottom-up: starting from a small problem instance
    - e.g. dynamic programming

4. Memoization is linked to one of the algorithm patterns considered in Chapter 3. Which one? In what respect?
Bottom up - as they start with small problem solved instances and reuses them to compute the following results; memoization is needed for this

In more detail difference dynamic programming vs. memoization

- Memoization opportunistically computes and stores argument/result pairs on a by-need basis ('lazy' approach).
- Dynamic programming systematically precomputes and stores argument/result pairs before they are needed ('eager' approach).

Benefits of dynamic programming

- memory efficiency - limited history recurrence (limited number of preceding values remembered)
- run-time performance - less recomputations -> faster

Benefits of memoization

- freedom of conceptual overhead - no need to think
- freedom of computational overhead - no systematic precomputation of values that might still not be needed

5. Some problems and algorithm patterns seem to be a good fit. But some characteristics of the problem can make the pattern inadequate for solving the problem. What are examples of such problem/pattern pairs? What are the general problems behind these problem/pattern pairs? What are concrete problem/pattern pair examples, where these problems materialize?

- e.g. divide and conquer and fibonacci problem; as this results in exponential runtime

6. What are the ingredients, i.e., the parameters resp. arguments of the divide-and-conquer and backtracking search algorithm patterns?

- Backtracking
  - search space
  - successors (connections)
  - initial node
  - goal node
- Divide-and-conquer
  - divide
  - combine
  - solve
  - check if small enough (indiv)

7. Consider the syntactical signatures of the higher-order functions for

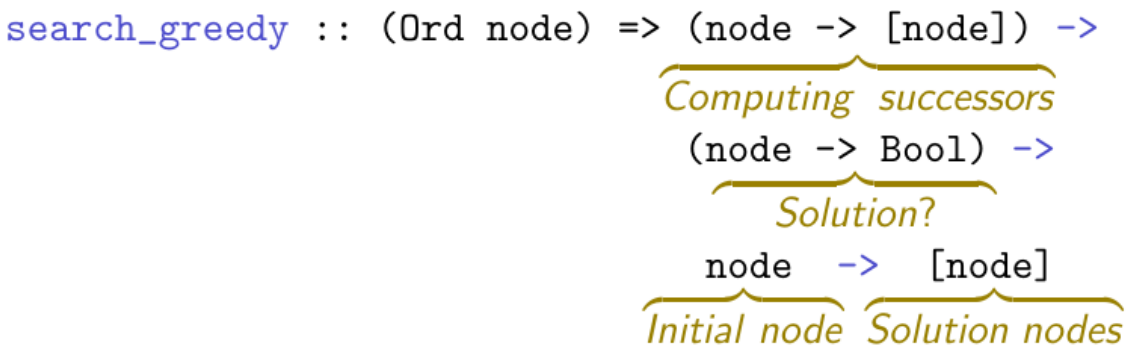
a) priority-first search:

search dfs :: (Eq node) => (node -> [node]) -> (node -> Bool) -> node -> [node]

```
search_dfs :: (Eq node) => (node -> [node]) ->
                           Computing successors
                           (node -> Bool) ->
                            Solution?
                           node -> [node]
                           Initial node  Solution nodes
```

b) greedy search:

search greedy :: (Ord node) => (node -> [node]) -> (node -> Bool) -> node -> [node]



Explain the meaning of all items occurring in the two signatures in detail. Why is the type context in search ds different from the one in search greedy (i.e., (Eq node) vs. (Ord node))?
Ord node - we can compare (<, =, <=) nodes with each other to see what node has the highest priority
Eq node - can only check if nodes are equal

8. What are application fields, problems often amenable to dynamic programming?

- graph algorithms
- search algorithms
- Examples: shortest path for all pairs of nodes of a graph, fibonacci

Part II, Chapter 4 'Equational Reasoning for Functional Pearls'

1. The symbol = is used by both functional programming languages like Haskell and imperative / object-orientied programming languages like C or Java. How do the 'functional' and the 'imperative/object-oriented' = differ from each other?

- imperative: assignment, destructive assignment, command / instruction
- functional: equal by definition, genuine mathematical equation; both sides have the same value

2. What does equational reasoning refer to? Explain and illustrate it by a striking example.

- a well-known mathematical means for reasoning about and proving the validity of e.g. arithmetical statements
- Example: Binom formula proof

4. Referential transparency supports equational reasoning. Why?
As there are no side-effects the code can be treated as mathematical definitions which can be used for proofing using equational reasoning

5. Why is equational reasoning important for functional pearls?

- Functional pearl: the calculation and proof process that leads to a pretty implementation
- To show that we can replace bad functional programs with functional pearls; this means you need a proof to show that a simple but not so elegant program were

everyone can easily see the correctness is equal to the elegant pearl that solves the problem better

6. What constitutes a functional pearl ? Can you illustrate your answer by means of an example?

- it is elegant and good: e.g. it is fast, short,... and showcases why functional programming is cool
- e.g. fast_reverse instead of reverse or fast fibonacci calculation instead of exponential one

7. What is meant if someone speaks of 'wholemeal' programming? What are its benefits?

- Benefits
  - helps avoiding indexitis and encourages lawful program construction
- Treat a matrix as a complete entity in itself instead of thinking about matrices in terms of indices and doing arithmetic on indices to identify rows, columns and boxes
  - think about Hidoku solver

11. Gofer, an overloaded term. Why?

Go F(or) E(quational) R(easoning) - is both name and acronym of a functional programming language

12. What is the SFN problem?

Smallest free number; X is set of natural numbers; compute the smallest number natural number that is not in X

Solution:

- sort list
- find first gap

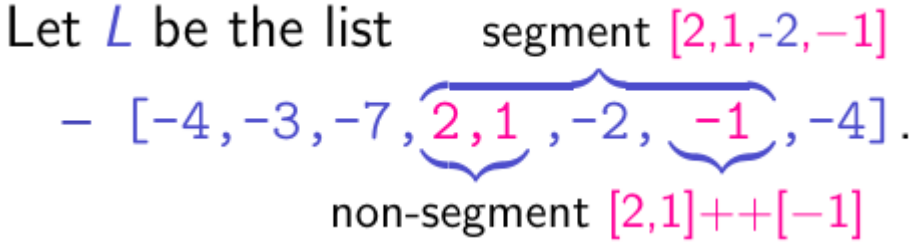
13. Is solving the SFN problem efficiently of practical relevance? Why? Or, why not?

memory re-assignment, memory allocation

14. The MNSS problem and the MSS problem are closely related. Why? In what respect?

- Maximum segment sum
  - L is a list of positive and negative integers
  - What is the maximum of the sums of all possible segments of L

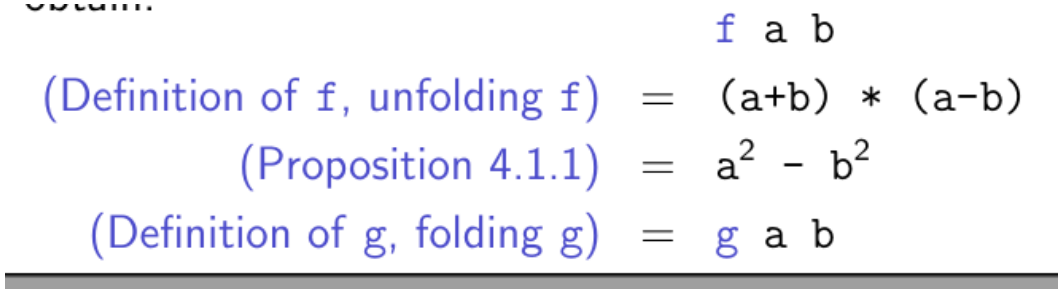
- Maximum non-segment sum
  - L is a list of positive and negative integers
  - What is the maximum of the sums of all possible non-segments of L



15. Comparing the initial algorithms for the MNSS and MSS problem, which one is computationally more complex? Why?
The pearl is more complex but faster

Interesting:

- Folding / Unfolding
  - Folding - right-to-left unrolling of functional definitions
  - Unfolding - left-to-right unrolling of functional definitions



  -
  - be aware of the Haskell ordering on the equation when showing correctness (cases are listed one after another)

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

Part IV, Chapter 9 ‘Monoids’

1. What is a monoid? What can be made an instance of Monoid?

- monoids are instances of type class Monoid obeying the monoid laws
- type class Monoid: has operators - mempty, mappend, mconcat (reducing a list of monoid values to a single monoid value using mappend)

2. Monoid foresees an operation called mappend for its instances. Is the name mappend well chosen? Why? Or, why not?

- no it is not well chosen as for most monoids the effect of mappend cannot be thought of in terms of appending values

- think of a function that takes two m values and maps them to another m value instead

4. What are the monoid laws? What do they require?

Monoid Laws

```
mempty `mappend` x           = x          (MonoL1)
x `mappend` mempty           = x          (MonoL2)
(x `mappend` y) `mappend` z  =
  x `mappend` (y `mappend` z)             (MonoL3)
```

- MonoL1 and MonoL2 require that mempty is a left-unit and a right-unit of mappend (hence it is a unit)
- MonoL3 requires that mappend is associative

5. Who is in charge that Monoid instances satisfy the monoid laws?
The programmer

6. Why is it not just a matter of taste, an issue of a 'good' or 'bad' programming style but dangerous to implement a monoid instance failing the monoid laws? Illustrate your reasoning by an example.
You would expect the monoid to have certain properties (especially being associative) if this is not fulfilled it cannot be used in places where it is expected to have this properties

7. The implementation of mappend of proper monoids must be:
(a) commutative. - No
(b) associative. - Yes
(c) distributive. - No?

8. The below can be made monoids:
(a) Int - Yes
(b) Bool - Yes
(c) Char - Yes
(d) Ordering - Yes
(e) IO
(f) [Int] - Yes
(g) [a] - Yes
(h) []
(i) Maybe Int - Yes
(j) Maybe a - Yes
(k) Maybe
(l) Either Int
(m) Either a
(n) Either
(o) (Int -> Int)
(p) (a -> Int)
(q) (Int -> b)

(r) (a -> b)
(s) (a ->)
(t) (-> b)
(u) (->)

10. Monoids are made for folding values. Why?
because folding requires an associative operation which monoids need to offer as part of the monoid laws

Part IV, Chapter 10 ‘Functors’

1. What is a functor?
is a representative of a new kind of type classes = higher-order type classes - type constructor class
enables bundling all types whose values can be mapped over in a functor, offers over-loaded function fmap

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

2. What (in principle) can be made an instance of Functor?
types whose values can be mapped over compositionally with a neutral element should be made instances of type constructor class Functor
The idea is that i have one type where I can just use mmap without having to reimplement it all over again.

Argument f of Functor is applied to a type variable - hence, f must be a 1-ary type constructor variable (not a type variable!!) - instances of Functor are thus type constructors not types

3. Give the default implementation of the list functor.
fmap f [] = []
fmap f (x:xs) = (f x) : (fmap (f xs))

4. The below can be made functors:
(a) Int
(b) Bool
(c) Char
(d) Ordering
(e) IO - yes
(f) [Int]
(g) [a]

(h) [] - yes
(i) Maybe Int - no, not a type constructor
(j) Maybe a - no, not a type constructor
(k) Maybe - yes
(l) Either Int - yes
(m) Either a - yes
(n) Either - no
(o) (Int -> Int) - no, not a type constructor
(p) (a -> Int) - no, not a type constructor
(q) (Int -> b) - no, not a type constructor
(r) (a -> b) - no, not a type constructor
(s) (a ->) - yes
(t) (-> b) - yes
(u) (->) - no - this is the map function
Right or wrong? Meaningful or not? Why?

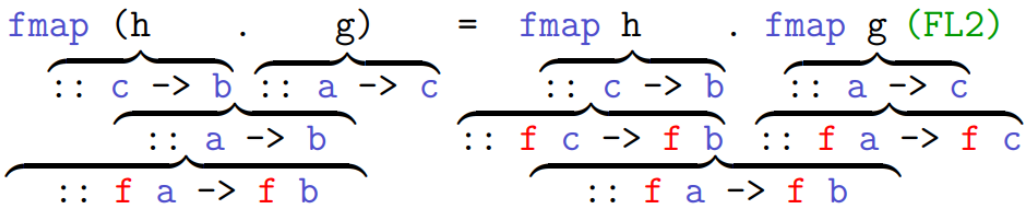
5. fmap :: (a -> b) -> f a -> f b can be read in a curried and uncurried fashion. Explain the two views of fmap.

`fmap :: (Functor f) => (a -> b) -> f a -> f b`

can be thought of in two ways. As a map which takes a map `g :: a -> b` and

1. lifts `g` to a new function `h :: f a -> f b` operating on functor values ⇝ curried view.
2. a functor value `v :: f a` and maps `g` over `v` ⇝ uncurried view.

6. Show that fmap (h . g) = fmap h . fmap g is type-correct.



7. What is the meaning of fmap of the map functor?
changes output type?

8. Give the standard implementation of the Maybe functor together with the laws it has to satisfy.
fmap g (Just x) = Just (g x)
fmap g Nothing = Nothing

Part IV, Chapter 11 ‘Applicatives’

1. What is an applicative?
an applicative is an instance of the type constructor class Applicative that obeys the applicative laws

applicatives are functors and hence 1-ary type constructors

2. What functions have to be implemented for an applicative?
pure - takes a value of any type and returns an applicative
(<*>) - angle brackets

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

3. The below can be made applicatives:
(a) Int - no
(b) Bool - no
(c) Char - no
(d) Ordering - no
(e) IO - yes
(f) [Int] - no
(g) [a] - no
(h) [] - yes
(i) Maybe Int - no
(j) Maybe a - no
(k) Maybe - yes
(l) Either Int - yes
(m) Either a - yes
(n) Either - no
(o) (Int -> Int) - no
(p) (a -> Int) -no
(q) (Int -> b) - no
(r) (a -> b) no
(s) (a ->) yes
(t) (-> b) yes
(u) (->) no
Right or wrong? Meaningful or not? Why?

5. The list applicative and list comprehension are closely linked to each other. Why? In what respect?

```
[x*y | x <- [2,5,10], y <- [8,10,11]]
 ->> [16,20,22,40,50,55,80,100,110]
```

...can alternatively be written using (<$>) and <*> and vice versa:

```
(*) <$> [2,5,10] <*> [8,10,11]
 ->> [16,20,22,40,50,55,80,100,110]
```

Part IV, Chapter 14 'Kinds'

1. What are kinds? What are they good for?
kinds are types of types and type constructors; they are represented by expressions over the symbol * (star)
two see how many values a type or type constructor needs

2. What is the kind of
(a) Maybe Int - *
(b) Maybe a
(c) Maybe - * -> *
(d) Either Int - * -> *
(e) Either a
(f) Either - * -> * -> *
(g) (Float -> Int) - *
(h) (a -> b)
(i) (a->) -
(j) (->) - * -> * -> *
(k) [] - * - *
(l) [a]
(m) [Int] - *

4. Let t1 , t2 , t3 be of kind *, * -> * and * -> * -> *, respectively. Are t1, t2, t3 eligible (in principle) as instances of
(a) Eq? - t1, t2, t3
(b) Monoid? - t2k, t3
(c) Functor? - t2
(d) Applicative? - t2
Why? Or, why not?
5. There can be

(a) applicatives t2
(b) functors and applicatives t2
(c) monoids and functors t2
differing in their kind. Right or wrong? Why?

Part IV, Chapter 12 ‘Monads’

1. What are the monad laws? What do they require, ensure?

Monad Laws

```
return x >>= f              = f x                (ML1)
c >>= return                = c                  (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g      (ML3)
```

- 1-ary type constructors
- >>= sequence operator

2. What does it mean to make a monad?

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  c >> k  =  c >>= \_ -> k
  fail s  =  error s
```

- >>= monadic function composition
- return monadic pure
- fail error handling
- >> simplified sequencing
- Monad is a subtype of applicative

3. Can every type be made a monad? (Do not just answer ‘yes’ or ‘no’).
no - only unary type constructors that fulfil the monadic laws

5. The (standard definitions of the)
(a) identity - correct; this monad maps a type to itself
(b) list - incorrect; fail is newly defined;
(c) maybe; incorrect; fail is newly defined (Nothing); useful for computation sequences that produce a result but might also produce an error

(d) map (->) - incorrect; (>>) newly defined;
(e) state - both inherited; The State monad in Haskell is a tool for managing and encapsulating mutable state within a functional programming paradigm. It allows developers to thread state through a sequence of computations while maintaining referential transparency.
(f) input/output - both reimplemented
monad uses the default implementations of return and fail of Monad. Right or wrong? Give the actual implementation if a default implementation is not used.
(>>=) and return almost always defined; (>>=) is the minimum complete implementation
(>>) and fail are usually just used from the Monad constructor class

11. What is a monad-plus?
Monads with a “plus” operation and a zero element. Which is a unit for the plus operation and a zero for the monadic bind.

12. What are the monad-plus laws?

Monad-Plus Laws

```
m >>= (\_ -> mzero) = mzero          (MPL1)
mzero >>= m         = mzero          (MPL2)
m `mplus` mzero     = m              (MPL3)
mzero `mplus` m     = m              (MPL4)
```

The IO monad cannot be made monad plus, because it lacks an appropriate mzero element.

13. How is the list monad-plus defined?

```
instance MonadPlus [] where      -- note the over-
  mzero = []                     -- loading of Id
  mplus = (++)
```

14. Why is input/output a challenge for purely functional languages?
IO inherently contains side-effects hence making functional programs impure
However, monadic implementation of IO allows specifying the evaluation order of functions in interaction shell making it controllable

15. Why is the monad concept appealing for handling input/output in Haskell?

Interesting:
- (>@>) operator

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c)
                                -> (a -> m c)
f >@> g = \x -> (f x) >>= g
```

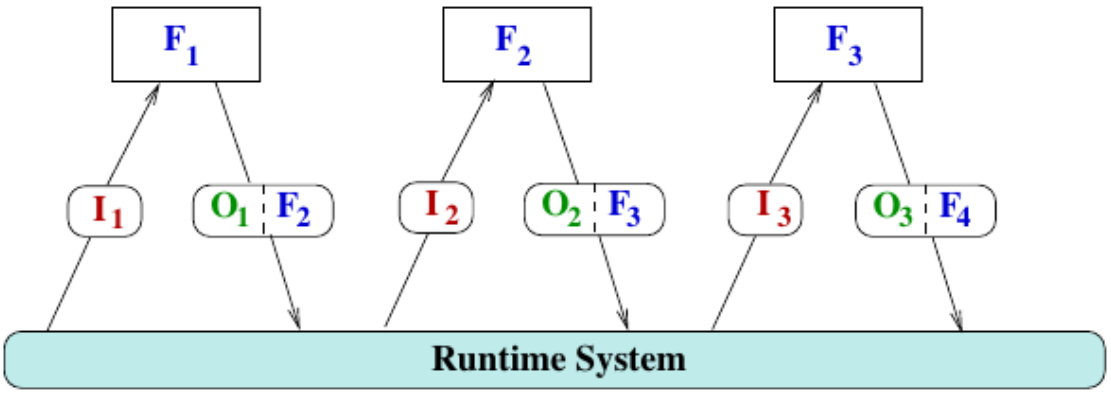
  - makes left-hand side of associativity of (>>=) look better
- If a type constructor should be an instance of Monad and Functor they need to fulfil this additional law:

```
fmap g xs  =  xs >>= return . g
           ( =  do x <- xs; return (g x) )
```

- Utility functions defined for Monads - they use the monad functions
  - sequence (uses return and foldr)
  - mapM (uses sequence)
  - mapF (uses return) - (equals map on lists)
  - joinM (equals concat on lists

- do-Notation
  - replace (>>=) and (>>) to express the imperative flavour of this operations in an even more imperative way
  - makes monadic sequencing syntactically more compelling and concise
  - can contain a let expression for storing a value in front of the do-block
  - Conversion rules

- the list comprehension is syntactic sugar for monadic syntax
- IO Monad
  - Type read commands: (IO a) - type instances a whose value can be read
  - Type write commands: (IO ()) - () = null tuple type as there is no type as we have an output
  - all data must be provided at the very beginning
  - no interaction between a program and a user
  - input output is a state transform triggered by input resulting in output



  -

```
(R1) do e <=> e

(R2) do e1;e2;...;en <=> e1 >>= \_ -> do e2;...;en
                     <=> e1 >> do e2;...;en

(R3) do let decl_list;e2;...;en <=> let decl_list
                                    in do e2;...;en

(R4) do pattern <- e1;e2;...;en <=>
          let ok pattern = do e2;...;en
              ok _       = fail "..."
              in e1 >>= ok
```

- Example:

B) The monad laws using do-notation:

```
do x <- return a; f x     =  f a                (ML1)
do x <- c; return x       =  c                  (ML2)
do x <- c; y <- f x; g y  =
             do y <- (do x <- c; f x); g y      (ML3)
```

- do x <- return a - we extract the a part of m a and assign it to x
- at the end of the do block we make it into a monad again

- Instances of Monad Plus (Examples)
  - List
  - Maybe

Part IV, Chapter 13 ‘Arrows’

1. What is an arrow? What is a sound arrow?

Arrows are like Monads but for 2 ary Type constructors.

```
pure id >>> f = f                              (ArrL1): identity
f >>> pure id = f                              (ArrL2): identity
(f >>> g) >>> h = f >>> (g >>> h)               (ArrL3): associa-
                                                        tivity

pure (g . f) = pure f >>> pure g               (ArrL4): functor
                                                        composition

first (pure f) = pure (f × id)                 (ArrL5): extension
first (f >>> g) = first f >>> first g          (ArrL6): functor
first f >>> pure (id × g) = pure (id × g) >>> first f
                                               (ArrL7): exchange
first f >>> pure fst = pure fst >>> f          (ArrL8): unit
first (first f) >>> pure assoc = pure assoc >>> first f
                                               (ArrL9): association
```

2. The below can be made arrows:
(a) Int - No
(b) Bool - No
(c) Char - No
(d) Ordering - No
(e) IO - No
(f) [Int] - No
(g) [a] - No
(h) [] - No
(i) Maybe Int - No
(j) Maybe a - No
(k) Maybe - No
(l) Either Int - No
(m) Either a - No
(n) Either - Yes
(o) (Int -> Int) - No
(p) (a -> Int) - No
(q) (Int -> b) - No
(r) (a -> b) - No
(s) (a ->) - No
(t) (-> b) - No
(u) (->) - Yes
Right or wrong? Meaningful or not? Why?

4. What is intuitively the meaning, the purpose of the arrow operations? What is their meaning for the map arrow?

Arrow composition boils down to ordinary functional composition ie. (>>>) = (.)

5. Where is Arrow sitting in the type class hierarchy of Haskell'98? Where in more recent versions of Haskell?

I am interesting:

- Utility functions for arrows
    - Swap
    - assoc
    - unassoc

Part III, Chapter 5 'Testing'

1. What are arguments in favour of testing? What are arguments in favour of verification?

- Testing = showing the presence of errors
    - types:
        - ad hoc: controllable effort but usually no quantifiable quality statement
        - systematically: controllable effort, quantifiable quality statements
- Verification = proving the absence of errors
    - formal correctness proofs -> high confidence
    - higher effort

2. What are properties or features a test system should fulfil to call it a system for systematic testing?

- Must:
    - specification-based
    - tool-supported
    - automatic
- Nice to have:
    - reporting (what was tested)

    - reproducibility, repeatability

3. What are possibilities we have for defining the meaning of a program? What are advantages, disadvantages of these possibilities?
specifications:

- informally: e.g. as commentary or separately in another document
    - disadvantage: often ambiguous, open to interpretation
    - advantages: quick and easy
- formally: e.g. with pre- and post-conditions; in formal specification language with a precise semantics
    - advantages: precise
    - disadvantages: often more complex and more work

4. What are possibilities to specify program properties of Haskell programs in QuickCheck? How do they differ in their pros and cons?

-

6. Experience shows that QuickCheck to reveal common kinds of flaws in occuring often during program development. Name a few.

- Type rounding errors

7. Is QuickCheck an appropriate tool for testing that the initial and the final implementation of a functional pearl problem are equivalent? Why? Or, why not? Or, does it depend on the problem under consideration?
it depends on the test data that is generated; standard generation functions do not suffice to show that as they are usually not generated in a way that considers edge cases; it kinda depends on the problem - if the problem is super basic then the test cases will probably suffice but not in general

8. Compare the approaches of testing against abstract models and algebraic specifications. Illustrate the idea underlying these approaches (also) by an example.

- abstract models
    - testing against a reference implementation
    - model defined with (executable) specification
- algebraic specifications
    - goal: testing correctness of an implementation
    - idea: any proper definition can satisfy certain predefined constraints

9. What are methods offered by QuickCheck to control the size of test data it generates?

- size - defines for example lists the upper bound of the list lengths
- resize - allows to supply an explicit size argument to a generator:

10. What are reporting facilities supported by QuickCheck?

- trivial - counts the trivial cases
- classify - classifies the test cases into groups (empty list; one element list, rest)

- collect - classify based on function (eg length of list) (histogram of testcases)

16. Working with QuickCheck you sometimes get messages like Arguments exhausted after 68 tests. Why? What does it mean? Is a QuickCheck terminating with such a message useless?

- That means that quickcheck has generated all defined number of tests (1000) but only 68 fulfilled the precondition and the goal of 100 tests was not reached
- That indicates that a self written generator may be the better option

17. Give an example, where a QuickCheck run is likely to terminate with such a message.

- precondition: list must be ordered and from length 100

18. QuickCheck supports random testing, i.e., testing with randomly generated test data. Richard Hamlet investigated the efficacy of random testing in the 1990s. What were some of his key findings?

- useful
- cheap

19. QuickCheck has been presented in 2000. Is it still unmodified since then? If not, name a few examples of changes.

No, there where changes

- Trivial -> cover

Interesting:

- Quickcheck simple predicate example:

  Define inside of a Haskell program the (predicate) property:

  ```
  prop_PlusAssociative :: Int -> Int -> Int -> Bool
  prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
  ```

  Double-checking prop_PlusAssociative with Hugs yields:

  ```
  Main>quickCheck prop_PlusAssociative
  OK, passed 100 tests
  ```

  -
- type signatures are needed to generate the test data
- QuickCheck allows preconditions
  - example:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
  = is_ordered xs ==> is_ordered (insert x xs)
```

-
- is_ordered is the precondition; test data that do not fulfil the precondition are discarded
- ==> selection of test data

- forAll operator
  - Example

```
prop_InsertOrdered x =
 forAll orderedLists $ \xs -> is_ordered (insert x xs)
```

generates randomly a set of sorted lists
tested to satisfy : `is_ordered (insert x xs)`

  -
  - this helps to only create meaningful test data instead of discarding unusable one
- multiple properties can be tested at once

- choose - take random value from list n to m (choose n m)
- arbitrary - generate arbitrary value
- oneof - randomly use one of the given generators
- frequency - can set a weight to the selection of the generators (biased selection)
- One can make default generator by defining the type as a new type and writing a generator for it. For that the type needs to be instance of arbitrary
-

Part III, Chapter 6 'Verification'

6. Give one statement each where
(a) natural induction - gaußsche summenformel beweisen
(b) strong induction - fibonacci function equal to other formula
(c) structural induction - proving stuff for trees
seems (most) appropriate for proving it.

7. What is the principle of natural induction? Explain the various parts it is composed of.

Let $\mathbb{N}$ be the set of natural numbers, and P be a property of natural numbers.

The Principle of Natural (or: Mathematical) Induction

$$\underbrace{P(1)}_{\text{Base Case}} \wedge [\overbrace{\forall n \in \mathbb{N}.\ \underbrace{P(n)}_{\text{Induction Hypothesis}} \Rightarrow P(n+1)}^{\text{Inductive Case}}] \Rightarrow \underbrace{\forall n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

Induction Step

8. Prove:
∀ n ∈ IN. Xn k=1k2 =n(n + 1)(2n + 1)/6

9. What is the German term for strong induction?
verallgemeinerte Induktion

10. What is Agda? What is it good for?
advanced programming language based on type theory - verified functional programming

11. What are partial lists? What are they good for?
finite sequences of values built from the undefined list
good for: so that one can work even if some parts did not terminate

12. What are proof principles at our disposal for proving properties of defined lists?

- structural induction
- natural induction
- strong induction

13. What changes, adaptions are required in these proof principles if lists may contain undefined values?

C) Proof pattern for lists and partial lists w/ possibly undefined values:

1. Base case: Prove that $P(\bot)$ and $P([\,])$ are true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\bot : \mathtt{xs})$ and $P(\mathtt{x} : \mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

14. Explain why the value of an expression can be undefined.
it needs infinite time to evaluate for example

15. What are proof principles at our disposal for proving equality of streams? How does one proceed when using these principles to show the equality of two streams?

- structured induction
  - approximants / approximants sets
  - reduce proving the equality of streams to proving the equality of sets and equivalent statements amenable to mathematical inductin
- coinduction
  - prove that heads of the streams are the same
  - prove their tails exhibit the same observational behaviour

18. What is a predicate? What is an admissible predicate?

Let $(C, \sqsubseteq)$ be a complete partial order (CPO) (or: domain), and ψ be a predicate on C, i.e., $\psi : C \to \mathbb{B}$.

Definition 6.6.1 (Admissible Predicate)

ψ is called admissible iff for every chain $D \subseteq C$ holds:

$$(\forall\, d \in D.\ \psi(d)) \ \Rightarrow\ \psi(\bigsqcup D)$$

Lemma 6.6.2

ψ is admissible, if it is expressible as an equation.

19. What is the relevance of admissible predicates for fixed point induction?

Interesting:

- Induction types
  - Natural induktion - vollständige Induktion
  - Strong induction - verallgemeinerte Induktion

The Principle of Strong Induction

$$\forall n \in \mathbb{N}.\ [\overbrace{(\forall m < n.\ P(m))}^{\text{Induction Hypothesis}} \ \overbrace{\Rightarrow P(n)}^{\text{Induction Step}}]^{\text{(Inductive) Case}} \ \Rightarrow\ \underbrace{\forall n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der verallgemeinerten Induktion)

  - Structural induction - strukturelle Induktion

The Principle of Structural Induction

$$\forall s \in S.\ [\overbrace{(\forall s' \in sub(s).\ P(s'))}^{\text{Induction Hypothesis}} \underbrace{\Rightarrow P(s)}_{\text{Induction Step}}] \Rightarrow \underbrace{\forall s \in S.\ P(s)}_{\text{Conclusion}}$$

(Inductive) Case

(dtsch. Prinzip der strukturellen Induktion)

- Coinduction
  - proving equality of infinite objects such as streams
- Fixed point induction
  - prove properties of the least fixed point of continuous functions

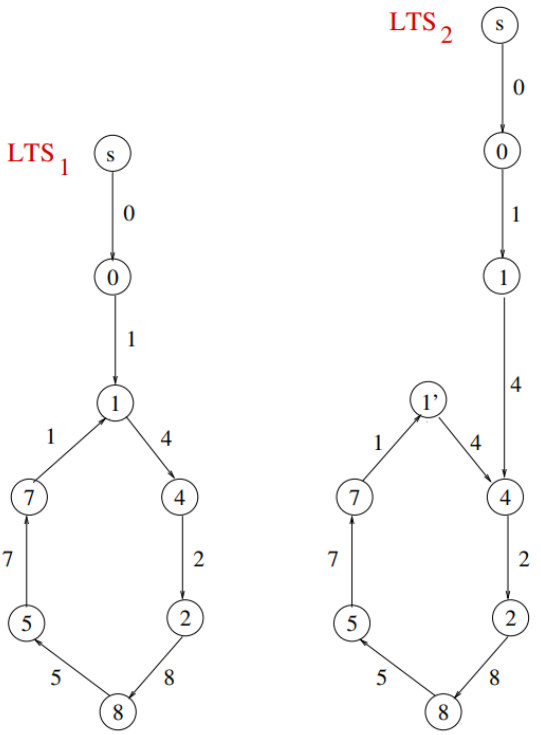
- Induction and recursion closely related
  - induction = starting form something simple and building up from there - bottom-up
  - recursion = starting from the whole thing and working backward to the simple cases - top-down
  - data structures often follow an inductive definition pattern while algorithms on datay structures mostly follow recursive definition pattern
- In haskell there is a data type Undefined - $\bot$ :: a for faulty or non-terminating computations; can be considered an approximation of any ordinary value of a data type
- Labelled Transition System

Definition 6.5.2.2 (Labeled Transition System)

A labelled transition system (LTS) is a tripel (Q, A, T) with

- Q a set of states.
- A a set of action labels.
- $T \subseteq Q \times A \times Q$ a ternary transition relation.

  -



- Verified programming
  - Correctness by construction
    - inductive proof principles
    - equational reasoning
    - use transformation rules
    - e.g. Functional pearls
  - Provers / proof assitents
    - Zeno, Leon, MoChiX, Catch (lazy haskell), agda

Part V, Chapter 15 ‘Parsing’

1. What is parsing concerned with?
lexical and syntactical analysis of structure of text (e.g. source code text of a program)
Parsing problem: read sequence of objects of type a -> yield object of a type b

2. What is the type of a parser (combinatory) ?

```
                output type
type Parse a b = [a] -> [(b,[a])]
                 input type
```

with [a] = left-over string from input

b is the output

Example:

```
Parser  Input       Output
bracket "(xyz" ->> [('(',"xyz")]
number  "234"  ->> [(2,"34"),(23,"4"), (234,"")]
bracket "234"  ->> []
```

empty list signals failure; non-empty success

4. When do we speak of a universal parser basis?
combinator library for parsing composed of

- 4 primitive parser functions
  - input-independent - none and succeed
  - input-dependent - token and spot
- 3 parser combinators
  - alternatives - alt
  - sequencing - (>*>)
  - transforming (build)

allow t build any parser

5. What parsers constitute the combinator parser basis of Chapter 15.2?

1. none, the always failing parser

```
none :: Parse a b
none _ = []
```

2. succeed, the always succeeding parser

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

3. token, the parser recognizing single objects (so-called tokens):

```
token :: Eq a => a -> Parse a a
token t (x:xs)
 | t == x      =  [(t,xs)]
 | otherwise   =  []
token t []     =  []
```

4. spot, the parser recognizing single objects enjoying some property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
 | p x         =  [(x,xs)]
 | otherwise   =  []
spot p []      =  []
```

6. Do the monadic parsers of Chapter 15.3 a parser basis, too?

```
newtype Parser a = Parse (String -> [(a,String)])
                   output type    input type
```

yes, there is one:

- counterpart for spot:

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

- counterpart for token

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

- mbuild

7. What could be arguments in favour of combinator parsing, what in favour of monadic parsing?

Combinator parsing

- Advantages
    - free choice of parser and combinator names (monadic only free choice for char, sat, mbuild)
    - input and output type of parsers both polymorphic

Monadic parsing
- do notation for sequencing parsers

8. Are all parsers used by monadic parsing truely monadic operations?
no char, sat and mbuild are not

9. Which parsers are truely monadic operations? Which ones are not?
mzero, return, mplus, (>>=)

10. What is recursive descent parsing?
parsers can be constructed from other parsers; you start at the toplevel parser and go down to the more specific parsers

13. What is the purpose of the always succeeding parser?
representing the symbol ε (the empty word)

14. Why is it good to have the build parser?
it can be used to build types from the parsed input for example

15. The monad-plus operations of the parser instance represent two parsers. Which ones?
mplus - alternatives
mzero - always failing

17. The type classes Show and Read are related to the parsing problem. Why? In what respect?
read is a combinatorial parser (has signature of a parser)
show - reverse function of read

18. What is the 'white space' problem of parsing?
white spaces can be put, in front, between, after everything and there can be multiple of them; they need to be handled accordingly

21. If the general parser type is a two-ary type constructor and monads are on-ary type constructor, how does this go along with monadic parsing?
we bind the input to be a string and hence getting a 1ary type constructor

22. Can you pairwise oppose the parsers of the combinator and the monadic parser approach?

	Combinator Parsing	Monadic Parsing
Primitive Parsers	none succeed token spot	mzero return char sat
Parser Combinators	alt (>*>) build	mplus (>>=) mbuild

Note: `mzero`, `return`, `mplus`, `(>>=)` are monad and monad-plus operations, respectively; `char`, `sat`, and `mbuild` are not!

23. Parsing is sometimes (or even often) considered a show-case for the elegance of functional programming. Can you explain why?
combinators are super useful; not much code is needed

Interesting

- two different approaches - both well-suited for building recursive descent parsers
  - combinator parsing
    - higher-order functions for parsing
  - monadic parsing
    - monadic parser combinators
- Parser combinators
  - are higher-order polymorphic functions

1. `alt`, the parser combining parsers as alternatives:

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 input = p1 input ++ p2 input
```

2. `(>*>)`, the parser combining parsers sequentially:

```
infixr 5 >*>
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 input
  = [((y,z),rem2) | (y,rem1) <- p1 input,
                    (z,rem2) <- p2 rem1]
```

3. build, the parser transforming obtained parse results:

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f input = [(f x,rem) | (x,rem) <- p input]
```

- Structure of combinator parsers
  - none, succeed, token, spot, alt, (>*>), build, list, topLevel
    - topLevel evaluates parsing result; e.g. if there is an error -> raise error
- Combinator parsers are well-suited for recursive descent parsers
  - are structured similar to BNF (bachus naur form)
  - provide for every operator of BNF-grammar a corresponding parser function
- Monadic parsers
  - return (success) and (>>=) (sequencing) already represent two primitive parser
  - monadPlus additionally mzero (failing) and mplus (alternatives)
- Important for parsers
  - higher-order functions
  - polymorphism
  - lazy evaluation

Part V, Chapter 16 ‘Logical Programming Functionally’

1. Functional and logic programming are representatives of the same programming paradigm. Which one?
declarative programming

2. What are characteristics of this paradigm? What is its essence?

- characterizing: programs are declarative assertions about a problem rather than imperative solution procedures
  - properties of a program are defined - what
  - instead of how

3. What is the rôle, the relevance of abstraction in the evolution of programming languages?

- hiding computer hardware and the details of program execution become more and more import as hardware and program execution get more complicated (e.g. The von Neumann Architecture)
- declarative programming would not be possible without abstraction (assembly?!)

5. Functional and logic programming (languages) belong to the same programming paradigm but are built upon different concepts and foundations. Can you explain them in more detail?

- Functional
  - based on mathematical functions

    - programs are sets of functions that operate on data structures and are defined by equations using distinction and recursion
    - efficient demand-driven evaluation strategies
- Logic
    - based on predicate logic
    - programs are sets of predicats defined by restricted forms of logic formulas (e.g. horn clauses)
    - provide non-determinism and predicates with multiple input/output modes for code reuse

6. What are key problems to be solved when extending a functional programming language towards enabling (to some extent) logic programming?

- logic programs yielding multiple answers -> using lists of successes technique
    - lists of successes = lazy lists; results are provided as they are found using lazy evaluation
- evaluation / search strategy inherent to logic programs -> encapsulating the search strategy in search monads
- logical variables (no distinction between input and output vars) -> realizing unification

7. Diagonalization and diagonalization based reasoning is an important proof technique but is also important for coping with infinite search spaces as often occur in logic programming problems. Explain the idea of diagonalization in the context of search space exploration in more detail.

	1	2	3	4	5	6	7	…
1	$(1,1)_1$	$(1,2)_2$	$(1,3)_4$	$(1,4)_7$	$(1,5)_{11}$	$(1,6)_{16}$	$(1,7)_{22}$	…
2	$(2,1)_3$	$(2,2)_5$	$(2,3)_8$	$(2,4)_{12}$	$(2,5)_{17}$	$(2,6)_{23}$	$(2,7)_{30}$	…
3	$(3,1)_6$	$(3,2)_9$	$(3,3)_{13}$	$(3,4)_{18}$	$(3,5)_{24}$	$(3,6)_{31}$	$(3,7)_{39}$	…
4	$(4,1)_{10}$	$(4,2)_{14}$	$(4,3)_{19}$	$(4,4)_{25}$	$(4,5)_{32}$	$(4,6)_{40}$	$(4,7)_{49}$	…
5	$(5,1)_{15}$	$(5,2)_{20}$	$(5,3)_{26}$	$(5,4)_{33}$	$(5,5)_{41}$	$(5,6)_{50}$	$(5,7)_{60}$	…
6	$(6,1)_{21}$	$(6,2)_{27}$	$(6,3)_{34}$	$(6,4)_{42}$	$(6,5)_{51}$	$(6,6)_{61}$	$(6,7)_{72}$	…
7	$(7,1)_{28}$	$(7,2)_{35}$	$(7,3)_{43}$	$(7,4)_{52}$	$(7,5)_{62}$	$(7,6)_{73}$	$(7,7)_{85}$	…
8	$(8,1)_{36}$	$(8,2)_{44}$	$(8,3)_{53}$	$(8,4)_{63}$	$(8,5)_{74}$	$(8,6)_{86}$	$(8,7)_{99}$	…
9	$(9,1)_{45}$	$(9,2)_{54}$	$(9,3)_{64}$	$(9,4)_{75}$	$(9,5)_{87}$	$(9,6)_{100}$	$(9,7)_{114}$	…
…	…	…	…	…	…	…	…	…

avoiding unfair search order in infinite search spaces

9. The approach for logic programming functionally of Chapter 16 makes intensive use of monads. To which purpose?

Uses a bunch and a diag, matrix,... monad so that the do-notation can be used

10. If solutions occur rarely in huge search spaces, it can be difficult for a user to distinguish an active program just not finding a(nother) solution for some time from a hanging one. How can this problem be dealt with?

by indicating search progress as we go

Interesting

- Functional logic languages
  - Examples: Curry, Toy, mercury, escher, oz, HAL
- Curry
  - operators
    - ? - nondeterministic choice
    - =:= - equation is to be solved rather than an operation to be defined

Example: Regular expressions and their semantics

```
data RE a = Lit a
          | Alt (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)

sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt r s)  = sem r ? sem s
sem (Conc r s) = sem r ++ sem s
sem (Star r)   = [] ? sem (Conc r (Star r))
```

  -
- Handling logical variables in functional languages
  - introduce type (e.g. term)
  - add utility functions for transforming e.g. strings, list of integers,... in terms
  - add type for substitutions (mappings from variables to terms)
  - add functions for substitutions e.g. created a list of mapping tuples,....
  - introduce type for predicates (type)
  - introduce type for answers (newtype)
- Combinators for Logic Programs
  - =:= - Equality - implemented as a unify, unify can be done if they are the same or they are variables that can be instantiated so that the resulting term is equal
  - &&& - Conjunction - implemented as a bind operation (>>=)
  - ||| - Disjunction - implemented as an alt operation
  - exists - existential quantificat - allows introducing new variables in predicates

Part V, Chapter 17 ‘Pretty Printing’

1. Pretty printing and parsing are closely linked to each other. Why? In what respect?
pretty printing is reversing the parsing step; that is making values of types (or tree-like structures) into text

2. What are important design goals and features of a pretty printer?

- ease of use
- flexibility of layout
- beauty of output

- the code of the printer itself should also be pretty

3. Explain and illustrate the idea of pretty printing by means of some examples. – The prettier printer of Philip Wadler is another fine example of

- monadic programming.
- abstract data type programming
- stream programming.
- algorithm pattern programming.
- generate/prune programming.
- lawful programming.

4. The prettier printer of Philip Wadler shall improve on a pretty printer by John Hughes, often considered the 'standard pretty printer of functional programming.' In what respect?

1. simple pretty printer (cf. Chapter 17.2)
   - implements `Doc` as strings.
   - supports for every document only one possible layout, in particular, no attempt is made to compress structure onto a single line.
2. prettier printer (cf. Chapter 17.3)
   - implements `Doc` in terms of suitbable algebraic sum data types.
   - allows multiple layouts of a document and to pick a best one out of them for printing a document.

Interesting:

- pretty printer
  - basic document operators

```
Associative operator for concatenating documents:
 (<>) :: Doc -> Doc -> Doc
The empty document being a right and left unit for (<>):
 nil :: Doc
Converting a string into a document (arguments of
function text shall not contain newline characters):
 text :: String -> Doc
The document representing a line break:
 line :: Doc
Adding indentation to a document:
 nest :: Int -> Doc -> Doc
Layouting a document as a string:
 layout :: Doc -> String
```

- The prettier printer
  - algebraic documents
    - document is a concatenation of items where each item is a text or a line break indented a given amount
    - documents are implemented as an algebraic sum data type

```
data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
```

  - multiple layouts
    - document should be considered equivalent to a set of strings
    - group operator needed
      - uses flatten (line break -> single space) and <|> (union of two sets of layouts)
        - follow distributive laws
        - union can only be applied to two layouts that result in the same layout when flattened
      - everything is compressed on one line by replacing each newline with text consisting of a single space; but the newline could also be replaced by alternate text e.g. empty text, symbols,...
      - is only used for line not for string in texts
  - setting preferred maximum line width
    - using pretty operator instead of layout (has int for max line width)
  - best operator to find best layout - converts a union-afflicted document into a union-free document
  - fits operator - checks if the first document line stays within the maximum line length
  - utility functions
    - separating documents with space, line break; folding a document,...
- performance of pretty printing

- should be doable in time O(s) .. s is size of the document
- and in space O(w max d) w width for printing, d depth of document
- inefficiency sources
    - document concatenation
        - solution: add explicit representation for concatenation and generalizing each operation to act on a list of concatenated documents
    - nesting of documents
        - solution: add explicit representation for nesting - maintaining a current indentation that is increment as nesting operators are processes
    - combined solution: indentation-document pairs

Part V, Chapter 18 ‘Functional Reactive Programming’

Hybrid systems

- are composed of continuous and discrete components.
- can be also considered cyber-physical systems.

Functional reactive programming as presented in Chapter 19 is another fine example of

- monadic programming

4. abstract data type programming.
5. stream programming.
    - functorial programming.
7. algorithm pattern programming.
    - arrow programming.
9. generate/prune programming.
10. lawful programming.

Right or wrong?

What are examples of continuous and discrete physical components

- continuous components: voltage-controlled motors, batteries, range finders
- discrete components: microprocessors, bumper switches, digital communication

logical notions

- continuos notions: wheel speed, orientation, distance from a wall
- discrete notions: running into another object, receiving a message, achieving a goal

What are examples of continuous and discrete often occurring in reactive systems?

robot systems, cyber-physical systems

What can be said about the origins of functional reactive programming?

functional reactive animation

Part VI, Chapter 19 ‘Parallel and ‘Real World’ Functional Programming’

What kinds of parallelism can be distinguished, are predominant in

- imperative programming?
    - libraries, message passing model, data-parellel languages
- functional programming?
    - implicit expression parallelism
        - if f(e1,...,en) is a functional expression then arguments can be evaluated in parallel
        - parallelism for free
        - results often unsatisfying, e.g. granularrity, load distribution,..
    - explicit parallelism
        - using meta-statements
        - often very good results
        - high programming effort and loss of functional elegance
    - algorithmic skeletons
        - compromise between the two above
        - represent typical patterns for parallization (farm, map, reduce,...)
        - easy to instantiate
        - high level of abstraction (hiding parallelism in skeleton)
        - skeletons on distributed data structures are data-parallel skeletons

5. What are important examples of algorithmic skeletons?

- map
- farm
- reduce
- branch & bound
- divide & conquer

6. Functional programs shall be rich of implicit parallelism. Why?
because it is side-effect free and parallelism is free if defined that way

9. Why is explicit parallelism often considered inadequate for functional programming?
Because it destroys the elegance of functional programming, we need metastatements that come with side effects