Advanced Programming Paradigms

N. Kälin

January 29, 2021

Contents

1 Introduction

1.1 Programming Paradigms

Paradigm: (definitions from Merriam-Webster)

- a theory or group of ideas about how something should be done, made, or thought about
- example, pattern; especially: an outstandingly clear or typical example or archetype (a perfect example of something)

Programmin Paradigm: fundamental style of programming

- In which notions do we think about a program?
- Which aspects can be **explicitly** described, which cannot?
  - state
  - concurrency and parallelism
  - nondeterminism

Software quality: (according to Bertrand Meyer)

- reliability
  - **correctness**
  - robustness
- modularity
  - extendibility
  - reusability
- compatibility, efficiency, portability, ease of use, timeliness

1.1.1 Imperative Programming

- based on explicitly reading and updating **state**
- immediate abstraction of von Neumann computer
- theoretical base: *Turing* machine
- concepts:
  - data structures: variables, records, arrays, pointers
  - computation:
    - expressions:
      literal, identifier, operation, function call
    - commands (instructions, "statements"):
      assignment, composition, goto, conditional, loop, procedure call
  - abstraction: functions, procedures
- examples: Ada, Algol, C, Cobol, Fortran, Moudla, Pascal

1.1.2 Object-Oriented Programming

- strongly based on imperative paradigm
- further support for abstraction and modularization
  - Abstract Data Types (ADTs)
  - information hiding
- concepts:
  - objects as instances of classes: data + procedures put together
  - encapsulation (private, protected, public)
  - inheritance for modularity and for variant records
  - subtyping, polymorphism and dynamic binding
  - genericity (from some imperative and most functional languages)
- examples: C++, C#, Eiffer, Java, Objective-C, Simula, Smalltalk

1.1.3 Functional Programming

- based on λ-calculus and reduction
- subexpressions are replaced by simpler, but equivalent subexpressions until no longer possible
- concepts:
  - no state, no commands; just expression
  - identifiers denote values, not variables (storage cells)
  - no commands implies no loops; just recursion
  - functions: recursive, anonymous, curried, higher-order (DSLs)
  - recursive algebraic data types and pattern matching
  - polymorphic and overloaded types
  - type inference
  - eager or lazy evaluation
  - simple equational reasoning about programs
- examples: F#, Haskell, Lisp, ML, OCaml

1.1.4 Logic Programming

- based on first-order logic (predicate logic)
- logical formulas express relations declaratively
- machine solves formulas through resolution
- works for specialized formulas like *HORN* clauses
- efficient only if programmer guides the solution process
- example: Prolog

1.1.5 Further Programming Paradigms

- constraint programming
- concurrent programming
- parallel programming

1.1.6 Multiparadigm Programming

- several paradigms can be combined into a single language
- each paradigm has its realm; today's large applications embrace many such realms; a single language simplifies interoperability
- examples:
  - functional with imperative features: ML
  - object-oriented with functional features: C#
  - functional with object-oriented features: F#, OCaml
  - functional + object-oriented: Scala
  - functional + logic: Curry (based on Haskell)

1.2 Correctness and Verification

1.2.1 Correctness

- prime quality, *conditio sine qua non*
- relative notion: program should be correct with respect to its **specification**
  - example: program that computes the sine perfectly well but should compute the root is clearly not correct
- but how can one know whether a program is correct or not?
  - by *testing*, one can find faults (bugs)
  - by *proving*, one can show the absence of faults

1.2.2 Testing versus Proving

better: Tests **and** Proofs

- testing
  - choose particular input
  - determine correct result for that input using test oracle
  - run program under test on the chosen input
  - compare obtained and correct result
    - if different: fault found
    - if equal: no relevant information obtained
- proving
  - do **not** choose a particular input
  - do **not** execute the program
  - instead apply mathematical rules to program and specification

1.2.3 Verification As a Matter Of Course (VAMOC)

(according to Bertrand Meyer)

- software controls more and more of our daily lives
- software becomes more and more complicated
- testing does not suffice; verification is needed in addition
- verification tools become more and more powerful
- examples: Spec# and Dafny for specification and verification of object-oriented programs

1.2.4 Types

- 'good' expressions can be typed at compile time
- ill-typed expressions will not compile
- thus corresponding run-time errors cannot occur
- type checking and inference is mostly fully automatic
- light-weight formal method
- first step towards program verification

2 Functional Programming

2.1 Correctness

(see 1.2.1)

2.1.1 Obtaining Mathematical Knowledge

1. Conjecture
   The product of all prime numbers between and including 2 and p, increased by 1, is again a prime number.

2. Examples
   For $p = 2, 3, 5, 7, 11, 379$ the conjecture is confirmed.

3. Counterexample
   For $p = 17$ the conjecture is refuted.

1. Theorem
   $(a + b)^2 = a^2 + 2ab + b^2$

2. Proof
   $(a + b)^2 = (a + b)(a + b) = a(a + b) + b(a + b) = aa+ab+ba+bb = aa+ab+ab+bb = aa+2ab+bb = a^2 + 2ab + b^2$

   with a **finite** number of steps we have thus shown that something holds for an **infinite** number of values

2.1.2 Consequence

- programming languages should simplify proofs (and therefore also program development itself)
- and thus may enhance program reliability

2.2 Referential Transparency

2.2.1 A More Formal Proof

$$
\begin{aligned}
& (a+b)^2 \\
=&\{\text{def square}\} \\
& (a+b)\cdot(a+b) \\
=&\{\text{distri}\} \\
& a\cdot(a+b)+b\cdot(a+b) \\
=&\{\text{distri twice}\} \\
& a\cdot a+a\cdot b+b\cdot a+b\cdot b \\
=&\{\text{commu multi}\} \\
& a\cdot a+a\cdot b+a\cdot b+b\cdot b \\
=&\{\text{neutral multi twice}\} \\
& a\cdot a+1\cdot(a\cdot b)+1\cdot(a\cdot b)+b\cdot b \\
=&\{\text{distri}\} \\
& a\cdot a+(1+1)\cdot(a\cdot b)+b\cdot b \\
=&\{\text{def 2}\} \\
& a\cdot a+2\cdot(a\cdot b)+b\cdot b \\
=&\{\text{def square twice}\} \\
& a^2+2ab+b^2
\end{aligned}
$$

- this proof sill handles associativity implicitly
- this format for *calculational proofs* is due to FEIJEN and DIJKSTRA
- a corresponding `calc` statement is available in Dafny

2.2.2 Equality

A fundamental mathematical concept

- four inference rules of a logic
- Reflexivity: $\frac{}{X=X}$
- Symmetry: $\frac{X=Y}{Y=X}$
- Transitivity: $\frac{X=Y, Y=Z}{X=Z}$
- LEIBNIZ: $\frac{X=Y}{E[v\leftarrow X]=E[v\leftarrow Y]}$
- X, Y, Z, E: expressions, v: variable, $E[v \leftarrow X]$: textual substitution of all (free) occurrences of v by (X) in E

2.2.3 Example LEIBNIZ

- from numbers: $x \cdot (y + z) = x \cdot y + x \cdot z$
- therefore, by LEIBNIZ (and Substitution):

$$\underbrace{(a \cdot (a+b))}+b \cdot (a+b) \quad (1)$$

$=\langle$LEIBINIZ, with $a \cdot (a+b) = a \cdot a + a \cdot b\rangle$

$$\overbrace{(a \cdot a + a \cdot b)} + \underbrace{(b \cdot +(a+b))} \quad (2)$$

$=\langle$LEIBNIZ, with $b \cdot (a+b) = b \cdot a + b \cdot b\rangle$

$$a \cdot a + a \cdot b + \overbrace{b \cdot a + b \cdot b} \quad (3)$$

- therefore, since (1) = (2) and (2) = (3), by Transitivity: (1) = (3)

2.2.4 Referential Transparency

three synonymous terms
- LEIBNIZ
- substitution of equals for equals
- referential transparency

2.2.5 Functional Program

- a functional program consists of
  1. a set of value and function declarations
  2. a single expression
- functional programming is referentially transparent
  - values and functions are declared via equality
  - equality then means mathematical equality (if using eager evaluation modulo termination)
- referential transparency employed for
  - program development, transformation, and proof
  - evaluation

2.2.6 Program Transformation

- to transform a program means to rewrite it according to given rules into an equivalent program
- Example:
  - with declaration $x = f(a)$ and arithmetic $x + x = 2 \cdot x$, expression $x + x$ can be safely rewritten into either of
    - $2 \cdot x$
    - $f(a) + x$
    - $x + f(a)$
    - $f(a) + f(a)$
    - $2 \cdot f(a)$

2.2.7 Evaluation

- execution of a program means evaluation of the expression
- Example:
  - declarations: $f(x) = 2 \cdot x + 1, a = 3$
  - expressions: $a + f(a)$
  - evaluation:

$$\begin{aligned} & a + f(a) \\ =& a + (2 \cdot a + 1) \\ =& 3 + (2 \cdot 3 + 1) \\ =& 3 + (6 + 1) \\ =& 3 + 7 \\ =& 10 \end{aligned}$$

- order of evaluation has no influence on result (modulo termination)

2.3 Imperative Programming

- Example:

```
y := 0; a := 3;
.
.
.
function f(x) begin y := y + 1;
   return x + y end
```

- execution:
  - $f(a) + f(a)$ returns $4 + 5 = 9$
  - $2 \cdot f(a)$ returns $2 \cdot 4 = 8$
- no referential transparency: even the most basic arithmetic cannot be performed
- syntax: expressions + commands
- semantics: values + environment + state
- expressions are *evaluated* in the environment and current state, yielding a value
- commands are *executed* in the environment and current state, yielding a new state
- Example:
  - assignment command with variable v and Expression E v := E
  - E is evaluated in the environment and current state, yielding value t; then t is assigned to the storage cell denoted by v in the environment, thus yielding a new state
- proofs of imperative programs are well possible too, but are by far more complicated
- possible using HOARE logic
- HOARE triple, with P, Q predicates and C command $\{P\}C\{Q\}$
- means: if execution of C starts in a state satisfying P, and execution terminates, then the resulting state satisfies Q
- Example:

  - proof rule for assignment command $v := E$ $\{Q[v \leftarrow E]\}v := E\{Q\}$

2.3.1 Progress in Programming Languages

- by adding features
  - expressions
  - procedures, functions
  - types
  - data structures
  - abstract data types
- by removing features
  - gotos
  - pointers
  - **state and assignment**

2.3.2 Imperative versus Functional Programming

- imperative paradigm
  - syntax: expressions + commands
  - semantics: values + environment + state
  - expressions are *evaluated* in the environment and current state, yielding a value
  - commands are *executed* in the environment and current state, yielding a new state
- functional paradigm
  - syntax: expressions
  - semantics: values + environment
  - expressions are *evaluated* in the environment, yielding a value

2.3.3 Misuse of the Symbol for Equality =

- assignment like $x := x + 1$ has not the slightest similarity to equality
- it is pronounced "x becomes (gets, receives) $x+1$" ...
- ... but **never ever** "x equals (is, is equal to) $x+1$"
- a different symbol like := of $\leftarrow$ should be used instead
- using the symbol for equality = to denote assignment is a horrendous design error of too many programming languages, since
  - by our very basic education, it is virtually impossible to see = and to not think of equality
  - equality is such a fundamental concept that it deserves a unique non-overloaded symbol

2.4 Evaluation Strategies

2.4.1 Evaluation

- strategies
  - innermost (call-by-value)
  - outermost (call-by-name)
  - lazy (outermost + sharing)
- reducible expressions, or *redex*
  - application of a function to its argument expressions
- Example: $\text{mult}(x, y) = x \cdot y$
- $\text{mult}(1 + 2, 2 + 3)$ has three redexes
  - $1 + 2$, yielding $\text{mult}(3, 2 + 3)$
  - $2 + 3$, yielding $\text{mult}(1 + 2, 5)$
  - $\text{mult}(1 + 2, 2 + 3)$, yielding $(1 + 2) \cdot (2 + 3)$

innermost	**outermost**
innermost redex first; if several, choose leftmost one first	outermost redex first; if several, choose leftmost one first
$\text{mult}(1 + 2, 2 + 3)$ $= \text{mult}(3, 2 + 3)$ $= \text{mult}(3, 5)$ $= 3 \cdot 5$ $= 15$	$\text{mult}(1 + 2, 2 + 3)$ $= (1 + 2) \cdot (2 + 3)$ $= 3 \cdot (2 + 3)$ $= 3 \cdot 5$ $= 15$

- Example: $\text{square}(x) = x \cdot x$
- innermost:

$$\begin{aligned} &\text{square}(1 + 2) \\ =&\text{square}(3) \\ =&3 \cdot 3 \\ =&9 \end{aligned}$$

- with innermost evaluation, each argument is evaluated exactly once
- outermost:

$$\begin{aligned} &\text{square}(1 + 2) \\ =&(1 + 2) \cdot (1 + 2) \\ =&3 \cdot (1 + 2) \\ =&3 \cdot 3 \\ =&9 \end{aligned}$$

- argument expressions might be evaluated more than once if the corresponding formal parameters occur several times in the body of the function
- solution to this problem via sharing:
  - keep only a single copy of the argument expression, and maintain a pointer to it for each corresponding formal parameter
  - evaluate the expression once, and replace it by its value
  - access this value through the pointers

2.4.2 Evaluation

- Example:

1. $f(x) = 17$
2. $\inf(x) = \inf(x)$

- inf(0) obviously yields an endless recursion
- What is $f(\inf(0))$?
- What is $f(1\text{div}0)$?
- innermost:
  - $f(\inf(0))$ yields an endless recursion
  - $f(1\text{div}0)$ aborts
- outermost (and thus lazy):
  - $f(\inf(0))$ yields 17
  - $f(1\text{div}0)$ yields 17
- an argmument is evaluated
  - innermost: exactly once
  - outermost: zero or more times
  - lazy: at most once
- whenever there exists an order of evaluation that terminates, outermost (and thus lazy) evaluation will find it

3 Programming in Haskell

3.1 First Steps

3.1.1 List functions

input	output
head [1,2,3,4,5]	1
tail [1,2,3,4,5]	[2,3,4,5]
[1,2,3,4,5] !! 2	3
take 3 [1,2,3,4,5]	[1,2,3]
drop 3 [1,2,3,4,5]	[4,5]
length [1,2,3,4,5]	5
sum [1,2,3,4,5]	15
product [1,2,3,4,5]	120
[1,2,3] ++ [4,5]	[1,2,3,4,5]
reverse [1,2,3,4,5]	[5,4,3,2,1]

3.1.2 Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space:
$f(a, b) + cd$
In Haskell, function application is denoted using space, and multiplication is denoted using $*$:
f a b + c*d
Moreover, function application is assumed to have higher priority than all other operators:

f a + b	(f a) + b, **not** f(a + b)

Examples:

Mathematics	Haskell
$f(x)$	f x
$f(x, y)$	f x y
$f(g(x))$	f (g x)
$f(x, g(y))$	f x (g y)
$f(x)g(y)$	f x * g y

3.1.3 Haskell Scripts

- As well as the functions in the standard library, you can also define your own functions
- New functions are defined within a script, a text file comprising a sequence of definitions
- By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes.

3.1.4 My First Script

```
double x = x + x
quadruple x =
  double (double x
  )
```

```
factorial n =
  product [1..n]
average ns = sum ns
  `div` length ns
```

Note:

- **div** is enclosed in back quotes, not forward
- x 'f' y is just syntactic sugar for f x y.

To start up GHCi with the script, type the following in a terminal:

```
$ ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"
```

3.1.5 Useful GHCi Commands

Command	Meaning
:load name	load script name
:reload	reload current script
:**set** editor name	set editor to name
:edit name	edit script name
:edit	edit current script
:**type expr**	show type of **expr**
:?	show all commands
:quit	quit GHCi

3.1.6 Naming Requirements

- Function and argument names must begin with a lower-case letter:
  myFun, fun1, arg_2, x'
- By convention, list arguments usually have an s suffix on their name:
  xs, ns, nss

3.1.7 The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

correct:

```
a = 10
b = 20
c = 30
```

wrong:

```
a = 10
 b = 20
c = 30
```

wrong:

```
 a = 10
b = 20
 c = 30
```

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

implicit grouping:

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

explicit grouping:

```
a = b + c
    where
    {b = 1;
     c = 2}
d = a * 2
```

3.2 Types and Classes

3.2.1 What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type **Bool** contains the two logical values **False** and **True**.

3.2.2 Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.

```
> 1 + False
error ...
```

1 is a number and **False** is a logical value, but + requires two numbers.

3.2.3 Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written e :: t
- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.
- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at runtime.
- In GHCi, the :**type** command calculates the type of an expression, without evaluating it:

```
> not False
True
> :type not False
not False :: Bool
```

3.2.4 Basic Types

Haskell has a number of basic types, including:

Bool	logical values
Char	single characters
String	strings of characters
Int	integer numbers
Float	floating-point numbers

3.2.5 List Types

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
```

In general: [t] is the type of lists with elements of type t.
Note:

- The type of a list says nothing about its length:

```
[False,True] :: [Bool]
[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

3.2.6 Tuple Types

```
(False,True) :: (Bool,Bool)
(False,'a',True) :: (Bool,Char,Bool)
```

In general: (t1,t2 ,..., tn) is the type of n-tuples whose ith components have type ti for any i in 1..n.
Note:

- The type of a tuple encodes its size:

```
(False,True) :: (Bool,Bool)
(False,True,False) :: (Bool,Bool,
    Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,
    Char))
(True,['a','b']) :: (Bool,[Char])
```

3.2.7 Function Types

A function is a mapping from values of one type to values of another type:

```
not :: Bool -> Bool
even :: Int -> Bool
```

In general: t1 -> t2 is the type of functions that map values of type t1 to values of type t2.
Note:

- The arrow -> is typed at the keyboard as ->.
- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using list or tuples:

```
add :: (Int,Int) -> Int
add (x,y) = x+y
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

3.2.8 Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.
Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

  ```
  add :: (Int,Int) -> Int
  add' :: Int -> (Int -> Int)
  ```

- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.
- Functions with more than two arguments can be curried by returning nested functions:

  ```
  mult :: Int -> (Int -> (Int -> Int))
  mult x y z = x*y*z
  ```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

3.2.9 Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.
For example:

```
add' 1 :: Int -> Int
take 5 :: [Int] -> [Int]
drop 5 :: [Int] -> [Int]
```

3.2.10 Currying Conventions

To avoid excess parantheses when using curried functions, two simple conventions are adopted:

- The arrow -> associates to the right.
  Int -> **Int** -> **Int** -> **Int**
  Means **Int** -> (**Int** -> (**Int** -> **Int**)).
- As a consequence, it is then natural for function application to associate to the left. mult x y z
  Means ((mult x) y) z.
  Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

3.2.11 Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] -> Int
```

For any type a, **length** takes a list of values of type a and returns an integer.
Note:

- Type variables can be instantiated to different types in different circumstances:

  ```
  > length [False,True] -- a = Bool
  2
  > length [1,2,3,4] -- a = Int
  4
  ```

- Many of the functions defined in the standard prelude are polymorphic. For example:

  ```
  fst :: (a,b) -> a
  head :: [a] -> a
  take :: Int -> [a] -> [a]
  zip :: [a] -> [b] -> [(a,b)]
  id :: a -> a
  ```

3.2.12 Overloaded Functions

A polymorpic function is called overloaded if its type contains one or more class constraints.

```
(+) :: Num a => a -> a -> a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.
Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

  ```
  > 1 + 2 -- a = Int
  3
  > 1.0 + 2.0 -- a = Float
  3.0
  > 'a' + 'b' -- Char is not a numeric
      type
  ERROR
  ```

- Haskell has a number of type classes, including:

Num	Numeric types
Eq	Equality types
Ord	Ordered types

- For example:

  ```
  (+) :: Num a => a -> a -> a
  (==) :: Eq a => a -> a -> Bool
  (<) :: Ord a => a -> a -> Bool
  ```

3.2.13 Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;
- Within a script, it is good practice to state the type of every new function defined;
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

3.3 Defining Functions

3.3.1 Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and −n otherwise.
Conditional expressions can be nested:

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must always have an **else** branch, which avoids any possible ambiguity problems with nested conditionals.

3.3.2 Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n >= 0     = n
      | otherwise = -n
```

As previously, but using guarded equations.
Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Note:

- The catch all condition **otherwise** is defined in prelude by **otherwise** = **True**.

3.3.3 Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not :: Bool -> Bool
not False = True
not True  = False
```

not maps **False** to **True**, and **True** to **False**.
Functions can often be defined in many different ways using pattern matching. For example:

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is **False**:

```
True  && b = b
False && _ = False
```

Note:

- The underscore symbol _ is a wildcard pattern that matches any argument value.
- Patterns are matched in order. For example, the following definition always returns **False**:

  ```
  _    && _    = False
  True && True = True
  ```

- Patterns may not repeat variables. For example, the following definition gives an error:

  ```
  b && b = b
  _ && _ = False
  ```

3.3.4 List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of the list.

```
[1,2,3,4]
```

Means 1:(2:(3:(4:[]))).
Functions on lists can be defined using x:xa patterns.

```
head :: [a] -> a
head (x:_) = x
tail :: [a] -> [a]
tail (_:xs) = xs
```

head and **tail** map any non-empty list to its first and remaining elements.
Note:

- x:xs patterns only match non-empty lists:

```
> head []
*** Exception: empty list
```

- x:xs patterns must be parenthesised, because application has priority over (:). For example, the following definition gives an error:

```
head x:_ = x
```

3.3.5 Lambda expressions

Functions can be constructed without naming the functions by using lampda expressions.

```
\x -> x + x
```

Note:

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the $\mapsto$ symbol, as in $x \mapsto x + x$.
- In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

3.3.6 Why are Lambda's useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.
For example:

```
add x y = x + y
```

means

```
add = \x -> (\y -> x + y)
```

Lambda expressions can be used to avoid naming functions that are only referenced once.
For example:

```
odds n = map f [0..n-1]
         where
             f x = x*2 + 1
```

can be simplified to

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

3.3.7 Operator Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.
For example:

```
> 1 + 2
3
> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.
For example:

```
> (1+) 2
3
> (+2) 1
3
```

In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called sections.

3.3.8 Why are Sections useful?

Useful functions can sometimes be constructed in a simple way using sections.
For example:

(1+) - successor function
(1/) - reciprocation function
(*2) - doubling function
(/2) - halving function

3.4 List Comprehensions

3.4.1 Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.
$\{x^2 | x \in \{1...5\}\}$ The set $\{1, 4, 9, 16, 25\}$ of all numbers x^2 such that x is an element of the set $\{1...5\}$.

3.4.2 Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x <- [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5] .
Note:

- The expression x <- [1..5] is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)
    ]
```

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)
    ]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.
- For example:

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)
    ]
```

x <− [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

3.4.3 Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y,) such that x,y are elements of the list [1..3] and $y \geq x$.
Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

For example:

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

3.4.4 Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | <- [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.
Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors :: Int -> [Int]
factors n =
    [x | y <- [1..n], n `mod` x == 0]
```

For example:

```
> factors 15
[1,3,5,15]
```

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime :: Int -> Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False
> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

For example:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

3.4.5 The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns that the list of all pairs of adjacent elements from a list:

```
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Using pairs we can define such a function that decides if the elements in a list are sorted:

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs
    xs]
```

For example:

```
> sorted [1,2,3,4]
True
> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs =
  [i | (x',i) <- zip xs [0..], x == x']
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

3.4.6 String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

```
"abc" :: String
```

Means ['a','b','c'] :: [Char].
Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
5
> take 3 "abcde"
"abc"
> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

```
count :: Char -> String -> Int
count x xs = length [x' | x' <- xs, x ==
    x']
```

For example:

```
> count 's' "Mississippi"
4
```

3.5 Recursive Functions

3.5.1 Introduction

As we have seen, many functions can naturally be defined in terms of other functions.

```
fac :: Int -> Int
fac n = product [1..n]
```

fac maps any integer n to the product of the integers between 1 and n.
Expressions are evaluated by a stepwise process of applying functions to their arguments.
For example:

```
fac 4
=
product [1..4]
=
product [1,2,3,4]
=
1*2*3*4
=
24
```

3.5.2 Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.
For example:

```
fac 3
=
3 * fac 2
=
3 * (2 * fac 1)
=
3 * (2 * (1 * fac 0))
=
3 * (2 * (1 * 1))
=
3 * (2 * 1)
=
3 * 2
=
6
```

Note:

- fac 0 = 1 is appropriate because 1 is the identity for multiplication: 1∗x = x = x∗1.
- The recursive definition diverges on integers < 0 because the base case is never reached:

  ```
  > fac (-1)
  *** Exception: stack overflow
  ```

3.5.3 Why is Recursion useful?

- Some functions, such as factorial , are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

3.5.4 Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product :: Num a => [a] -> a
product []     = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.
For example:

```
product [2,3,4]
=
2 * product [3,4]
=
2 * (3 * product [4])
=
2 * (3 * (4 * product []))
=
2 * (3 * (4 * 1))
=
24
```

Using the same pattern of recursion as in product we can define the **length** function on lists.

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.
For example:

```
length [1,2,3]
=
1 + length [2,3]
=
1 + (1 + length [3])
=
1 + (1 + (1 + length []))
=
1 + (1 + (1 + 0))
= 3
```

Using a similar pattern of recursion we can define the **reverse** function on lists.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.
For example:

```
reverse [1,2,3]
=
reverse [2,3] ++ [1]
=
(reverse [3] ++ [2]) ++ [1]
=
((reverse [] ++ [3]) ++ [2]) ++ [1]
=
(([] ++ [3]) ++ [2]) ++ [1]
=
[3,2,1]
```

3.5.5 Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
Zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs
    ys
```

- Remove the first n elements from a list:

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

- Appending two lists:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

3.5.6 Quicksort

The quicksort algorithm for sorting a list of values can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

Using recursion, this specification can be translated directly into an implementation:

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

For example (abbreviating qsort as q):

3.6 Higher-Order Functions

3.6.1 Introduction

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

3.6.2 Why are they useful?

- Common programming idioms can be encoded as functions within the language itself.
- Domain specific languages can be defined as collections of higher-order functions.
- Algebraic properties of higher-order functions can be used to reason about programs.

3.6.3 The map function

The higher-order library function called **map** applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

For example:

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

The **map** function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x <- xs]
```

Alternatively, for the purposes of proofs, the **map** function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

3.6.4 The filter function

The higher-order library function **filter** selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p []     = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise = filter p x
```

3.6.5 The foldr function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some function $\oplus$ applied to its head and f of its tail.
For example:
$(v = 0; \oplus = +)$

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$(v = 1; \oplus = *)$

```
product []     = 1
product (x:xs) = x * product xs
```

$(v = True; \oplus = \&\&)$

```
and []     = True
and (x:xs) = x && and xs
```

The higher-order library function **foldr** (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.
For example:

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
```

foldr itself can be defined using recursion:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

However, it is best to think of **foldr** non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:

sum [1,2,3]
=
foldr (+) 0 [1,2,3]
=
foldr (+) 0 (1:(2:(3:[])))
= (Replace each (:) by (+) and [] by 0)
1+(2+(3+0))
=
6

For example:

product [1,2,3]
=
foldr (∗) 1 [1,2,3]
=
foldr (∗) 1 (1:(2:(3:[])))
= (Replace each (:) by (∗) and [] by 1)
1∗(2∗(3∗1))
=
6

3.6.6 Other foldr examples

Even though **foldr** encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

For example:

length [1,2,3]
=
length (1:(2:(3:[])))
= (Replace each (:) by $\lambda_n \rightarrow 1+n$ and [] by 0)
1+(1+(1+0))
=
3

Hence, we have:
length = **foldr** $(\lambda_n \rightarrow 1+n)$ 0

Now recall the **reverse** function:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

reverse [1,2,3]
=
reverse (1:(2:(3:[])))
= (Replace each (:) by λx xs→xs ++ [x] and [] by [])
(([] ++ [3]) ++ [2]) ++ [1]
=
[3,2,1]

Hence, we have:
reverse = **foldr** (λx xs → xs ++ [x]) []

Finally, we note that the append function (++) has a particularly compact definition using **foldr**:
(++ ys)= **foldr** (:) ys (Replace each (:) by (:) and [] by ys.)

3.6.7 Why is foldr useful?

- Some recursive functions on lists, such as **sum**, are simpler to define using **foldr**.
- Properties of functions defined using **foldr** can be proved using algebraic properties of **foldr**, such as fusion and the banana split rule.
- Advanced program optimizations can be simpler if **foldr** is used in place of explicit recursion.

3.6.8 Other library functions

The library function (.) returns the composition of two functions as a single function.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

For example:

```
odd :: Int -> Bool
odd = not . even
```

The library function **all** decides if every element of a list satisfies a given predicate.

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

For example:

```
> all even [2,4,6,8,10]
True
```

Dually, the library function **any** decides if at least one element of a list satisfies a predicate.

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

For example:

```
> any (== ' ') "abc def"
True
```

The library function **takeWhile** selects elements from a list while a predicate holds of all the elements.

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

For example:

```
> takeWhile (/= ' ') "abc def"
"abc"
```

Dually, the function **dropWhile** removes elements while a predicate holds of all the elements.

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

For example:

```
> dropWhile (== ' ') "    abc"
"abc"
```

3.7 Declaring Types and Classes

3.7.1 Type declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type [**Char**].
Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int, Int)
```

we can define:

```
origin :: Pos
origin = (0,0)
left :: Pos -> Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int -> Int
mult (m,n) = m*n
copy :: a -> Pair a
copy x = (x,x)
```

Type declarations can be nested:

```
type Pos = (Int, Int)
type Trans = Pos -> Pos
```

However, they **cannot** be recursive:

```
type Tree = (Int,[Tree])
```

3.7.2 Data declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values **False** and **True**.
Note:

- The two values **False** and **True** are called the constructors for the type **Bool**.
- Type and constructor names must always begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]
flip :: Answer -> Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
  | Rect Float Float
```

we can define:

```
square :: Float -> Shape
square n = Rect n n
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

```
Circle :: Float -> Shape
Rect :: Float -> Float -> Shape
```

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

3.7.3 Recursive types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat -> Nat.
Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequences of values:

  ```
  Zero
  Succ Zero
  Succ (Succ Zero)
  ⋮
  ```

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function 1+.
- For example, the value

  ```
  Succ (Succ (Succ Zero))
  ```

  represents the natural number

  ```
  1 + (1 + (1 + 0)) = 3
  ```

Using recursion, it is easy to define functions that convert between values of type Nat and **Int**:

```
nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n
    )
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

```
                 add (Succ (Succ Zero)) (Succ Zero)
             =
                 Succ (add (Succ Zero)) (Succ Zero)
For example: =
                 Succ (Succ (add Zero (Succ Zero)))
             =
                 Succ (Succ (Succ Zero))
```

Note:

- The recursive definition for add corresponds to the laws 0+n=n and (1+m)+n = 1+(m+n).

3.7.4 Arithmetic expressions

Consider a simple form of expressions built up from integers using addition and multiplication.
Using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr -> Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

- The three constructors have types:

  ```
  Val :: Int -> Expr
  Add :: Expr -> Expr -> Expr
  Mul :: Expr -> Expr -> Expr
  ```

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable fold function. For example:

```
eval folde id (+) (*)
```

3.8 The Countdown Problem

3.8.1 What is Countdown?

- A popular quiz programme on British television that has been running since 1982.
- Includes a numbers game that we shall refer to as the countdown problem.

3.8.2 Example

Using the numbers

1 3 7 10 25 50

and the arithmetic operators

+ − * /

construct an expression whose value is 765.

3.8.3 Rules

- All the numbers, including intermediate results, must be positive naturals $(1, 2, 3, \ldots)$.
- Each of the source numbers can be used at most once when constructing the expression.
- We abstract from other rules that are adopted on television for pragmatic reasons.

For our example, one possible solution is

(25 − 10) * (50+1) = 765

Notes:

- There are 780 solutions for this example.
- Changing the target number to 831 gives an example that has no solutions.

3.8.4 Evaluating Expressions

Operators:

```
data Op = Add | Sub | Mul | Div
```

Apply an operator:

```
apply :: Op -> Int -> Int -> Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

Decide if the result of applying an operator to two positive natural numbers is another such:

```
valid :: Op -> Int -> Int -> Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

Expression:

```
data Expr = Val Int | App Op Expr Expr
```

Return the overall value of an expression, provided that it is a positive natural number:

```
eval :: Expr -> [Int]
eval (Val n)
eval ( App o l r) = [apply o x y | x <-
    eval l
                                  , y <-
                                    eval
                                     r
                                  , valid
                                     o x
                                     y]
```

Either succeeds and returns a singleton list, or fails and returns the empty list.

3.8.5 Formalizing the problem

Return a list of all possible ways of choosing zero or more elements from a list:

```
choices :: [a] -> [[a]]
```

For example:

```
> choices [1,2]
[[],[1],[2],[1,2],[2,1]]
```

Return a list of all the values in an expression:

```
values :: Expr -> [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values
    r
```

Decide if an expression is a solution for a given list of source numbers and a target number:

```
solution :: Expr -> [Int] -> Int -> Bool
solution e ns n = elem (values e) (
    choices ns)
                  && eval e == [n]
```

3.8.6 Brute force solution

3.9 Exercises:

3.9.1 Exercise 1

Types of Lists and Tuples:
Given the declaration x = 'x', which expressions are correctly typed?

x	**Char**
'x'	**Char**
"x"	[**Char**]
['x']	[**Char**]
[x, 'x']	[**Char**]
[x, x, x, x]	[**Char**]
['x', "x"]	**Not correct.**
[x == 'x', **True**]	[**Bool**]
[["True"]]	[[[**Char**]]]
[[**True**, **False**], **True**]	**Not correct.**
[[**True**, **False**], []]	[[**Bool**]]
('x')	**Char**
(x, 'x')	(**Char**, **Char**)
(x, x, x, x)	(**Char**, **Char**, **Char**, **Char**)
('x', "x")	(**Char**, [**Char**])
(x, **True**)	(**Char**, **Bool**)
(x == 'x', **True**)	(**True**, **True**)
(("True"))	[**Char**]
((**True**, **False**), **True**)	((**Bool**, **Bool**), **Bool**)
((**True**, **False**), ())	((**Bool**, **Bool**), ())

Types of Lists
Given the declaration a = [**True**], which expressions are correctly typed?

a	[**Bool**]
a ++ a ++ [**True**]	[**Bool**]
a ++ []	[**Bool**]
head a	**Bool**
tail a	[**Bool**]
head 'x'	**Not correct.**
head "x"	**Char**
tail "x"	[**Char**]
"dimdi" !! 2	**Char**
"dimdi" ++ "ding"	[**Char**]

Types of Lists and Tuples Mixed
Given the declaration:

```
einkaufsliste =
  [(3, "Widerstand 10kOhm"),
   (5, "Kondensator 0.1microFarad"),
   (2, "Zahnrad 38 Zaehne")]
preisliste =
  [("Zahnrad 38 Zaehne", 1200),
   ("Widerstand 10 kOhm", 50),
   ("Widerstand 20 kOhm", 50),
   ("Kondensator 0.1microFarad", 50)]
```

Which expressions are correctly typed?

[(**True**, 'a'), (**False**, 'b')]	[(**Bool**, **Char**)]
[(**True**, 'a'), ('b', **False**)]	**Not correct.**
[(**True**, 'a'), ('a' == 'b', **head** "a")]	[(**Bool**, **Char**)]
([**True**, 'a' == 'b'], ['a'])	([**Bool**],[**Char**])
('a', 'b', 'c', 'd')	(**Char**, **Char**, **Char**, **Char**)
('a', ('b', ('c', ('d'))))	(**Char**, (**Char**, (**Char**, **Char**)))
(('a', 'b'), 'c', 'd')	((**Char**, **Char**), **Char**, **Char**)
['a', 'b', 'c', 'd']	[**Char**]
einkaufsliste	[(**Integer**, [**Char**])]
preisliste	[([**Char**], **Integer**)]
(einkaufsliste , preisliste)	([(**Integer**, [**Char**])], [([**Char**], **Integer**)])

Types of Functions and Lists
Given the declarations:

```
f1 :: Int -> Int
f1 x = x^2 + x + 1

f2 :: Int -> Int
f2 x = 2*x + 1
```

Which expressions are correctly typed?

f1	**Int -> Int**
f1 5	**Int**
f1 f2	**Not correct.**
f1 (f2 5)	**Int**
[f1 5, f2 6, 5, 6]	[**Int**]
[f1, f2, f1]	[**Int -> Int**]
[f1 5, f2]	**Not correct.**
(f1 5, f2)	(**Int**, **Int -> Int**)
([f1, f2, f1] !! 1) 3	**Int**
([f1, f2, f1] !! 5) 3	**Int**

Types of Functions with Currying
Given the declarations:

```
g1 :: Int -> Int -> Int -> Int
g1 x y z = x^2 + y^2 + z^2

g2 :: Int -> Int -> Int
g2 x y = 2*x + 2*y
```

Which expressions are correctly typed?

g1	**Int** –> **Int** –> **Int** –> **Int**
g1 2	**Int** –> **Int** –> **Int**
g1 2 3	**Int** –> **Int**
g1 2 3 4	**Int**
g1 2 3 4 5	**Not correct.**
(g1, g2)	(**Int** –> **Int** –> **Int** –> **Int**, **Int** –> **Int** –> **Int**)
(g1 2, g2)	(**Int** –> **Int** –> **Int**, **Int** –> **Int** –> **Int**)
(g1 2 3, g2 4)	(**Int** –> **Int**, **Int** –> **Int**)
(g1 2 3 4, g2 4 5)	(**Int**, **Int**)
[g1, g2]	**Not correct.**
[g1 2, g2]	[**Int** –> **Int** –> **Int**]
[g1 2 3, g2 4]	[**Int** –> **Int**]
[g1 2 3 4, g2 4 5]	[**Int**]

Polymorphic Types

Given the declarations:

```
h1 x = (x, x, x)
h2 x = [x, x, x]
h3 x = [(x, x), (x, x)]
h4 x y = (x, y)
h5 x y = [x, y]
```

Which expressions are correctly typed?

h1	c –> (c, c, c)
h2	a –> [a]
h3	b –> [(b, b)]
h4	a –> b –> (a, b)
h5	a –> a –> [a]
h1 'a'	(**Char**, **Char**, **Char**)
h1 **True**	(**Bool**, **Bool**, **Bool**)
h4 'a' "True"	(**Char**, [**Char**])
h5 'a' "True"	**Not correct.**
h5 **True True**	[**Bool**]
[]	[a]
()	()
head []	a
head ()	**Not correct.**

Programming Exercise: two-dimensional vectors

```
module Exer01Sol where

-- Develop some functions to work with two-dimensional vectors.

-- a type for two-dimensional vectors
type Vec = (Double, Double)

-- the zero vector
zeroVec :: Vec
zeroVec = (0, 0)

-- some example vectors
a, b, c, d :: Vec
a = (3, 0)
b = (0, 4)
c = (sqrt2, sqrt2)
      where sqrt2 = sqrt 2
d = (3, 4)

-- lengthVec computes the length of a vector.
exaLengthVec =
  lengthVec a == 3 && lengthVec c == 2

lengthVec :: Vec -> Double
lengthVec (x, y) = sqrt (x^2 + y^2)

-- negVec negates a vector.
exaNegVec =
  negVec d == (-3, -4)
```

```
negVec :: Vec -> Vec
negVec (x, y) = (-x, -y)

-- negVecCurry negates a vector, but uses currying.
-- Note: this is a bad use of currying, since
-- the two components of a vector belong together.
exaNegVecCurry =
  negVecCurry (fst d) (snd d) == (-3, -4)

negVecCurry :: Double -> Double -> Vec
negVecCurry x y = (-x, -y)

-- addVec adds two vectors.
exaAddVec =
  a `addVec` b == d

addVec :: Vec -> Vec -> Vec
addVec (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

-- subVec subtracts two vectors.
exaSubVec =
  a `subVec` b == (3, -4)
-- Implement this function using negVec and addVec.

subVec :: Vec -> Vec -> Vec
subVec v1 v2 = addVec v1 (negVec v2)

subVecTerrible :: Vec -> Vec -> Vec
subVecTerrible (v1x, v1y) (v2x, v2y) = addVec (v1x, v1y) (negVec (v2x, v2y))
-- This is terrible, since there is no need to resolve
-- the components of the vectors.

-- subVecCurry subtracts two vectors.
-- Note: this example clearly demonstrates the bad use
-- of currying in negVecCurry.
-- The two components of a vector simply belong together.
exaSubVecCurry =
  a `subVecCurry` b == (3, -4)
-- Implement this function using negVecCurry and addVec.

subVecCurryV1 :: Vec -> Vec -> Vec
subVecCurryV1 v1 v2 = addVec v1 (negVecCurry (fst v2) (snd v2))
-- Whenever we see usage of fst and/or snd, we should automatically
-- strive to use pattern matching instead.
-- This is usually more elegant; see the next version.

subVecCurryV2 :: Vec -> Vec -> Vec
subVecCurryV2 v1 (v2x, v2y) = addVec v1 (negVecCurry v2x v2y)

subVecCurry = subVecCurryV2
```

```
-- distance computes the distance between two vectors.
exaDistance =
  distance a d == 4
-- Implement this function using subVec and lengthVec.

distance :: Vec -> Vec -> Double
distance v1 v2 = lengthVec (v1 `subVec` v2)

-- Scales a vector with a factor.
exaScaleVec =
  scaleVec d 3 == (9, 12)

scaleVec :: Vec -> Double -> Vec
scaleVec (x, y) s = (s*x, s*y)
```

3.9.2 Exercise 2

Types of numeric literals
Which expressions are correctly typed?

2	**Num** p =>p
2 + 2	**Num** a =>a
2 :: **Int**	**Int**
2 :: **Float**	**Float**
(2 + 2) :: **Double**	**Double**
2.0	**Fractional** p =>p
2.0 :: **Int**	**Not correct.**
2 + 2.0	**Fractional** a =>a
(2 :: **Int**) + (2 :: **Double**)	**Not correct.**
(2 :: **Int**) + 2	**Int**
(2, 2)	(**Num** a, **Num** b) =>(a, b)
[2, 2]	**Num** a =>[a]
[2, 2.0]	**Fractional** a =>[a]
[2 :: **Float**, 2 :: **Double**	**Not correct.**

Types of overloaded functions
Given the declarations:

```
f1 x = 2
f2 x = 2*x
f3 x y z = x == y && y == z
f4 x y z = x < y && y < z
f5 x y z = x == y && y < z
f6 x y = 2 * x < y
f7 x y = min (abs x) (negate y)
f8 x y = [x, y, 2]
f9 x y = x 'div' y + x / y
```

Which expressions are correctly typed?

f1	**Num** p1 =>p2 -> p1
f1 'a'	**Num** p1 =>p1
f1 "a"	**Num** p1 =>p1
f1 f1	**Num** p1 =>p1
f2	**Num** a =>a -> a
f2 2	**Num** a =>a
f2 2.0	**Fractional** a =>a
f2 'a'	**Not correct.**
('a', 'b') == ('c', 'd')	**Bool**
('a', 'b') < ('c', 'd')	**Bool**
('a', 'b') < ('c', 'd', 'e')	**Not correct.**
['a', 'b'] < ['c', 'd', 'e']	**Bool**
f3	**Eq** a =>a -> a -> a -> **Bool**
f3 ('a', 'b') ('a', 'b') ('a', 'b')	**Bool**
f4	**Ord** a =>a -> a -> a -> **Bool**
f4 2 2	(**Ord** a, **Num** a)=>a -> **Bool**
f5	**Ord** a =>a -> a -> a -> **Bool**
f5 [2] [] [2,2]	**Bool**
f6	(**Ord** a, **Num** a)=>a -> a -> **Bool**
(f6) 2	(**Ord** a, **Num** a)=>a -> **Bool**
f7	(**Ord** a, **Num** a)=>a -> a -> a
f7 (2 :: **Int**) (2 :: **Integer**)	**Not correct.**
f8	**Num** a =>a -> a -> [a]
f8 2 2.0	**Fractional** a =>[a]
f9	(**Integral** a, **Fractional** a)=>a -> a -> a
f9 2 2	(**Integral** a, **Fractional** a)=>a

Programming Exercise: List comprehensions

```
module Exer02Sol where

-- Develop some functions to work with matrices, using list comprehensions.
-- Higher-order functions are not required yet.

-- Helpful functions from the Prelude:
--   and, (!!), zip
```

```
toBeImplemented = undefined

-- the type for values
type Value = Double
-- the type for matrices
type Matrix = [[Value]]

-- We represent matrices as lists of lists,
-- where each inner list represents a row
-- (rather than a column) of the matrix.
-- Example:
-- [[a11, a12, a13],
--  [a21, a22, a23]]
-- Indexing, as often done in matrix computations,
-- starts with 1 rather than 0 (beg Dijkstra's forgiveness).

-- We assume as precondition for most of the
-- following functions that all inner lists
-- have the same length (if there are any).
-- Here is a function that might be used to
-- check this condition.
exaIsMat =
  isMat [[1, 2], [3, 4]] &&
  isMat [[1, 2]] &&
  isMat [] &&
  isMat [[], []] &&
  not (isMat [[1, 2], []])

isMat :: Matrix -> Bool
isMat [] = True
isMat (row1 : rows) =
    and [length row == lengthRow1 | row <- rows]
  where lengthRow1 = length row1

-- Sometimes we use square matrices.
-- Here is a function that might be used to
-- check whether a list of lists represents a square matrix.
exaIsSquareMat =
  isSquareMat [[1, 2], [3, 4]] &&
  isSquareMat [[1]] &&
  isSquareMat [] &&
  not (isSquareMat [[]]) &&
  not (isSquareMat [[1, 2, 3], [4, 5, 6]])

isSquareMat = isSquareMatV2

isSquareMatV1 :: Matrix -> Bool
isSquareMatV1 [] = True
isSquareMatV1 mat@(row1 : _) = -- example of an as-pattern
  isMat mat && length mat == length row1
```

```
-- The following version uses head instead of the as-pattern;
-- but I consider the as-pattern much more elegant.
-- The as-pattern also pinpoints the condition that mat must be non-empty
-- in the second case.
isSquareMatV2 :: Matrix -> Bool
isSquareMatV2 [] = True
isSquareMatV2 mat =
  isMat mat && length mat == length (head mat)

-- zeroMat generates an m * n zero matrix, that is,
-- a matrix with all entries 0.
exaZeroMat =
  zeroMat 2 3 == [[0, 0, 0], [0, 0, 0]]

zeroMat :: Int -> Int -> Matrix
zeroMat m n = [zeroRow | _ <- [1 .. m]]
  where
    zeroRow = [0 | _ <- [1 .. n]]

-- Check whether a list of lists represents a zero matrix or not.
isZeroMatV1 :: Matrix -> Bool
isZeroMatV1 mat =
  isMat mat && and [a == 0 | row <- mat, a <- row]

isZeroMatV2 :: Matrix -> Bool
isZeroMatV2 mat =
  isMat mat && and [a == 0 | a <- concat mat]

-- Version 1 seems to be more natural than version 2, since version 1
-- displays the two-dimensional nature of the problem, whereas
-- version 2 flattens the matrix to a one-dimensional structure.

-- unitMat generates an n * n unit matrix, that is,
-- a square matrix in which all entries are zero,
-- except of the elements on the main diagonal, which are 1.
exaUnitMat =
  unitMatV1 3 == [[1, 0, 0], [0, 1, 0], [0, 0, 1]] &&
  unitMatV2 3 == [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

unitMat = unitMatV1

unitMatV1 :: Int -> Matrix
unitMatV1 n = [row i | i <- [1 .. n]]
  where
    row i = [if j == i then 1 else 0 | j <- [1 .. n]]

unitMatV2 :: Int -> Matrix
unitMatV2 n = [row i | i <- [1 .. n]]
  where
    row i = [0 | _ <- [1 .. i-1]] ++ 1 : [0 | _ <- [i+1 .. n]]
```

```
-- Check whether two matrices are equal of not.
equalMat :: Matrix -> Matrix -> Bool
equalMat = (==)

-- Check whether a list of lists represents a unit matrix or not.
isUnitMat :: Matrix -> Bool
isUnitMat mat = mat == unitMat (length mat)

-- neg negates a matrix, that is,
-- negates all corresponding components of it.
exaNeg =
  neg [[1, -2], [0, 4]] == [[-1, 2], [0, -4]]

neg :: Matrix -> Matrix
neg mat = [[-a | a <- row] | row <- mat]

-- plusMat adds two matrices, that is,
-- adds all corresponding components of them.
-- precondition: mat1 and mat2 have the same size
exaPlusMat =
  unit3 `plusMat` (neg unit3) == zeroMat 3 3
    where unit3 = unitMat 3

plusMat = plusMatV1

plusMatV1 :: Matrix -> Matrix -> Matrix
mat1 `plusMatV1` mat2 =
    [row1 `plusRow` row2 | (row1, row2) <- zip mat1 mat2]
  where
    row1 `plusRow` row2 = [a1 + a2 | (a1, a2) <- zip row1 row2]

plusMatV2 :: Matrix -> Matrix -> Matrix
mat1 `plusMatV2` mat2 =
  [zipWith (+) row1 row2 | (row1, row2) <- zip mat1 mat2]

-- Version 2 is much shorter and much more 'functional' than version 1,
-- but uses zipWith, a higher-order function, and we have not yet studied
-- this concept.

-- Reads the element in row i and column j of matrix mat.
-- precondition: i and j are in range of mat
readMat :: Matrix -> Int -> Int -> Value
readMat mat i j = (mat !! i) !! j

-- updateMat 'updates' matrix mat.
-- The updated matrix is the same as mat, except of
-- position row i column j, where the new value is val.
-- Note: there is no imperative update; the function returns a new matrix.
-- precondition: i and j are in range of mat
exaUpdateMat =
```

```
  updateMat (unitMat 3) 3 1 5 == [[1,0,0],[0,1,0],[5,0,1]]

updateMat :: Matrix -> Int -> Int -> Value -> Matrix
updateMat mat i j val =
  [if i' == i then updateRow row j else row | (i', row) <- zip [1 ..] mat]
    where
      updateRow row j =
        [if j' == j then val else val' | (j', val') <- zip [1 ..] row]
```

3.9.3 Exercise 3

List sugaring
Rewrite the expressions so they don't contain the constrctor : (cons) any longer:

1:2:3:[4]	[1,2,3,4]
1:2:[3,4]	[1,2,3,4]
(1:2:[]) :(3:[]) :[]	[[1,2],3]
(1,2) :(3,4) :[(5,6)]	[(1,2) ,(3,4) ,(5,6)]
[] : []	[[]]
[] : [] : []	[[],[]]
([] : []) : []	[[[]]]
(([] : []) : []) : []	[[[[]]]]
'a' : 'b' : []	"ab"

List desugaring
Rewrite the expressions so they contain the square brackets only as list constructor [] (nil):

[1,2,3]	1 : 2 : [3]
[[1,2],[],[3,4]]	(1:2:[]) :([]) :(3:4:[]) :[]
[[], ["a"], [[]]]	???

Pattern Matching
Given the function and value declarations, give the type of each function and evaluate the expressions in the value declarations.

f1 (x : y : z) = (x, y, z)	[b] -> (b, b, [b])
f2 [x, y] = (x, y)	[b] -> (b, b)
f3 (x : y : []) = (x, y)	[b] -> (b, b)
a11 = f1 []	**Not correct.**
a21 = f2 []	**Not correct.**
a31 = f3 []	**Not correct.**
a12 = f1 [1]	**Not correct.**
a22 = f2 [1]	**Not correct.**
a32 = f3 [1]	**Not correct.**
a13 = f1 [1, 2]	(1,2,[])
a23 = f2 [1, 2]	(1,2)
a33 = f3 [1, 2]	(1,2)
a14 = f1 [1, 2, 3]	(1,2,[3])
a24 = f2 [1, 2, 3]	**Not correct.**
a34 = f3 [1, 2, 3]	**Not correct.**
a15 = f1 (1 : 2 : 3 : [])	(1,2,[3])
a25 = f2 (1 : 2 : 3 : [])	**Not correct.**
a35 = f3 (1 : 2 : 3 : [])	**Not correct.**
a16 = f1 ['a', 'b', 'c']	('a', 'b', "c")
a17 = f1 [[1], [2,3],[]]	([1], [2,3], [[]])
a18 = f1 (1 : 2 : 3 : [4,5])	(1, 2, [3,4,5])
a19 = f1 [1 .. 100]	(1,2,[3,4,...,100])

f4 (x, y) = [x, y]	(a, a) -> [a]
a41 = f4 ([1, 2], [3, 4, 5])	[[1,2],[3,4,5]]

g1 = "dimdi"= 1	
g1 ['d', 'o', 'm', 'd', 'o'] = 2	
g1 ('d' : 'i' : 'n' : 'g' : []) = 3	
g1 ('d' : 'i' : 'n' : 'g' : _) = 4	
g1 (x : y) = 5	
g1 _ = 6	
b11 = g1 "domdo"	2
b12 = g1 "ding"	3
b13 = g1 "dingdimdi"	4
b14 = g1 "dumdu"	5
b15 = g1 ""	6

g2 (d : "imdi") | d == 'd' || d == 'D' = 1	
g2 (z : "umsel") | z == 'z' || z == 'Z' = 2	
g2 _ = 3	
b21 = g2 "dimdi"	1
b22 = g2 ['D', 'i', 'm', 'd', 'i']	1
b23 = g2 ('Z' : 'u' : "msel")	2
b24 = g2 "dimdiding"	3

h1 ['a', 'b'] = 'a'	
h1 ['a', b] = b	
h1 (_ : _ : 'm' : _) = 'm'	
h1 (a : b) = a	
c11 = h1 "ab"	'a'
c12 = h1 "ac"	'c'
c13 = h1 "dimdi"	'm'
c14 = h1 "zumsel"	'm'
c15 = h1 "schnurpsel"	's'

h2 [(a, b), c] = c	
h2 (a : b : c) = a	
c21 = h2 [(1, 2), (3, 4)]	(3,4)
c22 = h2 [(1, 2), (3, 4), (5, 6)]	(1,2)
c23 = h2 [(1, 2)]	**Not correct.**

h3 ((x : y) : z) = y	
h3 ([] : _) = "2"	
h3 [] = "3"	
c31 = h3 ["dimdi"]	"imdi"
c32 = h3["", "dimdi", "domdo"]	"2"
c33 = h3 [[]]	"2"
c34 = h3 []	"3"

Programming Exercise: Recursion over lists

module Exer03Sol **where**

```
-- Develop some functions using recursion over lists.
-- Higher-order functions are not required yet.

toBeImplemented = undefined

--delDups deletes duplicates from a list
testDelDups =
  delDups [1,2,3,4,5] == [1,2,3,4,5] &&
  delDups [1,1,1,1,1] == [1] &&
  (delDups [1,2,2,4,1] == [1,2,4] || delDups [1,2,2,4,1] == [2,4,1]) &&
  delDups []          == ([] :: [Int])

delDups :: Eq a => [a] -> [a]
delDups [] = []
delDups (x : xs)
  | x `elem` xs = delDups xs
  | otherwise   = x : delDups xs

-- removeEachSnd removes each second element from a list.
testRemoveEachSnd =
  removeEachSnd [1,2,3,4,5,6,7,8] == [1,3,5,7] &&
  removeEachSnd [1,2,3,4,5,6,7]   == [1,3,5,7] &&
  removeEachSnd [1]               == [1] &&
  removeEachSnd []                == ([] :: [Int])

removeEachSnd :: [a] -> [a]
removeEachSnd (x : _ : xs) = x : removeEachSnd xs
removeEachSnd xs = xs

-- makePairs pairs adjacent elements of a list
testMakePairs =
  makePairs [1,2,3,4,5,6,7,8] == [(1,2),(3,4),(5,6),(7,8)] &&
  makePairs [1,2,3,4,5,6,7]   == [(1,2),(3,4),(5,6)] &&
  makePairs [1,2]             == [(1,2)] &&
  makePairs [1]               == [] &&
  makePairs []                == ([] :: [(Int,Int)])

makePairs :: [a] -> [(a, a)]
makePairs []          = []
makePairs xs@(_ : ys) = removeEachSnd (zip xs ys)

testMakePairsV2 =
  makePairsV2 [1,2,3,4,5,6,7,8] == [(1,2),(3,4),(5,6),(7,8)] &&
  makePairsV2 [1,2,3,4,5,6,7]   == [(1,2),(3,4),(5,6)] &&
  makePairsV2 [1,2]             == [(1,2)] &&
  makePairsV2 [1]               == [] &&
  makePairsV2 []                == ([] :: [(Int,Int)])

makePairsV2 :: [a] -> [(a, a)]
makePairsV2 (x : y : xs) = (x, y) : makePairsV2 xs
```

```
makePairsV2 xs            = []

-- halve divides a list into two lists containing each second element,
-- the first list beginning with the first,
-- the second list beginning with the second
testHalve =
  halve [1,2,3,4,5,6] == ([1,3,5], [2,4,6]) &&
  halve [1,2,3,4,5]   == ([1,3,5], [2,4]) &&
  halve [1]           == ([1], []) &&
  halve []            == ([], [] :: [Int])

halve :: [a] -> ([a], [a])
halve xs = h xs [] []
  where
    h (x1 : x2 : xs) accu1 accu2 = h xs (x1 : accu1) (x2 : accu2)
    h [x]            accu1 accu2 = h [] (x : accu1) accu2
    h []             accu1 accu2 = (reverse accu1, reverse accu2)

testHalveV2 =
  halveV2 [1,2,3,4,5,6] == ([1,3,5], [2,4,6]) &&
  halveV2 [1,2,3,4,5]   == ([1,3,5], [2,4]) &&
  halveV2 [1]           == ([1], []) &&
  halveV2 []            == ([], [] :: [Int])

halveV2 :: [a] -> ([a], [a])
halveV2 [] = ([], [])
halveV2 xs@(_ : ys) = (removeEachSnd xs, removeEachSnd ys)

-- divideList divides a list into chunks of length n each, except
-- of the last chunk, which might be shorter
-- Precondition:
--   n > 0
-- Theorem:
--   For all n > 0 and all xs: concat (divideList n xs) == xs
testDivideList =
  divideList 3 [1 .. 10] == [[1,2,3],[4,5,6],[7,8,9],[10]] &&
  divideList 3 [1 .. 9] == [[1,2,3],[4,5,6],[7,8,9]] &&
  divideList 3 [1] == [[1]] &&
  divideList 3 [] == ([] :: [[Int]])

divideList :: Int -> [a] -> [[a]]
divideList _ [] = []
divideList n xs = take n xs : divideList n (drop n xs)
```

3.9.4 Exercise 4

Given the declarations, give the most general type of each value, and if the value is not a function, then evaluate it.

Lambda expressions

f01 :: **Num** a =>a −> a	
f01 = \x −> 2∗x	**Num** a =>a −> a
f01' = \x −> 2∗x	**Num** a =>a −> a
f01'' () = \x −> 2∗x	**Num** a =>()−> a −> a
f01''' _ = \x −> 2∗x	**Num** a =>p −> a −> a
f02 = \x −> \y −> x + y	**Num** a =>a −> a −> a
f03 = \x y −> x + y	**Num** a =>a −> a −> a
f04 x = \y −> x + y	**Num** a =>a −> a −> a
f05 = \(x,y) −> x + y	**Num** a =>(a,a)−> a
f06 = \[x,y] −> x + y	**Num** a =>[a] −> a
f07 = [\x −> x+1, \x −> 2∗x, \x −> x^2]	**Num** a =>[a −> a]
f08 = **head** f07 5	**Num** a =>a
f09 = \x −> x	p −> p
f10 = [f09, \x −> x+1]	**Num** a =>[a −> a]
f11 = _ −> (\x −> x +1, \()−> 'a')	**Num** a =>p −> (a −> a, ()−> **Char**)

x ^+^ y = x^2 + y^2	
g01 = (^+^)	**Num** a =>a −> a −> a
g02 = (^+^ 2)	**Num** a =>a −> a
g03 = (3 ^+^)	**Num** a =>a −> a
g04 = (3 ^+^ 2)	**Num** a =>a 13
g05 x y = 2∗x + 3∗y	**Num** a =>a −> a −> a
g06 = (`g05` 2)	**Num** a =>a −> a
g07 = (2 `g05`)	**Num** a =>a −> a
g08 = g06 3	**Num** a =>a 12
g09 = g07 4	**Num** a =>a 16
g10 x y z = 2∗x + 3∗y + 4∗z	**Num** a =>a −> a −> a −> a
g11 = (`g05` 2)	**Num** a =>a −> a
g12 = g06 3	**Num** a =>a 12
g13 = g07 4	**Num** a =>a 16
g14 x = (g10 (x+1))	**Num** a =>a −> a −> a −> a
g15 = g14 2 3 4	**Num** a =>a 31
g16 n = x−> ([(+), (−), (∗)] !! n) x 2	**Num** a =>a −> a −> a
g17 = g16 1 5	**Num** a =>a 3

Sections

List comprehensions

h01 = [x | x <− [1 .. 5]]	(**Num** a, **Enum** a)=>[a] [1,2,3,4,5]
h02 = [2∗x | x <− [1 .. 5]]	(**Num** a, **Enum** a)=>[a] [2,4,6,8,10]
h03 = [x − y | x <− [1 .. 3], y <− [1 .. 4]]	(**Num** a, **Enum** a)=>[a] [0,−1,−2,−3,1,0,−1,−2,2,1,0,−1]
h04 = [x − y | y <− [1 .. 3], x <− [1 .. 4]]	(**Num** a, **Enum** a)=>[a] [0,1,2,3,−1,0,1,2,−2,−1,0,1]
h05 = [x + y | x <− [1 .. 3], y <− [1 .. 4], x >= y]	(**Num** a, **Enum** a, **Ord** a)=>[a] [2,3,4,4,5,6]
h06 = [**head** x | x <− ["dimdi", "schnurpsel", "zumsel"]]	[**Char**] "dsz"
h07 = [x | (x : _) <− ["dimdi", "schnurpsel", "zumsel"]]	[**Char**] "dsz"
h08 = [xs | ('s' : xs) <− ["dimdi", "schnurpsel", "zumsel"]]	[[**Char**]] ["chnurpsel"]

3.9.5 Exercise 7

```
-- Exercise 7 (Higher-Order Functions)
import Prelude hiding ((.), ($))

-- function composition
testComposition = (head . tail) [1,2,3] == head (tail [1,2,3])

(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)

-- function composition reverse
testCompositionRev = (tail .> head) [1,2,3] == head (tail [1,2,3])

(.>) :: (a -> b) -> (b -> c) -> (a -> c)
(.>) = flip (.)

-- function application
testFunApp = (head $ [1,2,3]) == head [1,2,3]
```

```
($) :: (a -> b) -> (a -> b)
f $ x = f x

-- function application reverse
testFunAppRev = ([1,2,3] $> head) == head [1,2,3]

($>) :: a -> (a -> b) -> b
($>) = flip ($)

infixl 9 .>
infixl 0 $>

test01 = (5 $> (+2) .> (*3) .> (+4)) == ((+4) . (*3) . (+2) $ 5)
```

Higer-Order Functions: Types

f01 = **curry** . **fst**	((a, b1) -> c, b2)-> a -> b1 -> c
f02 = **uncurry** . **fst**	(a -> b1 -> c, b2)-> (a, b1)-> c
f03 = **fst** . **curry**	**Not correct.**
f04 = **fst** . **uncurry**	**Not correct.**
f05 = **curry** . **curry**	(((a, b1), b2) -> c)-> a -> b1 -> b2 -> c
f06 **map** ($5)	**Num** a =>[a -> b] -> [b]
v07 = **map** ($5)[(+1), (*2)	**Num** b =>[b] [6,10]

Programming Exercise: Higher-order function

```
module Exer04Sol where

-- Exercise (Higher-Order Functions)

import Prelude hiding (flip, curry, uncurry)

toBeImplemented = undefined

-- flip f takes its (first) two arguments in the reverse order of f
exaFlip = flip take [1,2,3,4,5] 3 == [1,2,3]

flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x

-- curry converts a function on pairs to a curried function
exaCurry = curry (\(x,y) -> x + y) 3 4 == 7

curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)

-- uncurry converts a curried function to a function on pairs
exaUncurry = uncurry (\x y -> x + y) (3, 4) == 7

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y

exaReverseFr = reverseFr [1 .. 20000] == [20000, 19999 .. 1]
```

```
exaReverseFl = reverseFl [1 .. 20000] == [20000, 19999 .. 1]

-- implement reverse using foldr and (++)
reverseFr :: [a] -> [a]
reverseFr = foldr (\x accu -> accu ++ [x]) []

-- implement reverse using foldl, (:), and flip
reverseFl :: [a] -> [a]
reverseFl = foldl (flip (:)) []

-- revAppend prepends the first list in reverse order before the second list
exaRevAppend = revAppend [3,2,1] [4,5,6] == [1,2,3,4,5,6]
-- implement revAppend using foldl and flip

revAppend :: [a] -> [a] -> [a]
revAppend = flip (foldl (flip (:)))

exaMapFr = mapFr (*2) [1 .. 10] == map (*2) [1 .. 10]
exaMapFl = mapFl (*2) [1 .. 10] == map (*2) [1 .. 10]

mapFr :: (a -> b) -> [a] -> [b]
mapFr f = foldr (\x accu -> f x : accu) []

mapFl :: (a -> b) -> [a] -> [b]
mapFl f = foldl (\accu x -> accu ++ [f x]) []

-- for estimating performance
lr = length (mapFr (*2) [1 .. 20000])
ll = length (mapFl (*2) [1 .. 20000])

exaFilterFr = filterFr odd [1 .. 10] == filterFr odd [1 .. 10]
exaFilterFl = filterFl odd [1 .. 10] == filterFl odd [1 .. 10]

filterFr :: (a -> Bool) -> [a] -> [a]
filterFr p = foldr (\x accu -> if p x then x : accu else accu) []

filterFl :: (a -> Bool) -> [a] -> [a]
filterFl p = foldl (\accu x -> if p x then accu ++ [x] else accu) []

exaLengthFr = lengthFr [1 .. 1000] == 1000
exaLengthFl = lengthFl [1 .. 1000] == 1000

lengthFr :: [a] -> Int
lengthFr = foldr (\_ accu -> 1 + accu) 0

lengthFl :: [a] -> Int
lengthFl = foldl (\accu _ -> accu + 1) 0

exaAppendFr = appendFr [1,2,3] [4,5,6] == [1 .. 6]
exaAppendFl = appendFl [1,2,3] [4,5,6] == [1 .. 6]
```

```
appendFr :: [a] -> [a] -> [a]
appendFr = flip (foldr (:))

-- strange version, since it uses the library version (++) of exactly
-- what is to be implemented
appendFl :: [a] -> [a] -> [a]
appendFl = foldl (\accu x -> accu ++ [x])
```

3.9.6 Exercise 8

```
-- Exercise 8 (Test Framework and Permutations)

-- general test framework
-- first parameter of function test should be removed
-- as soon as we have a set type as instance of class Eq
-- at our disposal

-- intended type:
--test :: (f -> (a -> b)) -> (b -> b -> Bool) -> f -> [(a, b)] -> Bool
-- most general type:
test :: (sut -> (a -> b)) -> (b -> c -> Bool) -> sut -> [(a, c)] -> Bool
-- Note:
--   the type of sut is completely unrestricted;
--   in particular, it need not be a function type
test cast eq sut tests =
  and [cast sut input `eq` expected | (input, expected) <- tests]

testV2 cast eq sut tests = all check tests
  where
    check (input, expected) =
      (cast sut) input `eq` expected

-- function setEq
-- setEq compares two sets represented by lists for equality
-- both lists are permitted to contain duplicates

setEq :: Eq a => [a] -> [a] -> Bool
setEq xs ys = all (`elem` ys) xs && all (`elem` xs) ys

check1SetEq = setEq [1,2,3,2] [2,3,1,1]
check2SetEq = not $ setEq [1,2,3] [2,3,4]

test2 :: (b -> c -> Bool) -> (a1 -> a2 -> b) -> [((a1, a2), c)] -> Bool
test2 = test uncurry

uncurry3 :: (a -> b -> c -> d) -> ((a, b, c) -> d)
uncurry3 f (x, y, z) = f x y z

test3 ::
  (b -> b -> Bool) -> (a1 -> a2 -> a3 -> b) -> [((a1, a2, a3), b)] -> Bool
test3 = testV2 uncurry3

-- function revApp
-- revApp rs xs puts the reverse of rs in front of xs

revApp :: [a] -> [a] -> [a]
revApp = flip (foldl (flip (:)))

checkRevApp = revApp [3,2,1] [4,5,6] == [1,2,3,4,5,6]
```

```
-- function insert
-- insert x ys n inserts x into list ys at position n
-- precondition: 0 <= n <= length ys

type Insert a = a -> [a] -> Int -> [a]

insertTests =
  [((99, [1,2,3], 0), [99,1,2,3]),
   ((99, [1,2,3], 1), [1,99,2,3]),
   ((99, [1,2,3], 2), [1,2,99,3]),
   ((99, [1,2,3], 3), [1,2,3,99])]

insertV1 :: Insert a
insertV1 x ys       0 = x : ys
insertV1 x (y : ys) n = y : insertV1 x ys (n-1)

insertV2 :: Insert a
insertV2 x ys n = ls ++ x : rs
  where
    (ls, rs) = splitAt n ys

insertV1Test = test3 (==) insertV1 insertTests
insertV2Test = test3 (==) insertV2 insertTests

insert :: Insert a
insert = insertV1

-- function inserts
-- inserts x ys inserts x at all possible positions in ys
-- the order of the insertions remains unspecified

type Inserts a = a -> [a] -> [[a]]

-- actually just one test
insertsTests =
  [((99, [1,2,3]), [[99,1,2,3],
                    [1,99,2,3],
                    [1,2,99,3],
                    [1,2,3,99]])]

insertsV1 :: Inserts a
insertsV1 x ys = map (insert x ys) [0 .. length ys]

insertsV2 :: Inserts a
insertsV2 x ys = h [] ys []
  where
    h rs yys@(y : ys) accu =
      h (y : rs) ys (revApp rs (x : yys) : accu)
    h rs [] accu = revApp rs [x] : accu

insertsV1Test = test uncurry setEq insertsV1 insertsTests
```

```
insertsV2Test = test uncurry setEq insertsV2 insertsTests

inserts :: Inserts a
inserts = insertsV1

-- function outsert
-- outsert xs n yields the element at position n and
-- list xs shortened by that element
-- precondition: 0 <= n < length xs

outsertTests =
  [(([1,2,3], 0), (1, [2,3])),
   (([1,2,3], 1), (2, [1,3])),
   (([1,2,3], 2), (3, [1,2]))]

outsert :: [a] -> Int -> (a, [a])
outsert (x : xs) 0 = (x, xs)
outsert (x : xs) n = (y, x : ys) where (y, ys) = outsert xs (n-1)

outsertTest = test uncurry (==) outsert outsertTests

-- function perms
-- perms xs yields the list of all permutations of xs
-- the order of the permutations remains unspecified
-- precondition: all elements in xs are pairwise distinct

permsTests =
  [([],       [[]]),
   ([1],      [[1]]),
   ([1,2],    [[1,2], [2,1]]),
   ([1,2,3], [[1,2,3], [1,3,2],
              [2,1,3], [2,3,1],
              [3,1,2], [3,2,1]])]

permsV1 :: [a] -> [[a]]
permsV1 [] = [[]]
permsV1 (x : xs) = concatMap (inserts x) (permsV1 xs)

permsV2 :: [a] -> [[a]]
permsV2 [] = [[]]
permsV2 (x : xs) = [insert x ps n | ps <- permsV2 xs,
                                    n <- [0 .. length xs]]

permsV2a :: [a] -> [[a]]
permsV2a [] = [[]]
permsV2a (x : xs) = [ls ++ x : rs | ps <- permsV2a xs,
                                    n <- [0 .. length xs],
                                    let (ls, rs) = splitAt n ps]

permsV3 :: [a] -> [[a]]
permsV3 [] = [[]]
```

```
permsV3 xs = concatMap h [0 .. length xs - 1]
  where
    h n = map (x:) (permsV3 (ls ++ rs))
      where
        (ls, x : rs) = splitAt n xs

permsV4 :: [a] -> [[a]]
permsV4 [] = [[]]
permsV4 xs = [x : ps | n <- [0 .. length xs - 1],
                       let (ls, x : rs) = splitAt n xs,
                       ps <- permsV4 (ls ++ rs)]

permsV5 :: [a] -> [[a]]
permsV5 [] = [[]]
permsV5 xs = [x : ps | n <- [0 .. length xs - 1],
                       let (x, ys) = outsert xs n,
                       ps <- permsV5 ys]

-- based on a solution by Daniel Krni
-- extremely compact, but requires an equality test
permsV6 :: Eq a => [a] -> [[a]]
permsV6 [] = [[]]
permsV6 xs = [x : ps | x <- xs,
                       ps <- permsV6 (filter (/=x) xs)]

-- original solution Daniel Krni
permsV7 :: Eq a => [a] -> [[a]]
permsV7 [] = [[]]
permsV7 xs = concat $ map (\x -> map (x:) (permsV7 $ filter (/=x) xs)) xs

permsTest =
  all (\perms -> test id setEq perms permsTests)
    [permsV1, permsV2, permsV2a, permsV3, permsV4, permsV5, permsV6, permsV7]

perms :: [a] -> [[a]]
perms = permsV1

-- an application: concurrent programming
-- determine the results of all possible schedulings
-- of some atomic computations

atomA1 x = x + 1
atomA2 x = 2 * x
atomA3 x = x * x

atomB1 x = 2*x + 1
atomB2 x = 3*x + 2
atomB3 x = 4*x + 3

compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

```
allResultsA, allResultsB :: Int -> [Int]
allResultsA x = map (($x).compose) (perms [atomA1, atomA2, atomA3])
allResultsB x = map (($x).compose) (perms [atomB1, atomB2, atomB3])

singleResults :: Int -> [Int]
singleResults x = map ($x) [atomA1, atomA2, atomA3]

t01 :: [Int -> Int]
t01 = [atomA1, atomA2, atomA3]

t02 :: Int -> Int
t02 = compose [atomA1, atomA2, atomA3]

t03 :: [[Int -> Int]]
t03 = perms [atomA1, atomA2, atomA3]

t04 :: [Int -> Int]
t04 = map compose (perms [atomA1, atomA2, atomA3])
```

3.9.7 Hutton Exercises

Hutton02

```
module Hutton.HaskellHutton02 where

dimdi = 19

double x = x + x

quadruple x = double (double x)

factorial n = product [1 .. n]

averageV1 ns = sum ns `div` length ns
averageV2 ns = div (sum ns) (length ns)

a1 = b + c
     where
       b = 1
       c = 2
d1 = a1 * 2

a2 = b + c
     where
       { b = 1 ; c = 2 }
d2 = a2 * 2

fact :: Integer -> Integer
fact 0 = 1                    -- Gleichung A
fact n = n * fact (n - 1)     -- Gleichung B
```

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

lastV1 xs = head (reverse xs)
lastV2 xs = head (drop (length xs - 1) xs)
lastV3 xs = xs !! (length xs - 1)

initV1 xs = take (length xs - 1) xs
initV2 xs = reverse (tail (reverse xs))

halve xs = (take n xs, drop n xs)
  where
    n = length xs `div` 2
```

Hutton03

```
module Hutton.HaskellHutton03 where

type Vector = (Int,Int)

addvec :: (Vector,Vector) -> Vector
addvec ((x1, y1), (x2, y2)) = (x1 + x2, y1 + y2)

add :: (Int, Int) -> Int
add (x, y) = x + y

add' :: Int -> (Int -> Int)
add' x y = x + y

add5 :: Int -> Int
add5 y = add (5, y)

add'5 :: Int -> Int
add'5 = add' 5

dave = take 5 -- Dave Brubeck

multV0 x y z = x * y * z
multV1 x y = \z -> x * y * z
multV2 x = \y -> \z -> x * y * z
--multV3 :: Num a => a -> a -> a -> a
multV3 = \x -> \y -> \z -> x * y * z

second xs = head (tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

double x = x * 2
```

```
palindrome xs = reverse xs == xs

twice f x = f (f x)

neg :: (Int, Int) -> (Int, Int)
neg (x, y) = (-x, -y)
```

Hutton04

```
module Hutton.HaskellHutton04 where

--signumV1 :: (Num a, Ord a) => a -> Int
signumV1 n | n < 0      = -1
           | n == 0     =  0
           | otherwise =  1

--pred :: Int -> Int
--pred (n + 1) = n

safetailV1 xs = if null xs then []
                else tail xs

safetailV2 xs | null xs = []
              | otherwise = tail xs

safetailV3 []       = []
safetailV3 (_ : xs) = xs

andV1 b1 b2 = if b1 then
                if b2 then True
                else False
              else False

andV2 b1 b2 = if b1 then
                b2
              else False
```

Hutton05

```
module Hutton.HaskellHutton05 where

pairs xs = zip xs (tail xs)

sorted xs = and [x <= y | (x, y) <- pairs xs]

s1 = sorted ([] :: [Int])

isPyth :: (Integer, Integer, Integer) -> Bool
isPyth (x,y,z) = x^2 + y^2 == z^2

pythsV1 :: Integer -> [(Integer, Integer, Integer)]
```

```
pythsV1 n =
  [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n], isPyth (x,y,z)]

-- finds all triples with x <= y
pythsV2 :: Integer -> [(Integer, Integer, Integer)]
pythsV2 n =
  [(x,y,z) | z <- [1..n],
             y <- [1..z-1],
             x <- [1..y], isPyth (x,y,z)]
    where
      isPyth (x,y,z) = x^2 + y^2 == z^2

allWithXeqY :: [(Integer, Integer, Integer)] -> [(Integer, Integer, Integer)]
allWithXeqY xs = [(x, y, z) | (x, y, z) <- xs, x == y]

allWithXltY :: [(Integer, Integer, Integer)] -> [(Integer, Integer, Integer)]
allWithXltY xs = [t | t@(x, y, _) <- xs, x < y]

factorsWOn :: Integer -> [Integer]
factorsWOn n = [x | x <- [1..n-1], n `mod` x == 0]

perfectsV1 :: Integer -> [Integer]
perfectsV1 n = [x | x <- [1..n], x == sum (factorsWOn x)]

perfectsV2 :: Integer -> [Integer]
perfectsV2 n = [x | x <- [1..n], x == sum (factorsWOn x)]
  where
    factorsWOn n = [x | x <- [1..n-1], n `mod` x == 0]

scalProd :: [Int] -> [Int] -> Int
scalProd xs ys = sum [x * y | (x,y) <- zip xs ys]

positions start x xs =
  [i | (x', i) <- zip xs [start ..], x' == x]
```

Hutton06

```
module Hutton.HaskellHutton06 where

import Prelude hiding ((++), and, concat, replicate, (!!), elem)

infixr 5 ++

myDrop n xs
  | n <= 0 = xs
myDrop _ [] = []
myDrop n (x : xs) = drop (n-1) xs

(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
reversePre :: [a] -> [a]
reversePre []       = []
reversePre (x : xs) = reversePre xs ++ [x]

reverseAccu :: [a] -> [a]
reverseAccu xs = h xs []
  where
    h []       accu = accu
    h (x : xs) accu = h xs (x : accu)

-- as specified in the Prelude
reverseFold :: [a] -> [a]
reverseFold = foldl (flip (:)) []

and :: [Bool] -> Bool
and []       = True
and (x : xs) = x && and xs

concat :: [[a]] -> [a]
concat []         = []
concat (xs : xss) = xs ++ concat xss

replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x

(!!) :: [a] -> Int -> a
_        !! n | n < 0 = error "negative index"
[]       !! _         = error "index too large"
(x : _)  !! 0 = x
(x : xs) !! n = xs !! (n-1)

elem :: Eq a => a -> [a] -> Bool
elem _ []                    = False
elem x (y : ys) | y == x     = True
                | otherwise = elem x ys

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

msort :: Ord a => [a] -> [a]
msort []  = []
--msort [x] = [x]
msort xs  = merge (msort (take n xs)) (msort (drop n xs))
  where n = length xs `div` 2

halve :: [a] -> ([a], [a])
```

```
halve xs = h xs [] []
  where
    h (x1 : x2 : xs) accu1 accu2 = h xs (x1 : accu1) (x2 : accu2)
    h [x]            accu1 accu2 = (x : accu1, accu2)
    h []             accu1 accu2 = (accu1, accu2)

msortV2 []  = []
msortV2 [x] = [x]
msortV2 xs = merge (msortV2 hsl) (msortV2 hsr)
  where (hsl, hsr) = halve xs
```

Programming Exercises: Higher-order functions and recursion

```
module Exer05Sol where

-- Develop some functions to work with order lists.
-- Make use of higher-order functions and/or recursion.

type ArtName = String    -- name of article
type Number = Int        -- number of ordered articles
type Order = (ArtName, Number)

type Price = Int    -- price of an article in Rappen
type Pricing = (ArtName, Price)
type PricedOrder = (ArtName, Number, Price)

-- Note: Order and Pricing are exactly the SAME type.
-- However, we distinguish them on the software engineering level,
-- but we must be careful.

ol01 :: [Order]
ol01 =
  [("Schraube M4", 100),
   ("Mutter M4", 100),
   ("Unterlegscheibe M4", 200)]

pl01 :: [Pricing]
pl01 =
  [("Schraube M4", 5),
   ("Unterlegscheibe M4", 2),
   ("Mutter M4", 5),
   ("Zahnrad 36Z", 1300)]

-- Given a name and a list of name item pairs, myLookup returns the first
-- item in the list that matches the given name.
-- If the list does not contain the given name, myLookup fails.
-- Later we will write a better function that returns a value indicating
-- whether an item has been found or not.
exa_myLookup =
  myLookup 39 [(5, 'a'), (39, 'b'), (7, 'c'), (39, 'd')] == 'b' &&
  myLookupV2 39 [(5, 'a'), (39, 'b'), (7, 'c'), (39, 'd')] == 'b'
```

```
myLookup :: Eq a => a -> [(a, b)] -> b
myLookup x ((x', y) : xys)
  | x' == x = y
  | otherwise = myLookup x xys
myLookup _ [] = error "lookup failed"

myLookupV2 :: Eq a => a -> [(a, b)] -> b
myLookupV2 x xys = snd (head (filter (\(x', _) -> x' == x) xys))

-- Given an order list and a pricing list, addPrices adds the prices
-- according to the pricing list to the order list.
-- Precondition:
--   All article names in the order list occur in the pricing list.
exa_addPrices =
  addPrices ol01 pl01 ==
    [("Schraube M4",100,500),
     ("Mutter M4",100,500),
     ("Unterlegscheibe M4",200,400)]

addPrices :: [Order] -> [Pricing] -> [PricedOrder]
addPrices ol pl = map (\(name, num) -> (name, num, myLookup name pl * num)) ol

-- totalPrice determines the total price of an order list.
exa_totalPrice =
  totalPrice (addPrices ol01 pl01) == 1400

totalPrice :: [PricedOrder] -> Price
totalPrice pol = sum (map (\(_, _, price) -> price) pol)

-- totalNumPrice determines the total number of items and the total price
-- of an order list.
totalNumPrice :: [PricedOrder] -> (Number, Price)
totalNumPrice pol = (sum nums, sum prices)
  where (_, nums, prices) = unzip3 pol

-- Returns items that (for the number ordered) cost more than a given maxPrice.
exa_tooExpensive =
  tooExpensive 450 (addPrices ol01 pl01) ==
    [("Schraube M4",100,500),
     ("Mutter M4",100,500)]

tooExpensive :: Price -> [PricedOrder] -> [PricedOrder]
tooExpensive maxPrice pol = filter (\(_, _, price) -> price > maxPrice) pol

-- Adds an order to an order list.
-- If the article name added already occurs in the order list,
-- the number is accordingly incremented.
exa_add =
  addOrder ("Mutter M4", 250) ol01 ==
    [("Schraube M4", 100),
     ("Mutter M4", 350),
```

```
    ("Unterlegscheibe M4", 200)]

addOrder :: Order -> [Order] -> [Order]
addOrder (name, num) ((name', num') : ol)
  | name == name' = (name, num + num') : ol
  | otherwise = (name', num') : addOrder (name, num) ol
addOrder nameNum [] = [nameNum]

-- addOrderList adds all orders of a new order list to an old order list.
exa_addOrderList =
  addOrderList (tail ol01) ol01 ==
    [("Schraube M4", 100),
     ("Mutter M4", 200),
     ("Unterlegscheibe M4", 400)]

addOrderList :: [Order] -> [Order] -> [Order]
addOrderList olNew olOld = foldr addOrder olOld olNew

-- Removes an order with a given article name from a given order list.
exa_removeOrder =
  removeOrder "Schraube M4" ol01 ==
    [("Mutter M4", 100),
     ("Unterlegscheibe M4", 200)]

removeOrder :: ArtName -> [Order] -> [Order]
removeOrder name ol = filter (\(name', _) -> name' /= name) ol
```

4 Verification

4.1 Correctness of Software

4.1.1 Software Development Process

Problem → Requirements → Specification → Implementation

Correctness issues:

- Do the requirements describe the problem in an adequate way?
- Is the specification an adequate formalization of the requirements? → Validation
- Is the implementation correct w.r.t. the specification? → Verification

4.1.2 Proving and Testing

- Testing: good for finding bugs
- Proving: good for showing there are no bugs
- A good practical method:
  - First: test your program to find as many possible errors as possible
  - Then: try to prove your program correct

4.1.3 Proving: The Very Idea

What do we need for such proofs?

- a completely formal specification language;
- an implementation language suited for this task;
- a mathematical theory that links specification and implementation language;
- tool support.

The most fundamental of all these approaches is the verification of imperative programs against specifications consisting of logical formulas using **Hoare** logic.

4.1.4 Dafny

Dafny is a specification and implementation language for proving the correctness of an implementation against a specification.

4.2 Specifications vs. Implementations

4.2.1 Implementation

The following Java method computes a partial function from int to int. ("greatest integer less or equal to square root of a")

```
int f(int a) {
  int t, s, i;
  t = 1; s = 1; i = 0;
  while (s <= a){
    t = t + 2;
    s = s + t;
    i = i + 1;
  }
  return i;
}
```

4.2.2 Specification

The following Dafny specification defines a partial function from int to int.

```
method F(a:int) returns (r:int)
  requires a >= 0
  ensures r*r <= a < (r+1)*(r+1)
```

- The requires clause declares a **precondition**. This is a condition we assume to hold before execution begins.
- The ensures clause declares a **postcondition**. This is a condition that is guaranteed to hold after execution ends, provided that the precondition holds before execution begins.

4.2.3 Key Difference

- The specification describes **what** the result of the function is **without** explaining how to compute it.
- The implementation describes **how** to compute the result of the function **without** explaining what it is.

4.2.4 Both in one, the spec informally (Java)

```
// Integer square root of an integer.
int iroot(int a)
  // Provided a >= 0, iroot(a) returns
  // the greatest integer r with r*r <=a
{
  int odd, square, root;
  odd=1; square=1; root=0;
  while (square <= a) {
    odd=odd+2;
    square=square+odd;
    root=root+1;
  }
  return root;
}
```

4.2.5 Both in one, the spec formally (Dafny)

```
method NatSquareRoot(a:int) returns (r:
    int)
  requires a >= 0
  ensures r*r <= a < (r+1)*(r+1)
{
  var d, s:int;
  d := 1; // Odd
  s := 1; // Square
  r := 0; // Root
  while s <= a
    invariant d == 2*r + 1
    invariant s == (r+1)*(r+1)
    invariant r*r <= a
  {
    d := d + 2;
    s := s + d;
    r := r + 1;
  }
}
```

4.2.6 Increased Readability through Redundancy

Since very special knowledge is needed for developing an efficient implementation from a specification, it is not reasonable to assume that it will be possible to construct a compiler that directly compiles a specification into efficiently executable code.

- Now we have two descriptions of the same problem → **Redundancy**
- The two descriptions provide very distinct points of view.

4.3 IML: Imperative (Model | Mini) Language

4.3.1 Top Level

⟨program⟩ ::= ⟨specification⟩⟨implementation⟩
⟨specification⟩ ::= 'specification'⟨precondition⟩ ⟨framecondition⟩⟨postcondition⟩
⟨precondition⟩ ::= 'requires'⟨assert⟩
⟨framecondition⟩ ::= 'modifies'⟨progvar⟩ (','⟨progvar⟩)*
⟨postcondition⟩ ::= 'ensures'⟨assert⟩
⟨implementation⟩ ::= 'implementation'⟨cmd⟩

A program variable ⟨progvar⟩ is a sequence of letters and digits not starting with a digit.

4.3.2 Commands

⟨cmd⟩ ::= ⟨skip⟩
|⟨assignment⟩
|⟨composition⟩
|⟨conditional⟩
|⟨loop⟩
⟨skip⟩ ::= 'skip'
⟨assignment⟩ ::= ⟨progvar⟩':='⟨aexpr⟩
⟨composition⟩ ::= ⟨cmd⟩';'⟨cmd⟩
⟨conditional⟩ ::= 'if'⟨bexpr⟩'then'⟨cmd⟩'else'⟨cmd⟩'end'
⟨loop⟩ ::= 'while'⟨bexpr⟩'invar'⟨assert⟩'do'⟨cmd⟩'end'

4.3.3 Terms and Arithmetic Expressions

⟨term⟩ ::= ⟨aliteral⟩
|⟨progvar⟩
|⟨oldvar⟩
|⟨boundvar⟩
|'-'⟨term⟩
|⟨term⟩⟨aopr⟩⟨term⟩
|'('⟨term⟩')'
|⟨funid⟩'('⟨term⟩ (','⟨term⟩)* ')'
⟨aopr⟩ ::= ' + '|' − '|' ∗ '

An arithmetic expression is the same as a term, except that ⟨oldvar⟩ and ⟨boundvar⟩ must not occur in it, and only implemented functions may be used.

- ⟨aliteral⟩ is a sequence of digits.
- ⟨oldvar⟩ is a sequence of letters and digits not starting with a digit, ending with a tilde.
- ⟨boundvar⟩ is a sequence of letters and digits not starting with a digit, ending with a prime.
- ⟨funid⟩ is a sequence of letters and digits not starting with a digit.

4.3.4 Assertions and Boolean Expressions

⟨assert⟩ ::= ⟨bliteral⟩
|⟨term⟩⟨ropr⟩⟨term⟩
|'not'⟨assert⟩
|⟨assert⟩⟨bopr⟩⟨assert⟩
|'('⟨assert⟩')'
|'(''forall'⟨boundvar⟩'|'⟨assert⟩')'
|'(''exists'⟨boundvar⟩'|'⟨assert⟩')'
⟨bliteral⟩ ::= 'true'|'false'
⟨ropr⟩ ::= ' = '|'/ = '|' < '|' ≤ '|' > '|' ≥ '
⟨bopr⟩ ::= '&&'|'||'|' ==> '|' <==> '

A boolean expression is the same as an assertion, except that terms are restricted to arithmetic expressions, and quantifications must not occur in it.

4.3.5 Terms, Assertions and Expressions

- Terms and arithmetic expressions are evaluated in a state to yield a value of type integer.
- Assertions and boolean expressions are evaluated in a state to yield a value of type boolean.
- Expressions have no side effects: their evaluation never changes state. (Think of them as being functional!) This property considerably simplifies verification.
- All terms, assertions, and expressions are total: they are defined in all possible states. (This is the reason why we have excluded division!)

Why so complicated?

- Why do we distinguish between terms and arithmetic expressions?
- Why do we distinguish between assertions and boolean expressions?
- The (arithmetic and boolean) expressions occur in the implementation context.
  - So they must be executable.
- The terms and assertions occur in the specification context.
  - There is no need for them to be executable.
- Recall: Verification is done at compiletime, not at runtime. The compiler will not generate any code for specification constructs!

4.3.6 Old-Variables

- We need a possibility in the postcondition to refer to the initial values of variables.
- Old-variables must not occur in preconditions - there they would make no sense.
- In order to record the initial values of variables, the verifier adds a corresponding assignment command for each old-variable to the very beginning of the command.
  - Note: This is done by the verifier; the compiler does not generate any code for such variables and thus for such assignment commands.
  - So there is no runtime overhead.
  - And there is not possibility to refer to old-variables in the implementation context.
  - In particular: Old-variables must not occur on the left-hand side of assignment commands.

4.3.7 Frameconditions

- A framecondition lists all program variables mentioned in precondition and postcondition whose values are allowed to be changed during execution.
- But the important point is this: The values of all other variables mentioned in precondition and postcondition must remain fixed.
- Variables local to the implementation are not considered here.
- Now the specification of our division program reads as follows:

```
requires a>=0 && b>0
modifies q, r
ensures a=b*q+r && 0<=r && r<b
```

- The verifier must perform a corresponding check: Each program variable occurring on the left-hand side of some assignment command and being mentioned in precondition or postcondition must be mentioned in the framecondition.
  - This is a simple syntactic check.

4.4 States

4.4.1 States and Assignment Commands

- The distinguishing feature of any imperative programming language is the explicit change of state by means of assignment commands.
- Execution of a program generates a sequence of states.
- A single state can be modelled by a function mapping the variables VAR of a program to their current values VAL: $STATE = VAR \to VAL$

4.4.2 States and Boolean Expressions

A boolean expression can be evaluated in a given state to yield a truth value.

4.4.3 States and Assertions

An assertion describes a set of states, namely the set of all states that satisfy the assertion, i.e., in which the assertion evaluates to *true*.

4.5 Recall Logic

4.5.1 Implication

- The notion of logical implication
  - 'if p then q', or
  - 'p implies q', or
  - 'q follows from p'

  is absolutely central for verification.
- It is formally written as $p \Rightarrow q$ (sometimes also as $q \Leftarrow p$.

- p is called the antecedent, and q the consequent of the implication.
- This is obviously true for all possible values of n.
  - Let $n = 6$. Thus $\underbrace{6 > 5}_{\text{true}} \Rightarrow \underbrace{6 > 3}_{\text{true}}$ is true.
  - Let $n = 4$. Thus $\underbrace{4 > 5}_{\text{false}} \Rightarrow \underbrace{4 > 3}_{\text{true}}$ is true.
  - Let $n = 2$. Thus $\underbrace{2 > 5}_{\text{false}} \Rightarrow \underbrace{2 > 3}_{\text{false}}$ is true.
- We observe that the implication is true if the antecedent is false, irrespective of the consequent.
- If the antecedent is false, the implication as a whole simply tells us nothing about the consequent - in particular nothing false.
- Therefore, if the antecedent is false, we say that the implication is vacuously true.
- With the antecedent we restrict our attention to the interesting cases.
- The only case in which an implication is false is if the antecedent is true, but the consequent is false.
- Of the following four statements (Of course, everybody knows that the moon is made of cheese...)
  - If the moon is made of chocolate, then 5 is a prime number.
  - If the moon is made of chocolate, then 5 is not a prime number.
  - If the moon is made of cheese, then 5 is a prime number.
  - If the moon is made of cheese, then 5 is not a prime number.

  only the last one is false.
- The example also tells us that there need not be any causal relationship between antecedent and consequent.
- Implication can be expressed by disjunction and negation: $p \Rightarrow q \equiv \neg p \vee q$.
- Example:
  - Let us read $\neg p$ as 'Hands up' and q as 'I shoot'. Then we see: 'Hands up or I shoot' is equivalent to 'If you don't take up your hands, then I shoot'.
- Consider the implications
  - $x = 5 \wedge y = 7 \Rightarrow x = 5$
  - $x = 5 \Rightarrow x = 5 \vee y = 7$
- Both are obviously true for all possible values of x and y.
- We see that the antecedent is more restrictive than the consequent.
- The set of states given by the antecedent is a subset of the set of states given by the consequent.
- We say that the antecedent is stronger than the consequent, or that the consequent is weaker than the antecedent.
- What is the strongest, what the weakest possible condition?
  - Think of a doorkeeper.
  - The strongest possible doorkeeper is a closed door (without doorkeeper).
  - The weakest possible doorkeeper is an open door (without doorkeeper).

4.5.2 Validity versus Truth

- An assertion p is called valid if it is true in all states.
- In this case we write: $\models p$.
- Otherwise, i.e., in case there exists a state in which the assertion is not true, we say the assertion is not valid and write: $\not\models p$
- Examples:
  - $\models n > 5 \Rightarrow n > 3$
  - $\not\models n > 3 \Rightarrow n > 5$

4.6 Hoare Triples

4.6.1 Syntax

- Essential for verification is the concept of a Hoare Triple.
- ⟨Hoare_triple⟩ ::= '{'⟨assert⟩'}'⟨cmd⟩'{'⟨assert⟩'}'
- The first assertion is called the precondition of the Hoare triple.
- The second assertion is called the postcondition.
- Examples:
  - $\{x = 5\}x := x + 1\{x = 17\}$

4.6.2 Semantics

- A Hoare triple is itself a boolean formula, depending on some state.
- But: We must consider two states in a Hoare triple:
  - the state before execution of the command, called the prestate, and
  - the state after ececution of the command, called the poststate.
- Let us consider the occurrences of variable x in the following Hoare triple: $\{x > 5\}x := x + 1\{x > 6\}$
  - the first and the third occurrence denote the value of x in the prestate;
  - the second occurrence denotes the address of variable x; and
  - the fourth occurrence denotes the value of x in the poststate.

- We see that variable x occurs with three different meanings in a single formula - a quite unusual situation in mathematics.
- Now we are ready to define the semantics.
- A Hoare triple: $\{P\}C\{Q\}$ is true in a given prestate, if the following implication is true:
  - If the prestate satisfies precondition P and execution of the command C terminates, then the poststate satisfies the postcondition Q.

  Or more formally:
  - prestate satisifes $P\wedge$ execution of C terminates $\Rightarrow$ poststate satisfies Q
- Example:
  - $\{x > 0\}x := x + 1\{x > 2\}$
    in prestate σ_0 with $\sigma_0(x) = 0$ this yields true
    in prestate σ_1 with $\sigma_1(x) = 1$ this yields false
    in prestate σ_2 with $\sigma_2(x) = 2$ this yields true

4.6.3 Validity

- A Hoare triple $\{P\}C\{Q\}$ is called valid if it is true in all prestates.
- In this case we write: $\models \{P\}C\{Q\}$
- In other words, it is valid if the following implication holds:
  - If execution of command C begins in any state that satisfies the precondition P and execution terminates, then the resulting state satisfies the postcondition Q.
- In this case, the command C is called partially correct with respect to precondition P and postcondition Q.
- Note that a valid Hoare triple does not provide any information concerning the resulting state if execution begins in any state that does not satisfy the precondition.
- Examples:
  $\not\models \{x = 5\}x := x + 1\{x = 17\}$
  $\models \{x > 5\}x := x + 1\{x > 6\}$
  $\not\models \{j = 0\}$ while $i = 0$ do skip end $\{k = 0\}$

4.6.4 Total Correctness

- Consider a Hoare triple be valid and execution of command C terminates in all prestates satisfying the precondition P.
- In other words, the following implication holds:
  - If execution of command C begins in any state that satisfies the precondition P, then execution terminates and the resulting state satisfies the postcondition Q.
- In this case, the command C is called totally correct with respect to precondition P and postcondition Q.
- It is often practically first to prove partial correctness and after this termination.
- Both together yield total correctness.

4.6.5 Partial versus Total Correctness

- Partial correctness means:
  - If the program ever terminates, then the result it produces is correct.
  - Or: the program cannot give a wrong answer - but we don't know whether it gives an answer at all.
- Total correctness means:
  - The program terminates and the result it produces is correct.
- Example:
  - You are in a hurry and ask a person for the way to the train station.
    - If it is a 'partially correct person', then you obtain either the correct way, or an excuse that the person does not know the way.
    - If it is a 'totally correct person', then you obtain the correct way.
    - But the person might be not even partially correct ... and you miss your train!
- Partial correctness is probably more important than just ensuring termination, since it guarantees that you will not be mislead - just ask another person.
- A program that does not 'crash' but produces a wrong result is generally by far more dangerous than a program that crashes and produces no result at all.

4.7 Weakest Preconditions

4.7.1 Hoare Logic and Weakest Preconditions

- Program verification essentially means to prove a Hoare triple $\{P\}C\{Q\}$ valid.
- Hoare logic is a logic for obtaining valid Hoare triples by purely deducting reasoning, that is, by mathematical proof.
- Deductive reasoning means successively applying inference rules to axioms and already obtained conclusions to obtain new conclusions.
- Usually such proofs are extremely long and boring, and must therefore be performed as automatically

as possible (by programs that are by themselves reliable ...)
- However, the proof is undecidable in the general case - there is no algorithm for this problem that works in all cases.
- But fortunately that does not stop engineers to find algorithms working in many special, but relevant, cases.
- The concept of weakest preconditions helps automating verification.

4.7.2 Hoare Triples and Weakest Preconditions

- Consider a Hoare triple $\{P\}C\{Q\}$.
- We can begin execution of C in any of all possible states. So the set of prestates is the same as the set of all possible states.
- Let us partition this set of prestates into six classes:
  - first according to P,
  - second according to termination of C,
  - third according to Q, provided C terminates.

  1. prestate satisfies P, execution terminates in poststate satisfying Q
  2. prestate satisfies P, execution terminates in poststate satisfying $\neg Q$
  3. prestate satisfies P, execution loops
  4. prestate satisfies $\neg P$, execution terminates in poststate satisfying Q
  5. prestate satisfies $\neg P$, execution terminates in poststate satisfying $\neg Q$
  6. prestate satisfies $\neg P$, execution loops
- The Hoare triple is false in all states of class 2, and true in all states of the remaining five classes. (Truth for classes 3,4,5,6 is vacuous.)
- Now consider a valid Hoare triple $\{P\}C\{Q\}$.
- Here are the resulting classes:

  1. prestate satisfies P, execution terminates in poststate satisfying Q
  2. empty
  3. prestate satisfies P, execution loops
  4. prestate satisfies $\neg P$, execution terminates in poststate satisfying Q
  5. prestate satisfies $\neg P$, execution terminates in poststate satisfying $\neg Q$
  6. prestate satisfies $\neg P$, execution loops
- Now again consider an arbitrary (valid or not) Hoare triple $\{P\}C\{Q\}$.
- Let us put together classes 1 and 4. Then we would have all prestates in which execution terminates in poststate satisfying Q.
- Let us put together classes 3 and 6. Then we would have all prestates in which execution loops.
- An assertion that describes exactly the classes 1,4,3,6 is called a weakest precondition of C and Q.
- Let W be a weakest precondition of C and Q. Then $\neg W$ describes exactly the classes 2 and 5 together.

An immediate consequence of all these definitions is the following:

Theorem 1:
Let W be a weakest precondition of a Command C and a postcondition Q. Then the Hoare triple $\{W\}C\{Q\}$ is valid. More formally: $\models \{W\}C\{Q\}$

- Let us consider the connection between valid Hoare triples and weakest preconditions.
- This connection is one of the key ideas for verification.

Theorem 2:
Let W be a weakest precondition of command C and postcondition Q, and P an assertion. Then: $\models \{P\}C\{Q\}$ if and only if $\models P \Rightarrow W$

- Why is the connection between valid Hoare triples and weakest preconditions one of the key ideas of verification?
- To prove a program correct w.r.t its specification essentially means to prove a corresponding Hoare triple to be valid.
- Given a command and a postcondition, the weakest precondition can be mechanically determined. Caveat:
  - At least, this is possible for commands not containing loops.
  - But loops require annotated invariants and correspondingly, a subtle modification of the notion of weakest precondition.
  - However, we will not go into these details.
- Thus, we can prove a Hoare triple $\{P\}C\{Q\}$ valid in two steps:
  - The programming part: Determine the weakest precondition, say W, of C and Q. Then construct the implication $P \Rightarrow W$.
  - The mathematical part: Prove the implication $P \Rightarrow W$ valid.
- But if the implication $P \Rightarrow W$ is not valid, then, according to our theorem, the Hoare triple $\{P\}C\{Q\}$ is not valid either.

4.7.3 Rules of Inference

- Let $P_1, P_2, \ldots, P_n$ and C be boolean formulas, here assertions or Hoare triples ($n \geq 0$).
- An inference rule is a construct of the following form: $\frac{P_1 P_2 \ldots P_n}{C}$.
- The formulas above the line are called premises, the formula below the line is called conclusion.
- If there are no premises ($n = 0$), the rule is called an axiom.
- The inference rule is called correct, if the validity of the conclusion follows from the validity of the premises.

4.7.4 For IML

Following are inference rules as well as weakest preconditions for the commands of IML:

- skip,
- assignment,
- composition,
- conditional, and
- loop.

In addition, the Rule of Consequence is presented. This is not a rule concerning any particular command, but a rule that links programming with mathematics. It is of utmost importance.

Skip Theorem (Skip Axiom)
Let P be an assertion. The inference rule $\frac{}{\{P\}\ \text{skip}\ \{P\}}$ is correct.
Examples:

$\models \{x > 6\}$ **skip** $\{x > 6\}$
$\models \{x + 2 > 5\}$ **skip** $\{x + 2 > 5\}$

- What about the following Hoare triples?
  $\{x > 6\}$ **skip** $\{6 < x\}$
  $\{x + 2 > 5\}$ **skip** $\{x > 3\}$
  $\{x^2 + 4x + 4 = 0\}$ **skip** $\{x = -2\}$
- The first two of them are 'obviously' valid.
- However, their validity does not follow from the Skip Axiom alone: Though the two assertions $x > 6$ and $6 < x$ are equivalent, they are not the same assertions. (The same holds for $x + 2 > 5$ and $x > 3$.)
- Assertions are syntactic objects - think of them as strings!
- In fact, the third Hoare triple is valid too - but we must know something about the solution of quadratic equations.

Rule of Consequence

- Obviously, some mathematical knowledge that has nothing directly to do with programming itself is required (and that not only for the third example, but also for the first two).
- That knowledge must be brought into the play - and this is done with the Rule of Consequence (RoC).
- The RoC allows us to plug in ordinary mathematics into Hoare logic.

Theorem (Rule of Consequence)
Let P, Q, R, S be assertions, and C a command. The inference rule $\frac{P \Rightarrow Q \{Q\} C \{R\} R \Rightarrow S}{\{P\} C \{S\}}$ is correct.
Example:

$$\frac{x>7 \Rightarrow x>6 \{x>6\}\ \textbf{skip}\ \{x>6\} x>6 \Rightarrow x>5}{\{x>7\}\ \textbf{skip}\ \{x>5\}}$$

We have three premises and one conclusion:

- The implications $x > 7 \Rightarrow x > 6$ and $x > 6 \Rightarrow x > 5$ are premises. They are purely mathematical formulas having nothing to do with programming directly. They are both valid.
- The Hoare triple $\{x > 6\}$ **skip** $\{x > 6\}$ is a premise. It is a pure programming construct having nothing to do with mathematics directly. It is valid, because of the Skip Axiom.
- The Hoare triple $\{x > 7\}$ **skip** $\{x > 5\}$ is the conclusion. It is valid, because of the RoC.

Skip and Rule of Consequence This proof rewritten more concisely and much closer to normal programming:

1. $\{x > 7\}$
2. $\{x > 6\}$
3. skip
4. $\{x > 6\}$
5. $\{x > 5\}$

- Premises: We read
  - lines 1 and 2 as implication (first premise of RoC);
  - lines 2 and 4 as Hoare triple (second premise of RoC);
  - lines 4 and 5 as implication (third premise of RoC).
- Conclusion. We read
  - lines 1 up to 5 as Hoare triple.
- Two other - and shorter - proofs of the same validity:

1. $\{x > 7\}$
2. skip
3. $\{x > 7\}$
4. $\{x > 5\}$

1. $\{x > 7\}$
2. $\{x > 5\}$

3. skip
4. $\{x > 5\}$

- All three proofs do their job.
- There is a more systematic way to obtain a proof → next section.

General Proof Procedure

- To prove $\{P\}C\{Q\}$ valid, we start with the postcondition Q, go backwards over the command C to determine a weakest precondition W, and construct the implication $P \Rightarrow W$.
- This implication is called a verification condition (VC).
- If this VC is valid, then the original Hoare triple $\{P\}C\{Q\}$ is valid too, because of the RoC.
- Weakest preconditions, and thus VCs, can be determined fully automatically.
- And they are the best preconditions, since they assume as little as possible to arrive at the postcondition.

Computing Weakest Preconditions

- We define a function wp, taking a command and an assertion as input, and producing an assertion as output.
- This function, applied to a command and a postcondition, returns a weakest precondition of the given command and postcondition.
- Such a function is called a predicate transformer, since it transforms a predicate (that is, an assertion) into a predicate.
- So the function transforms syntactic objects into syntactic objects.

Skip: Weakest Precondition

Theorem (wp of skip)

Let Q be an assertion. Then $wp\,(\textbf{skip}, Q) = Q$.

General Proof Procedure

- Our example revisited using the general proof procedure:
  1. $\{x > 7\}$ // given precondition $x > 7$
  2. $\{x > 5\}$ // compute $wp\,(\text{skip}, x > 5) = x > 5$
  3. skip // go backwards over command skip
  4. $\{x > 5\}$ // start here with given postcondition $x > 5$
- Finally, construct the verification condition $x > 7 \Rightarrow x > 5$.

Working Backwards and Software Engineering

- It looks strange in the first moment to start with the postcondition and go backwards to arrive at the precondition.
  - But the postcondition describes the actual task of the program and is thus given as starting point of program development.
  - And the precondition will be determined to find out under which circumstances the postcondition can be established.
- Happily, determining preconditions from postconditions is much simpler than the other way around.

General Proof Procedure

- Determining VCs (via computing weakest preconditions) can be done automatically by a tool called a verification condition generator.
- The VCs can then (hopefully automatically) be discharged (that is, proved valid) by a second tool called a theorem prover.

Assignment

- The basic ingredient of the backwards approach is textual substitution (search and replace of a text editor).
- Textual substitution however is quite subtle in presence of the bound variables occurring in quantifications.
- However, we solved this problem elegantly by choosing different kinds of identifiers for program variables and bound variables.

Textual Substitution

- Let E and R be expressions and let v be a variable.
- $E[v \leftarrow R]$ denotes the expression that is the same as E but with all (free) occurrences of v replaced by (R), that is, by R in parentheses, in order to maintain precedences.
- This process is purely textual - the meaning of the symbols is totally irrelevant.

Assignment

Theorem (Assignment Axiom)

Let Q be an assertion, E an arithmetic expression, and v a program variable. The inference rule $\frac{}{\{Q[v \leftarrow E]\}v := E\{Q\}}$ is correct.

Example:

$\models \{(x = 5)\,[x \leftarrow x + 1]\}x := x + 1\{x = 5\}$
perform textual substitution:
$\models \{((x + 1) = 5)\}x := x + 1\{x = 5\}$
with $\models x = 4 \Rightarrow ((x + 1) = 5)$ and the RoC we obtain:
$\models \{x = 4\}x := x + 1\{x = 5\}$

- Why is $\{Q[v \leftarrow E]\}v := E\{Q\}$ valid?
- Example:
  $\{(x + 1) = 7\}$
  $x := x + 1$
  $\{x = 7\}$
- Execution of the assignment command just re-

names the value of $x + 1$ in the prestate to the value x in the poststate.
  - In other words: What was called $x+1$ in the prestate is called x in the poststate.
- More generally: What was called E in the prestate is called v in the poststate.

1. Example:
$$\models \{(x = 5)\,[x \leftarrow 5]\}x := 5\{x = 5\}$$
$$\models \{(5 = 5)\,x := 5\{x = 5\}\}$$
$$\frac{\text{true}\Rightarrow 5{=}5\{(5{=}5)\}x{:=}5\{x{=}5\}}{\{\text{true}\}x{:=}5\{x{=}5\}}$$
2. Example:
$$\models \{(x \neq 5)\,[x \leftarrow 5]\}x := 5\{x \neq 5\}$$
$$\models \{(5 \neq 5)\}x := 5\{x \neq 5\}$$
$$\frac{\text{false}\Rightarrow 5{\neq}5\{(5{\neq}5)\}x{:=}5\{x{\neq}5\}}{\{\text{false}\}x{:=}5\{x{\neq}5\}}$$
3. Example:
$$\models \{\left(x^4 = 256\right)\,[x \leftarrow x{\cdot}x]\}x := x{\cdot}x\{x^4 = 256\}$$
$$\models \{(x \cdot x)^4 = 256\}x := x \cdot x\{x^4 = 256\}$$
$$\frac{x^8{=}256\Rightarrow(x{\cdot}x)^4{=}256\{(x{\cdot}x)^4{=}256\}x{:=}x{\cdot}x\{x^4{=}256\}}{\{x^8{=}256\}x{:=}x{\cdot}x\{x^4{=}256\}}$$

Assignment: Weakest Precondition

Theorem (wp of assignment)

Let Q be an assertion, E an arithmetic expression, and v a program variable. Then $wp(v := E, Q) = Q[v \leftarrow E]$.

Composition