Advanced Programming Paradigms

N. Kälin

October 4, 2020

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):

$$\underbrace{(a \cdot (a+b))} + b \cdot (a+b) \tag{1}$$
$$= \langle \text{LEIBINIZ, with } a \cdot (a+b) = a \cdot a + a \cdot b \rangle$$
$$\overbrace{(a \cdot a + a \cdot b)} + \underbrace{(b \cdot +(a+b))} \tag{2}$$
$$= \langle \text{LEIBNIZ, with } b \cdot (a+b) = b \cdot a + b \cdot b \rangle$$
$$a \cdot a + a \cdot b + \overbrace{b \cdot a + b \cdot b} \tag{3}$$

- 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 ← 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:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
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.

```haskell
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:

```haskell
> 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:

```haskell
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.

```haskell
(+) :: 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:

```haskell
> 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):