Lecture notes

Advanced Programming

WS 2018/19

Prof. Dr. Michael Hanus
Priv.Doz. Dr. Frank Huch

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

December 6, 2018

Pre-Preface

These lecture notes are a translation of the german lecture notes associated with the "Advanced Programming" lectures held by Prof. Michael Hanus. Moreover, these notes do not translate the entire lecture by Prof. Hanus but only the chapters relevant according to the newly established pre-masters conditions for international students in Computer Science at the University of Kiel.

Preface

This lecture introduces advanced programming concepts which go beyond the first semesters of programming. The following is presented on the basis of various programming languages that represent the most important programming paradigms. Modern functional programming techniques are developed using the language Haskell. Logical and constraint-oriented programming is shown in the language Prolog. Concepts for concurrent and distributed programming are presented using the language Java.

This script is a revised version of a transcript, originally written and typeset in LaTeX by Nick Prühs in SS 2009. My thanks go to Nick Prühs for the first LaTeX template as well as Björn Peemöller and Lars Noelle for corrections.

One more important note: This script is only intended to give an overview of what is covered in the lecture. It does not replace attending the lecture, which is important for understanding the concepts and techniques of advanced programming. For in-depth self-study, a look at the textbooks and references given in the lecture is recommended.

Kiel, February 2017 Michael Hanus

PS: Anyone who did not find any errors while reading the notes probably has not paid enough attention. I am thankful for all hints on errors I receive, both in person as well as in writing or by e-mail.

Contents

1 Java Generics

Since version 5.0 (released in 2004) Java supports *generic programming*: Classes and methods can be parameterized with types. This opens up similar possibilities as templates in C++. As an easy example we consider a very simple container class which can store either a value or no value. In Java, this could be defined as follows.

```java
public class Optional {

        private Object value;
        private boolean present;

        public Optional() {
                present = false;
        }

        public Optional(Object v) {
                value = v;
                present = true;
        }

        public boolean isPresent() {
                return present;
        }

        public Object get() {
                if (present) {
                        return value;
                }
                throw new NoSuchElementException();
        }
}
```

We use the fact that `NoSuchElementException` is derived from `RuntimeException`, that is, it is a so-called *unchecked exception* and does not need to be declared.

The class could be used as follows.

```java
Optional opt = new Optional(new Integer(42));

if (opt.isPresent()) {
    Integer n = (Integer) opt.get();
        System.out.println(n);
}
```

Each time the stored object of the container class is accessed, an explicit type cast is done. There is *no type security*: In case of a false cast a `ClassCastException` is thrown at runtime.

Keep in mind that the definition of the class `Optional` is independent of the type of the stored value; the type of `value` is `Object`. This allows using arbitrary types that are represented in an abstract way in the definition of the class: Passing a type as a parameter is called *parametric polymorphism*.

Type parameters are marked by angle brackets and are used in place of `Object`. The adapted class definition is.

```
public class Optional<T> {

        private T value;
        private boolean present;

        public Optional() {
                present = false;
        }

        public Optional(T v) {
                value = v;
                present = true;
        }

        public boolean isPresent() {
                return present;
        }

        public T get() {
                if (present) {
                        return value;
                }
                throw new NoSuchElementException();
        }
}
```

The class is now used like this.

```
Optional<Integer> opt = new Optional<Integer>(new Integer(42));

if (opt.isPresent()) {
        Integer n = opt.get();
    System.out.println(n);
}
```

Explicit type casts are not necessary and an expression like

```
opt = new Optional<Integer>(opt);
```

returns a type error at compile time.

Naturally, multiple type parameters are also possible.

```
public class Pair<A, B> {
```

```
        private A first;
        private B second;

        public Pair(A first, B second) {
                this.first = first;
                this.second = second;
        }

        public A first() {
                return first;
        }

        public B second() {
                return second;
        }
}
```

1.1 Interaction with Inheritance

It is also possible to restrict type parameters so that only classes can be used that provide certain methods. As an example we will extend the class `Optional` so that it implements the interface `Comparable`.

```
public class Optional<T extends Comparable<T>>
implements Comparable<Optional<T>> {
    ...
   @Override
   public int compareTo(Optional<T> o) {
       if (present) {
           return o.isPresent() ? value.compareTo(o.get()) : 1;
       } else {
            return o.isPresent() ? -1 : 0;
        }
    }
}
```

For interfaces as well as inheritance the keyword `extends` is used. It is also possible to list multiple restrictions of type variables. For three type parameters (`T`, `S` and `U`) this would look like the following.

```
<T extends A<T>, S, U extends B<T,S>>
```

In this example, `T` has to provide the methods of `A<T>` and `U` has to provide the methods of `B<T,S>`. `A` and `B` need to be classes or interfaces, not type variables.

1.2 Wildcards

The class `Integer` is a sub class of the class `Number`. Therefore, the following code should be valid.

```
Optional<Integer> oi = new Optional<Integer>(new Integer(42));
Optional<Number> on = oi;
```

However, the type system does not allow this because `Optional<Integer>` is **no** subtype of `Optional<Number>`! The problem becomes more obvious when we add a setter method to the class `Optional`.

```
public void set(T v) {
    present = true;
    value = v;
}
```

In consequence, this would allow the following.

```
on.set(new Float(42));
Integer i = oi.get();
```

Since `oi` and `on` are names for the same objects, executing this code would lead to a type error. For this reason the type system does not allow the above code.

Nevertheless, sometimes a supertype of polymorphic classes, that is, a type that includes all other types, is needed. An `Optional` of unknown type is described like this.

```
Optional<?> ox = oi;
```

The symbol "`?`" is known as a *wildcard* type. `Optional<?>` represents all other instances of `Optional`, for example `Optional<Integer>`, `Optional<String>` and `Optional<Optional<Object>>`.

One downside of this approach is that only methods, which fit every type, work.

```
Object o = ox.get();
```

We can use the method `set` with `ox` but we need a value that belongs to every type: `null`.

```
ox.set(null);
```

This is the only possible `set`-call.

Unfortunately, the wildcard type "`?`" is often not sufficient. For example, when multiple subtypes like GUI elements are stored in a collection. This can be solved by using *bounded wildcards*.

`<? extends A>` includes *all* subtypes of `A` *(covariance)*

`<? super A>` includes *all* supertypes of `A` *(contra variance)*

Keep in mind that a type is sub- and supertype of itself, that is, for all types T the properties `<T extends T>` and `<T super T>` hold.

In the following, a few examples of using bounded wildcards are shown.

Expression	valid?	Reasoning
`Optional<? extends Number> on = oi;`		
`Integer i = on.get();`	no!	`?` could also be `Float`
`Number n = on.get();`	yes	
`on.set(n);`	no!	`?` coulde be more specific than `Number`
`on.set(new Integer(42));`	no!	`?` could also be `Float`
`on.set(null);`	yes	
`Optional<? super Integer> ox = oi;`		
`Integer i = ox.get();`	no!	`?` could also be `Number` or `Object`
`Number n = ox.get();`	no!	`?` could also be `Object`
`Object o = ox.get();`	yes	
`ox.set(o);`	no!	`?` could also be `Number`
`ox.set(n);`	no!	`n` could also have the type `Float`
`ox.set(i);`	yes	

Figure 1.1: Expressions with wildcards

This means that we can extract objects of type `A` from an `Optional<? extends A>` but only insert `null`, while we can insert objects of the type `A` into an `Optional<? super A>` but only extract objects of type `Object`.

1.3 Type Inference

When initializing a variable with a generic type parameter, the parameter is stated twice.

```
Optional<Integer>        o = new Optional<Integer>(new Integer(42));
List<Optional<Integer>> l = new ArrayList<Optional<Integer>>(o);
```

Since version 7, the Java compiler is able to infer the generic type when initializing an object with `new` in most cases. Therefore, the type only needs to be stated in the variable declaration. When calling the `new` operator, the type is calculated by the diamond operator `<>`.

```
Optional<Integer> mv = new Optional<>(new Integer(42));
List<Optional<Integer>> l = new ArrayList<>(mv);
```

Only the whole type within the angle brackets can be omitted, types like `<Optional<>>` are not allowed. If the compiler is not able to infer the type, it still needs to be stated manually.

Besides the shorter code, the diamond operator also enables creating generic singleton objects, which was not possible before.

```
Optional<?> empty = new Optional<>();
```

This is especially useful when only one instance of immutable objects should be kept in memory.

2 Concurrent Programming in Java

An application in computer science is described as *concurrent* if it does not have a strictly sequential flow, but when the application has several activities which run concurrently, that is, (almost) in parallel. These activities are often referred to as tasks, threads, or processes. These tasks are sequentially running programs themselves. In a concurrent program, there are therefore not only one program counter which determines the next instruction to be processed but many program counters. We will see that the development of concurrent software is associated with many pitfalls. In this chapter we will discuss how to develop concurrent software that is as reliable as possible. Building on concurrent concepts, we will later also consider distributed software. We will use Java as programming language, but the concepts can also be applied to other languages.

Why do we need concurrent programming? We would often like one application to take over several tasks. At the same time the *reactivity* of the application should be preserved. Examples of such applications are listed in the following.

- GUIs
- operating system routines
- distributed applications (web servers , chat, ...)

Solution We achieve this by means of *Concurrency)*. By the use of threads or processes, individual tasks of an application can be programmed and executed independently of other tasks.

Parallelism The parallel execution of several processes intends to achieve faster execution (high-performance computing).

Distributed system Several components in a network work on a problem together. Usually there is a distributed task, sometimes distributed systems are used for parallelization.

When we speak of *multitasking*, we mean that the processor time is distributed by a scheduler among concurrent threads or processes. We distinguish between two types of multitasking.

Cooperative multitasking A thread continues to calculate until it returns control (for example with `yield()`) or until it waits for messages (`suspend()`). In Java we find this in so-called *green threads*.

Preemptive multitasking The scheduler can take control from tasks. Here we often enjoy more programming comfort because we do not need to think about where we should give up control.

2.1 Synchronization

2.1.1 Interprocess Communication and Synchronization

In addition to the generation of threads or processes, communication between them is also important. This is usually done via shared memory or variables. We consider the following example in pseudo code.

```
int i = 0;
par
  { i = i + 1; }
  { i = i * 2; }
end par;
print(i);
```

Concurrency makes programs non-deterministic, that is, different results can be obtained depending on the scheduling. Thus, the above program can generate outputs 1 or 2, depending on how the scheduler executes the two concurrent processes. If the result of a program run depends on the order of the scheduling, this is called a *race condition*.

In addition to the two results 1 and 2, another result, namely 0, is also possible. This is due to the fact that it has not yet been clearly specified which actions are really executed atomically. By translating the program into byte or machine code, the following instructions can result.

1. `i = i + 1;` → `LOAD i; INC; STORE i;`
2. `i = i * 2;` → `LOAD i; SHIFTL; STORE i;`

Then the following sequence leads to output 0.

```
(2) LOAD i;

(1) LOAD i;
(1) INC;
(1) STORE i;

(2) SHIFTL;
(2) STORE i;
```

So we need synchronization to ensure the atomic execution of certain sections of code that work concurrently on the same resources. We call such code sections *critical sections*.

2.1.2 Synchronization with Semaphores

A well-known concept for synchronizing concurrent threads or processes goes back to Dijkstra from 1968. Dijkstra developed an abstract data type with the aim of synchronizing the atomic (uninterrupted) execution of certain program sections. These *semaphores* provide two atomic operations.

```
p(s) {
  if   s >= 1
  then s = s - 1;
  else add executing thread to queue for s and suspend it;
}

v(s) {
  if   waiting list for s not empty
  then wake first process in queue
  else s = s + 1;
}
```

`p(s)` stands for pass or *passeer*, `v(s)` stands for leave or *verlaat*. Now we can prevent the output 0 of our above program as follows.

```
int i = 0;
Semaphore s = 1;
par
  { p(s); i = i + 1; v(s); }
  { p(s); i = i * 2; v(s); }
end par;
```

The initial value of the semaphore determines the maximum number of processes in the critical area. Usually we find the value 1 here. We also call such semaphores *binary semaphore*.

Another application of semaphores is the *producer-consumer problem*: n producers produce goods that are consumed by m consumers. One simple solution to this problem uses an unrestricted buffer.

```
Buffer buffer = ...
Semaphore num = 0;
```

The producer's code looks like this.

```
while (true) {
  newproduct = produce();
  push(newproduct, buffer);
  v(num);
}
```

The consumer's code looks like this.

```
while (true) {
  p(num);
  prod = pull(buffer);
  consume(prod);
}
```

What is still missing is the synchronization to `buffer`. The synchronization can be realized by adding another semaphore.

```
Buffer buffer = ...
Semaphore num = 0;
```

```
Semaphore bufferAccess = 1;
```

Below is the adapted code of the producer.

```
while (true) {
  newproduct = produce();
  p(bufferAccess);
  push(newproduct, buffer);
  v(bufferAccess);
  v(num);
}
```

Below is the adapted code of the consumer.

```
while (true) {
  p(num);
  p(bufferAccess);
  prod = pull(buffer);
  v(bufferAccess);
  consume(prod);
}
```

However, the use of semaphores also has some disadvantages. The code with semaphores quickly looks unstructured and confusing. In addition, we cannot use semaphores compositionally: the simple code `p(s); p(s);` can already generate a *deadlock* on a binary semaphore `s`.

The concept of *monitors* which may be familiar from lectures like "Operating and Communication Systems" offers an improvement here. In fact, Java uses a mechanism similar to these monitors for synchronization.

2.1.3 Dining Philosophers

The *dining philosophers* problem with n philosophers can be modelled as follows using semaphores.

```
Semaphore stick1 = 1;
Semaphore stick2 = 1;
Semaphore stick3 = 1;
Semaphore stick4 = 1;
Semaphore stick5 = 1;

par { phil(stick1, stick2); }
    { phil(stick2, stick3); }
    { phil(stick3, stick4); }
    { phil(stick4, stick5); }
    { phil(stick5, stick1); }
end par;
```

The code for philosopher `i` looks like the following.

```
public phil(stickl,stickr) {
  while (true) {
    think();
```

```
      p(stickl);
      p(stickr);

      eat();

      v(stickl);
      v(stickr);
    }
  }
```

However, a deadlock can occur if all philosophers take their left stick at the same time. We can avoid this deadlock by putting it back.

```
  while (true) {
    think();

    p(stickl);

    if (lookup(stickr) == 0) {    # look up integer value of stick
      v(stickl);
    } else {
      p(stickr);

      eat();

      v(stickl);
      v(stickr);
    }
  }
```

Here `lookup(s)` denotes a lookup function of the abstract data type semaphore that returns the integer value of a semaphore `s`.

The program now has a livelock, that is, individual philosophers can starve to death. We do not want to discuss this problem further here.

2.2 Threads in Java

2.2.1 The Class `Thread`

The API of Java offers a class `Thread` in the package `java.lang`. Own threads can be derived from this. The code to be executed in parallel is written to the `run()` method. After we have created a new thread with the help of its constructor, we can start it for concurrent execution with the method `start()`.

We consider the following simple thread as an example.

```
  public class ConcurrentPrint extends Thread {
    private String s;
```

```
  public ConcurrentPrint(String s) {
    this.s = s;
  }

  public void run() {
    while (true) {
      System.out.print(s + " ");
    }
  }

  public static void main(String[] args) {
    new ConcurrentPrint("a").start();
    new ConcurrentPrint("b").start();
  }
}
```

The above program flow can lead to many possible outputs:

```
a a b b a a b b ...
a a a b b ...
a b a a a b a a b b ...
a a a a a a a a a a ...
```

The latter is guaranteed if cooperative scheduling is used.

2.2.2 The Interface `Runnable`

Java does not offer inheriting from multiple classes. Therefore, an extension of the class `Thread` is often unfavorable. An alternative is the Interface `Runnable`:

```
public class ConcurrentPrint implements Runnable {
  private String s;

  public ConcurrentPrint(String s) {
    this.s = s;
  }

  public void run() {
    while (true) {
      System.out.print(s + " ");
    }
  }

  public static void main(String[] args) {
    Runnable aThread = new ConcurrentPrint("a");
    Runnable bThread = new ConcurrentPrint("b");

    new Thread(aThread).start();
    new Thread(bThread).start();
  }
}
```

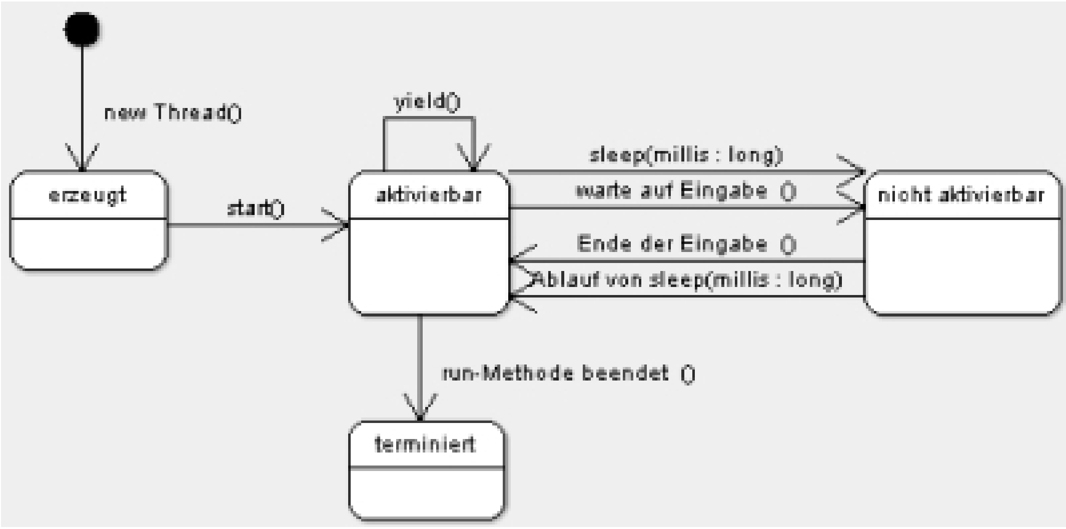

Figure 2.1: States of threads

Note that within the above implementation of `ConcurrentPrint`, `this` no longer returns an object of type `Thread`. The current thread object can instead be reached via the static method `Thread.currentThread()`.

2.2.3 Properties of Thread Objects

Every thread object in Java has a number of properties.

Name Each thread has a name, such as `"main-Thread"`, `"Thread-0"` or `"Thread-1"`. Access to the name of a thread is defined using the methods `getName()` and `setName(String)`. You can use these for example as names for debugging.

State Each thread is always in a certain state. An overview of these states and the state transitions is shown in figure 2.1. A thread object remains in the *terminated* state until all references to it have been discarded.

Demon A thread can be created as a background thread by calling `setDaemon(true)` *before* calling the `start()` method. The JVM terminates when only daemon threads are running. Examples of such threads are AWT threads or the garbage collector.

Priority Each thread in Java has a specific priority that is platform-specific.

Thread group Threads can also be divided into groups for simultaneous management.

Method `sleep(long)` Lets the thread sleep for the specified time. A call to this method can throw a `InterruptedException` which must be caught.

2.2.4 Synchronization of Threads

Java offers a monitor-like concept for thread synchronization that allows to set and release locks on objects.

The methods of a thread in Java can be declared as `synchronized` . In all synchronized methods of an object there may be a maximum of one thread at a time. This also includes calculations that are called in a synchronized method and unsynchronized methods of the same object. A synchronized method is not left by calling `sleep(long)` or `yield()`.

Furthermore, each object has its own *lock* . When attempting to execute a method declared as `synchronized`, we distinguish three cases.

1. If the lock is released, the thread takes it.
2. If the thread already has the lock, it continues.
3. Otherwise, the thread is suspended.

The lock is released again if the method is exited in which it was acquired. Compared to semaphores, the Java approach seems more structured, you can't forget to unlock. Yet it is less flexible.

2.2.5 The Example Class `Account`

A simple example is intended to illustrate the use of `synchronized` methods. We are looking at an implementation of a class for a bank account.

```
public class Account {
  private int balance;

  public Account(int initialDeposit) {
    balance = initialDeposit;
  }

  public synchronized int getBalance() {
    return balance;
  }

  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

We now want to use the class as follows.

```
Account a = new Account(300);
...
a.deposit(100); // concurrent, first thread
...
a.deposit(100); // concurrent, second thread
...
System.out.println(a.getBalance());
```

The calls of the method `deposit(int)` should be made concurrently from different threads. Without the keyword `synchronized`, the output would be `500`. The output `400` would also be possible (see section 2.1.1), Interprocess communication and synchronization). With `synchronized`, an output of `500` is guaranteed.

2.2.6 A Closer Look at `synchronized`

Inherited synchronized methods do not necessarily have to be synchronized again. If you overwrite such methods, you can omit the keyword `synchronized`. This is called *refined implementation*. The method of the superclass remains `synchronized`. On the other hand, unsynchronized methods can also be overwritten by synchronized ones.

Class methods declared as synchronized (`static synchronized`) have no interaction with synchronized object methods. The class therefore has its own lock.

You can also synchronize individual statements.

```
synchronized (expr) block
```

The expression `expr` must evaluate an object, whose lock is then used for synchronization. Strictly speaking, synchronized methods are only syntactic sugar: The method declaration

```
synchronized A m(args) block
```

is expanded to

```
A m(args) {
  synchronized (this) block
}
```

Synchronizing individual statements is useful to reduce the amount of code that needs to be synchronized or sequentialized.

```
private double state;

public void calc() {
  double res;
  // do some really expensive computation
  ...
  // save the result to an instance variable
  synchronized (this) {
    state = res;
  }
}
```

Synchronizing individual statements is also useful to synchronize on other objects. We consider a simple implementation of a synchronized collection as an example.

```
class Store {
  public synchronized boolean hasSpace() {
    ...
  }

  public synchronized void insert(int i)
```

```
            throws NoSpaceAvailableException {
      ...
    }
  }
```

We now want to use this collection as follows.

```
Store s = new Store();
...
if (s.hasSpace()) {
  s.insert(42);
}
```

But this leads to problems, because we cannot exclude that a re-schedule happens between the calls of `hasSpace()` and `insert(int)`. Since defining special methods for such cases often turns out to be impracticable, we better use the collection like this.

```
synchronized(s) {
  if (s.hasSpace()) {
    s.insert(42);
  }
}
```

2.2.7 Differentiating Synchronization in the Context of OO

We call synchronized methods and synchronized instructions in object methods *server-side synchronisation*. Synchronization of the calls of an object is called *client-side synchronisation*.

For efficiency reasons, Java API objects, especially collections, are no longer synchronized. For Collections, however, there exist synchronized versions via wrappers such as `synchronizedCollection`, `synchronizedSet`, `synchronizedSortedSet`, `synchronizedList`, `synchronizedMap` or `synchronizedSortedMap`.

Safely copying a list into an array is possible in two different ways now. First, we create an instance of a synchronized list.

```
List<Integer> unsyncList = new List<Integer>();
... // fill the list
List<Integer> list = Collections.synchronizedList(unsyncList);
```

Now we copy this list to an array with either a single line

```
Integer[] a = list.toArray(new Integer[0]);
```

or with

```
Integer[] b;

synchronized (list) {
  b = new Integer[list.size()];
  list.toArray(b);
}
```

With the second, two-line variant, synchronization to the list is indispensable: We access the collection in both lines and cannot guarantee that another thread will not change the collection in the meantime. This is a classic example of client-side synchronization.

2.2.8 Communication Between Threads

Threads communicate with each other via shared objects. How to find we find out when a variable contains a value? For this there are several possible solutions.

The first option is to display the modification of a component of the object, for example, by setting a flag (`boolean`). However, this has the disadvantage that checking the flag leads to busy waiting. Busy waiting means that a thread is waiting for the occurrence of an event, thereby continuing to calculate and thus consuming resources like processor time. Therefore, a thread is suspended using the method `wait()` of the object and woken up later by using `notify()` or `notifyAll()`.

```java
public class C {
  private int state = 0;

  public synchronized void printNewState()
    throws InterruptedException {

    wait();
    System.out.println(state);
  }

  public synchronized void setValue(int v) {
    state = v;
    notify();
    System.out.println("value set");
  }
}
```

Two thread call the methods `printNewState()` and `setValue(42)` concurrently. Now the *only* possible output is

```
value set
42
```

If the call of `wait()` only comes after the method `setValue(int)` has already been left by the first thread, this leads to the output `value set`.

The methods `wait()`, `notify()` and `notifyAll()` may only be used within `synchronized` methods or blocks and are methods of the locked object with the following semantics.

`wait()` puts the executing thread to sleep and releases the lock of the object again.

`notify()` awakens *one* sleeping thread of the object and continues with its own calculation. The awakened thread now applies for the lock. If no thread sleeps, the `notify()` is lost.

`notifyAll()` does the same as `notify()`, only for all threads that were laid to sleep with `wait()` for this object.

Please note that these three methods may only be called on objects whose lock has been received before. The call must therefore be made in a `synchronized` method or in a `synchronized` block, otherwise a `IllegalMonitorStateException` is thrown at runtime.

We would like to write a program that outputs all changes of the state.

```
...
private boolean modified = false; // signals state changes
...
public synchronized void printNewState()
  throws InterruptedException {
  while (true) {
    if (!modified) {
      wait();
    }

    System.out.println(state);
    modified = false;
  }
}

public synchronized void setValue(int v) {
  state = v;
  notify();
  modified = true;
  System.out.println("value set");
}
```

One thread now executes `printNewState()`, other threads change the state using `setValue(int)`. This leads to a problem: With several setting threads, the output of individual intermediate states can be lost. So `setValue(int)` also has to wait and wake up if necessary.

```
public synchronized void printNewState()
  throws InterruptedException {
  while (true) {
    if (!modified) {
      wait();
    }

    System.out.println(state);
    modified = false;
    notify();
  }
}

public synchronized void setValue(int v)
  throws InterruptedException {
  if (modified) {
    wait();
  }
```

```
    state = v;
    notify();
    modified = true;
    System.out.println("value set");
  }
```

But now it is not guaranteed that the call of `notify()` in the method `setValue(int)` wakes up the `printNewState` thread! In Java we solve this problem with `notifyAll()` and accept a little busy waiting.

```
  public synchronized void printNewState()
    throws InterruptedException {
    while (true) {
      while (!modified) {
        wait();
      }

      System.out.println(state);
      modified = false;
      notify();
    }
  }

  public synchronized void setValue(int v)
    throws InterruptedException {
    while (modified) {
      wait();
    }

    state = v;
    notifyAll();
    modified = true;
    System.out.println("value set");
  }
```

The `wait()` method is also overloaded several times in Java:

`wait(long)` interrupts execution for the specified number of milliseconds.

`wait(long, int)` interrupts the execution for the specified number of milli- and nanoseconds.

Note: It is strongly discouraged to base the correctness of the program on these overloads! The calls `wait(0)`, `wait(0, 0)` and `wait()` all cause the thread to wait until it wakes up again.

2.2.9 Case Study: Single-element Buffer

A single-element buffer is convenient for communication between threads. Since the buffer is single-element, it can only be empty or full. A value can be written into an empty buffer via a method `put` from a full buffer. The value can be removed using `take`.

`take` suspends on an empty buffer, `put` suspends on a full buffer.

```
public class Buffer1<T> {
  private T content;
  private boolean empty;


  public Buffer1() {
    empty = true;
  }

  public Buffer1(T content) {
    this.content = content;
    empty = false;
  }


  public synchronized T take() throws InterruptedException {
    while (empty) {
      wait();
    }

    empty = true;
    notifyAll();

    return content;
  }

  public synchronized void put(T o) throws InterruptedException {
    while (!empty) {
      wait();
    }

    empty = false;
    notifyAll();
    content = o;
  }

  public synchronized boolean isEmpty() {
    return empty;
  }
}
```

What is unfortunate about the above solution is that too many threads are awakened, that is, `notifyAll()` always awakens all reading threads as well as all writing threads, most of which are immediately put to sleep again. Can we wake up threads in a targeted way? Yes! We use special objects to synchronize the `take` and `put` threads.

```
public class Buffer1<T> {
  private T content;
  private boolean empty;
```

```
  private Object r = new Object();
  private Object w = new Object();


  public Buffer1() {
    empty = true;
  }

  public Buffer1(T content) {
    this.content = content;
    empty = false;
  }


  public T take() throws InterruptedException {
    synchronized (r) {
      while (empty) {
        r.wait();
      }

      synchronized (w) {
        empty = true;
        w.notify();

        return content;
      }
    }
  }

  public void put(T o) throws InterruptedException {
    synchronized(w) {
      while (!empty) {
        w.wait();
      }

      synchronized (r) {
        empty = false;
        r.notify();
        content = o;
      }
    }
  }

  public boolean isEmpty() {
    return empty;
  }
}
```

Here the `while` is very important! Another thread entering the method from the outside could overtake a waiting (and just awakened) thread!

2.2.10 Exiting and Interrupting Threads

Java offers several ways to terminate threads:

1. terminating the `run()` method
2. aborting the `run()` method
3. calling the `destroy()` method (deprecated, partly no longer implemented)
4. demon thread and program end

With 1 and 2 all locks are released. With 3, locks are not released which makes this method uncontrollable. For this reason, this method should not be used. With 4, the locks do not matter.

Java also provides a way to interrupt threads via *interrupts* . Each thread has a flag indicating interrupts.

The `Thread` method `interrupt()` sends an interrupt to a thread, the flag is set. If the thread is sleeping due to a call to `sleep()` or `wait()`, it is awakened and a `InterruptedException` is thrown.

```
synchronized (o) {
  ...
  try {
    ...
    o.wait();
    ...
  } catch (InterruptedException e) {
    ...
  }
}
```

In an interrupt after calling `wait()`, the `catch` block is not entered until the thread has recovered the lock on the `o` object of the surrounding `synchronized` block!

In contrast, in the suspension with `synchronized` the thread is not awakened, but only the flag is set.

The method `public boolean isInterrupted()` tests if a thread has received interrupts. `public static boolean interrupted()` tests the current thread for an interrupt and clears the interrupted flag.

Handling interrupts in a `synchronized` method is possible as follows:

```
synchronized void m(...) {
  ...
  if (Thread.currentThread().isInterrupted()) {
    throw new InterruptedException();
  }
}
```

If a `InterruptedException` is caught, the flag is also cleared. Then the flag needs to be set again!

2.3 Distributed Programming in Java

As an abstraction of network communication, Java offers the *Remote Method Invocation (RMI)* . This allows remote objects to be used on other computers as if they were local objects. In order for the data to be sent over a network, it must be converted (arguments and results of method calls) into byte sequences, usually referred to as serialization.

2.3.1 Serialization/Deserialization of Data

The serialization of an object `o` returns a byte sequence, the deserialization of the byte sequence returns a new object `o1`. Both objects should be the same with regard to their behavior, but have different object identities, i.e. `o1` is a copy of `o`.

The (de-)serialization takes place recursively. Thus, contained objects must also be (de)serialized. To specify that an object can be serialized, Java offers the interface `Serializable`.

```
public class C implements java.io.Serializable { ... }
```

This interface does not contain any methods and is therefore only a "marker interface" that specifies that objects of this class can be (de-)serialized.

During serialization, certain time- or security-critical parts of an object can be hidden using the keyword `transient` :

```
protected transient String password;
```

Transient values should be set explicitly after deserialization (for example a timer) or should not be used (for example passwords).

For serializing you use the class `ObjectOutputStream` , for deserializing the class `ObjectInputStream` . Their constructors are a `OutputStream` and `InputStream` respectively. Objects can be written

```
public void writeObject(Object o)
```

using and read by means of

```
public final Object readObject()
```

and a cast of appropriate type.

This could be used to "manually" transfer objects from one machine to another (for example by using socket connections) and then work with them on the other machine. You can avoid this manual work, however, if you only want to use certain functionalities of an object somewhere else. This is made possible in Java by Remote Method Invocation.

2.3.2 Remote Method Invocation (RMI)

In OO-programming we have a client-server view of objects. When a method is called, the calling object is seen as the client and the called object as the server.

In a distributed context, messages become real messages on the Internet (transmitted via TCP). Processes that communicate with each other are

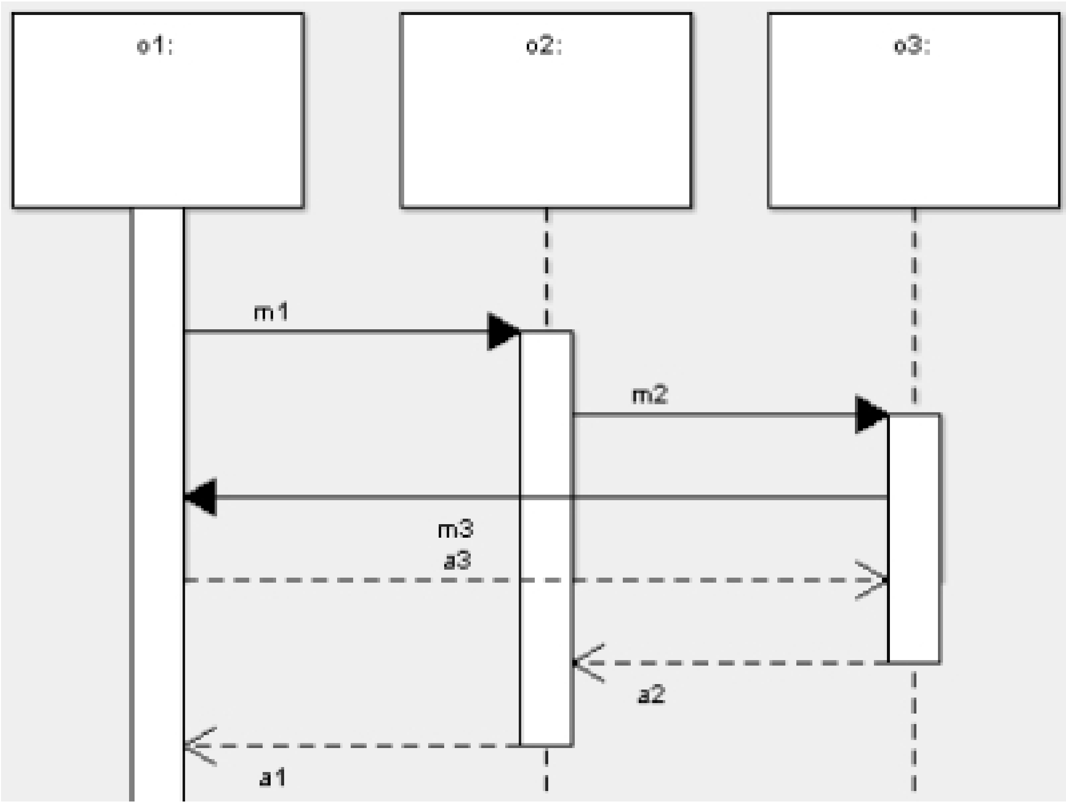

Figure 2.2: Remote Method Invocation in Java

- servers that provide information.
- clients that request information.

Any communication pattern (for example peer-to-peer) can be represented. The idea of RMI goes back to *Remote Procedure Call (RPC)*, which was developed for C.

First, an RMI client needs a reference to the remote object. The RMI registration serves this purpose. It requests the reference using a URL:

```
rmi://hostname:port/servicename
```

`hostname` can be a computer name or an IP address, `servicename` is a string describing an object. The default port of RMI is 1099.

There is also a second way to access a remote object in a program: as the argument of a method call. Usually you can use the above registration only for the "‘first contact”, then (remote) objects are exchanged and used transparently (like local objects).

Such objects can be distributed on any computer. Method calls for remote objects are implemented by network communication.

The interface for RMI is divided into stub and skeleton. Since Java 5 these are no longer visible, but are generated implicitly at runtime.

The network communication takes place via TCP/IP, but this is also not visible for the application programmer.

The parameters and the return value of a remotely called method must of course be converted into bytes and be transferred. The following rules apply.

- For `Remote` objects only a reference of the object is transferred.
- `Serializable` objects are converted to `byte[]`. On the other side a copy of the object will be created.
- Primitive values are copied.

To use objects from remote nodes, an RMI Service Interface must first be defined for this object. This interface describes the methods that can be called from another computer. Consider a simple "Flip" server as an example, that is, an object containing a Boolean state which can be changed by a `flip` message. The RMI service interface can then be defined as follows.

```
import java.rmi.*;

public interface FlipServer extends Remote {
  public void flip() throws RemoteException;
  public boolean getState() throws RemoteException;
}
```

An implementation of the RMI interface on the server side could look like the following.

```
import java.rmi.server.*;
import java.rmi.*;

public class FlipServerImpl extends UnicastRemoteObject
                            implements FlipServer {

  private boolean state;

  public FlipServerImpl() throws RemoteException {
    state = false;
  }

  public void flip() {
    state = !state;
  }

  public boolean getState() {
    return state;
  }
}
```

In older Java versions it was necessary to generate the stub and skeleton classes with the RMI compiler `rmic` - this is no longer the case. The class `UnicastRemoteObject` represents the simplest way of realizing remote objects. All necessary translation steps are done automatically. In some cases, it may be necessary to intervene in the (de-)serialization process yourself to define properties of the `RemoteObjects`. For this purpose, other interfaces exist.

2.3.3 The RMI Registry

To access a server object from another host, Java provides an RMI registry. To access a remote object, an RMI registry server must be initialized on the host on which the server object is executed. This RMI registry server can be started explicitly using the command `rmiregistry` (for example, in a UNIX shell). Its use for object registration is shown in the following example.

```
import java.rmi.*;
import java.net.*;

public class Server {
  public static void main(String[] args) {
    try {
      String host = args.length >= 1 ? args[0] : "localhost";
      String uri  = "rmi://" + host + "/FlipServer";

      FlipServerImpl server = new FlipServerImpl();
      Naming.rebind(uri, server);
    } catch (MalformedURLException|RemoteException e) { ... }
  }
}
```

`rebind` registers the name of the server object. On the client side, the RMI registry must be contacted to get a reference to the remote object so that its `flip` method can be called.

```
import java.rmi.*;
import java.net.*;

public class Client {
  public static void main(String[] args) {
    try {
      String host = args.length >= 1 ? args[0] : "localhost";
      String uri = "rmi://" + host + "/FlipServer";

      FlipServer s = (FlipServer) Naming.lookup(uri);
      s.flip();
      System.out.println("State: " + s.getState());
    } catch (MalformedURLException|NotBoundException|RemoteException e) {
      ...
    }
  }
}
```

However, remind that the RMI registry is only used for a "‘first contact” und usually only on server object is registered. After that, other remote objects are passed as prameters or return values and nodes in the distributed system get aware of all relevant objects of the whole system.

Thus, RMI represents a continuation of the sequential programming on distributed objects. This means that several distributed processes can access an object "‘simultaneously". So you also have to consider the problem of concurrent synchronization! However, synchronization can be more difficult here, as the following example shows.

As we have seen above, *client-side synchronization* is to guarantee the atomic execution of several methods which can already be synchronized. For this purpose we consider the simple Flip-server once again and a client that uses the `flip` method twice without interrupt calls.

```
...
FlipServer s = (FlipServer) Naming.lookup(uri);
synchronized(s) {
    System.out.println("State1: " + s.getState());
    s.flip();
    Thread.sleep(2000);
    s.flip();
    System.out.println("State2: " + s.getState());
}
...
```

Since the client synchronizes both calls on the flip server `s`, no one else should be able to change the state of the flip server during this time, so that the results of both outputs are always the same. This would also be the case in the purely concurrent context, but in the distributed context this no longer applies, since synchronization does not take place on the remote objects, but on the local stub objects!

Dynamic loading, security concepts or distributed memory cleanup (garbage collection) are other aspects of Java RMI that we do not consider here.

3 Functional Programming

One focus of this lecture is this chapter on functional programming. The practical relevance of functional programming is emphasized, for example, by Thomas Ball and Benjamin Zorn of Microsoft in the article [3], which bears the meaningful subtitle "Industry is ready and waiting for more graduates educated in the principles of programming languages".

> Second, would-be programmers (CS majors or non-majors) should be exposed as early as possible to functional programming languages to gain experience in the declarative programming paradigm. The value of functional/declarative language abstractions is clear: they allow programmers to do more with less and enable compilation to more efficient code across a wide range of runtime targets.

Another interesting recommendation of the authors is the following.

> First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, its application in design formalisms, and experience the creation and debugging of formal specifications with automated tools...

Therefore, the study of mathematical foundations and logic is an important aspect of a computer science degree, but it is not part of this lecture.

Functional programming offers a number of advantages over classical imperative programming.

- High level of abstraction, no manipulation of memory cells
- No side effects, therefore easier code optimization and better comprehensibility
- Programming via properties, not via the execution sequence
- Implicit memory management
- Simpler proof of correctness and verification
- Compact source programs, therefore shorter development time, more readable programs, better maintainability
- Modular program structure, polymorphism, higher-order functions, reusability of code

For practical programming, knowledge of functional programming is important, as functional programming techniques and language constructs lead to better structured programs, as explained in the article [15]. Therefore, functional concepts can also be found in many modern programming languages in a limited form. However, we will first introduce purely functional programming.

In functional programs, a *variable* represents an unknown value. A *program* is a set of function definitions. The memory is not explicitly usable, but is automatically managed and cleared. A *program flow* consists of the reduction of expressions. This goes back to the mathematical theory of the λ calculus from Church [5]. In the following we introduce the purely functional programming using the Haskell programming language [13].

3.1 Expressions and Functions

In mathematics, a variable represents unknown (arbitrary) values, so that we often use expressions like

$$x^2 - 4x + 4 = 0 \Leftrightarrow x = 2$$

In imperative languages, on the other hand, we often see expressions such as the following.

$$x = x + 1 \qquad \text{or} \qquad x := x + 1$$

This represents a contradiction to mathematics. In functional programming, as in mathematics, variables are interpreted as unknown values (and not as names for memory cells)!

While functions in mathematics are used for calculation, we use procedures or functions in programming languages for structuring. There, however, is no real connection due to side effects. In functional programming languages, however, there are no side effects, so every function call with the same arguments produces the same result.

Functions can be defined in Haskell as follows.

```
f x1 ... xn = e
```

`f` is the function name, `x1` to `xn` are formal parameters or variables, and `e` is the body, an expression over `x1` to `xn`.

Expressions are constructed in Haskell by combining elements from the (incomplete) list below. gebildet werden:

1. Numbers: `3`, `3.14159`
2. Basic operations: `3 + 4`, `5 * 7`
3. Function application: `(f e1 ... en)`. Parentheses can be omitted if the context allows.
4. Conditional expressions: `(if b then e1 else e2)`

Haskell almost looks like a scripting language. In contrast to script languages like PHP, Ruby or Python, Haskell has all elements to realize even large software systems. In particular, unlike scripting languages, Haskell is *strictly typed*, that is, all values and expressions have a type that is checked by the Haskell system *before* the program is executed. We will discuss the different data types in more detail in chapter 3.2. Here, we want to annotate the type for all functions to make clear what they are used for.[1]

A basic data type is `Int`, the set of integers (or a finite subset of it). The type of a function is annotated by "`::`", where several argument types and the result type are separated by "$\rightarrow$". Furthermore, the intuitive meaning of functions should be explained by a *comment* before the function definition, where comments are introduced by two minus signs and reach until the end of the line.

As an example, consider a function for calculating squares. We can define this in Haskell as follows.

```haskell
-- Computes the square of a number.
square :: Int -> Int
square x = x * x
```

If this definition is stored in the file `Square.hs`, then you can use the Haskell interpreter `ghci`, which belongs to the Glasgow Haskell Compiler (GHC), as follows to use this function.

```
> ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :l Square
[1 of 1] Compiling Main             ( Square.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 3
9
*Main> square (3 + 1)
16
*Main> :q
Leaving GHCi.
```

A function for calculating the minimum of two numbers can be defined in Haskell in this way.

```haskell
-- Computes the mininum of two numbers.
min :: Int -> Int -> Int
min x y = if x <= y then x else y
```

Next, we have look at the factorial function. It is defined mathematically as follows.

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

The Haskell implementation looks like this.

```haskell
-- Computes the factorial of a non-negative number.
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n - 1)
```

[1] Haskell can, unlike many other languages, infer function types, so that it is not necessary write the type down. Nevertheless, it is a better programming style to add function types for program documentation.

3.1.1 Evaluation

The evaluation of function definitions in Haskell is done by oriented calculation from left to right: First the current parameters are bound, that is, the formal parameters are replaced by the current ones. Then the left side is replaced by the right side.

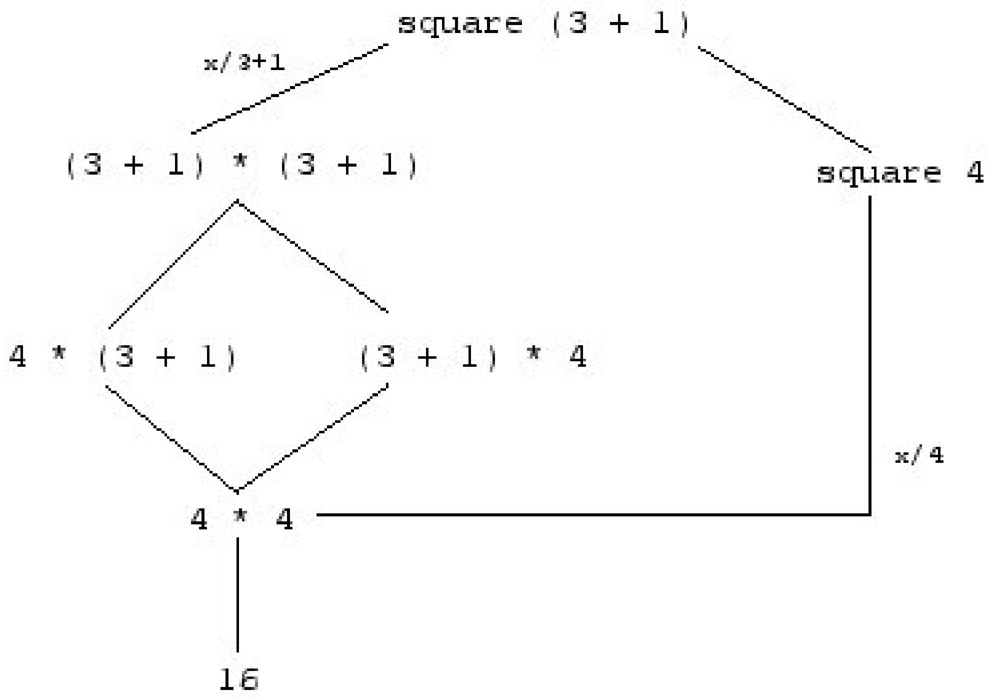

Figure 3.1: Ways to evaluate a function

Figure 3.1 shows how functions can be evaluated. The right branch shows how a function is evaluated in Java, the left branch resembles the evaluation in Haskell if double calculations of `3 + 1` are omitted.

Another example is the evaluation of a call to our function `fac`.

$$
\begin{aligned}
\texttt{fac 2} &= \texttt{if 2 == 0 then 1 else 2 * fac (2 - 1)} && (3.1)\\
&= \texttt{if False then 1 else 2 * fac (2 - 1)} && (3.2)\\
&= \texttt{2 * fac (2 - 1)} && (3.3)\\
&= \texttt{2 * fac 1} && (3.4)\\
&= \texttt{2 * (if 1 == 0 then 1 else 1 * fac (1 - 1))} && (3.5)\\
&= \texttt{2 * (if False then 1 else 1 * fac (1 - 1))} && (3.6)\\
&= \texttt{2 * 1 * fac (1 - 1)} && (3.7)\\
&= \texttt{2 * 1 * fac 0} && (3.8)\\
&= \texttt{2 * 1 * (if 0 == 0 then 1 else 0 * fac (0 - 1))} && (3.9)\\
&= \texttt{2 * 1 * (if True then 1 else 0 * fac (0 - 1))} && (3.10)\\
&= \texttt{2 * 1 * 1} && (3.11)\\
&= \texttt{2 * 1} && (3.12)\\
&= \texttt{2} && (3.13)
\end{aligned}
$$

We now want to develop an efficient function for calculating Fibonacci numbers. Our first version is directly motivated by the mathematical definition.

```
-- Computes the n-th Fibonacci number.
fib1 :: Int -> Int
fib1 n = if n == 0
         then 0
         else if n == 1
              then 1
              else fib1 (n - 1) + fib1 (n - 2)
```

However, this variant is extremely inefficient: its execution time is $O(2^n)$.

How can we improve our first version? We calculate the Fibonacci numbers from the bottom: The numbers are enumerated from 0 until the nth number is reached: 0 1 1 2 3 ... $fib(n)$.

This programming technique is known by the name of *accumulator technique*. To do this, we must always keep the two previous numbers as parameters.

```
-- An accumulator function to compute n-th Fibonacci number more efficiently.
fib2' :: Int -> Int -> Int -> Int
fib2' fibn fibnp1 n = if n == 0
                      then fibn
                      else fib2' fibnp1 (fibn + fibnp1) (n - 1)

-- Computes the n-th Fibonacci number.
fib2 :: Int -> Int
fib2 n = fib2' 0 1 n
```

Here, `fibn` is the nth Fibonacci number and `fibnp1` the $(n+1)$th. Thus we achieve a linear runtime.

From a software-technical point of view, our second variant is unattractive: We want to avoid other external calls to `fib2'`. How this works is described in the next section.

3.1.2 Local Definitions

Haskell offers several ways to define functions locally. One possibility is the keyword `where`:

```
fib2 :: Int -> Int
fib2 n = fib2' 0 1 n
  where fib2' fibn fibnp1 n =
          if n == 0
          then fibn
          else fib2' fibnp1 (fibn + fibnp1) (n - 1)
```

`where` definitions are visible in the previous equation, outside they are invisible.

Alternatively, the keyword `let` can be used.

```
fib2 n =
  let fib2' fibn fibnp1 n =
        if n == 0
        then fibn
        else fib2' fibnp1 (fibn + fibnp1) (n - 1)
  in fib2' 0 1 n
```

In contrast to `where`, `let` is an *expression*. The `fib2'` defined by `let` is only visible within the `let` expression.

`let` ... `in` ... can occur as an arbitrary expression.

```
(let x = 3
     y = 1
 in x + y) + 2
```

The above expression evaluates to 6.

The syntax of Haskell does not require parentheses and no separation of the individual definitions (for example by a semicolon) when defining such blocks. In Haskell, the *off-side rule* applies: The next symbol after `where` or `let` that is not a whitespace defines a *block*.

- If the next line starts to the right of the block, it belongs to the same definition.
- If the next line starts at the edge of the block, a new definition in the block starts here.
- However, if the next line starts to the left of the block, the block before it ends here.

Local definitions offer a number of advantages.

- Name conflicts can be prevented
- Wrong usage of helper functions can be prevented
- Better readability
- Redundant computations can be avoided

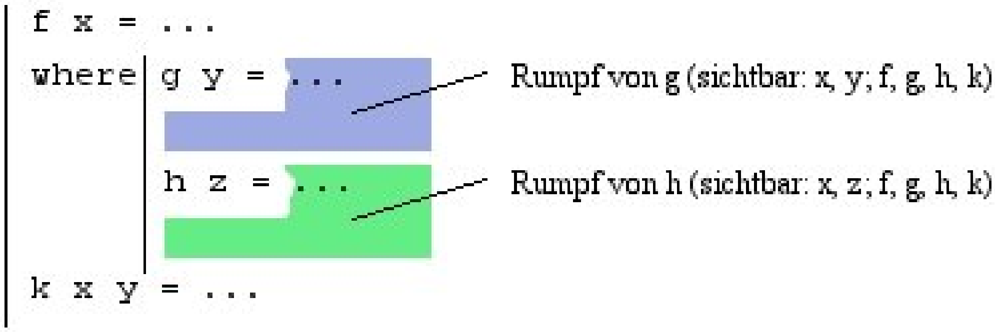

Figure 3.2: Off-side rule in Haskell

- Helper functions with less parameters

Let us look at an example to avoid multiple calculations. Instead of the confusing function definition

```
f x y = y * (1 - y) + (1 + x * y) * (1 - y) + x * y
```

we write instead

```
f x y = let a = 1 - y
            b = x * y
        in y * a + (1 + b) * a + b
```

By using the local declaration constructs `let` and `where` it is also possible to save parameters for helper functions. This is illustrated by the following example.

The predicate `isPrim` is supposed to check if the given number is a prime number. In Haskell we can express this as follows.

```
-- Checks whether a number is prime.
isPrim :: Int -> Bool
isPrim n = n /= 1 && checkDiv (div n 2)
  where checkDiv m =
          m == 1 || mod n m /= 0 && checkDiv (m - 1)
```

Here `/=` expresses inequality, `&&` stands for a conjunction, `||` for the logical or and `div` for integer division.

In the last line we do not need further parentheses, because `&&` binds stronger than `||`. The `n` is visible in `checkDiv`, because the latter is locally defined.

3.2 Data Types

3.2.1 Basic Data Types

Integers

We have already used a basic data type of Haskell before: integers. In fact, Haskell distinguishes between two types of integers:

`Int`: Values -2^{31} ... $2^{31}-1$
`Integer`: arbitrary size (limited by memory)

Arithmetic operators: `+ - * div mod`
Comparison operators: `< > <= >= == /=`

Booleans

Booleans are another basic data type.

`Bool`: `True False`

Operators: `&& || not == /=`

`==` means equivalence, `/=` exclusive or (XOR).

Floats

Floating point numbers are a basic data type as well.

`Float`: `0.3 -1.5e-2`

Operators: similar to `Int`, but `/` instead of `div`, no `mod`

Chars

Unicode characters are a basic data type, too.

`Char`: `'a' '\n' '\NUL' '\214'`

The operators are defined in the library `Data.Char` and can be imported into the program by adding "`import Data.Char`" to the beginning of the Haskell program.

```
chr :: Int -> Char
ord :: Char -> Int
```

The operator "`::`" is used for optional type annotations of expressions, as explained below.

3.2.2 Type Annotations

As already mentioned, Haskell is a strictly typed programming language, that is, all values and expressions in Haskell have a type which can also be annotated.

```
3 :: Int
```

In this example, `3` is a value or expression, and `Int` is a type expression. Other examples of type annotations are as follows.

```
3 :: Integer
(3 == 4) || True :: Bool
(3 == (4 :: Int)) || True :: Bool
```

We can also specify type annotations for functions. These are written in a separate line:

```
square :: Int -> Int
square x = x * x
```

But what is the type of `min` (see section **??**), considering that the function has two arguments?

```
min :: Int -> Int -> Int
```

A function arrow is therefore also used between the argument types written. We will see later why this is necessary.

3.2.3 Algebraic Data Structures

Own data structures can be defined as new data types. Values are built using *constructors*. Constructors are freely interpreted functions and therefore cannot be reduced.

Defining an Algebraic Data Type

Algebraic data types are defined in Haskell as follows.

$$
\texttt{data}\ \tau\ \texttt{=}\ C_1\ \tau_{11}\ \ldots\ \tau_{1n_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}
$$

where

- τ is the newly defined type,
- $C_1, ..., C_k$ are the constructors and
- $\tau_{i1}, ..., \tau_{in_i}$ are the argument types of the constructor C_i. Therefore

  $$C_i\ \texttt{::}\ \tau_{i1}\ \texttt{->}\ \cdots\ \texttt{->}\ \tau_{in_i}\ \texttt{->}\ \tau$$

  holds.

Note: Both type and constructor names must begin with a capital letter in Haskell!

Examples

1. Enumerated types (only nullary constructors):

   ```
   data Color = Red | Blue | Yellow
   ```

   This data type defines three values of the type `Color`. `Bool` is also an enumerated type, predefined as

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

2. Types with only one constructor:

   ```
   data Complex = Complex Float Float
   Complex 3.0 4.0 :: Complex
   ```

   This is allowed because Haskell works with separate namespaces for types and constructors.

   How do we select individual components? In Haskell we use *pattern matching* instead of explicit selection functions (this possibility will be explained later):

   ```
   -- Add two complex numbers:
   addC :: Complex -> Complex -> Complex
   addC (Complex r1 i1) (Complex r2 i2) = Complex (r1+r2) (i1+i2)
   ```

3. Lists (mixed types): For now, we consider only lists with elements of type `Int`:

   ```
   data List = Nil | Cons Int List
   ```

   `Nil` represents the empty list. The function `append` allows concatenating two lists.

   ```
   -- Concatenate two lists of integer elements:
   append :: List -> List -> List
   append Nil         ys = ys
   append (Cons x xs) ys = Cons x (append xs ys)
   ```

   This is a definition using several equations, where the first suitable one is selected. For example, a function call could be reduced in the following way.

   ```
     append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)
   = Cons 1 (append (Cons 2 Nil) (Cons 3 Nil))
   = Cons 1 (Cons 2 (append Nil (Cons 3 Nil)))
   = Cons 1 (Cons 2 (Cons 3 Nil))
   ```

   Haskell's predefined lists look like this.

   ```
   data [Int] = [] | Int:[Int]
   ```

   `[]` corresponds to `Nil`, "`:`" is equivalent to `Cons` and `[Int]` is `List`.

   The operator "`:`" is right-associative. Therefore, the following holds.

   ```
   1:(2:(3:[]))
   ```

   is equal to

   ```
   1:2:3:[]
   ```

   which can be written in the more compact way

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

   Haskell also offers the operator `++` instead of `append`, predefined in the `Prelude`.

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

Operators are binary functions that are used infix and begin with a special character. By means of parentheses, operators can be used like normal functions.

```
(++) ::  [Int] -> [Int] -> [Int]
```

`[1] ++ [2]` is equal to `(++) [1] [2]`.

The other way around, binary functions can be used infix with single inverted commas `‘...‘`.

```
div 4 2
```

can be written as

```
4 `div` 2
```

For user-defined data types, it is not automatically possible to compare or output them. For this the keyword `deriving` can be added after the data type definition.

```
data MyType = ... deriving (Eq, Show, Ord)
```

3.3 Polymorphism

For the definition of universally usable data structures and operations Haskell supports *type polymorphism*. To explain this in more detail, we will determine the length of a list as an example:

```
-- Compute the length of a list of integers:
length :: [Int] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Of course, this definition also works for other lists, for example of type `[Char]`, `[Bool]` or `[[Int]]`.

Generally, one could say

```
length :: ∀ Type τ. [τ] -> Int
```

which is expressed in Haskell by means of type variables.

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

What is the type of `(++)`?

```
(++) :: [a] -> [a] -> [a]
```

In consequence only lists with the same argument type can be concatenated. We consider another example.

```
-- Compute the last element of a list:
last :: [a] -> a
last [x]    = x
last (x:xs) = last xs
```

This works because `[a]` is a list type, and `[x]` is a one-element list (corresponds to `x:[]`). However, we must not swap the two rules, otherwise every call to the function will yield a run-time error!

How can we define polymorphic data types ourselves? Haskell provides *type constructors* for the construction of types.

$$
\begin{aligned}
&\texttt{data}\ K\ a_1 \ldots a_m\ \texttt{=} \\
&\quad C_1\ \tau_{11}\ \ldots\ \tau_{1n_1} \\
&\quad \texttt{|}\ \ldots \\
&\quad \texttt{|}\ C_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}
\end{aligned}
$$

These data type definitions therefore look similar to the previous ones but here

- K is a type constructor (not a type),
- $a_1, \ldots, a_m$ are type variables and
- τ_{ik} are type expressions like basic types, type variables or a type constructor applied to type expressions.

Functions and constructors are applied to values or expressions and generate values. Similarly, type constructors are applied to types and generate types.

As an example for a polymorphic datatype we want to model partial values in Haskell.

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

Then "`Maybe Int`" and also "`Maybe (Maybe Int)`" is a valid type.

If a type constructor in a type is applied to type variables, then the resulting type is also called *polymorphic*, as in the following example.

```haskell
isNothing :: Maybe a -> Bool
isNothing Nothing  = True
isNothing (Just _) = False
```

Another good example of a polymorphic datatype is the binary tree.

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

-- Compute the height of a binary tree:
height :: Tree a -> Int
height (Leaf _)     = 1
height (Node tl tr) = 1 + max (height tl) (height tr)
```

In Haskell, polymorphic lists are also predefined as syntactic sugar.

```haskell
data [a] = [] | a : [a]
```

Although this definition is not a syntactically valid Haskell program, it can be understood as follows: The square brackets around `[a]` are the type constructor for lists, `a` is the element type, `[]` is the empty list and `:` is the list constructor.

For this reason, the following expressions are all the same.

```haskell
(:) 1 ((:) 2 ((:) 3 []))
1 : (2 : (3 : []))
1 : 2 : 3 : []
[1,2,3]
```

According to the above definition, any complex list types such as `[Int]`, `[Maybe Bool]` or `[[Int]]` are possible.

Some functions on lists are for example `head`, `tail`, `last`, `concat` and `(!!)`.

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

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

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

concat :: [[a]] -> [a]
concat []     = []
concat (l:ls) = l ++ concat ls

(!!) :: [a] -> Int -> a
(x:xs) !! n = if n == 0 then x
                        else xs !! (n - 1)
```

The last function can also be defined as follows.

```
(x:_ ) !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

Strings are defined in Haskell as lists of characters.

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

Here, “`type`” initiates the definition of a *Typsynonyms*, that is, a new name (`String`) for another type expression (`[Char]`).

Thus the string `"Hello"` corresponds to the list `'H':'e':'l':'l':'o':[]`. For this reason, all list functions also work for strings.

```
length ("Hello" ++ " folks!")
```

The above expression is evaluated `12`.

Other predefined type constructors in Haskell are as follows.

- Sum of two types

  ```
  data Either a b = Left a | Right b
  ```

  This can be used, for example, to write values of "‘different types"’ to a list.

  ```
  [Left 42, Right "Hallo"] :: [Either Int String]
  ```

  Sum types can be processed via patterns, for example.

  ```
  valOrLen :: Either Int String -> Int
  valOrLen (Left  v) = v
  valOrLen (Right s) = length s
  ```

- Tuple

  ```
  data (,) a b = (,) a b
  ```

```
data (,,) a b c = (,,) a b c
```

Some functions have already been defined for this as well.

```
(3,True) :: (Int,Bool)

fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y

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

unzip :: [(a, b)] -> ([a], [b])
unzip []            = ([], [])
unzip ((x ,y):xys) = let (xs, ys) = unzip xys
                     in (x:xs, y:ys)
```

In principle `unzip` is the inversion of `zip`, that is, if "`unzip zs`" evaluates to the result `(xs,ys)`, then "`zip xs ys`" evaluates again to `zs`. However, conversely `unzip (zip xs ys)` does not always evaluate to `(xs,ys)`!

3.4 Pattern Matching

As we have already seen, functions can be defined by several equations using *pattern matching*.

$$
\begin{aligned}
&\texttt{f}\ pat_{11} \ldots pat_{1n} = e_1\\
&\vdots\\
&\texttt{f}\ pat_{k1} \ldots pat_{kn} = e_k
\end{aligned}
$$

This results in very concise programs, since you only have to define the right-hand sides for the special cases specified by the patterns. For multiple rules, Haskell selects and applies the textual first rule with a matching left side. Overlapping rules are allowed in principle, but they can lead to programs that are not very readable and should therefore be avoided.

3.4.1 Structure of Patterns

In principle, the patterns are data terms (that is, they contain no defined functions) with variables. The following patterns are possible.

- `x` *(Variable)*: matches always, the variable is bound to the current value.
- `_` (*Wildcard*): matches always, no binding.

- `C` $pat_1 ... pat_k$ where `C` is a k-ary Constructor: matches, if the same construktor and arguments match with $pat_1, ..., pat_k$
- `x@pat` (*as pattern*): matches if `pat` matches; Additionally, `x` is bound to the whole matched term.

  The as pattern also avoids overlapping patterns.

  ```
  last :: [a] -> a
  last [x]              = x
  last (x : xs@(_:_)) = last xs
  ```

In addition there are so-called `(n+`k`)` patterns for positive integers. We do not them explain here, because they are often not supported and not important.

Patterns can also be used with `let` and `where`:

```
unzip :: [(a, b)] -> ([a], [b])
unzip ((x,y) : xys) = (x:xs, y:ys)
 where
  (xs,ys) = unzip xys
```

3.4.2 Case Expressions

Sometimes it is also useful to branch expressions using pattern matching.

$$
\begin{array}{l}
\texttt{case } e \texttt{ of } pat_1 \texttt{ -> } e_1 \\
\qquad\qquad \vdots \\
\qquad\qquad pat_n \texttt{ -> } e_n
\end{array}
$$

The above defines an expression with type $e_1, ..., e_n$, which must all have the same type. $e, pat_1, ..., pat_n$ must also have the same type. After the keyword `of`, the off-side rule also applies, that is, the patterns $pat_1, ..., pat_n$ must all begin in the same column.

The result of `e` is matched against pat_1 to pat_n one after the other. If a pattern matches, the whole `case` expression is replaced by the corresponding e_i.

As an example, consider extracting the lines of a string.

```
-- Breaks a string into a list of lines where a line is terminated at a
-- newline character. The resulting lines do not contain newline characters.
lines :: String -> [String]
lines ""         = []
lines ('\n':cs) = "" : lines cs
lines (c:cs)     = case lines cs of
                      []      -> [[c]]
                      (l:ls) -> (c : l) : ls
```

3.4.3 Guards

Each pattern can have an additional boolean condition, also called *guard*.

```
fac :: Int -> Int
fac n | n == 0     = 1
      | otherwise = n * fac (n - 1)
```

`otherwise` is not a keyword, but a function that always evaluates to `True`.

By combining guards and case expressions, the first n elements of a list can be extracted.

```
-- Returns prefix of length n.
take :: Int -> [a] -> [a]
take n xs | n <= 0    = []
          | otherwise = case xs of
                             []     -> []
                             (x:xs) -> x : take (n-1) xs
```

Guards are also allowed for `let`, `where` and `case`.

3.5 Higher Order Functions

In Haskell, functions are not only used to define calculation methods, but functions are also "first class citizens" because they can be used like all other values (for example numbers). This means that functions can appear in data structures and also as parameters or results of other functions. In the latter case one speaks of "‘higher-order functions"’..

Higher-order functions can used for

- generic programming
- defining program schemes (control structures)

This enables us to achieve a better reusability and a higher modularity of the program code.

3.5.1 Example: Derivative Function

The derivative function is a function that returns a new function for a function. The numerical calculation looks like this.

$f'(x) = \lim\limits_{dx \to 0} \frac{f(x+dx)-f(x)}{dx}$

An implementation with a small dx could look like the following.

```
dx :: Float
dx = 0.0001

-- Computes the derivation of a (continuous) function.
derive :: (Float -> Float) -> (Float -> Float)
derive f = f'
  where f' :: Float -> Float
        f' x = (f (x + dx) - f x) / dx
```

Now, `(derive sin) 0.0` evaluates to `1.0` and `(derive square) 1.0` evaluates to `2.00033`.

3.5.2 Anonymous Functions (Lambda Abstraction)

Sometimes one does not want to give every defined function a name (like `square`), but to write it directly where is is needed, as shown in the example below.

```
derive (\x -> x * x)
```

The argument corresponds to the function $x \mapsto x^2$. Such a function without name is also called *lambda abstraction* or *anonymous function*. Here \ stands for λ, `x` is a parameter and `x * x` is an expression (the body of the function).

In general, anonymous functions in Haskell are defined as follows.

$$\backslash\ p_1\ \ldots\ p_n\ \texttt{->}\ e$$

where $p_1, \ldots, p_n$ are patterns and e is an expression.

We can also write the following.

```
derive f = \x -> (f (x + dx) - f x) / dx
```

When using this function, we can observe that it behaves approximately like the derived function $(y \mapsto 2y)$.

```
(derive (\x -> x * x)) 0.0 -> 0.0001
(derive (\x -> x * x)) 2.0 -> 4.0001
(derive (\x -> x * x)) 4.0 -> 8.0001
```

But `derive` is not the only function with a functional result. For example, the function `add` can be defined in three different ways.

```
add :: Int -> Int -> Int
add x y = x + y
```

or

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

or

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

So `add` can also be seen as a constant that returns a function as a result, or as a function that takes an `Int` and returns a function that takes another `Int` and only then returns an `Int`.

Therefore, the types `Int` → `Int` → `Int` and `Int` → `(Int` → `Int)` must be identical. In fact, the type constructor "→" is defined as right-associative, so that this binding always applies if no brackets are written. You should note that `(a` → `b)` → `c` *not* the same as `a` → `b` → `c` or `a` → `(b` → `c)`!

So it would make sense to write the following independently of the definition of `add`.

```
derive (add 2)
```

If a function is applied to "too few" arguments, this is called *partial application*application, partial. The partial application is made possible syntactically by currying. The name *currying* goes back to *Haskell B. Curry*, who discovered the following isomorphy in the 1940s:

$$[A \times B \to C] \;\simeq\; [A \to (B \to C)]$$

This means that a function with two arguments can also be interpreted as a function with one argument, which then returns another function for the second argument.

Actually, this technique was established much earlier by *Moses Schönfinkel* [16] in 1924. But because this article was published in German and "*Schönfinkeling*" cannot be pronounced so well in English, the term "Currying" has prevailed.

A number of functions can now be defined with the help of partial applications.

- `take 42 :: [a] -> [a]` yields the first up to 42 elements of a list.
- `(+) 1 :: Int -> Int` is the increment function.

For operators, so-called *sections* offer an additional, shortened notation.

- `(1+)` is the increment function.
- `(2-)` is equal to `\x -> 2-x`.
- `(/2)` is equal to `\x -> x/2`.
- `(-2)` is *not* equal to `\x -> x-2`, because the compiler cannot distinguish the minus sign from the unary minus operator.

Therefore, operators can also be partially applied to the second argument.

```
(/b) a = (a/) b = a / b
```

The order of arguments is a design decision because of partial application. The order can be modified with λ-abstractions and the function `flip` .

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

The function `flip` can be used, for example, with `take` in order to supply the list argument first.

```
(flip take) :: [a] -> Int -> [a]
(flip take) "Hello World!" :: Int -> [Char]
```

3.5.3 Generic Programming

We consider the following functions `incList` and `codeStr`.

```
incList :: [Int] -> [Int]
incList []     = []
incList (x:xs) = (x + 1) : incList xs

code :: Char -> Char
code c | c == 'Z'  = 'A'
       | c == 'z'  = 'a'
       | otherwise = chr (ord c + 1)
```

```
codeStr :: String -> String
codeStr ""     = ""
codeStr (c:cs) = code c : codeStr cs
```

The expression `codeStr "Informatik"` evaluates to `"Jogpsnbujl"`. We observe that the definitions of `incList` and `codeStr` are nearly identical. Only the function that is applied to the list elements differs.

A generalized version is the function `map`.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

`incList` and `codeStr` can be defined more elegantly by using `map`.

```
incList = map (+1)
codeStr = map code
```

We look at another two examples: A function that yields the sum of all elements in a list and a function that calculates the sum of a string's Unicode values.

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

checkSum :: String -> Int
checkSum ""     = 1
checkSum (c:cs) = ord c + checkSum cs
```

Is there a common pattern? Indeed, both functions can be defined more easily by means of the powerful function `foldr`.

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

sum = foldr (+) 0
checkSum = foldr (\c res -> ord c + res) 1
```

To understand `foldr`, the following perspective might help. The first argument of `foldr` of type `(a -> b -> b)` replaces the list constructor `(:)` in the list. The second argument of type `b` is the replacement for the empty list `[]`. Note that the type of the supplied function matches the type of `(:)`.

The following expressions are equivalent.

```
foldr f e [1,2,3]
= foldr f e ((:) 1 ((:) 2 ((:) 3 [])))
=           (f   1 (f   2 (f   3 e )))
```

The general approach when designing such functions is as follows: Identify a common pattern and implement it with functional parameters.

Another pattern that we know is the function `filter` that filter elements with certain properties from a list.

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

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

We can use `filter`, for example, to transform a list into a set, that is, removing all duplicates.:

```
nub :: [Int] -> [Int]
nub []      = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

By using `filter`, we can implement *Quicksort* to sort lists.

```
qsort :: [Int] -> [Int]
qsort []      = []
qsort (x:xs) =
  qsort (filter (<= x) xs) ++ [x] ++ qsort (filter (> x) xs)
```

`filter` can also be defined by using `foldr`.

```
filter p = foldr (\x ys -> if p x then x:ys else ys) []
```

The function `foldr` is a very generic skeleton, it is equivalent to a katamorphism in category theory.

Using `foldr` can have drawbacks sometimes. The expression

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

leads to a large calculation that is first built up on the stack and then evaluated in the end.

An improved version can be implemented by means of the accumulator technique.

```
sum xs = sum' xs 0
  where sum' :: [Int] -> Int -> Int
        sum' []     s = s
        sum' (x:xs) s = sum' xs (x + s)
```

The expression `sum [1,2,3]` is now being reduced to `((0 + 1) + 2) + 3`, which can be evaluated immediately.

An equivalent implementation can be achieved by using a different version of fold.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ e []      = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Hence, a call `foldl f e (`x_1 `:` x_2 `:` ... `:` x_n `: [])` is replaced with `f` ... `(f (f e` x_1`)` x_2`)` ... x_n.

Now `sum` can be defined even simpler.

```
sum = foldl (+) 0
```

3.5.4 Control Structures

Many control structures that we know from other programming languages can be modeled in Haskell. First, we consider the `while` loop.

```
x = 1;
while x < 100
do
x = 2*x
od
```

In general, a `while` loop consists of the following.

- a state before executing the loop (initial value)
- a condition
- a loop body for changing the state

In Haskell, the `while` loop looks as follows.

```
while :: (a -> Bool) -> (a -> a) -> a -> a
while p f x | p x       = while p f (f x)
            | otherwise = x
```

For example, the expression `while (<100) (2*) 1` evaluates to `128`.

Note that this is not a language extension! This control structure is nothing more than a function, a first class citizen.

3.5.5 Functions as Data

What are data structures? From an abstract point of view, data structures are objects with specific operations.

- Constructors (like `(:)` or `[]`)
- Selectors (like `head` or `tail`, and pattern matching)
- Test functions (like `null`, and pattern matching)
- Conjunctions (like `++`)

The important part is the functionality, that is, the interface, not the implementation. Therefore, a data structure corresponds to a set of functions.

As an example, we want to implement arrays with arbitrary elements in Haskell.

The constructs have the following type.

```
emptyArray :: Array a
putIndex :: Array a -> Int -> a -> Array a
```

Now we only need a single selector.

```
getIndex :: Array a -> Int -> a
```

Now we want to implement the interface. We can achieve this simply by not using other data structures, for example, lists or trees, but rather implement the array as a function. The implementation of this approach could look like the following.

```
type Array a = Int -> a

emptyArray i =
error ("Access to non-initialized component " ++ show i)
```

```
getIndex a i = a i

putIndex a i v = a'
where a' j | i == j    = v
           | otherwise = a j
```

The advantage of this implementation is its conceptual clarity because the implementation is the same as the specification. One drawback is the access time that grows with an increasing number of `putIndex` calls.

3.5.6 Useful Higher Order Functions

One useful higher-order function is the function composition operator `(.)`.

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

Two more interesting higher-order functions are `curry` and `uncurry`. They allow using functions that are defined on tuples with single elements and vice versa.

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

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

Finally we have look at the function `const` that takes two arguments and returns the first one.

```
const :: a -> b -> a
const x _ = x
```

3.5.7 Higher Order Functions in Imperative Languages

Modern imperative programming languages allow functions as parameters or return values, too. Although partial application – like Haskell offers – is often not supported and the integration is not as seamless, many algorithms can be defined in a more compact way using higher-order functions. Higher-order also allows additional ways of abstraction since functions can not only abstract values but also program behavior.

In the following, we have a look at the usage of functional parameters in Ruby and Java 8. Besides the concrete syntax, we also discuss the problems that arise from the interplay of functions as values and mutable variables, objects or memory cells.

Higher-order functions are part of Ruby's syntax. They are available in the form of "'blocks"'. A block is a sequence of commands that can be parameterized.

The simplest form of blocks are non-parameterized blocks like, for example, loop bodies. Such blocks can have parameters (noted between vertical lines), which makes them equivalent to anonymous functions. Methods and functions can, in addition to normal parameters, also have a block parameter. For some simple examples, we consider the methods `each` and `map` for arrays.

```
a = [1,2,3,4,5]

b = a.map do |x| x + 1 end

b.each do |x| puts x end
```

The program outputs the numbers from 2 to 6 on the screen by means of `puts`. [2].

The `map`-Method applies the supplied unary block to every element of the array. The original array is not modified but instead, an array of the same length is created. The `each` method also applies the supplied unary block to every element of the array but does not create a new one.

To understand how blocks are used, we want to define our own version of `map` that mutates the supplied array. In order to not get too deep into specific Ruby details at this time, we define the function not as a method but rather as an independent procedure `map!` [3] that uses the array as an additional parameter.

```
def map!(a)
  for i in 0..a.size-1 do
    a[i] = yield(a[i])
  end
end

a = [1,2,3,4,5]

map!(a) do |x| x + 1 end

a.each do |x| puts x end
```

A block that is supplied to `map!` is applied with the keyword `yield`; in the above example it is applied to the argument `(a[i])`. The procedure `map!` is then applied to the array `a` and given the increment function block as an additional parameter. [4]

Thus, blocks are equivalent to functional parameters. Multiple blocks as parameters or returning a block is problematic. Therefore, Ruby allows transforming a block into a value of the class `Proc` by using `lambda`. The result can be used like any other value, that is, as an ordinary parameter or as part of a data structure.

Besides, the class `Proc` offers the method `call` that applies the function, or rather – the block, to parameters. As an example we define the function `foldr` with an explicit functional parameter instead of a block.

```
def foldr(f,e,a)
  if a == [] then
    e
  else
```

[2]Blocks in Ruby can be noted alternatively by using curly brackets, for example `b = a.map {|x| x+1}`.

[3]The bang in the name of methods and functions is a Ruby custom that indicates that the method is mutating, that is, the supplied object is modified. Many methods exist in both versions.

[4]Ruby also offers a predefined method `map!`.

```
    f.call(a[0],foldr(f,e,a[1,a.size-1]))
  end
end
```

The sum of the elements of an array can then be calculated as follows.

```
puts foldr(lambda do |x,y| x+y end,0,[1,2,3,4,5,6,7,8,9,10])
```

Next, we want to define an array of functions. The `i`-th position of the array is supposed to be the function that adds the value `i` to its argument. Intuitively, this can be implement like this.

```
a = [0,1,2,3,4,5,6,7,8,9]

for i in 0 .. a.size-1 do    # iterate over array, begin with 0
  a[i] = lambda {|x| x+i }   # Write respective increment function to i-th
end                          # position

puts a[3].call(70)           # Apply function at index 3 to the value 70
```

When the program is executed, the result is unexpectedly not `73` but `79`. This is because variables correspond to memory cells in Ruby which are modified while the program executes. The program does not add the value that `i` has in the loop body but instead when the function is executed, which is 9 in the example above. Thus, function bodies in imperative languages should not contain mutable variables. This can be achieved by, for example, creating the array of functions by using `map!`.

```
a = [0,1,2,3,4,5,6,7,8,9]

a.map! { |i| lambda {|x| x + i }}

puts a[3].call(70)    # Hier erhalten wir nun 73.
```

Now the variable `i` is no longer a memory cell that can be modified over time. Instead, the variable is a block parameter that is created each time the block is applied, similar to scoping. The program now returns the expected value `73`.

Anonymous functions can also be defined in Java, starting with version 8. Using functional parameters by means of anonymous inner classes was possible before but the new lambda notation increases the code readability drastically and adds to the feeling of functional programming. As an example we define the class `Higher` that offers a non-mutating `map` function for lists.

```
import java.util.*;

class Higher {

  interface Fun<A,B> {  // define interface for objects of functional type
    B call (A arg);     // that requires a call method
  }

  static <A,B> List<B> map (Fun<A,B> f, List<A> xs) {
  List<B> ys = new ArrayList<B> (xs.size());
```

```
    for (A x : xs) {
      ys.add(f.call(x));    // the interface is used here
    }
    return ys;
    }
  }
```

When using the `map` method, we now can use the lambda notation to define anonymous implementations of the `Fun` interface, as shown below.

```
  public static void main (String[] args) {
    List<Integer> a = Arrays.asList(1,2,3,4,5);
    a = map(x -> x + 1, a);
    System.out.println(a);  // Result: [2,3,4,5,6]
  }
```

The lambda function `x -> x + 1`[5] is an elegant alternative to the definition of an anonymous inner class that implements the interface `Fun`.

We want to find out next, if the problem of using mutable variables within function bodies (as before in Ruby) is present in Java, too. Therefore, we again implement a loop that returns a list of increment functions.

```
  public static void main (String[] args) {
    List<Fun<Integer,Integer>> fs = new ArrayList<>(10);
    for (int i = 0; i<10; i++) {
      fs.add(x -> x + i);
    }
    System.out.println(fs.get(3).call(70));
  }
```

At compile time, the following error message appears.

```
  local variables referenced from a lambda expression must be final or
  effectively final
  fs.add(x -> x + i);
```

Thus Java recognizes that the variable `i` is modified in the program and therefore is cannot be used in the body of a lambda expression. The simple solution is creating a `final` copy of `i` for every loop iteration.

```
  public static void main (String[] args) {
    List<Fun<Integer,Integer>> fs = new ArrayList<Fun<Integer,Integer>>();
    for (int i = 0; i<10; i++) {
      final int j = i;
      fs.add(x -> x + j);
    }
  System.out.println(fs.get(3).call(70));
  }
```

Now the program compiles and returns `73` as expected.

[5] Multiple function arguments are separated by commas within parentheses, for example `(x,y) -> x + y`. Type annotations like `int x -> x + 1` are also possible.

Modern imperative programming languages often offer the possibility to use functional parameters and values, as demonstrated in the examples. The resulting code is often shorter and more comprehensible, especially when predefined functions like `map` and `fold` can be used for lists or arrays. The resulting way of programming often differs significantly from the familiar imperative way because control structures are less important and data plus function that manipulate data are moved to the foreground, that is, the core of the implemented algorithms.

We have also seen the major pitfall of using higher-order functions in imperative programming languages: mutable variables in function bodies. The problem can often be mitigated by not modifying variables in this context and programming in a more functional way. Especially in Java, using a `final` copy of a mutable variable is a solution, too.

3.6 Type Classes and Overloading

We consider the function `elem` that checks whether an element is contained in a list.

```
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

What are possible type of `elem`? Some examples are listed below.

```
Int  -> [Int]  -> Bool
Bool -> [Bool] -> Bool
Char -> String -> Bool
```

Unfortunately, "`a -> [a] -> Bool`" is not a valid type since an arbitrary type `a` is too general: `a` also includes functions, for which defining equality is difficult (there is no correct, general definition in Haskell). Thus, we need a way to restrict types to types for which value equality is defined. This can be written in Haskell as follows.

```
elem :: Eq a => a -> [a] -> Bool
```

"`Eq a`" is called a *type constraint*. By adding parentheses and commas, multiple type constraints can be used.

The class `Eq` is defined in Haskell like this.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

A *class* contains one or more functions that need to be defined for all instances of the class, that is, types. In case of `Eq`, the functions are `(==)` and `(/=)`.

Types can be defined as *instances* of a class by implementing the functions of the class.

```
data IntTree = Empty | Node IntTree Int IntTree

instance Eq IntTree where
  Empty         == Empty            = True
  Node tl n tr  == Node tl' n' tr'  = tl == tl' && n == n'
                                      && tr == tr'
  _             == _                = False
```

```
  t1 /= t2 = not (t1 == t2)
```

Now `(==)` and `(/=)` can be used for the type `IntTree` and, for example, the above function `elem` can also be used with lists of elements of the type `IntTree`.

Class instances can also be defined for polymorphic types. However, this might require type constraints when defining the instance, as shown in the next example.

```
data Tree a = Empty | Node (Tree a) a (Tree a)

instance Eq a => Eq (Tree a) where
  ...<as above>...
```

Note that infinitely many types become instances of the class `Eq` this way.

3.6.1 Predefined Functions of a Class

The definition of `(/=)` looks like the above version for almost all instances. Therefore, a default definition is often useful for class functions. The default definition can be overwritten (and needs to be in some cases).

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x1 == x2 = not (x1 /= x2)
  x1 /= x2 = not (x1 == x2)
```

3.6.2 Predefined Classes

For some types it is useful to define a total order on values of the type. This functionality is provided in Haskell by an extension of `Eq`, .

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

... -- predefined implementations
```

A minimal instance definition needs at least `compare` or `(<=)`.

Other predefined classes are `Num`, `Show` and `Read`.

- `Num` represents numbers for calculations. (`(+) :: Num a => a -> a -> a`)
- `Show` transforms values into strings. (`show :: Show a => a -> String`)
- `Read` constructs values from strings. (`read :: Read a => String -> a`)

Other predefined classes are presented in the lecture "'Funktionale Programmierung"' (functional programming).

Instanzes can be derived automatically (except for `Num`) for self-defined data types by appending the keyword `deriving` and a list of type classes to the data type definition.

deriving ($\kappa_1, \ldots, \kappa_n$)

Exercise: Check the types of all functions which were defined in the lecture for the most general type. This is possible by deleting the type signature for self-defined functions; Haskell will then derive the most general type. A few examples are the following.

```
(+)   :: Num a => a -> a -> a
nub   :: Eq a  => [a] -> [a]
qsort :: Ord a => [a] -> [a]
```

3.6.3 The Class `Read`

The class `Show` is used to output data as text. To read data, that is, retrieve a value from a string, the string needs to be *parsed*, which is a difficult task. Luckily, there exists a predefined class. Nevertheless, using the class requires some understanding about parsing strings.

We consider the following type definition that defines the type of functions that parse strings to values.

```
type ReadS a = String -> [(a,String)]
```

What is the intention behind the return type of `ReadS`? The first part of the tuple is the actual result of the parser while the second part is the remainder of the string that has not been parsed yet. For example, when we consider the string `"Node Empty 42 Empty"`, it becomes clear that after reading the initial string `Node`, a tree needs to follow. In this situation, we would like to receive the remaining unparsed string to use it later.

If the string can be parsed multiple ways, the alternatives are returned as elements of a list. If the supplied string cannot be parsed, the returned value is the empty list.

Using `ReadS` can look like the following.

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a] -- predefined
```

The two functions `reads` and `read` are defined as follows.

```
reads :: Read a => ReadS a
reads = readsPrec 0

read :: Read a => String -> a
read str = case reads str of
             [(x,"")] -> x
             _        -> error "no parse"
```

Thus, `reads` allows transforming a string into a value while checking whether the input is syntactically correct. On the other hand, `read` can be used when there is no doubt that the input is syntactically correct.

The evaluation of `reads` and `read` expressions looks like this.

```
reads "(3,'a')" :: [((Int,Char),String)]
= [((3,'a'),"")]

reads "(3,'a')" :: [((Int,Int),String)]
= []

read "(3,'a')" :: (Int,Char)
= (3,'a')

read "(3,'a')" :: (Int,Int)
= error: no parse

reads "3,'a')" :: [(Int,String)]
= [(3,",'a'")]
```

3.7 Lazy Evaluation

We consider the following Haskell program.

```
f x = 1
h = h
```

The expression `f h` can be evaluated in multiple ways.`f` can be evaluated first, which leads to the result `1`, or an attempt is made to evaluate `h` – which never terminates.

From this example we see that not every computation path terminates, but this depends on the evaluation strategy. We distinguish two excellent *reduction strategies*:

- *leftmost-innermost* (*LI*): applicative order (strict functions)
- *leftmost-outermost* (*LO*): normal order (non-strict functions)

One advantage of LO reduction is that it is complete. Anything that *can* somehow be computed *will* be computed. However, LO can be inefficient because computations can be doubled.

Can LO still have advantages in practice? Indeed, LO offers the following.

- Avoidance of superfluous (possibly infinite) calculations
- Computing with infinite data structures

For example, the following function `from` defines the ascending, infinite list of natural numbers, starting with the number `n`.

```
from :: Num a => a -> [a]
from n = n : from (n + 1)
```

As another example, we recall the function `take`.

```
take :: Int -> [a] -> [a]
take n _      | n <= 0 = []
take _ []              = []
take n (x:xs)          = x : take (n - 1) xs
```

`take 1 (from 1)` evaluates to `[1]` since LO works like this.

```
take 1 (from 1)
= take 1 (1:from 2)
= 1 : take 0 (from 2)
= 1 : []
```

The advantage lies in the separation of control (`take 1`) and data (`from 1`).

As a further example, we consider the prime number calculation using the sieve of Eratosthenes. The idea is as follows.

1. Consider the list of all numbers greater or equal to 2.
2. Remove all multiples of the first (prime) number.
3. The first element of the list is a prime number. Return to step 2. proceed the same with the remaining list.

This algorithm can be implement in Haskell as follows.

```
sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve (filter (\x -> x `mod` p > 0) xs)

primes :: [Int]
primes = sieve (from 2)
```

The argument of `sieve` is an input list which starts with a prime number and in which all multiples of smaller prime numbers are missing. The result is a list of all prime numbers!

Now the expression `take 10 primes` yields the first ten prime numbers.

```
[2,3,5,7,11,13,17,19,23,29]
```

By using `(!!)`, we can output the tenth prime number directly: `primes!!9` evaluates to `29`.

Infinite data structures can be used as an alternative to the accumulator technique. We consider the Fibonacci function as an example. To retrieve the n-th Fibonacci number, we create the list of all Fibonacci numbers and look up the n-th element.

```
fibgen :: Int -> Int -> [Int]
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)

fibs :: [Int]
fibs = fibgen 0 1

fib :: Int -> Int
fib n = fibs !! n
```

Now `fib 10` evaluates to `55`.

However, one disadvantage of the LO strategy remains: Calculations can be duplicated. We have another look at the simple function `double`.

```
double x = x + x
```

If we pass `(double 3)` as an argument to `double`, then the evaluation looks like this, according to the LI strategy.

```
double (double 3)
= double (3 + 3)
= double    6
= 6 + 6
= 12
```

According to the LO strategy, the result is instead as follows.

```
double (double 3)
= double 3 + double 3
= (3 + 3)  + double 3
=    6     + double 3
=    6     + (3 + 3)
=    6     +    6
=          12
```

Because of the obvious inefficiency, no real programming language uses the LO strategy.

One optimization of the strategy leads to *lazy evaluation*, where, instead of terms, graphs are being reduced. Variables of the program correspond to pointers to expressions and evaluating an expression updates the value of every variable that points to it. This is called *sharing*. Sharing can also be understood as normalization of the program, where for every subexpression a new variable is introduced. For the above example, this looks like the following.

```
double x = x + x

main = let y = 3
           z = double y
       in double z
```

The evaluation works as shown in figure 3.3. Black lines indicate a reduction step and blue lines are pointers to expressions.

This strategy was formalized by Launchbury in 1993 [10]. Lazy evaluation is optimal in respect to the length of the evaluation: There are no redundant computations as with LI and no duplicated expressions as with LO, although sometimes a lot of memory is needed.

Lazy evaluation is used in the programming language Haskell, while other languages like ML, Erlang, Scheme and Lisp use the LI strategy.

Another advantage of lazy evaluation is that functions can be composed nicely: If we assume the existance of a producer function, for example, of the type `gen ::` $\alpha \to \beta$, and a consumer function, for example, with the type `con ::` $\beta \to \gamma$. If we compose both functions `con . gen`, lazy evaluation does not create large intermediate data structures, but instead, only parts that are needed at the moment are stored in memory and freed as soon as they are no longer needed.

Haskell also allows *cyclic data structures* like the list

```
ones = 1 : ones
```

The list can be stored in a constant amount of memory, as shown in figure 3.4.

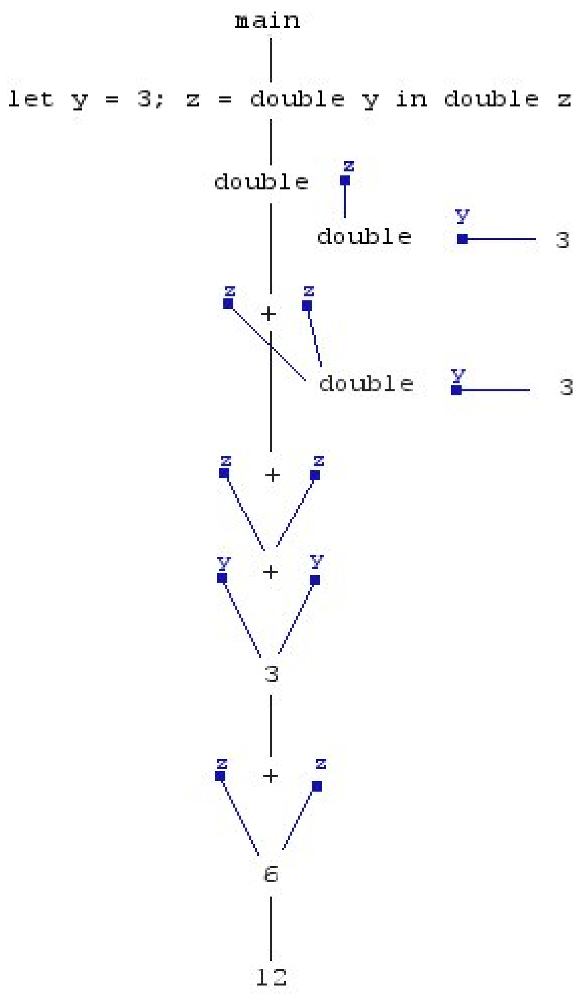

Figure 3.3: Sharing with lazy evaluation

3.8 Enumerated Lists

Because infinite structures can often be useful, Haskell has some predefined functions to create infinite lists. For example, `repeat` yields an infinite list of identical elements.

```
repeat :: a -> [a]
repeat x = x : repeat x
```

Thus, `take 70 (repeat '-')` yields a textual line.

An infinite list of repeated application of a function can be created with the function `iterate`.

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

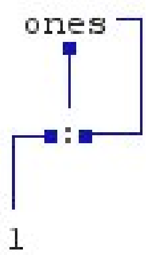

Figure 3.4: Cyclic list `ones`

As an exercise, one should think about what `iterate (+1) 0` evaluates to.

The example `from` has shown us that it is simple to create infinite arithmetic sequences in Haskell. We can also define *arithmetic sequences* and *intervals* with the step size `1` or arbitrary step size.

```
from :: Int -> [Int]
from n = n : from (n + 1)

fromThen :: Int -> Int -> [Int]
fromThen n1 n2 = let d = n2-n1 in n1 : fromThen (n1+d) (n2+d)

fromTo :: Int -> Int -> [Int]
fromTo n m = if n>m then [] else n : fromTo (n + 1) m

fromThenTo :: Int -> Int -> Int -> [Int]
fromThenTo n1 n2 m =
  let d = n2-n1
  in if d>=0 && n1>m || d<0 && n1<m
     then []
     else n1 : fromThenTo (n1+d) (n2+d) m
```

Since these functions[6] are rather useful, Haskell has a special syntax `[n..m]` for this functionality.

`[n1 .. ]`	means	`from n`
`[n1,n2 .. ]`	means	`fromThen n1 n2`
`[n .. m]`	means	`fromTo n m`
`[n1,n2 .. m]`	means	`fromThenTo n1 n2 m`

The following equalities hold.

```
[1..4]          == [1,2,3,4]
take 5 [2..]    == [2,3,4,5,6]
take 5 [2,4..]  == [2,4,6,8,10]
[1,3..10]       == [1,3,5,7,9]
take 5 [3,1..]  == [3,1,-1,-3,-5]
```

[6] In Haskell, these functions are defined with the prefix `enum`.

Not only integers but also floating point numbers and characters can be enumerated.

```
> [1,1.5 .. 10]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5,7.0,7.5,8.0,8.5,9.0,9.5,10.0]
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
> take 20 ['A' ..]
"ABCDEFGHIJKLMNOPQRST"
```

For this reason, Haskell's Prelude contains the type class `Enum`.

```
class Enum a where
  succ, pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a] -- [n..]
  enumFromThen :: a -> a -> [a] -- [n1,n2..]
  enumFromTo :: a -> a -> [a] -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n1,n2..m]
```

Therefore, the notation can be used for sequences of all instances of `Enum`. There are instances, for example, for `Int`, `Float`, `Char`, `Bool` and `Ordering`. Therefore, the following holds.

```
> [LT ..]
[LT,EQ,GT]
```

For data types with a finite number of members, the instance only yield finite sequences, too.

Arithmetic sequences are have many uses. For example, the factorial function can be defined as follows.

```
fac n = foldr (*) 1 [1 .. n]
```

It is also possible to enumerate a list of objects by pairing each element with an index.

```
> zip [1..] "abcde"
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
```

The combination with other familiar functions is possible too, of course.

```
> map (uncurry (++)) (zip (map show [1..]) (take 5 (repeat ". Zeile")))
["1. Zeile","2. Zeile","3. Zeile","4. Zeile","5. Zeile"]
```

3.9 List Comprehensions

As we already saw in chapter 3.8 on page 59, Haskell provides some syntactic sugar for list definitions with arithmetic sequences. Furthermore, we can even describe lists with a mathematical notation called *list comprehension*.

```
[(i,j) | i <- [1..3], j <- [2..4], i /= j]
```

The above expression evaluates to `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]`. Allowed are *generatores* like `i <- [1..3]` and *boolean conditions* like `i /= j`. `let` is also possible.

Thus, the following expression yields all finite initial sequences of natural numbers.

```
[[0..n] | n <- [0..]]
```

Finally, we consider another definition of the function `concat`, which turns a list of lists into a result list containing the lists' entries.

```
concat :: [[a]] -> [a]
concat xss = [y | ys <- xss, y <- ys]
```

A practical application where list comprehensions are very useful is parsing values.

As an example, we consider CSV files (comma separated values), which we can use to store a list of lists of values in one file.

```
data CSV a = CSV [[a]]
```

First we define a `Show` instance.

```
instance Show a => Show (CSV a) where
  show (CSV xss) = unlines (map separate xss)
    where
      separate []     = ""
      separate [x]    = show x
      separate (x:xs) = show x ++ "," ++ separate xs
```

The following is the instance for `Read`. Here we again use the idea of parsing introduced in the chapter on classes.

```
readsPrec :: Read a => Int -> String -> [(a, String)]
```

The function uses an additional parameter that can be used to note precedents in parentheses. Here we simply ignore it or forward it unchanged to the sub-parser.

```
instance Read a => Read (CSV a) where
  readsPrec p s = case s of
    []      -> [ (CSV []           , []) ]
    '\n':s1 -> [ (CSV ([]:xs)      , s2)
               | (CSV xs       , s2) <- readsPrec p s1 ]
    ',' :s1 -> [ (CSV ((x:xs):xss), s3)
               | (x            , s2) <- readsPrec p s1
               , (CSV (xs:xss), s3) <- readsPrec p s2 ]
    _       -> [ (CSV ((x:xs):xss), s2)
               | (x            , s1) <- readsPrec p s
               , (CSV (xs:xss), s2) <- readsPrec p s1 ]
```

Since we do not want to consider the precedent parameter `p`, we simply pass it on with recursive calls using `readsPrec p s`. Alternatively, we could have written a shorter `reads s`, which would result in the initial value being assigned to the precedent parameter.

3.10 Input and Output

Haskell is a purely functional language, that is, functions have no side effects. How can input and output be integrated into such a language?

A first idea: As in other languages (for example ML, Erlang or Scheme), with true side effects.

```
main = let str = getLine
       in putStr str
```

Here `getLine` should read a line from the keyboard and `putStr` is supposed to output a string on the standard output. What could be the type of `main` or `putStr`?

In Haskell, the smallest type is `()` (pronounced: unit), whose only value is `()::()` (corresponds to `void` in languages like Java). But if `main` had the result type `()`, then no input or output would be necessary to calculate the result `()` because of lazy evaluation...

Therefore, `putStr` must output the passed string as a side effect before the result `()` is returned.

We consider another example.

```
main = let x = putStr "Hi"
       in x; x
```

Here the semicolon "`;`" is supposed to cause `Hi` to appear twice in a row as output. The problem is, however, that due to lazy evaluation `x` is calculated only once, which means that the side effect, the output, is executed only once.

Another problem arises from the following, common scenario.

```
main = let dataBase = readDBFromUser
           request  = readRequestFromUser
       in print (lookup request dataBase)
```

Here the lazy evaluation prevents the desired input sequence. All these problems are solved in Haskell with monads.

3.10.1 IO monad

The input and output is done in Haskell *not* as a side effect. Instead, IO functions provide an *action* for input or output as a result, that is, as a value.

```
putStr "Hi"
```

For example, the above expression is an action that outputs `Hi` to the standard output *if* it is executed.

In principle, an action is a mapping of the following type.

```
World -> (a, World)
```

Here, `World` describes the present state of the whole "outer world". An action takes the current state of the world and returns a value (for example the read input) and a changed world state. Important is the fact that the world is not directly accessible. Therefore this type is abstractly called "`IO a`".

Output actions only change the world and return nothing. For this reason they have the type `IO ()`. There are, for example, the following predefined output actions.

```
putStr :: String -> IO ()
putChar :: Char -> IO ()
```

IO actions, like all other functions, are "first class citizens". They can be stored in data structures, for example. An interactive program thus conceptually defines a large IO action, which is applied to the initial world at program start and finally delivers a changed world.

```
main :: IO ()
```

Combining IO actions is achieved with the sequence operator.

```
(>>) :: IO () -> IO () -> IO ()
```

This yields the following equivalences.

```
main = putStr "Hi" >> putStr "Hi"

main = let pHi = putStr "Hi"
        in pHi >> pHi

main = let actions = repeat (putStr "Hi")
        in actions !! 0 >> actions !! 42
```

An infinite list of output actions is generated by `repeat (putStr "Hi")`.

IO actions can be combined with purely functional calculations as usual. For example, we can output the results of calculations.

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n - 1)

main = putStr (show (fac 42))
```

Or, in a more simple way, like this.

```
main = print (fac 42)
```

`print` is a function that first converts a passed value into a string and then writes it to the standard output.

```
print :: Show a => a -> IO ()
print x = putStr (show x) >> putChar '\n'
```

We consider another example. The following definitions of `putStr` are equivalent.

```
putStr :: String -> IO ()
putStr ""     = return ()
putStr (c:cs) = putChar c >> putStr cs

putStr = foldr (\c -> (putChar c >>)) (return ())
```

`return ()` is here, so to speak, the "empty IO action" which does nothing and only returns its argument.

```
return :: a -> IO a
```

There are corresponding actions for the input of data, where the type of the return value corresponds to the type of the input data.

```
getChar :: IO Char
getLine :: IO String
```

How can the result of an IO action be further used in a subsequent IO action? For we can use the "bind operator".

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

The bind operator is used as shown in the following example.

```
getChar >>= putChar
```

Here `getChar` has the type `IO Char`, `putChar` has the type `Char -> IO ()`. So `getChar >>= putChar` has the type `IO ()`. It reads a character and outputs it again.

Next we want to read a whole line.

```
getLine :: IO String
getLine =
  getChar >>= \c -> if c == '\n'
                    then return ""
                    else getLine >>= \cs -> return (c:cs)
```

On a closer look, the string

```
... >>= ...
```

resembles an assignment in an imperative language, except that the left and right sides are swapped. For this reason, a special notation has been introduced for better readability.

3.10.2 do Notation

With `do {` $a_1; \ldots; a_n$ `}` or

```
do a1
   ⋮
   an
```

(note the layout rule after the `do`!) there is an alternative, "imperative" notation.

The expression

```
do p <- e1
  e2
```

is equivalent to the bind expression

```
e1 >>= \p -> e2
```

and equivalent to the instruction sequence

```
do e1
   e2
```

for the sequence operator

```
e1 >> e2
```

This allows us to define the `getLine` operation as follows.

```
getLine = do c <- getChar
             if c == '\n'
               then return ""
               else do cs <- getLine
                       return (c:cs)
```

3.10.3 Printing Intermediate Results

We want to write a function that calculates the faculty, outputting all intermediate results of the calculation. This is possible if we reformulate the function to an IO action.

```
fac :: Int -> IO Int
fac n | n==0      = return 1
      | otherwise = do f <- fac (n-1)
                       print (n-1,f)
                       return (n * f)

main :: IO ()
main = do
  putStr "n: "
  str <- getLine
  facn <- fac (read str)
  putStrLn ("Factorial: " ++ show facn)
```

Using the function looks like the following.

```
> main
n: 6
(0,1)
(1,1)
(2,2)
(3,6)
(4,24)
(5,120)
Factorial: 720
```

But such programs should be avoided! It is always better to separate input and output on one side from calculations on the other. A common, established scheme is the following.

```
main = do input <- getInput
          let res = computation input
          print res
```

Here, the line

```
          let res = computation input
```

is a purely functional computation. The `let` does not need an `in` in a do block.

3.10.4 Reading and Writing Files

The Haskell libraries, based on the monadic IO concept, offer many possibilities to communicate with the environment, for example, reading and writing files, databases or socket connections for network applications. A very easy to use method for reading and writing files is predefined with the following IO actions.

```
readFile  :: String -> IO String        -- reads a file with a given name
writeFile :: String -> String -> IO () -- writes a file with given contents
```

For example, we could copy a file like this.

```
readFile "oldfile" >>= writeFile "newfile"
```

Because these operations are performed lazily, even large files can be copied without significant memory consumption.

With the functions already known to us, we can analyze files in a simple way. The size of a file can be computed as follows.

```
> readFile "FILE" >>= print . length
```

We can calculate the number of lines in a file in this way.

```
> readFile "FILE" >>= print . length . lines
```

Here we can see how the function composition and the associated combination style can be applied well. Another example is counting the number of blank lines in a file.

```
> readFile "FILE" >>= print . length . filter (all (==' ')) . lines
```

Here, the predefined function `all` checks whether all elements of a list meet a given predicate (one has to understand why this definition expresses this exactly).

```
all :: (a -> Bool) -> [a] -> Bool
all p = foldr (&&) True . map p
```

Finally, we want to see how easy it is to output the lines of a file numbered by using higher order functions and infinite data structures. For this we define a function that numbers a text line by line.

```
enumerateLines :: String -> String
enumerateLines = concat . map (++"\n")
                        . map (uncurry (++))
                        . zip (map (\n->show n ++ ": ") [1..])
                        . lines
```

If the file `FILE` contains the content

```
Dies ist
eine Datei
mit drei Zeilen.
```

it is numbered as follows:

```
> readFile "FILE" >>= putStrLn . enumerateLines
1: Dies ist
2: eine Datei
3: mit drei Zeilen.
```

By using some predefined functions, we can still improve the code. For example, since the combination of `concat` and `map` is often used, this combination is predefined as follows.

```
-- Maps a function from elements to lists and merges the result into one list.
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

This allows us to simplify the definition of `enumerateLines` as follows.

```
enumerateLines :: String -> String
enumerateLines = concatMap (++"\n")
                        . map (uncurry (++))
```

```
                             . zip (map (\n->show n ++ ": ") [1..])
                             . lines
```

If we take a closer look at the first function, we see that it merges a list of strings into one string, line by line, by adding line breaks between the individual strings. Since this is often used, this combination is predefined.

```
-- Concatenates a list of strings with terminating newlines.
unlines    :: [String] -> String
unlines ls = concatMap (++"\n") ls
```

This allows us to simplify the definition of `enumerateLines` once again.

```
enumerateLines :: String -> String
enumerateLines = unlines . map (uncurry (++))
                         . zip (map (\n->show n ++ ": ") [1..])
                         . lines
```

3.11 Modules

Like almost every other programming language, Haskell supports large-scale programming by dividing a program into several modules. Since modules in Haskell are organized similarly to other languages, we will only give a brief overview below.

A *module* defines a set of names (of functions or data constructors) that can be used when importing this module. You can limit the set of names by *export declarations* so that, for example, the names of only locally relevant functions are not exported, that is, are not visible on the outside.

The source code of a module therefore begins as follows.

```
module MyProgram (f, g, h) where
```

Here `MyProgram` is the *module name*, which must be identical to the filename (exception: hierarchical module names, which we will not discuss further here). Thus this module is stored in the file `MyProgram.hs`. The list of exported names follows in brackets, in this case the names `f`, `g` and `h` are exported. Normally the declarations after a `where` have to be indented further, but in the case of a module they can also start in the first column, so that we can write programs as usual. So our whole module could look like this.

```
module MyProgram (f, g, h) where

f = 42

g = f*f

h = f+g
```

Note that module names must always begin with a capital letter!

It is possible to deviate from this general scheme as follows.

- The export list (i.e. “`(f, g, h)`”) can also be missing; in this case *all* names defined in the module will be exported.

- Type constructors can also be listed in the export list. In this case, the type constructor name is also exported, but not the corresponding data constructors. If these are also to be exported, then one must write "`(..)`" after type constructor names in the export list, as shown in the following example.

```
module Nats(Nat(..),add,mult) where

data Nat = Z | S Nat
 deriving Show

add Z     y = y
add (S x) y = S (add x y)

mult Z     _ = Z
mult (S x) y = add y (mult x y)
```

- If no module header is specified in the source file, the system implicitly uses the module header

```
module Main(main) where
```

We have used an interactive Haskell system like `ghci` to test small programs before.It is also possible to create an executable file from a Haskell program using the `ghc`. For this, the program (or the main part of the program) must be stored in the module `Main`, which in turn must contain a main function, that is, an IO action that is executed as the main program. [7]

```
main :: IO ()
```

Then one can generate an executable machine program `myexec` with the following command, for example.

```
ghc -o myexec Main.hs
```

Typically, the main program uses a number of other modules that can be imported using an `import` declaration. For example, our main program could look like this.

```
module Main where

import Nats

main = print (add (S Z) (S Z))
```

If we execute the command

```
ghc -o myexec Main.hs
```

we get an executable file `myexec` which we can then execute directly.

```
> ./myexec
S (S Z)
```

A `import` declaration makes those names visible in the current module which are exported from the imported module.

[7] This is how an IO action is associated with a world state!

There are a number of variants:

- The imported names can be limited by an enumeration. For example, the following import does not include the name `mult`.

```
import Nats (Nat(..),add)
```

- One can also import all names except for a few exceptions by means of a `hiding`-constraint.

```
import Nats hiding (mult)
```

- The standard module `Prelude` is always implicitly imported if it is not explicitly specified, for example in the following program the functions `map` and `foldr` are not imported.

```
import Prelude hiding (map,foldr)
```

- To avoid ambiguities, you can also access imported names within a module in a qualified manner, for example like this.

```
module Main where

import Nats

main = Prelude.print (Nats.add (S Z) (S Z))
```

- If one wants to force that imported names must always be accessed in a qualified way, this can indicated by the restriction `qualified`.

```
module Main where

import qualified Nats

main = print (Nats.add (Nats.S Nats.Z) (Nats.S Nats.Z))
```

- If the names are too long, the module can be renamed during import.

```
module Main where

import qualified Nats as N

main = print (N.add (N.S N.Z) (N.S N.Z))
```

In addition, there are further possibilities and rules for the use of modules, which cannot all be described here. For this you should consult Haskell's language definition.

3.12 Data Abstraction and Abstract Data Types

As we have seen from the example of integers, there are different ways to implement a data type. In fact, a central idea of programming with data is

> **data abstraction**: It is not important how the data is represented internally, it is only important how the data is used, that is, which operations are available for this purpose.

For example, in higher programming languages it is not interesting in which bit order integer values are represented internally. What is important is that the addition and multiplication works "as usual". The same applies to other data. For example, in chapter 3.5.5 we have seen that we can implement fields differently than we know from imperative programming languages. Nevertheless, the field operations provide the correct results. Since data abstraction is an important programming technique (in all programming languages!), we will explain this in more detail below.

Abstraction from the internal structure or representation of the data allows using it without knowing the implementation. Instead, one only needs to know the interface, sometimes called API (Application Programming Interface), that is, a set of operations that can be used to work with the data. To give the interface more structure, the interface operations are classified as follows.

Constructors: Operations for constructing data

Selectors: Operations for extracting partial information from data

Operators: Operations to combine data

As a simple example, we consider rational numbers, consisting of a numerator and a denominator. A concrete data type can be immediately defined for this.

```
data Rat = Rat Int Int
  deriving Eq
```

For a "natural" output, we define our own show instance.

```
instance Show Rat where
  show (Rat n d) = show n ++ "/" ++ show d
```

A *constructor* for rational numbers is `Rat`. But if we want to hide the actual implementation so that we can change it later, then it is better to use a self-defined constructor operation instead of this constructor:

```
rat :: Int -> Int -> Rat
rat n d = Rat n d
```

In addition, we also need *selector* operations if we want to access the components of a rational number:

```
numerator :: Rat -> Int
numerator (Rat n _) = n

denominator :: Rat -> Int
denominator (Rat _ n) = n
```

A *operator* on rational numbers is, for example, an arithmetic combination like addition or multiplication. These are quite easy to define with the school knowledge on fractions.

```
addR :: Rat -> Rat -> Rat
addR (Rat n1 d1) (Rat n2 d2) = rat (n1*d2 + n2*d1) (d1*d2)

mulR :: Rat -> Rat -> Rat
mulR (Rat n1 d1) (Rat n2 d2) = rat (n1*n2) (d1*d2)
```

To prevent the user from looking into the details of the implementation and defining functions that depend on it, we hide them by exporting only the type name `Rat`. Our module `Rat` looks like this.

```
module Rat(Rat,rat,numerator,denominator,addR,mulR) where

data Rat = Rat Int Int
  deriving Eq

instance Show Rat where
  show (Rat n d) = show n ++ "/" ++ show d

rat :: Int -> Int -> Rat
rat n d = Rat n d

numerator :: Rat -> Int
numerator (Rat n _) = n

denominator :: Rat -> Int
denominator (Rat _ n) = n

addR :: Rat -> Rat -> Rat
addR (Rat n1 d1) (Rat n2 d2) = rat (n1*d2 + n2*d1) (d1*d2)

mulR :: Rat -> Rat -> Rat
mulR (Rat n1 d1) (Rat n2 d2) = rat (n1*n2) (d1*d2)
```

We can now use rational numbers. For example, we can define fractions and link them.

```
oneThird :: Rat
oneThird =  rat 1 3

twoThird :: Rat
twoThird = addR oneThird oneThird
```

If we output the value of `twoThird`, we get

```
ghci> twoThird
6/9
```

This is not wrong, but we would have expected the output `2/3`. Intuitively, rational numbers should always be reduced. We achieve this by a simple change. When constructing rational numbers, numerators and denominators are always reduced with the largest common divisor. Therefore, we simply change the definition of the constructor.

```
rat :: Int -> Int -> Rat
rat n d = let g = gcd n d in Rat (div n g) (div d g)

ghci> twoThird
2/3
```

However, one disadvantage remains: negative denominators are not uniquely represented. For example, the expression

```
ghci> rat (-3) (-2)
-3/-2
```

yields -3/-2 although this is the same value as 3/2. In order to make the representation unique and thus as simple as possible, we avoid the construction of rational numbers with negative denominators with the following redefinition.

```
rat :: Int -> Int -> Rat
rat n d = let g  = gcd n d in posDenom (div n g) (div d g)
 where
  posDenom n d = if d<0 then Rat (0 - n) (abs d)
                      else Rat n d

ghci> rat (-3) (-2)
3/2
```

Here we already see an important advantage of data abstraction. We have only improved one operation and can continue to use the remaining operations with the same interface.

However, if the implementation of a data type is hidden, then how do we know how the operations will behave or what kind of results they produce? This is where another principle of data abstraction comes into play: The behavior of the operations is determined by certain laws which every implementation must meet. This means that we can use different implementations, but still rely on the fact that they work in a certain way. In principle, these laws could be any logical formula, but as a rule, we limit ourselves to equations like "`addR x y == addR y x`". Such a description of a data type is also called *abstract data type*.

Definition 3.1 (Abstract data type) *An* abstract data type *(*ADT*) is a tuple* (Σ, X, E) *consisting of the following components.*

- *A program signature* Σ *(contains operations on the data type),*
- *a set* X *of variables that is disjoint with* Σ*, and*
- *a set* E *of equations of the form* $t = t'$*, where* $t, t' \in T_\Sigma(X)_s$ *are terms of the same signature* s*.*

Let us look at the rational numbers again as an example. The ADT for rational numbers is[8] (Σ, X, E) with

- $\Sigma = (S, F)$ with
  $S = \{\texttt{Rat}, \texttt{Int}\}$
  $F = \{\texttt{rat} :: \texttt{Int}\ \texttt{Int} \to \texttt{Rat}, \texttt{numerator} :: \texttt{Rat} \to \texttt{Int}, \texttt{denominator} :: \texttt{Rat} \to \texttt{Int}\}$
- $X = \{n :: \texttt{Int}, d :: \texttt{Int}\}$
- $E = \left\{ \dfrac{\texttt{numerator}\ \ (\texttt{rat}\ n\ d)}{\texttt{denominator}\ (\texttt{rat}\ n\ d)} = \dfrac{n}{d} \right\}$

Typically, an ADT has an explicit signature (here: `Rat`) that describes the elements of the data type (and other signatures used in the ADT operations). Frequently, the use of this ADT signature also reveals the category to which the ADT operations belong.

[8]In fact, this ADT also includes the ADT for integers, since we use it. But in the following we leave out ADTs which are already known and which we only use. But formally you would have to import them.

- *Constructors* Constructors have the ADT signature as the result signature. (here: `Rat`).
- *Selectors* Constructors have the ADT signature as the argument signature. (here: `numerator` and `denominator`).
- *Operators* Constructors have the ADT signature as the result and argument signature (for example "$\texttt{addR} :: \texttt{Rat}, \texttt{Rat} \to \texttt{Rat}$", which is not specified further here.)

It is often the case that selectors and constructors are in a unique relationship, that is, the selector returns exactly the data that was constructed with a constructor. In our case, however, the equations

$$\texttt{numerator}\ (\texttt{rat}\ n\ d) = n$$

and

$$\texttt{denominator}\ (\texttt{rat}\ n\ d) = d$$

would not be correct since we represent fractions in a reduced form. The equation in the ADT therefore specifies that it is irrelevant how rational numbers are represented. What is important is that the ratio of numerator and denominator is always the same.

An *implementation of an ADT* is a program that implements all functions in the program signature so that the equations for all values are always satisfied instead of the variables.

Many important data structures can be specified as abstract data types. For example, we have the following ADT components for list structures:

- Constructors: `[]` und `(:)`
- Selectors: `head` und `tail`
- Operators: `(++)`
- Equations:

$$\begin{aligned} \texttt{head}\ (x : xs) &= x \\ \texttt{tail}\ (x : xs) &= xs \end{aligned}$$

  (and other equations for `(++)`)

Case Study: Set Implementations

To show how data abstraction helps exchange the implementation of data types without changing anything for the user, consider sets as an example. Sets are collections of objects where there is no order and no duplicate elements. For sets we can define many operations. Here we only look at the following functions.

- `empty`: empty set
- `insert x s`: insert element `x` into set `s`
- `isElem x s`: is `x` an element of the set `s`?
- `union s1 s2`: union of the sets `s1` und `s2`

Regardless of a concrete implementation, the following laws should always hold.

$$
\begin{aligned}
\texttt{isElem}\ x\ \texttt{empty} &= \texttt{False}\\
\texttt{isElem}\ x\ (\texttt{insert}\ x\ s) &= \texttt{True}\\
\texttt{isElem}\ x\ (\texttt{insert}\ y\ (\texttt{insert}\ x\ s)) &= \texttt{True}\\
\texttt{isElem}\ x\ (\texttt{union}\ s_1\ s_2) &= \texttt{isElem}\ x\ s_1\ \texttt{||}\ \texttt{isElem}\ x\ s_2
\end{aligned}
$$

A first simple and clear reference implementation of sets is obtained by the representation of sets as characteristic functions, as already done in the exercises.

```
module Sets(Set, empty, insert, isElem, union) where

data Set a = Set (a -> Bool)

empty :: Set a
empty = Set (const False)

insert :: Eq a => a -> Set a -> Set a
insert x (Set s) = Set (\y -> x == y || s y)

isElem :: Eq a => a -> Set a -> Bool
isElem x (Set s) = s x

union :: Eq a => Set a -> Set a -> Set a
union (Set s1) (Set s2) = Set (\x -> s1 x || s2 x)
```

We get an alternative implementation of sets if we represent sets, for example, as lists, in which no duplicates occur. With this representation, the implementation is also quite simple.

```
module SetsByLists(Set, empty, insert, isElem, union) where

import Test.QuickCheck

data Set a = Set [a]
 deriving Show

empty :: Set a
empty = Set []

insert :: Eq a => a -> Set a -> Set a
insert x (Set s) = Set (if x `elem` s then s else x:s)

isElem :: Eq a => a -> Set a -> Bool
isElem x (Set xs) = x `elem` xs

union :: Eq a => Set a -> Set a -> Set a
union (Set [])     s2 = s2
union (Set (x:xs)) s2 = insert x (union (Set xs) s2)
```

Note that this implementation has exactly the same interface as the reference implementation and the ADT. This means that we can replace the set implementation in a program by simply replacing `import Sets` with `import SetsByLists`. Here we see an essential advantage of data abstraction: We can exchange the implementation without changing anything in the application programs (except for the name of the imported module).

A slightly more efficient implementation of sets can be achieved by representing **sets as ordered lists** by making sure that elements are inserted at the correct position in the list.

```
insert :: (Eq a, Ord a) => a -> Set a -> Set a
insert x (Set s) = Set (oinsert s)
 where
  oinsert []     = [x]
  oinsert (y:ys) | x==y      = y:ys
                 | x<y       = x:y:ys
                 | otherwise = y : oinsert ys
```

This has the advantage that we only have to compare on average half of the elements to check whether an element is present in a set.

```
isElem :: (Eq a, Ord a) => a -> Set a -> Bool
isElem x (Set xs) = oelem xs
 where
  oelem []     = False
  oelem (y:ys) | x==y      = True
               | x<y       = False
               | otherwise = oelem ys
```

However, this is not yet an improvement in complexity, that is, the runtime for finding an element is still linear in the number of elements. However, what we can significantly improve is the union operation. While the previous set union was quadratic in the number of elements of both lists (because of the repeated call of `insert` and thus `elem`), we can reduce this to a linear runtime for ordered lists by going through both lists at the same time.

```
union :: (Eq a, Ord a) => Set a -> Set a -> Set a
union (Set s1) (Set s2) = Set (ounion s1 s2)
 where
  ounion []        ys     = ys
  ounion xs@(_:_)  []     = xs
  ounion (x:xs)    (y:ys) | x==y = x : ounion xs ys
                          | x<y  = x : ounion xs (y:ys)
                          | x>y  = y : ounion (x:xs) ys
```

Unfortunately, the (sometimes most important) operations `insert` and `isElem` are still linear in the number of set elements. We could reduce this to a logarithmic complexity by using balanced search trees. This makes the implementation more complex, but we can test it again with the existing ADT properties, because the interface of the implementation remains the same.

Bibliography

[1] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[2] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[3] T. Ball and B. Zorn. Teach foundational language principles. *Communications of the ACM*, 58(5):30–31, 2015.

[4] L. Byrd. Understanding the control flow of Prolog programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

[7] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[8] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *Proc. IFIP '83*, pages 909–914. North-Holland, 1983.

[9] M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.

[10] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[11] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[12] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

[14] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[15] N. Savage. Using functions for easier programming. *Communications of the ACM*, 61(5):29–30, 2018.

[16] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

List of Figures