Elements of Functional Computing

Dan R. Ghica

2015

Contents

1 Introduction

1.1 Basic concepts

Computer Science (CS) studies computation and information, both from a theoretical point of view and for applications in constructing computer systems. It is perhaps not very helpful to try and define these two basic, and deeply connected, notions of "computation" and "information", but it is perhaps helpful to talk about some properties they enjoy.

Information is what is said to be exchanged in the course of communication. Information is said to be used to represent data, knowledge or ideas. A wonderful example of information is the DNA: it is a pattern of molecules that defines the formation and development of an organism without any need for a conscious or insightful mind. This lack of insight is essential for the understanding of *computation*, which is the deliberate, automated, mechanical processing or transformation of information. So the questions that CS answers are commonly about *what* can be computed and *how* it can be computed. The answers to these questions can be both qualitative, indicating that something can/cannot be computed, and quantitative, indicating how much of a certain resource (time, space, energy) must be consumed in order to perform a computation. The engineering and design side of CS is concerned with how to improve quantitative measures of computation.

Computation is carried out by a mindless, uninsightful *program* executed by a *machine*. The machines can be either physical (case in which they are called *computers*) or virtual (a machine which is in some logical sense running on top of another machine) or abstract (a theoretical contraption which embodies certain ideals of what can be computed). Programs are always written in a *language* which needs to be well-defined in at least two ways:

Syntax. Just like English, or any other *natural* language, it is important to distinguish between well-formed phrases and gibberish. Note that to make this distinction it is not important to understand the meaning, just the rules of what a well-formed phrase looks like. Consider the opening stanza of the poem *Jabberwocky* by Lewis Caroll:

> 'Twas brillig, and the slithy toves
> Did gyre and gimble in the wabe;
> All mimsy were the borogoves,
> And the mome raths outgrabe.

It has the syntactic form of English despite using nonsensical made-up words.

Programming languages, unlike natural languages, have much stricter and clearly defined syntax. A good programming language has no ambiguity as to whether a phrase is legal or not, and if it is legal as to what its syntactic structure is. In a program a single-character typo can usually render a program illegal.

Semantics. If syntax is concerned with what a meaningful program should look like, in semantics the concern is what the program actually means. Just like the syntax of a programming language must have a complete, unambiguous specification so its semantics must define completely and unambiguously what each program means.

Defining, or even just understanding, the syntax and semantics of languages is an advanced topic with which we will not deal in this module. Instead we will focus on the activity of *writing programs*, i.e. *programming.*

1.2 Programming the machine, a historical outlook

Programming is actually a very old activity. The oldest recognisable device used to perform calculations, the *abacus*, is known since 2400 BCE, although all its calculations are manual. The first mechanical calculators were used to perform astrological calculations, for example the *astrolabe*, which was invented around 150 BCE.

More modern efforts are Pascal's 1642 calculating machine and Leibniz's *Stepped Reckoner* from 1672. The latter was made into a commercial product, *the Arithmometer* by C.X. Thomas in 1820. Perhaps the most famous was Babbage's *analytical engine*, a design continuously improved from

1833 until 1871, the year of Babbage's death. Notably, its programs were stored on punchcards and it had control features similar to modern computers. Ada Lovelace is officially the world's first computer programmer, having written a program for the analytical engine to compute a sequence of numbers known as Bernoulli numbers.

The first *electronic* computers appeared before and during World War II. Replacing mechanical components with electronic circuits allowed higher speeds and smaller sizes. Variously claiming to be *the first* computer are Zuse's Z3 (Germany, 1941), the Atanasoff-Berry computer (USA, 1937) or the Colossus (UK, 1943).

ENIAC (US, 1946) was the first computer to resemble in internal structure today's computers. It executed a few thousands instructions per second, it weighed 27 tons and it consumed 150 kW of power. For comparison, a smartphone contains a computer which executes billions of instructions per second, weighs 150 g and consumes a few Watts.

The first modern programming language was Fortran, created in 1956 by J. Backus. It was designed especially to program the IBM 704.

1.3 Abstraction

Abstraction is a process by which essential concepts are separated or derived from detailed properties. A good way of seeing abstraction at work is in tree-like directories, where more abstract concepts are closer to the root. The shirt of footballer Craig Bellamy is found in the following hierarchy of abstractions:

1. Sports memorabilia
2. Football shirts
3. English League shirts
4. Liverpool shirts.

The detailed description is something like *Liverpool England Football XL Extra Large Mens Craig Bellamy Wales 2007 Home*. An abstraction is not unique, the same shirt being possibly abstracted as *red football shirt* or *shirt of famous Welshmen*.

In programming, abstraction is tremendously important. Because programs can be very complicated, the more detail we need to manage manually the more likely we are to make mistakes. Consider the following two ways of giving directions.

Low level:

1. Head northwest for 276 ft
2. turn left, continue for 0.5 mi
3. take 2nd exit at roundabout, continue for 0.2 mi
4. slight left, continue for 0.1 mi
5. turn left, continue for 0.2 mi
6. turn right, continue for 482 ft.

High level:

1. Go to Selly Oak train station via Harborne Ln

If correctly applied by the driver of a car stationed in front of University Station they will have the same effect: driving to Selly Oak train station. However, the second formulation leaves out many details and, because of that, it is more likely to be carried out correctly.

A computer, like a map, can be seen at many different levels of abstraction. At the lowest level we have, like in any physical object, elementary particles, their physical and chemical properties, and their energetic states. Higher up we have the geometrical structures of the processor and the memory etched onto wafers of silicone. Higher still, we have logical gates and connectors.

Higher still we have the architecture of the computer, consisting of abstract blocks with distinct functionalities. Programming can happen at this level, but it is incredibly detailed and tedious.

Administering the computer is one important program called an *operating system* (OS), and it is a program written at this level of abstraction, or at least its key components are. It gives a far more abstract view of the computer, quite similar to a *virtual machine*. Because the same OS can run on quite different computers this higher level of abstraction makes programming far easier, and also portable, because we program the OS and not the actual physical computer.

On top of the OS we can have further programming language infrastructure, which gives an even more abstract view of the system. This makes programs even more concise and even more portable, because we do not program an OS, but just a highly abstract programming language. These languages have been called, conventionally, *functional programming languages.*

1.4 Functional programming: a historical outlook

Functional programming (FP) languages are utterly top-down. They abstract away as much as possible from the physical computer and its OS. The focus of FP is not to reflect the capabilities of a certain machine model, but to give a direct way of representing information and computation. As a result, FP is concise, elegant and less error-prone than machine-oriented programming.

The roots of FP are not in engineering, but in mathematics, and especially in logic. The father of logic is Aristotle, a Greek philosopher who lived in 3rd Century BCE. His aim was to distinguish between what it means to be *true* as compared to what is simply widely believed or popular. The study of logic was an important aspect of philosophy throughout history, but the idea of *calculation* in logic was famously emphasised by Leibniz (17th Century) who proposed *lingua characteristica universalis*, a universal diagrammatic language for all human knowledge, and an associated *calculus ratiocinator*, calculational rules to be carried out by machines in order to derive all propositions:

> If controversies were to arise, there would be no more need of disputation between two philosophers than between two accountants. For it would suffice to take their pencils in their hands, and say to each other: Calculemus – Let us calculate.

The first to make logic more like mathematics was Boole, who in 1847 proposed an algebra of logical objects – *Boolean Algebra*. This project of formalising logic was much extended by Frege in 1879, and used to provide a logical foundation for all mathematics. It is possible that Frege's work would have remained obscure if their importance was not recognised by Russell, who promoted it while finding a critical flaw in it. Frege was devastated by this, and likely driven insane. Russell proceeded to fix this flaw, also known as Epimenides's paradox, since it was first formulated in classic Greece. The result was *Principia Mathematica*, a 2000-page book that took 10 years to write. It derived all of mathematics from logic, it had no obvious paradoxes, but it was still not proved to be free of contradictions!

By the late 20th Century the need to show that mathematics is *consistent* became more and more stringent. This need was formulated by Hilbert as a program to show that mathematics is consistent (no false statement can be proved true), complete (all true statements can be proved), and decidable (the proof can be done as a mechanical computation).

Hilbert's program was soon shown to be flawed by Gödel, who famously proved that arithmetic is incomplete and that mathematics cannot be proved to be consistent (1930-1933). Soon after Church and Turing proved that mathematics is not decidable either (1936-1937).

Out of the work of Church, Gödel, Kleene, Post and Turing the modern approach to FP was born, starting from the question: what mathematical functions can be computed? Most important, the first programming language for "computable mathematics" was defined by Church, the λ-calculus, a universal language for computation. Note that the language had only two operations: defining a function and applying a function. All computation can be reduced to this!

The first FP language was McCarthy's LISP (1950s), around the same time with FORTRAN, but it is based on a flawed understanding of the λ-calculus. The first modern FP language was Milner's ML (1970s), which grew in several dialects, most importantly OCaml and Microsoft's F#. The most modern widely used FP language is Haskell (1987). Both it and the ML-like languages are based directly on Church's λ-calculus.

Moreover, the ideas of the λ-calculus are becoming more and more prevalent. Many new programming languages such as Python have FP features, and existing programming languages such as Java and C++ are being extended to incorporate such features. Also, industry is beginning to recognise the advantages of FP: the average salary of a F# programmer last year was £65,000 compared with £47,500.[1]

As modern computing devices become more complex, having programming languages that reflect their architecture is no longer feasible. The level of abstraction must be raised, this is why FP is extremely important for the education of a future computer scientist or programmer.

FP like OCaml have another great advantage: we can prove, like in math, that a program is correct! We can do that by hand or we can use specialised tools. In this module we will do it by hand, so we will only write simple-enough programs. But in a world of crashing software, guaranteed-to-be-correct programs are growing more and more appreciated.

[1] `itjobwatch.co.uk`

2 "Formal" programming: rewrite systems

When programming, in general, we can ignore the physical computer an think of a a program in game-like terms:

- there is an initial position of the game (*initial configuration*);
- there are a set of rules of the game; each rule applies in a certain configuration and creates a new configuration;
- the game is played until a winning or losing position is reached; the winning configuration is also called *a result*.

Note that this is a lot like Maths: we are working with *structured data* and *theories*. Because the rules can be seen as *equations*, and their totality is a *theory*.

Consider the following puzzle:[2]

> Two llamas traveling on a very narrow ledge encounter two llamas camels coming the other way. Llamas never go backwards, especially when on a precarious ledge. The llamas will jump over each other, but only if there is a llama-sized space on the other side. The llamas didn't see each other until there was only exactly one llama's width between the two groups.
>
> How can all llamas pass, allowing both groups to go on their way?

How could we represent this puzzle?

Right-facing llama is represented by the letter R;

Left-facing llama is represented by the letter L;

Llama-sized space is represented by the letter O.

The situation at any given moment can be therefore represented by a *word* (sequence of letters). Initially, this is:

$$\cdots OOORROLLOOO \cdots$$

We actually need only one space to work with before and after the llamas, so we can use a simpler initial configuration:

$$ORROLLO$$

The rules of the game are the following:

Right-moving llama advances into a space, so the configuration in which the rule applies is RO. After the move it becomes OR. We write this rule as

$$mr: \qquad RO \Rightarrow OR$$

Left-moving llama advances into a space, so the configuration in which the rule applies is OL. After the move it becomes LO. We write this rule as

$$ml: \qquad OL \Rightarrow LO$$

Right-moving llama jumps over a left-moving llama, so the configuration in which the rule applies is RLO. After the move it becomes OLR. We write this rule as

$$jr: \qquad RLO \Rightarrow OLR$$

Left-moving llama jumps over a right-moving llama, so the configuration in which the rule applies is ORL. After the move it becomes LRO. We write this rule as

$$jl: \qquad ORL \Rightarrow LRO$$

[2] Adapted from `http://www.folj.com/folj.com/puzzles/`

Winning positions occur whenever all the right-moving llamas are to the right of all left-moving llamas, irrespective of the spaces.

$$\cdots L \cdots L \cdots R \cdots R \cdots$$

Losing positions occur whenever some right-moving llamas are stuck to the left of some left-moving llamas, like in

$$\cdots RRLL \cdots$$

A solution of the puzzle is given below. We underline the word to which we apply the rule, and we also indicate the rule that is being applied and its result:

$$\begin{aligned}
mr: &\quad ORROLLO \Rightarrow ORORLLO && (1)\\
jl: &\quad ORORLLO \Rightarrow ORLROLO && (2)\\
jl: &\quad ORLROLO \Rightarrow LROROLO && (3)\\
mr: &\quad LROROLO \Rightarrow LORROLO && (4)\\
mr: &\quad LORROLO \Rightarrow LORORLO && (5)\\
jl: &\quad LORORLO \Rightarrow LOROOLR && (6)\\
mr: &\quad LOROOLR \Rightarrow LOOROLR && (7)\\
mr: &\quad LOOROLR \Rightarrow LOOORLR && (8)\\
jl: &\quad LOOORLR \Rightarrow LOOLROR && (9)\\
win: &\quad LOOLROR && (10)
\end{aligned}$$

Note that we can look at this situation in two ways:

imperative The player must specify the rules which are to be applied, and the sequence of these rules are *the program*. For example above, the program is $mr; jl; jl; mr; mr; jl; mr; mr; jl$, resulting in *win*. This is how imperative programming languages such as C or Java work.

declarative The player must specify only the initial configuration, which is the program. It is up to the computer then to figure out what is the sequence of steps required to reach the winning position. This is how functional and logical programming languages such as OCaml, Lisp, Haskell or Prolog work.

The technical name for such a formal system is a *rewrite system*. A *good* such system has several properties:

Determinism means that in any configuration a single rule can be applied. The llama game obviously does not have this property. For example, ml can also be applied in the initial position.

Normalisation means that we are guaranteed to reach a winning position. The llama game does not have this property, since the moves $mr; mr$ result in a losing position $RRLL$.

Confluence means that if there are choices in how rules are to be applied, the order of the rules does not matter in terms of reaching a winning or losing position. The llama game doesn't have this property either, since from the initial configuration we can reach both winning and losing positions.

2.0.1 "Theorems"

Formal (rewrite) systems are very clearly specified and it is possible to formulate precise properties about them and *prove them*. Such properties are guaranteed to be always true, and we can rightfully call them *theorems* (like in Maths). The way we prove such theorems is by showing that:

1. The property is true in the initial configuration.

2. Any rule applied in any configuration (reachable from the initial configuration through repeated application of rules) results in a configuration that also has this property.

For example, how could we show that *it is not possible to reach LOROLORO.* We could perhaps examine all possible games and check that this configuration never occurs – but this is tremendously hard work. Or we can notice that

1. the initial configuration has 3 *O*s;
2. any rule keeps the number of *O*s constant;
3. so any configuration has 3 *O*s;
4. *LOROLORO* has 4 *O*s;
5. *LOROLORO* is not reachable.

2.1 Further reading

The Wikipedia entry for *Computer Science* is informative and well written. Many of the sub-topics are expanded in related articles which are also worth reading.

CS's origin in logic were explored by Prof. Moshe Vardi from Rice, who also gave a Distinguished Lecture on this topic in Birmingham in 2009. A version of the talk, including video and slides is posted online[3]. It is fun and very interesting.

Prof. Robert Harper at Carnegie Mellon has a blog on functional programming[4]. He is opinionated and writes very well. Some of the posts are technical but if you look up the section on *Teaching* you will find it informative and amusing.

The Wikipedia entry for *Rewriting*[5] is quite technical, but browsing it will give you a flavour about the tremendous scope and power of this technique.

[3] `http://www.cs.vt.edu/DistinguishedLectures/MosheVardi`
[4] `http://existentialtype.wordpress.com/`
[5] `http://en.wikipedia.org/wiki/Rewriting`

3 Starting with OCaml/F#

In the text below we assume that we have started the top-level OCaml shell. In typewriter font we see the input and the output of the shell.

3.1 Predefined types

One of the simplest OCaml programs we can write is this:

```
# 0;;
- : int = 0
```

It is the program that calculates the number zero, consisting only of the number 0 itself. Note the ;; terminator which indicates the end of the program. The execution of the program produces the following output:

- The dash indicates that the output value does not have a name (more about this later);

int This is a very important piece of information. Before calculating the value, OCaml must know what kind of value it is. It does that by examining your program very closely. In this case OCaml figured out that you want to calculate an integer value;

0 This is the actual value.

So the output told us that your program produces an anonymous integer value which is 0.
We can also give a name to a value. This name is called a *variable.*

```
# let zero = 0;;
val zero : int = 0
```

The program says "calculate 0 and let `zero` be the name of that value." In the output of the program, instead of the dash we now see:

`val zero` This says that the program is calculating a value and it is giving it the name `zero.`

We can calculate with integers using the common arithmetic operations.

```
# 8 * 5;;
- : int = 40
# 9 - 5 * 4;;
- : int = -11
# let squareof19 = 19 * 19;;
val squareof19 : int = 361
# squareof19 * 4 - 6;;
- : int = 1438
```

We can also use arithmetic-logic operations such as comparisons. Note the new type, **bool**:

```
# 9 > 8;;
- : bool = true
# 7 = 0;;
- : bool = false
# zero < squareof19;;
- : bool = true
```

In order to understand how OCaml evaluates an expression let's go back to the concept of *rewrite system.* The configurations are programs and the rules of rewriting are the rules of mathematic and logic plus a new rule: *substitution of names by the values they name.* In the case of the last program above, the succession of rule applications is:

```
zero < squareof19                 where zero=0, squareof19=1438
=> 0 < squareof19                 where zero=0, squareof19=1438
=> 0 < 1438                       where zero=0, squareof19=1438
true
```

The game is won when a *value* is produced and no more rules can be applied.

One cute thing: if a number is very large we can use _ to separate digits:

```
# let large = 10_000_000_000;;
val large : int = 10000000000
```

Other than integers we have the following common data types:

char characters

```
# 'a';;
- : char = 'a'
# let be = 'b';;
val be : char = 'b'
# let ce = 'c';;
val ce : char = 'c'
# be < ce;;
- : bool = true
```

Note that characters are ordered alphabetically.

strings of characters

```
# "hello focs";;
- : string = "hello focs"
# let greeting = "hello";;
val greeting : string = "hello"
# let students = "focs";;
val students : string = "focs"
# greeting < students;;
- : bool = false
# greeting ^ " " ^ students = "hello focs";;
- : bool = true
```

Strings are also ordered alpabetically, and can be *concatenated* using the *uparrow* symbol. We can extract a character from a string:

```
# greeting.[0];;
- : char = 'h'
# "hello".[0];;
- : char = 'h'
```

Here is, as an elaboration, how the last example is *evaluated*

```
greeting ^ " " ^ students = "hello focs"     where greeting="hello", students="focs"
"hello" ^ " " ^ students = "hello focs"      where greeting="hello", students="focs"
"hello " ^ students = "hello focs"           where greeting="hello", students="focs"
"hello " ^ "focs" = "hello focs"
"hello focs" = "hello focs"
true
```

floating point numbers, which are an approximation of real numbers.

```
# 1.4;;
- : float = 1.4
# 0.0;;
- : float = 0.
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312
```

Lets try to calculate the circumference of the Earth using the standard formula $c = 2\pi r$.

```
# 2 * pi * earthradius;;
Characters 4-6:
  2 * pi * earthradius;;
      ^^
Error: This expression has type float but an expression was expected of type int
```

3.2 Toplevel vs. local definition

This definition is top-level, i.e. the *scope* of the variable is until the end of the file or, if we are in the interactive toplevel, until we exit it:

```
let x = 1;;
```

This definition is *local to the expression*, i.e. the *scope* of the variable is until the expression finishes evaluation:

```
let y = 1 in y + y;;
```

Here is what happens:

```
# let x = 1;;
val x : int = 1
# x + x;;
- : int = 2
# let y = 1 in y + y;;
- : int = 2
# y + y;;
Characters 0-1:
  y + y;;
  ^
Error: Unbound value y
```

Note: "unbound" means "undefined".

Also note that variables *shadow* each other, and when the current one runs out of scope the previous one becomes visible. This is actually similar to how other languages deal with local variables.

3.3 Type errors

We just managed our first *type error*! Every expression in OCaml has a *type* and the types need to match. In this calculation above,

- 2 has the type of *integer*
- multiplication takes two *integers* and produces an *integer*
- `pi` is a *floating point number*.

The error says this: I was expecting an argument of type int (for the multiplication) but you are providing one of type float. Mismatch!

Let's have another go at it by using 2.0 instead of 2.

```
# 2.0 * pi * earthradius;;
Characters 0-3:
  2.0 * pi * earthradius;;
  ^^^
Error: This expression has type float but an expression was expected of type int
```

Of course, this is also an error because multiplication is expecting an integer argument. In OCaml integer multiplication and floating point multiplication use different symbols! [6]

```
# 2.0 *. pi *. earthradius;;
- : float = 40074.7842077247624
```

3.4 Defined types: variants

In OCaml we can easily define our own types. In this subsection we will only look at *enumeration*-like types, where the set of values is finite.

```
# type weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
type weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
# let d = Tue;;
val d : weekday = Tue
# let d' = Wed;;
val d' : weekday = Wed
```

One data type that only has two value is booleans. We can redefine that type! We don't need to, but it's just an exercise.

```
# type mbool = tt | ff;;
Error: Syntax error
```

Oops! The names that we use in defining types, called *constructors* (because they construct values of that type) must start with a capital letter.

```
# type mbool = TT | FF;;
type mbool = TT | FF
# let b = TT;;
val b : mbool = TT
# let b' = FF;;
val b' : mbool = FF
```

[6] F# is slightly different in that the arithmetical symbols are the same. They are said to be *overloaded* for float and int.

4 Functions

Just above we computed the circumference of the earth. How about the circumference of the moon, or that of a football:

```
# 2.0 *. pi *. 1737.4;;
- : float = 10916.4061526945334
# 2.0 *. pi *. 27.5;;
- : float = 172.78759594745
```

How about the circumference of a circle, in general, if we know the radius r?

```
# 2.0 *. pi *. r;;
Characters 13-14:
  2.0 *. pi *. r;;
               ^
Error: Unbound value r
```

Here OCaml is complaining that it doesn't know what r is. We need a way to tell OCaml not to worry about what r is, that it will be provided later. This is how we do it:

```
# fun r ->  2.0 *. pi *. r;;
- : float -> float = <fun>
```

The syntax `fun r -> ...` is very suggestive. The arrow tells us that r will need to be replaced in what follows by some actual value. This special kind of expression is called a **function**.

Important: You may have seen functions in Maths. OCaml functions are quite similar to mathematical functions, but they are not the quite the same thing. To make sure you understand exactly how they behave it is best to think about them formally, as an expression with a missing name, to be provided later.

The name is provided by *applying the function to an argument*. Let's say we need to calculate the circumference of a circle of radius 10:

```
# (fun r ->  2.0 *. pi *. r) 10.0;;
- : float = 62.8318530718
```

Here is the step-by-step *evaluation* of this program. As before, it consists of mathematical operations plus replacing names by their values.

```
(fun r ->  2.0 *. pi *. r) 10.0   where pi = 3.14...
=> 2.0 *. pi *. r                 where r = 10.0, pi = 3.14...
=> 2.0 *. 3.14.. *. 3             where r = 10.0
=> ... => 62.8318530718
```

Note that a function is itself a special kind of value, and can be given a name:

```
# let circumference = fun r ->  2.0 *. pi *. r;;
val circumference : float -> float = <fun>
```

And we use it just like before:

```
# circumference 1.5;;
- : float = 9.42477796077000107
```

Here is the step-by-step evaluation:

```
   circumference 1.5               where circumference= fun r->2.0 *. pi *. r, pi=3.14...
=> (fun r ->  2.0 *. pi *. r) 1.5  where pi=3.14...
=> 2.0 *. pi *. r                  where pi=3.14..., r=1.5
=> ...
```

Notation. For convenience, OCaml lets us define functions in two different, but perfectly equivalent ways:

```
let circumference = fun r ->  2.0 *. pi *. r
let circumference r =  2.0 *. pi *. r
```

The second one looks even more like a Maths function.

A technicality . The name of the argument is irrelevant, since it will be substituted for an actual value later. So these two functions are equal:

```
let circumference = fun r ->  2.0 *. pi *. r
let circumference = fun s ->  2.0 *. pi *. s
```

We can use the renaming of the argument as a *rule* if it clashes with other names in the program. For example, consider this:

```
let r = 10.0 in
let circumference = fun r ->  2.0 *. pi *. r in
circumference r
```

The step-by-step execution will go like this

```
let r = 10.0 in
let circumference = fun r ->  2.0 *. pi *. r in
circumference r
=>
let circumference = fun r ->  2.0 *. pi *. r in
circumference r
where r=10
=>
circumference r
where r = 10, circumference=fun r ->  2.0 *. pi *. r
=>
(fun r ->  2.0 *. pi *. r) r where r=10
```

At this stage we have two `r`'s which can be confusing, so we can simply rename the one which is used as argument to the function:

```
=> (fun r' ->  2.0 *. pi *. r') r where r=10
=> (fun r' ->  2.0 *. pi *. r') 10
=> 2.0 *. pi *. r' where r'=10
...
```

4.1 Pattern-matching and if

Pattern-matching resembles case statements in other languages, but it is more powerful let us "calculate" how we feel like on a given weekday:

```
type weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;

let how_i_feel day =
  match day with
| Mon -> "sleepy"
| Tue -> "grumpy"
| Wed -> "sneezy"
| Thu -> "dopey"
| Fri -> "happy"
| Sat -> "bashful"
```

```
| Sun -> "doc";;
val how_i_feel : weekday -> string = <fun>
# how_i_feel Tue;;
- : string = "grumpy"
```

To keep notation concise OCaml provides a simple notion of wildcards in patterns. Note that pattern matching is always attempted top-to-bottom!

```
type mbools = True | False;;

let mand b1 b2 =
  match (b1, b2) with
| True, True -> True
| _, _       -> False;;
val mand : mybools -> mybools -> mybools = <fun>
# mand True False;;
- : mybools = False
# mand True True;;
- : mybools = True
```

An aside: Comparison with Java The great advantage of pattern-matching over *case* statements in languages such as Java is the fact that patterns are checked at compile time both for exhaustiveness and reachability:

```
let how_i_feel day =
  match day with
(*| Mon -> "sleepy"*)
| Tue -> "grumpy"
| Wed -> "sneezy"
| Thu -> "dopey"
| Fri -> "happy"
| Sat -> "bashful"
| Sun -> "doc";;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Mon
val how_i_feel : weekday -> string = <fun>
```

Or

```
# let how_i_feel day =
  match day with
| Mon -> "sleepy"
| Tue -> "grumpy"
| Wed -> "sneezy"
| Thu -> "dopey"
| Fri -> "happy"
| Sat -> "bashful"
| Sun -> "doc"
| Mon -> "sleepy again";;
                  Characters 172-175:
   | Mon -> "sleepy again";;
     ^^^
Warning 11: this match case is unused.
val how_i_feel : weekday -> string = <fun>
```

Here is how an equivalent program must be written in Java:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumTest {
    public String how_i_feel(Day day) {
        switch (day) {
            case MONDAY:    return "sleepy";
            case TUESDAY:   return "grumpy";
            case WEDNESDAY: return "sneezy";
            case THURSDAY:  return "dopey";
            case FRIDAY:    return "happy";
            case SATURDAY:  return "bashful";
            case SUNDAY:    return "doc";
        }
        return "unreachable";
    }
}
```

Do you note something annoying? Yes, it's the last *return* statement. However, this is what happens if we remove it:

```
EnumTest.java:21: missing return statement
    }
    ^
1 error
```

The Java compiler cannot tell that we have exhausted all the possibilities in the *switch* statement. It will also equally not inform us if we forgot one of the cases – no warning or error. This is a significant source of possible programming errors!

Branching OCaml provides a branching expression which works uniformly across all types, `if-then-else`. However, whenever possible we should use pattern matching instead!

```
# if 4 > 0 then 7 + 8 else 4 - 5;;
- : int = 15
```

The step-by-step evaluation is

```
# if 4 > 0 then 7 + 8 else 4 - 5
=> if true then 7 + 8 else 4 - 5
=> 7 + 8
=> 15
```

Note that both branches need to have the same type!

```
# if 4 > 8 then 0 else 'a';;
Error: This expression has type char but an expression was expected of type
         int
```

This is because otherwise we wouldn't be able to make a function out of if statements:

```
# let min x y = if x < y then x else y;;
val min : 'a -> 'a -> 'a = <fun>
# min 4 5;;
- : int = 4
# min 'a' 'z';;
- : char = 'a'
```

Avoiding if sillinesses. If statements must be avoided because they have the same problem as the Java *switch* statements: no compiler help if a case statement is forgotten or redundant. They are also clumsier and harder to read:

```
let how_i_feel day =
  if day = Mon then "sleepy"
  else if day = Tue then "grumpy"
  else if day = Wed then "sneezy"
  else if day = Thu then "dopey"
  else if day = Fri then "happy"
  else if day = Sat then "bashful"
  else if day = Sun then "doc"
  else "error";;
val how_i_feel : weekday -> string = <fun>
```

Another thing to bear in mind is that *if* is a boolean operation which returns a boolean value, and can be used as a ternary operator. The following style of writing is unfortunately too common!

```
if ...some test... then true else false
```

instead of just

```
...some test...
```

or

```
if ...some test... then false else true
```

instead of

```
not (...some test...)
```

or

```
if sometest = true then something else somethingelse
```

instead of

```
if sometest then something else somethingelse
```

And so on.

5 Multiple arguments. Polymorphism. Tuples.

5.1 Multiple arguments

Using the `fun` syntax we can provide any number of arguments to an OCaml function. For example, the size of the hypotenuse in a right-angled triangle is $h = \sqrt{a^2 + b^2}$.

```
# fun a -> (fun b -> sqrt (a *. a +. b *. b));;
- : float -> float -> float = <fun>
```

We will need to provide first the `a`, then the `b` in order to calculate the answer. Purely for convenience, OCaml lets us write a named version of the function also like this:

```
let hyp a b = sqrt (a *. a +. b *. b);;
```

I have just introduced the *library function* `sqrt` which calculates the square root of a float.

Lets calculate with this function, step-by-step:

```
hyp 2.0 3.0 where hyp=fun a -> (fun b -> sqrt (a*.a+.b*.b))
=> (fun a -> (fun b -> sqrt (a*.a+.b*.b))) 2.0 3.0
=> (fun b -> sqrt (a*.a+.b*.b)) 3.0 where a=2.0
=> sqrt (a*.a+.b*.b) where a=2.0, b=3.0
=> sqrt (2.0 *. 2.0 +. 3.0 *. 3.0)
=> 4.0
```

A short comment on the type `float -> float -> float`. To make it clearer how the arrow associates, I will add brackets: `float -> (float -> float)`. This means that we are talking about a function that receives a `float` and returns a new function! This is unusual! We think of functions as expressions that take some data and return some data, but in fact a function can take anything, including functions, and return anything, including functions. Functions *are data* in OCaml.

If you think about it, it makes sense: we need to provide two arguments to `hyp`. If we provide just the first argument (`a`), we are still left with an expression which needs another argument (`b`). An expression which needs an argument is a function!

5.2 Polymorphism

Here is a rather funny function, which takes as argument another function and another argument and it applies the function to the argument:

```
# let eval = fun f -> fun x -> f x;;
val eval : ('a -> 'b) -> 'a -> 'b = <fun>
```

Note. The type of `eval` includes some funny names: `'a, 'b`. They are *type variables*. OCaml is trying to guess the types of `f` and `x` but can only guess the *shape* of the type. The data will need to be figured out later. This is called *polymorphism* and we will see a lot of it.

Here is another funny function, the *identity*:

```
# let id = fun z -> z;;
val id : 'a -> 'a = <fun>
```

It simply returns its argument – not useless as you will see! It also has a polymorphic type.

Now here is an interesting fact we can establish about these two silly functions, from the following observations:

```
# eval id 5;;
- : int = 5
# eval id 9.8;;
- : float = 9.8
# eval id 'c';;
- : char = 'c'
# eval id "got it?";;
- : string = "got it?"
```

Lets go through one of the evaluations step-by-step:

```
eval id 5 where eval=fun f->fun x-> f x, id=fun z->z
=> (fun f -> fun x -> f x) id 5 where id=fun z->z
=> (fun x -> f x) where f=id, id=fun z->z
=> f x where f=id, id=fun z->z, x=5
=> id x where id=fun z->z, x=5
=> (fun z->z) x where x=5
=> z where z=x, x=5
=> x where x=5
=> 5
```

We note that in general it should be the case that

```
eval id a = a
```

for any `a`.

However, this equality cannot be established by OCaml! We need to *prove it*:

```
# eval id a = a;;
Characters 8-9:
  eval id a = a;;
           ^
Error: Unbound value a
```

And the proof is exactly the evaluation above, where we can use `a` instead of 5:

```
eval id a where eval=fun f->fun x-> f x, id=fun z->z
=> (fun f -> fun x -> f x) id a where id=fun z->z
=> (fun x -> f x) where f=id, id=fun z->z
=> f x where f=id, id=fun z->z, x=a
=> id x where id=fun z->z, x=a
=> (fun z->z) x where x=a
=> z where z=x, x=a
=> x where x=a
=> a
```

So we have established formally that `eval id a` *alwyas reduces to* `a`, therefore they are equal.

5.3 Notation

There is another, cuter way to write function application (this is actually predefined in OCaml):

```
let (|>) x f = f x;;
```

Why is this neat? Because it makes *nested function application* less confusing. Compare the following which calculates the hypotenuse of an isosceles right-angle triangle:

```
let dbl x = 2. *. x;;
let sqr x = x  *. x;;
let hyp x = sqrt (dbl (sqr (x)));;
```

So many brackets! Here is the improved syntax:

```
let hyp x = x |> sqr |> dbl |> sqrt;;
```

This is also closer to how we think about such calculations:

> You take an `x`, then you square it, then you double the result, then you take the square root.

Finally, if some of the intermediate steps produced significant values, which you migth want to reuse, you can also write like this:

```
let hyp x =
    let sqrit = sqr x in
    let dblit = dbl sqrit in
    let sqrtit = sqrt dblit in
    sqrtit;;
```

The last is beginning to look similar to how you would do it in a Java-like language

```
public float hyp (float x) {
    float sqrit = sqr (x);
    float dblit = dbl (sqrit);
    float sqrtit = sqrt (dblit);
    return sqrtit;
    }
```

5.4 Tuples

A function that takes multiple arguments, such as

```
let hyp a b = sqrt (a *. a +. b *. b);;
val hyp : float -> float -> float = <fun>
```

can also receive all its arguments in one go, as a *tuple*:

```
let hyp' (a, b) = sqrt (a *. a +. b *. b);;
val hyp' : float * float -> float = <fun>
```

Function `hyp` takes two arguments, first one a `float` and the second one a `float`, whereas `hyp'` takes only one argument which is a *pair* with components `float` and `float` written as `float * float`: a very subtle difference!

In fact we can rewrite `hyp` yet again so that it only uses one argument variable, which is then *decomposed* using pattern-matching:

```
let hyp'' ab =
    let (a, b) = ab in
    sqrt (a *. a +. b *. b);;
val hyp'' : float * float -> float = <fun>
```

Note that `hyp'` and `hyp''` are perfectly equivalent, the difference is simply notational.

In OCaml tuples are a first-class data type.

```
# (1, 2);;
- : int * int = (1, 2)
```

OCaml tells us it has computed a value that is a tuple of an int with an int, having value (1, 2). Tuples can be defined, of course, using computations:

```
# (1 + 2, 3 + 4);;
- : int * int = (3, 7)
```

And they need not be of the same type:

```
#  (1, '1', "1");;
- : int * char * string = (1, '1', "1")
```

This is a tuple of three different types. But they can be of any type – even functions.

```
# (circumference, 5);;
- : (float -> float) * int = (<fun>, 5)
```

Moreover, tuples can be *of tuples*.

```
# (1, (2, 3));;
- : int * (int * int) = (1, (2, 3))
# ((1, 2), 3);;
- : (int * int) * int = ((1, 2), 3)
```

The first one is a tuple of an int with a tuple, the second one is a tuple of a tuple with an int. They are not only not equal, they are not even comparable, because they have different types:

```
# (1, (2, 3)) = ((1, 2), 3);;
Characters 14-25:
  (1, (2, 3)) = ((1, 2), 3);;
                ^^^^^^^^^^^
This expression has type (int * int) * int but is here used with type
  int * (int * int)
```

Note that in OCaml we can define a tuple also directly as a triple:

```
# (1, 2, 3);;
- : int * int * int = (1, 2, 3)
```

This is yet another type, which is not equal to any of the above!

5.5 More on pattern-matching

We can also use, if we want, the standard syntax for pattern-matching:

```
fun x ->
      match x with (x', y') -> x' + y' ;;
  - : int * int -> int = <fun>
```

Here is this function in action to calculate `1+2=3`:

```
(fun x -> (match x with (x', y') -> x' + y')) (1, 2)
=> match x with (x', y') -> x' + y' where x = (1, 2)
=> match (1, 2) with (x', y') -> x' + y'
```

Note that indeed the value (1, 1) "matches" with the pair of variables `(x', y')` which *simultaneously* get the two values.

```
=> x' + y' where x'=1, y'=2
=> 1+2 => 3
```

This syntax is clunky, this is why OCaml lets us bring up the pattern-matching directly into the function definition when we name it. All the functions below are identical. Just use whichever feels more convenient:

```
let sum (x, y) = x + y
let sum        = fun xy -> (match xy with (x, y) -> x + y)
let sum        = function (x, y) -> x + y
let sum xy     = let (x, y) = xy in x + y
```

One standard thing we can do with a tuple is to extract the components. This operation is called *projection*. Let us define the "first" and "second" projection for a pair.

```
# let fst (x, y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y;;
val snd : 'a * 'b -> 'b = <fun>
```

We can program with projections instead of pattern-matching:

```
let sum xy = (fst x) + (snd y)
```

But it is much more tedious because each tuple type needs its own projection functions, whereas pattern-matching is part of the language syntax in a more uniform way.

In Java-like languages where pattern-matching is not part of the language projections are the only option.

```
public class Tuple<X, Y> {
  public final X x;
  public final Y y;
  public Tuple(X x, Y y) {
    this.x = x;
    this.y = y;
  }
}

public Integer sum (Tuple<Integer, Integer> xy) { return xy.x + xy.y; }
```

which must be called like this:

```
sum (new Tuple (1, 1))
```

5.6 Some comments on type

Remember that a function of multiple arguments such as sum can be also written in another way:

```
# let sum' x1 x2 = x1 + x2 ;;
val sum' : int -> int -> int = <fun>
```

Although this calculates the same thing, the types are quite different. Lets examine the type, which can be clearer by adding superfluous brackets:

```
  int -> (int -> int)
```

It is a *function* that takes an integer and produces a *function* that takes an integer and produces an integer. This is a common way of writing functions in FP to avoid extra brackets, and it essentially the same as

```
  int * int -> int
```

but it is convenient because it avoids tuples. You can think of it either way, just make sure you write it properly so that the types match.

```
# let sum (x1, x2) = x1 + x2 ;;
val sum : int * int -> int = <fun>
# let sum' x1 x2 = x1 + x2 ;;
val sum' : int -> int -> int = <fun>
# sum (10, 20);;
- : int = 30
# sum' (10, 20);;
Characters 5-13:
  sum' (10, 20);;
       ^^^^^^^^
This expression has type int * int but is here used with type int
# sum' 10 20 ;;
- : int = 30
# sum' (10, 20) ;;
Characters 5-13:
  sum' (10, 20) ;;
       ^^^^^^^^
This expression has type int * int but is here used with type int
```

6 Isomorphism of types

Consider these two functions:

```
# let sum (x1, x2) = x1 + x2 ;;
val sum : int * int -> int = <fun>
# let sum' x1 x2 = x1 + x2 ;;
val sum' : int -> int -> int = <fun>
```

They clearly compute the same thing, `x1+x2`, but they have different types:

```
int * int -> int   versus   int -> int -> int
```

The first one is a type which expects one argument, a pair of ints. The second one expects two arguments, each an int. These types should be very similar, because "one pair of ints" and "two ints" should be the same thing.

Consider these three tuples:

```
# (1,2,3);;
- : int * int * int = (1, 2, 3)
# ((1,2),3);;
- : (int * int) * int = ((1, 2), 3)
# (1,(2,3));;
- : int * (int * int) = (1, (2, 3))
```

They are *essentially* the same although they again have different types and cannot even be compared (the equality operator expects things of the same type):

```
# (1,2,3)=(1,(2,3));;
Characters 8-17:
  (1,2,3)=(1,(2,3));;
          ^^^^^^^^^
Error: This expression has type int * (int * int)
       but an expression was expected of type int * int * int
```

There is a technical word for two things that are *essentially the same* without being equal: *isomorphic*. This comes from Greek and means "same shape". They are not equal, but have "the same shape".

Here we will make it clear what it means to have the same shape:

> Two types `'a` and `'b` have the same shape (*isomorphic*) if we can construct functions `f : 'a -> 'b` and `g : 'b -> 'a` such that `f(g(x)) = x` and `g(f(y)) = y` for any `x`, `y` of the appropriate type.

Showing that two types have the same shape is tricky: we need to guess the functions!

Lets see that `int * (int * int)` and `int * int * int` have the same shape. We just need to provide the functions:

```
let f (a, (b, c)) = (a, b, c)
let g (a, b, c) = (a, (b, c))
```

This seems quite trivial, isn't it? Indeed, most of the time the choice of the functions is going to be quite obvious given the types! Lets *prove* that this is an isomorphism.

```
f(g(x)) = ???
```

Here it is a little subtle. In order to be able to apply `g(x)` it must be that `x` can be pattern-matched into a triple like `(x1, x2, x3)`. Anything else will give a compile-time error, and we are only considering correct programs. So

```
f(g(x))
=> f(g(x1, x2, x3))
=> f(x1, (x2, x3))
=> (x1, x2, x3)
```

Similarly,

```
g(f(x))
=> g(f(x1, (x2, x3)))
=> g(x1, x2, x3)
=> (x1, (x2, x3))
```

6.1 Quick recap of OCaml syntax

These are the things we learned:

int is the type of integer numbers. Examples: `0, 7, -5`. On ints we can do arithmetic (`7-4`) and comparison (`8>9`).

bool is the type of booleans. Examples: `true, false.` On bools we can compute logical expressions such as "and" (`true && (7<9)`), "or" (`false || 6=0`).

char is the type of characters. Examples: `'a', 'b', 't'`. We can compare chars.

string is the type of strings of characters. Examples: `"hello"`. We can concatenate (`"go" ^ "  " ^ "away"`) or compare strings.

variables are names we give expressions, using `let`. For example: `let zero = 0`.

functions are expressions with a "missing name", specified by the `fun` keyword. For example `fun x -> x+1` is a "functions" that adds 1 to whatever *argument* is provided. Arguments are provided by applications, which means writing it just after the function `(fun x->x+1) 4`.

named functions can be created using `let` in two equivalent ways, for example: `let f x = x + 1` or `let f = fun x -> x+1`.

tuples or *pairs* are created using brackets, for example `(1, 2, 3)` or `(7, ("a", 'b'))`.

6.2 Further reading

The official tutorial introduction to OCaml is

`http://caml.inria.fr/pub/docs/manual-ocaml/manual003.html`

Beware, it moves very fast and it assumes a lot of prior knowledge of FP concepts. We have covered sections 1.1–1.3.

For the full list of basic OCaml operations and functions check out this link:

`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`

to the language definition. It is a *reference* not a *tutorial* so it will also include some things you don't understand yet. Don't worry about it. Just experiment with the things that you do understand.

Chapters 1 and 2 of Paulson's "ML for the Working Programmer" expand on the same issue, but don't be confused about the syntax – it is more old-fashioned and slightly different than that of OCaml. But the basic ideas are the same.

For a basic introduction to *mathematical* sets and functions this is a nice, extended tutorial:

`http://www.mth.pdx.edu/resources/set.pdf`.

7 Lists

Lists are to computer science what sets are to Maths: the most fundamental data structure. Unlike sets, in a list the order of elements matters and an element may occur several times.

We have already seen that we can create more complicated data structures using tuples. However, when we need to put together many elements tuples become awkward to work with. Imagine that you want to sum all the elements in this 8-tuple:

```
# (1, 3, 7, 8, 2, 3, 2, 9);;
- : int * int * int * int * int * int * int * int = (1, 3, 7, 8, 2, 3, 2, 9)
```

What if you wanted to somehow put together 100 or 1,000 or 1,000,000 elements? Tuples would be just incredibly awkward. And what if you wanted to put together a bunch of elements without knowing how many you have, only that it's a finite number of them? Then tuples would just not work.

In this case we will use *lists*, which are finite enumerations of elements. In OCaml we write lists like this:

```
# [1; 3; 7; 8; 2; 3; 2; 9];;
- : int list = [1; 3; 7; 8; 2; 3; 2; 9]
```

Note that the output of the evaluation gives `int list` as the type. Indeed, this is a `list` of `ints`. This should suggest you that the elements of a list should be always of the same type, unlike those of tuples:

```
# [1; '2'];;

Characters 4-7:
  [1; '2'];;
      ^^^
Error: This expression has type char but is here used with type int
```

What does this error actually mean? OCaml rules state that lists must be of elements of the same type. The interpreter looked at the first element, `1`, and figured it was an `int`. Which means that all the other elements should be `int` as well, which is not the case with `'2'` which is a `char`. Error!

Beware! It is easy here to make a small syntactic error which can lead to confusing errors, if you type "," instead of ";":

```
# [1, '2'];;
- : (int * char) list = [(1, '2')]
```

No error! But the type is all wrong. Because we used the comma, the interpreter thought that we are creating the *pair* (1, '2') so it constructed a list of pairs, containing that one element.

This tells you that we can make lists of anything: lists of basic data such as `bool, int, float`, or lists of pairs, or lists of functions, or even lists of lists!

```
# [[1; 2; 3]; [3; 2; 1; 0]];;
- : int list list = [[1; 2; 3]; [3; 2; 1; 0]]
```

The *empty list* is polymorphic, as the type of its elements is not fixed yet:

```
# [];;
- : 'a list = []
```

7.1 Arrays versus lists

Lists are one of the simplest *data structures* that we can use to process collections of items. In more machine-oriented programming languages you will have seen *arrays* as the simplest form of programming with collections. OCaml also has arrays, which are created with a slightly different syntax:

```
# [1; 3; 7; 8; 2; 3; 2; 9];;
- : int list = [1; 3; 7; 8; 2; 3; 2; 9]
# [|1; 3; 7; 8; 2; 3; 2; 9|];;
- : int array = [|1; 3; 7; 8; 2; 3; 2; 9|]
```

In fact, arrays and lists are *isomorphic* data structure. It looks like mapping between lists and arrays is just a matter of adding or removing the vertical bars in the definition, although formally things will turn out to be a bit more complicated. The key difference between lists and arrays is not what they store but in how they work.

With a *data structure* the two main questions you need to ask is, in a broad sense, how you *get* stuff from them and how you *put* stuff on them. Here is where lists and arrays differ a lot.

Getting data from an array. In an array every element has an *index*, which is a number. If you need to get to an element you need to know its index and use it to access it:

```
# let a = [|1; 3; 7; 8; 2; 3; 2; 9|];;
# a.(0);;
- : int = 1
val a : int array = [|1; 3; 7; 8; 2; 3; 2; 9|]
# a.(1);;
- : int = 3
# a.(4);;
- : int = 2
```

This may seem like very convenient and simple, but it turns out it is a rich source of errors in programming. Managing integer indices can be a pain. Here's what can happen:

```
# a.(10);;
Exception: Invalid_argument "index out of bounds".
```

`Exception` means run-time error; in other words, our program just crashed because we asked it to perform an illegal operation: access an array element which is not there.

```
# a.(-1);;
Exception: Invalid_argument "index out of bounds".
```

Beware! Calculating with numbers is always more complicated than it seems! We have already seen in the previous lecture how overflow and precision are important issues.

Runtime errors Throwing runtime errors is often useful to indicate the fact that a program has not been used properly. In OCaml there are very sophisticate mechanisms for doing this, but lets just look at the simplest possible way. It is a predefined function called `failwith()` which takes an error message. Calling it will abruptly terminate all program execution and issue that error message.

```
# failwith ("boom");;
Exception: Failure "boom".
```

7.2 Getting data from a list.

Now lets look at lists. There are several ways in which you can access lists but I will teach you the best way: *pattern matching*. Pattern matching will turn out to be our favourite way of accessing any data structure.

Any list will fit only two possible patterns:

empty The list is empty, case in which the *pattern* it matches `[]`

head/tail The list has a *head*, its first element, and a *tail*, the rest of the list, case in which it matches the pattern `hd::tl`.

So when we want to access elements from a list we only need to consider these two cases: either the list is empty, or it has a head element and a tail list.

Also note that any list can be written as `hd::tl`

```
[1;3;5;7] = 1::[3;5;7] = 1::(3::[5;7]) = 1::3::5::7::[].
```

Do not be confused about the *tail*: it is not the next element and it is not the last element! It is a smaller list consisting of the rest of the elements.

The way we write this in an OCaml function which operates on list is like this:

```
# let somefunction ls = match ls with
  | [] -> ...evaluate when the list is empty...
  | hd :: tl -> ... evaluate when the head is hd and the tail is tl...
```

Let us write a very simple function which checks if a list is empty, returning `true` if that is the case and `false` if that is not the case:

```
# let empty ls = match ls with
  | [] -> true
  | hd :: tl -> false;;
val empty : 'a list -> bool = <fun>
# empty [1; 2; 3];;
- : bool = false
# empty ['a'; 'b'];;
- : bool = false
# empty [];;
- : bool = true
```

Note that `empty` is polymorphic: it works on *any* list.

```
[[]];;
- : 'a list list = [[]]
# empty [[]];;
- : bool = false
```

Lets try another function, `single` which checks whether a function has precisely one element:

```
# let single ls = match ls with
  | [] -> false
  | hd :: tl -> empty tl;;
val single : 'a list -> bool = <fun>
# single [];;
- : bool = false
# single [1];;
- : bool = true
# single [1; 2];;
- : bool = false
```

We can express this function in words as follows: *a list has precisely one element if it is not empty and its tail is empty.* Here is a step-by-step evaluation:

```
single [1]
=> (match ls with | [] -> false | hd::tl -> empty tl) (1::[])
=> empty tl where tl = []
=> (match ls with | [] -> true | hd :: tl -> false) []
=> true.
```

8 Recursion.

8.0.1 Sums

Recall the function that summed the elements of a pair:

```
let sum2 (x, y) = x + y
```

Now consider summing the elements of a triple. We can write either

```
let sum3 (x, y, z) = x + y + z
```

or we can use the previous pattern:

```
let sum3 (x, (y, z)) = x + sum2 (y, z)
```

or just

```
let sum3 (first, rest) = x + sum2 rest
```

Adding the elements of a quadruple then is

```
let sum4 (x, (y, (z, u))) = x + sum3 (y, (z, u))
```

or just

```
let sum4 (first, rest) = first + sum3 rest
```

If we keep going, in general we will have that

```
let sum<n> (first, rest) = first + sum<n-1> rest
```

This is where the magic of lists kicks in! Unlike an n-tuple, which is made up of an element and a (n-1)-tuple, a list is made up of a first element (called *the head*) and another list, the rest of the list, (calle *the tail*):

```
let rec sum (head::tail) = head + sum tail
```

If we try to compile this we get a *warning*

```
# let rec sum (head::tail) = head + sum tail;;
Characters 12-42:
  let rec sum (head::tail) = head + sum tail;;
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val sum : int list -> int = <fun>
```

The function compiled successfully but Ocaml reminded us that we are forgetting a case: the empty list [], which is indeed not formed of a head and a tail. This is what happens if we want to run this function:

```
# sum [2;4];;
Exception: Match_failure ("//toplevel//", 1, 12).
```

To understand the error lets follow the step-by-step evaluation:

```
sum [2;4]
=> sum (2::[4])
=> head+sum tail where head=2, tail=[4]
=> 2 + sum [4]
=> 2 + sum (4::[])
=> 2 + 4 + sum []
=> Exception: Match_failure
```

To fix it we just need to add the missing case!

```
let rec sum x = match x with
     | [] -> 0
     | head::tail -> head + sum tail
```

or, equivalently and more succintly,

```
let rec sum = function
     | [] -> 0
     | head::tail -> head + sum tail
```

Now the evaluation is:

```
sum [2;4]
=> sum (2::[4])
=> head+sum tail where head=2, tail=[4]
=> 2 + sum [4]
=> 2 + sum (4::[])
=> 2 + 4 + sum []
=> 2 + 4 + 0
=> 6
```

Isn't pattern-matching quite amazing? It could detect at compile-time that a case was missing from our function definition!

8.0.2 Count

Let us try to write now a function which computes the number of elements in a list. Following the pattern of thinking in the previous section, it will look something like this:

*The **number of elements** of a list is:*

- *if the list is empty then it is 0*
- *if the list has head and tail, then it is 1 plus the **number of elements** in the tail*

If you find it confusing, try with tuples, triples, quadruples ... and generalise to n-tuples like before.

In OCaml we write it like this:

```
# let rec length ls = match ls with
  | [] -> 0
  | hd::tl -> 1 + length tl;;
val length : 'a list -> int = <fun>
# length [];;
- : int = 0
# length [1];;
- : int = 1
# length [1; 2; 3; 4];;
- : int = 4
```

This is called a *recursive* definition because it is defined in terms of itself. OCaml needs to know whether a definition is recursive or not, hence the new word `rec` in the definition of the function.

Here is how the function calculates in a step-by-step way:

```
length [1; 2; 3]
=> length (1::[2;3])
=> 1 + length tl where tl = [2;3]
=> 1 + length (2::[3])
=> 1 + 1 + length tl' where tl' = [3]
```

```
=> 2 + length (3::[])
=> 2 + 1 + length []
=> 3+0
=> 3.
```

Nth. Let us write a function `nth` which selects the n-th element of a list, much like an array index does. Note that we need to follow the same pattern, what to do for the empty list and what to do for the list made of a head and a tail:

empty The empty list has no elements so we cannot apply the index: this is an error! In this case we will call `failwith`.

head and tail There is only one case when we can simply return the answer, and that is if the index is 0, because the first element in a list has index 0. But what if the index we want is not zero? We need to look for it into the tail. However, we need to do a small adjustment, because when we look into the tail it means that we have already skipped past the first element, so the indices need to be off by one!

Here's how the function looks like:

```
# let rec nth ls n = match ls with
  | [] -> failwith "Index out of bounds"
  | hd :: tl -> if n = 0 then hd else nth tl (n-1);;
val nth : 'a list -> int -> 'a = <fun>
# nth [1;2;3;4] 3;;
- : int = 4
# nth [0;1;2;3;4;5] 4;;
- : int = 4
# nth [0;1;2] 3;;
Exception: Failure "Index out of bounds".
```

Here is an example of how the function calculates in a step-by-step way:

```
nth [1;2;3] 2 = if 2 = 0 then 1 else nth [2; 3] (2-1)     (n = 2, hd = 1, tl = [2; 3])
              = nth [2; 3] 1                              (2 = 0 is false)
              = if 1 = 0 then 2 else nth [3] (1-1)        (n = 1, hd = 2, tl = [3])
              = nth [3] 0                                 (1 = 0 is false)
              = if 0 = 0 then 3 else []                   (n = 0, hd = 3, tail = [])
              = 3
```

8.1 Creating lists

Another important aspect of *data structures* is how to construct them. This is where a list is much better than an array. Constructing an array is a one-shot deal. Because a list is made of a head (which is an element) and a tail (which is a list) we can always increase the size of a list. We cannot do that with arrays!

```
# 0 :: [1; 2; 3] ;;
- : int list = [0; 1; 2; 3]
```

Note that the `::` notation can be used both to pattern-match and to create lists.

Append. Another useful function is combining two lists to make a larger list. This operation is called `append` and it takes two arguments, lets call them `ls1, ls2`.

As usual, let us look at the cases we need to consider and figure out in English what needs to be done. Because we have two lists, there are actually four cases to consider, but as it will turn out we won't care whether `ls2` is empty or not, so we only need to consider `ls1.`

empty The result of appending to the empty list another list `ls2` is just `ls2`

head and tail In this case we append to the tail, then add the head back!

```
#  let rec append ls1 ls2 = match ls1 with
  | [] -> ls2
  | hd::tl -> hd :: (append tl ls2);;
    val append : 'a list -> 'a list -> 'a list = <fun>
# append [0;1;2;3;4] [5;6;7;8];;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]
# append [] [1;2;3;4];;
- : int list = [1; 2; 3; 4]
# append [1;2;3;4] [];;
- : int list = [1; 2; 3; 4]
```

Here is an example of step-by-step execution.

```
  append [1; 2] [3; 4] = 1 :: (append [2] [3; 4])
                       = 1 :: (2 :: (append [] [3; 4]))
                       = 1 :: (2 :: [3; 4])
                       = [1; 2; 3; 4]
```

Revert. Let us look at another useful and simple function, reversing the list.

empty The reverse of the empty list is the empty list.

head and tail We reverse the tail then we add the head at the end. Note that because we add the head *at the end* we cannot use `::` but we need to make it into a list and use `append` instead!

```
# let rec revert ls = match ls with
  | [] -> []
  | hd :: tl -> append (revert tl) [hd];;
val revert : 'a list -> 'a list = <fun>
# revert [];;
- : 'a list = []
# revert [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

The `revert` function is quite interesting because it uses another function, `append`. Lets calculate by hand the reverse of a list:

```
1  revert [1; 2; 3] = append (revert [2; 3]) [1]
2                   = append (append (revert [3]) [2]) [1]
3                   = append (append (append (revert []) [3]) [2]) [1]
4                   = append (append (append [] [3]) [2]) [1]
5                   = append (append [3] [2]) [1]
6                   = append (3 :: (append [] [2]) [1]
7                   = append (3 :: [2]) [1]
8                   = append ([3; 2]) [1]
9                   = 3 :: (append [2] [1])
10                  = 3 :: (2 :: append [] [1])
11                  = 3 :: (2 :: [1])
12                  = [3; 2; 1]
```

It took 12 computational steps to revert a list of 3 elements. This seems inefficient because it is!

Tail recursion. Functions normally return their values, but there is a very common functional programming tricks where we can use the arguments to "accumulate" a result. Let us try to implement `revert` like this and use an extra argument to store an intermediate result. Lets call this function `revap ls sl`; the second argument will hold the "partially reversed" list. As before, the cases concern only `ls`:

empty If `ls` is empty we return `sl`

head and tail If `ls` has a `hd` and a `tl` we "shift" `hd` from the first to the second argument.

```
#  let rec revap ls sl = match ls with
  | [] -> sl
  | hd :: tl -> revap tl (hd :: sl);;
    val revap : 'a list -> 'a list -> 'a list = <fun>
# revap [1; 2; 3] [] ;;
- : int list = [3; 2; 1]
```

Note that when we call `revap` we give it an empty list [] as second argument. This is where the result "accumulates".

Let us execute it by hand on the same example

```
1 revap [1; 2; 3] [] = revap [2; 3] (1 :: [])
2                    = revap [3] (2::(1::[]))
3                    = revap [] (3::(2::(1::[])))
4                    =  (3::(2::(1::[])))
5                    =  [3; 2; 1]
```

Much better! Only 5 steps.

8.2 Further reading

For extra information you may read Paulson's notes up to page 45 and pages 69-80 in Paulson's book (*ML for the Working Programmer*).

8.3 More on patterns

The pattern system of OCaml is very expressive. The basic recipe of writing a list function like this

```
# let rec f ls = match ls with
  | [] -> ...
  | hd :: tl -> ...  f tl ...
```

is enough but it is sometimes awkward.

Maximal element. Suppose that we need to find the maximum element in a list. Here is a first (wrong) attempt:

```
let rec maxl ls = match ls with
  | [] -> failwith "Empty list"
  | hd :: tl -> max hd (maxl tl)
```

The reasoning is:

- if the list is empty then there is no maximum element – error
- if the list is not empty we take the maximum between the head and the maximum of the rest of the list.

However, if we run it:

```
# maxl [3; 2; 9];;
Exception: Failure "Empty list".
```

What is going on? Lets evaluate by hand:

```
maxl [3; 2; 9] = max 3 (maxl [2; 9])
               = max 3 (max 2 (maxl [9]))
               = max 3 (max 2 (max 9 (maxl [])))
               = failwith "Empty list"
```

Our second case in the informal reasoning wasn't quite right. It should be:

- if the list is empty then there is no maximum element – error
- if the list is not empty and the tail is empty then the head is the maximum element
- if the list is not empty and the tail is not empty we take the maximum between the head and the maximum of the rest of the list.

Which is implemented as

```
let rec maxl ls = match ls with
  | [] -> failwith "Empty list"
  | hd :: tl -> if empty tl then hd else max hd (maxl tl)
```

This works correctly. However, we can improve this program by making it more elegant and getting rid of the ugly `if` statement. In the process you will also see how the pattern system of OCaml is more expressive then just the basic recipe:

```
let rec maxl ls = match ls with
  | [] -> failwith "Empty list"
  | hd :: [] -> hd
  | hd :: tl -> max hd (maxl tl)
```

Note pattern `hd::[]`. It matches successfully if there is a head (the list is non-empty) and the tail is empty. This is one of the special cases we needed to handle.

Here is a principle of good functional programming:

No ifs. No buts.

As a matter of principle it is good practice to avoid special situations. If special situations are unavoidable, try to handle them via patterns. If patterns cannot work, only then use `if`. The reason for this is that the type system of OCaml is very good at analysing patterns at compile time and reporting possible errors. Consider this implementation of `maxl`, and OCaml's warning:

```
let rec maxl ls = match ls with
  | hd :: [] -> hd
  | hd :: tl -> max hd (maxl tl);;
      Characters 19-85:
  ...................match ls with
    | hd :: [] -> hd
    | hd :: tl -> max hd (maxl tl)..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val maxl : 'a list -> 'a = <fun>
```

What happens is that we forgot to handle the empty-list pattern. OCaml can analyse a pattern-match and see if it is *exhaustive*, i.e. all cases are covered. If some cases are not covered a *warning* is given and the missing pattern is identified below. Because this is a warning, a smaller error, the program is still executable, but it may fail.

```
# maxl [6;8;3];;
- : int = 8
# maxl [];;
Exception: Match_failure ("", 49, 18).
```

Note that the exception is different, `Match failure.` This is not a serious problem because our correct `maxl` also fails on the empty list. But you can also get useful warnings. Consider this erroneous implementation:

```
let rec maxl ls = match ls with
  | [] -> failwith "Empty list"
  | hd :: tl -> max hd (maxl tl)
  | hd :: [] -> hd;;
        Characters 103-111:
    | hd :: [] -> hd;;
      ^^^^^^^^
Warning 11: this match case is unused.
val maxl : 'a list -> 'a = <fun>
```

In this case we have *redundant* patterns which cannot be matched. Why is this? Because whenever `hd::[]` could be matched, `hd::tl` can also be matched. As OCaml tries the patterns in order, this means that the last pattern is never tried. What happens at runtime:

```
# maxl [3;1;2];;
Exception: Failure "Empty list".
```

The special pattern of the empty tail is never matched.

Compare lists. Here is a comparison of two lists:

```
let rec listeq ls1 ls2 = match (ls1, ls2) with
| [], [] -> true
| hd::tl, hd'::tl' -> hd = hd' && listeq tl tl'
| _ -> false
```

Note the underscore pattern, which matches anything: in the case of list comparison all lists of unequal lengths will end up triggering the third pattern. Note that OCaml implements equality on lists, so this function is not really needed.

```
let listeq ls1 ls2 = (ls1 = ls2)
```

or even

```
let listeq = (=)
```

The underscore pattern also can be used when we don't actually care about what the pattern is, just that it matches. Here is a function that checks the first two elements of a list exist and are equal:

```
let firsteq ls = match ls with
| hd :: hd' :: _ -> hd = hd'
| _ -> false
```

The first occurrence of the underscore ignores the tail, the second matches any list with zero or one element.

9 Case study: graph search

9.1 Graph search

Those of you who take AI already know that graph search is an important problem. From Norvig's book you know that in general a search algorithm has the following shape:

```
1  General-Search(problem,strategy) RETURN a solution, or failure
2  initialise the search tree using the initial state of problem;
3  LOOP DO
4  IF there are no candidates for expansion THEN
5  RETURN failure;
6  choose a leaf node for expansion according to strategy;
7  IF the node contains a goal state THEN
8  RETURN the corresponding solution
9  ELSE
10 expand the node and add the resulting nodes to the search tree;
11 END
```

The algorithm pseudo-code is of course vague but we can spot some key operations, indicated with italics above:

initialise there must be a way to represent our problem as a graph (line 2)

expansion there must be a way to calculate what the next candidates are (lines 4, 10)

failure in case there is no solution a failure must be reported (line 5)

strategy there must be a way to prioritise or filter the candidate solutions (line 6)

goal we must be able to check whether the goal has been reached (line 7)

add we must be able to add a candidate (line 10).

In this lecture we shall see how to implement general search in OCaml.

Let us sketch out how the main search function should look like

```
let rec search
graph     (* the search graph representing the problem *)
expand    (* how to expand a node                       *)
strategy  (* strategy for prioritising candidates      *)
=
??
```

Looking at the algorithm we see that the set of candidates, which is the one we must check whether it's empty or not, is implicit. We should have that as an explicit argument.

```
let rec search
graph     (* the search graph representing the problem *)
expand    (* how to expand a node                       *)
fringe    (* the current nodes under consideration     *)
strategy  (* strategy for prioritising candidates      *)
=
if empty fringe then failwith "No solution."
...
```

Note that the algorithm suggests we "return" failure. However, "failure" is not a value so we just fail.

```
let rec search
graph     (* the search graph representing the problem *)
expand    (* how to expand a node                       *)
fringe    (* the current nodes under consideration      *)
goal      (* specification of the solution              *)
strategy  (* strategy for prioritising candidates       *)
=
if empty fringe then failwith "No solution."
else if exists goal fringe then find goal fringe
else ...
```

When considering the potential candidates we first check if we find a solution (line 7 of algorithm) and we return it. We now realise that the *goal* should also be given as an argument.

Here I departed a little bit from the algorithm as stated. I think I would rather *first* check if I don't already have a solution, *then* look for the next possible solution. So now we do line 6 of the algorithm and choose a node. Which one should we choose?

The easiest to choose is the first one, but is it reasonable? The algorithm says to choose *according to the strategy* so it seems that we should apply the strategy now to the fringe. But the fringe which we already have was presumably the result of some strategising so the first element should be the element which we want. So we either *expand the fringe, strategise, select the head* or *select the head, expand, strategise*. The only real problem would be if we *expand* then *select the head* without strategising, because some unwanted choices might be in the head position. So we don't even need to call *exists*, just look at the head! We can therefore rewrite what we wrote just as

```
let rec search
graph    (* the representation of the problem    *)
expand   (* how to expand a node                 *)
fringe   (* the current nodes to be investigated *)
goal     (* specification of the solution        *)
strategy (* how to prioritise the fringe         *)
=
match fringe with
| [] -> failwith "No solution"
| hd :: tl ->
if goal hd then hd
else ...
```

What we need to do is expand and strategise the fringe, then "loop."

```
let rec search
graph    (* the representation of the problem    *)
expand   (* how to expand a node                 *)
fringe   (* the current nodes to be investigated *)
goal     (* specification of the solution        *)
strategy (* how to prioritise the fringe         *)
=
match fringe with
| [] -> failwith "No solution"
| hd :: tl ->
if goal hd then hd
else search graph expand (strategy tl (expand hd graph)) goal strategy
```

This is the skeleton of the program, a *generic* algorithm. To actually use it we need:

graph a problem graph

expand a way to expand a node into a fringe

goal a goal definition

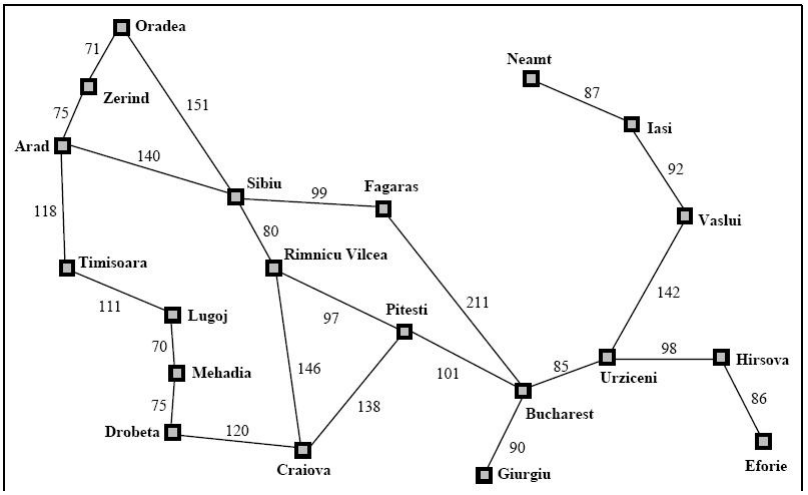
strategy a prioritisation strategy.

We can plug this program into OCaml and it is interesting to notice its *type*:

```
val search :
(* the graph:     *)    'a ->
(* the expansion: *)   ('b -> 'a -> 'c) ->
(* the fringe:    *)    'b list ->
(* the goal:      *)   ('b -> bool) ->
(* the strategy:  *)   ('b list -> 'c -> 'b list) ->
(* the solution:  *)    'b
= <fun>
```

Above, `'a` is the type of the graph, `'b` that of the node, and `'c` that of the un-strategised fringe and `'b list` that of the strategised fringe. This tells us what kind of arguments we need to provide when we construct the detailed search algorithm.

An example map you may have seen is from Norvig's book [7]:



Graph. First we need to implement the problem graph. The job is quite easy because this is actually a concrete graph. There are many (isomorphic) ways to represent a graph, but a conventional one is as a *list of edges*, where each edge is a *tuple* containing the source, destination and distance.

For example, *Sibiu to Fagaras, 99 kms* will be represented as

```
('S', 'F', 99)
```

Since all cities start with a different alphabet letter we will represent the cities by the initial, for brevity.

Then the map above is:

```
let roadmap =
[ ('A', 'Z', 75);
('A', 'S', 140);
('A', 'T', 118);
('T', 'L', 111);
('L', 'M', 70);
('M', 'D', 75);
('D', 'C', 120);
('C', 'R', 146);
```

[7] `http://aima.cs.berkeley.edu/`

```
('R', 'S', 80);
('S', 'O', 151);
('O', 'Z', 71);
('S', 'F', 99);
('F', 'B', 211);
('B', 'P', 101);
('P', 'C', 138);
('B', 'G', 90);
('B', 'U', 85);
('U', 'H', 98);
('H', 'E', 86);
('U', 'V', 142);
('V', 'I', 92);
('I', 'N', 87)]
```

Expansion. The next thing we need to construct is an expansion function, which given a city and the roadmap will return an un-strategised fringe. It makes sense to represent an un-strategised fringe as a list of cities.

```
let rec expand vertex graph =
match graph with
| [] -> []
| (v1, v2, _) :: tl ->
if v1 = vertex then (v2 :: expand vertex tl)
else if v2 = vertex then (v1 :: expand vertex tl)
else expand vertex tl
```

Points to keep in mind:

1. in this example search we will ignore distance information, hence the underscore pattern
2. the edges of this graph are undirected so any edge `(x, y, d)` means that we can get from `x` to `y` *or* from `y` to `x`.
3. we have something we can test, whether expansion works properly with the roadmap:

   ```
   # expand 'P' roadmap ;;
   - : char list = ['R'; 'B'; 'C']
   ```

   Indeed, from *Pitesti* we can get to *Ramnicu, Bucuresti, Craiova* only in one step.

The fringe. When we call the search algorithm the initial fringe will include the starting city only. The algorithm requires that this is a list, and it will be a list of cities. For example, `['B']` is a valid initial fringe.

The goal. This spells out what we search for. If it is a particular city we search for then the goal will test for equality with the city. For example, `fun x -> x = 'Z'` checks whether we found *Zerind*. But a goal can be any property of a town. For example we can look for "Northern cities", case in which the goal will be

```
fun x -> x = 'O' || x = 'N'
```

The strategy. One of the common search algorithms is *Depth-first search* (DFS) where the strategy simply is to stack the new fringe before the existing fringe.

```
let dfs oldf newf = newf @ oldf
```

9.2 Running and debugging

To recapitulate, the full program is:

```
let rec search
graph    (* the representation of the problem *)
expand   (* how to expand a node *)
fringe   (* the current nodes to be investigated *)
goal     (* specification of the solution *)
strategy (* how to prioritise the fringe *)
=
match fringe with
| [] -> failwith "No solution"
| hd :: tl ->
if goal hd then hd
else search graph expand (strategy tl (expand hd graph)) goal strategy

let roadmap =
[ ('A', 'Z', 75);
('A', 'S', 140);
('A', 'T', 118);
('T', 'L', 111);
('L', 'M', 70);
('M', 'D', 75);
('D', 'C', 120);
('C', 'R', 146);
('R', 'S', 80);
('R', 'P', 97);
('S', 'O', 151);
('O', 'Z', 71);
('S', 'F', 99);
('F', 'B', 211);
('B', 'P', 101);
('P', 'C', 138);
('B', 'G', 90);
('B', 'U', 85);
('U', 'H', 98);
('H', 'E', 86);
('U', 'V', 142);
('V', 'I', 92);
('I', 'N', 87)]

let rec expand vertex graph =
match graph with
| [] -> []
| (v1, v2, _) :: tl ->
if v1 = vertex then (v2 :: expand vertex tl)
else if v2 = vertex then (v1 :: expand vertex tl)
else expand vertex tl

let strategy oldf newf = newf @ oldf
```

We can now run our search, and lets see if we can find *Zerind* from *Bucharest*:

```
# search roadmap expand ['B'] (fun x->x='Z') strategy ;;
- : char = 'Z'
```

The search succeeded! Lets now try a failed search:

```
search roadmap expand ['Y'] (fun x->x='Z') strategy ;;
Exception: Failure "No solution".
```

Indeed the initial town starting with Y does not exist so we are starting from an impossible initial situation.

Another way to fail is to search from Y:

```
# search roadmap expand ['Z'] (fun x->x='Y') strategy ;;
^C Interrupted.
```

After waiting a few minutes I interrupted the program which became unresponsive.

What happened? The search program, unlike the previous programs is not defined using *structural* recursion but using a more general form of recursion. Note that in `search` the recursive call uses both the head and the tail. In a proper structurally recursive program the recursive call should only use the tail. This is why, although structurally recursive programs terminate in general recursive programs may *diverge*, which means never produce an answer.

Because this is a non-trivial OCaml program it is possible to get it wrong on the first attempt. It is useful to know how to *debug*. For this, you can learn how to use the OCaml debugger `ocamldebug`[8] or you can insert print statements to monitor intermediate values calculated in your program. I tend to prefer the latter.

In OCaml to print a value we use the `print_`*type* family of functions. For example,

```
# print_char 'A';;
A- : unit = ()
# print_int 123;;
123- : unit = ()
# print_string "I love FOCS";;
I love FOCS- : unit = ()
```

All the functions we have seen so far either *compute a value* or *fail*. In the case of print statements, the computed value is the trivial one, called `()`, belonging to the trivial type `unit`. The only value of the type `unit` is `()`, hence the name.

The print functions are not interesting through what they compute but through what they *do*: produce output, which you can see on the screen. These are called *imperative* functions, sometimes called *procedures* or *commands* in other languages. Sometimes they are incredibly useful (for example Java is a programming language focussing heavily on commands) but sometimes they lead to complicated low-level algorithms. Right now we will only use them for debugging.

Something that is useful to know in a search algorithm is, at any moment, what is the current fringe. We know that the fringe is a list and we can print it using an *iterator* which applies a command to each element of a list. This is pre-defined in the List module.

```
# iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

We modify the search algorithm to always print the current fringe:

```
1  let rec search
2  graph (* the representation of the problem *)
3  expand (* how to expand a node *)
4  fringe (* the current nodes to be investigated *)
5  goal (* specification of the solution *)
6  strategy (* how to prioritise the fringe *)
7  =
8  iter (fun x->print_char x) fringe;
9  print_endline ();
10 match fringe with
11 | [] -> failwith "No solution"
```

[8] `http://caml.inria.fr/pub/docs/manual-ocaml/manual030.html`

```
12 | hd :: tl ->
13 if goal hd then hd
14 else search graph expand (strategy tl (expand hd graph)) goal strategy
```

The added lines are italicised, on lines 8-9. Note the function `print_endline ()` which prints and end-of-line character to the console.

Also note a new notation, the *semicolon*. It sequences two commands and you have seen in in Java before. In OCaml, the semicolon stands for

```
# let semicolon c2 c1 = ();;
val semicolon : 'a -> 'b -> unit = <fun>
# semicolon (print_char 'b') (print_char 'a');;
ab- : unit = ()
```

So `c1; c2` is the same as `semicolon c2 c1`. A semicolon is a function which seems to ignore its arguments! This is quite funny, but the arguments are commands, which don't produce interesting values anyway, so there is not much to do with them. So the function returns the unit value `()`. However, in the process of applying the function the arguments are *evaluated* and in the course of the evaluation the commands produce their effects. Because arguments are evaluated right to left, to match the way function application associates, the first command to be executed must be put second.

For our successful search, the output is

```
# search roadmap expand ['B'] (fun x->x='Z') strategy ;;
B
FPGU
SBPGU
AROFBPGU
ZSTROFBPGU
- : char = 'Z'
```

This is useful because we can see that the algorithm behaves as expected, expanding the leading node and placing the new nodes at the top of the fringe.

For the failed search, the output is now:

```
# search roadmap expand ['Z'] (fun x->x='Y') strategy ;;
Z
AO
ZSTO
AOSTO
ZSTOSTO
AOSTOSTO
ZSTOSTOSTO
AOSTOSTOSTO
ZSTOSTOSTOSTO
AOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTOSTO
```

```
ZSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
ZSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
AOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTOSTO
^C Interrupted
```

The problem is that the search cycles between Arad, Oradea and Sibiu. This is something that you (should) know about DFS. So we need to improve it to keep track of visited states:

```
1  let rec search
2  graph    (* the representation of the problem    *)
3  expand   (* how to expand a node                 *)
4  fringe   (* the current nodes to be investigated *)
5  goal     (* specification of the solution        *)
6  strategy (* how to prioritise the fringe         *)
7  visited  (* remember visited nodes               *)
8  =
9  iter (fun x->print_char x) fringe;
10 print_newline ();
11 match fringe with
12 | [] -> failwith "No solution"
13 | hd :: tl ->
14 if goal hd then hd
15 else search graph expand
16 (strategy tl (expand hd graph) (hd::visited))
17 goal strategy (hd::visited)
```

In the generic algorithm we just track the visited nodes and we let the strategy function worry about eliminating the visited nodes in while strategising the fringe.

We have to rework our strategy function to remove visited nodes.

```
let strategy oldf newf visited= remove (newf @ oldf) visited
```

So we need a a function to *remove* all the elements of a list from another list. This is not in the library and needs to be implemented:

```
let rec remove fromls ls = match fromls with
| [] -> []
| hd::tl -> if mem hd ls then (remove tl ls) else (hd :: remove tl ls)
```

Another, more concise (but more abstract) version of the same function is:

```
let remove fromls ls = filter (fun x -> not (mem x ls)) fromls
```

Now the failed search output is

```
1  # search roadmap expand ['Z'] (fun x->x='Y') strategy [] ;;
2  Z
3  AO
4  STO
5  ROFTO
6  CPOFTO
7  DPPOFTO
8  MPPOFTO
9  LPPOFTO
10 TPPOFTO
```

```
11 PPOFO
12 BOFO
13 FGUOFO
14 GUOO
15 UOO
16 HVOO
17 EVOO
18 VOO
19 IOO
20 NOO
21 OO
22
23 Exception: Failure "No solution".
```

Note the initially empty *visited* list. Also note that a node may occur more than once in the fringe. This is not a problem, and it is the case only for nodes that have not been visited yet. As soon as such a node is visited, all the occurrences disappear from the fringe, for example on line 11 the removal of P. This is not a problem.

We are done. The final complete example is in Fig. 1. The search function (lines 3-19) is generic, the roadmap (lines 21-29) is for the particular example from Norvig's book, the associated expansion function (31-37) is specific to the roadmap, the list removal function is general purpose and could be added to a separate list library. The only thing specific to DFS is the strategy function on line 43.

```
1   open List
2
3   let rec search
4   graph    (* the representation of the problem    *)
5   expand   (* how to expand a node                 *)
6   fringe   (* the current nodes to be investigated *)
7   goal     (* specification of the solution        *)
8   strategy (* how to prioritise the fringe         *)
9   visited  (* remembe visited nodes                *)
10  =
11  iter (fun x->print_char x) fringe;
12  print_newline ();
13  match fringe with
14  | [] -> failwith "No solution"
15  | hd :: tl ->
16  if goal hd then hd
17  else search graph expand
18  (strategy tl (expand hd graph) (hd::visited))
19  goal strategy (hd::visited)
20
21  let roadmap =
22  [ ('A', 'Z', 75);  ('A', 'S', 140); ('A', 'T', 118);
23  ('T', 'L', 111); ('L', 'M', 70);  ('M', 'D', 75);
24  ('D', 'C', 120); ('C', 'R', 146); ('R', 'S', 80);
25  ('R', 'P', 97);  ('S', 'O', 151); ('O', 'Z', 71);
26  ('S', 'F', 99);  ('F', 'B', 211); ('B', 'P', 101);
27  ('P', 'C', 138); ('B', 'G', 90);  ('B', 'U', 85);
28  ('U', 'H', 98);  ('H', 'E', 86);  ('U', 'V', 142);
29  ('V', 'I', 92);  ('I', 'N', 87)]
30
31  let rec expand vertex graph =
32  match graph with
33  | [] -> []
34  | (v1, v2, _) :: tl ->
35  if v1 = vertex then (v2 :: expand vertex tl)
36  else if v2 = vertex then (v1 :: expand vertex tl)
37  else expand vertex tl
38
39  let rec remove fromls ls = match fromls with
40  | [] -> []
41  | hd::tl -> if mem hd ls then (remove tl ls) else (hd :: remove tl ls)
42
43  let strategy oldf newf visited = remove (newf @ oldf) visited
```

Figure 1: Depth-first search

10 Structural induction

It is more likely to get right a functional program than an imperative one (have a look at the two implementations of merge-sort above). More than that, in the case of functional programs we can often *prove*, using symbolic calculations, that our programs are correct, which means that they satisfy a certain property. We have already seen this in the case of isomorphisms on tuples. Now lets look at functions on lists.

Remember the functions `length` and `append`

```
let rec length ls = match ls with
| [] -> 0
| hd::tl -> 1 + length tl

let rec append ls1 ls2 = match ls1 with
| [] -> ls2
| hd::tl -> hd :: (append tl ls2)
```

Here is a property that we just know it should be true:

> The number of elements in the concatenation of two lists is always equal to the sum of the number of elements of the two lists.

In OCaml this is

```
length (append l1 l2) = length l1 + length l2
```

But we know that OCaml cannot compute symbolically, as `l1, l2` need to be assigned concrete values. We need to do it by hand!

We need to apply the function `append` to symbolic values. We have to look at the definition of the function and try each pattern separately.

Case 1: `l1 = [].` In this case we need to show that

```
length (append [] l2) = length [] + length l2 (?)
```

Lets try both sides separately.

```
length (append [] l2) = length l2
length [] + length l2 = 0 + length l2 = length l2
```

In this case they are indeed equal!

Case 2: `l1 = hd::tl.` We want to show

```
length (append (hd::tl) l2) = length (hd::tl) + length l2 (?)
```

We calculate both sides

```
length (append (hd::tl) l2) = length (hd::(append tl l2)) = 1 + length (append tl l2)
length (hd::tl) + length l2                                = 1 + length tl + length l2
```

It is quite clear that these expression will not be equal unless we can prove that

```
length (append tl l2) = length tl + length l2 (?)
```

Note that this is suspiciously similar to what we set out to prove:

```
length (append l1 l2) = length l1 + length l2 (?)
```

Are we running around in circles? No! The two are not quite similar. One of them is a statement about the *list* and the other about the *tail*! In proving things about the list we can assume the same thing to be true about the tail. So we are done!

□

This way of reasoning is called

STRUCTURAL INDUCTION

and it is very similar to recursion: we can compute something about the list assuming that we can compute the same thing about its tail. In fact recursion and induction are the same thing.

Why it is correct to reason like this requires deep mathematics. Essentially, it relies on proving mathematically that induction uniquely defines functions.

Lets try another one. From the definition of `append` we know that `append [] ls = ls`. But lets prove that appending an empty list *at the end* has the same effect `append ls [] = ls`. We need to take the two cases

Case 1: `ls = [].` In this case we need to prove

```
append [] [] = []
```

which is immediate by definition.

Case 2: `ls = hd::tl.` In this case we need to prove

```
append hd::tl [] = hd :: tl (?)
```

We compute on the left

```
append hd::tl [] = hd :: (append tl [])
```

If this is to be equal to

```
hd :: tl
```

then we need to show

```
append tl [] = tl (?)
```

Which is true by structural induction, because the main goal is:

```
append ls [] = ls (?)
```

and `tl` is the tail of `ls`.

□

10.1 Examples for structural induction

A membership testing function

```
let rec mem x = function
  | [] -> false
  | hd::tl -> (hd = x) || (mem x tl)
```

A function to remove all duplicates from a list, resulting in a list in which all elements occur exactly once:

```
let rec setify = function
  | [] -> []
  | hd::tl -> if mem hd tl then setify tl else hd::(setify tl)
```

A function that counts how many times an element occurs in a list:

```
let rec occ x = function
    | [] -> 0
    | hd::tl -> (if x=hd then 1 else 0) + occ x tl
```

10.1.1 Example

Prove that

```
mem x l = mem x (setify l)
```

Base case:

```
mem x [] = mem x (setify []) = false
```

For the inductive case, we can assume:

```
mem x tl = mem x (setify tl)
```

Proof:

```
mem x (setify hd::tl)
= mem x (if mem hd tl then setify tl else hd::(setify tl)) [Q2]
case 1) mem hd tl
   = mem x (setify tl)
   = mem x tl BY IND. HYP.
case 2) not mem hd tl
   = mem x ( hd::(setify tl))
   = (hd = x) || (mem x (setify tl)) [Q3]
   case 2.a) hd = x
        = true
        = mem x hd::tl
   case 2.b) hd <> x
        = mem x (setify tl) BY IND. HYP.
```

10.1.2 Example

Prove that

```
setify (setify l) = setify l
```

The basis of induction is

```
setify (setify []) = [] = setify []
```

For the inductive case we can assume:

```
setify (setify tl) = setify tl
```

Proof (Q1 assignment, ommitted):

10.1.3 Example

Prove that

```
if not (mem x l) then occ x l = 0
```

Base case:

```
      if not (mem x []) then occ x [] = 0
      if true then true
```

Inductive hypothesis:

```
if not (mem x tl) then occ x tl = 0
```

Proof of inductive case:

```
if not (mem x (hd::tl)) then occ x (hd::tl) = 0
= if not ( (hd=x) || (mem x tl) ) then (if x=hd then 1 else 0) + occ x tl
case 1) hd = x
     = true TRIVIAL if false then ... else true
case 2) hd <> x
     = if mem x tl then (if false then 1 else 0) + occ x tl
     = if mem x tl then occ x tl BY IND.HYP.
```

11 Search and sort

11.1 Searching and filtering

A very common operation on lists, and on data structures in general, is searching, which means finding an element with a particular property. In its simplest incarnation we may want to simply check if a list contains a given element.

```
let rec find x = function
  | [] -> false
  | hd :: tl -> x = hd || find x t

# find 3 [2; 4; 1; 3; 5; 8] ;;
- : bool = true
#  find 13 [2; 4; 1; 3; 5; 8] ;;
- : bool = false
```

We can *generalise* `find` so that we look for an element that satisfies *some property*. By *some property* we mean some function `p` which returns a `bool`, and which will be given as an argument. We can call this function `exists` becaus it is really an implementation of the existential quantifier over the list: $\exists x \in l.prop(x)$.

```
let rec exists prop = function
  | [] -> false
  | hd :: tl -> prop hd || exists prop tl
 val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

Something about OCaml: the `||` operator will not evaluate the second operator if the first one is `true`. So this function will only run until it encounters the first `hd` for which `f hd` is true.

Let's use this function to check whether a list has any negative elements.

```
# exists (fun x -> x < 0) [1; 2; 3; 4] ;;
- : bool = false
```

This is the same thing as evaluating the *predicate* $\exists x \in [1;2;3;4].x < 0$.

Or check whether a list has any elements which are even

```
# exists (fun x -> x mod 2 = 0) [1; 2; 3; 4] ;;
- : bool = true
```

This is the same thing as evaluating the *predicate* $\exists x \in [1;2;3;4].x \bmod 2 = 0$.

You can also see now why it is helpful to define anonymous functions.

We can, of course implement the old `find` function in terms of the new more general `exists`:

```
let find a l = exists (fun x -> x = a) l
```

Select. The `exists` function only tells us whether a list contains elements which satisfy a property. But it is often useful to select those elements in a separate list, and to *filter out* the elements that do not satisfy it.

```
let rec select prop = function
| [] -> []
| hd :: tl -> if prop hd then hd :: (select prop tl) else select prop tl
 val select : ('a -> bool) -> 'a list -> 'a list = <fun>
#   select (fun x -> x < 0) [1; 2; 3; 4];;
- : int list = []
#   select (fun x -> x mod 2 = 0) [1; 2; 3; 4];;
- : int list = [2; 4]
```

Above, we have just used `select` to select only the negative then only the even elements from a list. The result of select is empty if and only if there is no element satisfying the property.

These functions are tremendously useful because we can use them to test or select from lists given *any* property. We can in fact completely ignore the fact that the argument is a list, since we don't need to recurse explicitly anymore. We just need to give the property.

For example, select all the elements greater than a given value, where we want the *given value* to be also an argument. We know how to write this property:

```
let gt x y = x > y
val gt : 'a -> 'a -> bool = <fun>
         ^^^^^^^^^^^^^^^
```

The apparent problem here is that the type of gt does not seem to match to the type required by exist or select:

```
val exists : ('a -> bool) -> 'a list -> bool = <fun>
val select : ('a -> bool) -> 'a list -> 'a list = <fun>
              ^^^^^^^^^^
```

How can we get a function $a \to bool$ given a function $a \to a \to bool$? Here is where OCaml's partial function application comes in handy. All we need is to provide the second argument (y):

```
let select_gt y l = select (gt y) l
val select_gt : 'a -> 'a list -> 'a list = <fun>
# select_gt 3 [1;2;3;4;5;6];;
- : int list = [4; 5; 6]
```

Observation. Above I knew I was going to do a partial application of gt so I put y as the second argument. What if the second argument was x, but we needed to supply just y?

```
let gt' y x = x > y
```

In this case we can still do it using anonymous functions and partial application:

```
let select_gt' y l = select ((fun x y -> gt' y x) y) l
```

Also consider:

```
let leq x y = x <= y
let select_leq y l = select (leq y) l
```

11.2 Quick-sort

If we know how to select the elements greater than and smaller or equal than the head of a list we are just one step away from being able to sort the list, recursively!

```
# let rec sort = function
| [] -> []
| hd :: tl ->
    let prefix = sort (select_gt hd tl) in
    let suffix = sort (select_leq hd tl) in
    prefix @ (hd :: suffix);;
val sort : 'a list -> 'a list = <fun>
# sort [1;3;2;2;4;3;2;3;4;3;2;1];;
- : int list = [1; 1; 2; 2; 2; 2; 3; 3; 3; 3; 4; 4]
```

Parallelism. Note that the calculation of the prefix is independent of that of the suffix and could be conceivably done in parallel. However, the standard OCaml compiler will not do that for you.

11.3 Selection sort

Let us consider a different very common sorting algorithm, *selection-sort*, and its asymptotic complexity. For a change, now we will count *the number of comparison operations* rather than the number of function calls. As we discussed in the previous section, the choice of the measured *resource* is up to us. Because asymptotic complexity does away with many fixed parameters, more often than not the function ends up in the same complexity class anyway.

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

The way a list is processed by this algorithm is, informally, as follows. Each step corresponds to one step in the algorithm:

```
                      ls = [3;2;4;1;5]
Step 1, find minimum:  min = 1
Step 2, swap minimum:  1::[3;2;4;5]
Step 3, recurse     :  1::[2;3;4;5]
```

We will now implement this algorithm *naively*, with no concern for performance, and we will measure its asymptotic complexity.

We need a function to find the minimum:

```
let rec minelm = function
| [] -> failwith "Empty list"
| [x] -> x
| x::xs -> min x (minelm xs)
```

And we need a function to remove an element from a list

```
let rec remelm x = function
| [] -> []
| y::ys -> if x=y then ys else y::(remelm x ys)
```

With these two helper functions we can now put our program together:

```
let rec selsort = function
| [] -> []
| [x] -> [x]
| xs ->
    let x = minelm xs in
    let xs' = remelm x xs in
    x :: (selsort xs')
```

To convince ourselves this is correct we can run some tests, and it is especially helpful to run a test by hand.

```
 selsort [3;1;2]
= let x = minelm [3;1;2] in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let x = (min 3 (minelm[1;2])) in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let x = (min 3 (min 1 (minelm [2]))) in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let x = (min 3 (min 1 2)) in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let x = (min 3 1) in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let x = 1 in let xs' = remelm x [3;1;2] in x :: (selsort xs')
= let xs' = remelm 1 [3;1;2] in 1 :: (selsort xs')
= let xs' = (if 1=3 then [1;2] else 3::(remelm 1 [1;2])) in 1 :: (selsort xs')
```

```
= let xs' = 3::(remelm 1 [1;2]) in 1 :: (selsort xs')
= let xs' = 3::(if 1=1 then [2] else 1::(remelm 1 [2])) in 1 :: (selsort xs')
= let xs' = 3::[2] in 1 :: (selsort xs')
= 1 :: (selsort [3;2])
...
= 1::2::[3]
= [1;2;3]
```

We now determine the complexity class of the program by counting the number of comparisons, which includes the number of `min` operations.

The full mathematical details of computing complexity classes are complicated beyond the scope of this module, so we will proceed in an informal way. We need to first identify the *recurrence relations* from which we know the complexity classes. As before, lets call $T(n+1)$ the amount of resources (comparisons, here) consumed by computing the function for an argument of size $n+1$.

1. For `minelm` we have $T(0) = T(1) = 0$ as no comparisons are made for empty and singleton lists. In general, $T(n+1) = 1 + T(n)$ as each head-tail case involves one comparison and one recursive call on the tail, in the worst case. So we have a linear, $\mathcal{O}(n)$ function. Note that the best and the worst case are equal, since execution does not depend on data.

2. For `remelm` we have $T(0) = 0$ and in general $T(n+1) = 1 + T(n)$ because it involves one comparison (the = test) and recursive call on the tail. So we have a linear, $\mathcal{O}(n)$ function, in the worst case. Note that in the best case $T(n+1) = 1$, if the equality test succeeds, case in which we have a constant $\mathcal{O}(1)$ function.

3. For `selsort` we have $T(0) = T(1) = 0$ and in general $T(n+1) = n + n + T(n) = 2n + T(n)$, because of two linear calls to `remelm` and `minelm`. We can simplify the constant factors so $T(n+1) = n + T(n)$, which is a quadratic, $\mathcal{O}(n^2)$ algorithm. This is the worst case. For the best case, when `remelm` works in constant time, the recurrence relation is $T(n+1) = 1 + n + T(n)$, which is also quadratic.

   Note that in the above we considered that the size of `xs'` in `selsort xs'` is one less than the size of `xs`. This is true (and somewhat obvious) but we should be able to prove it. This is somewhat important because if the size of the list does not decrease by one in each recursive call we cannot guarantee termination (this is not a structurally recursive algorithm!).

   Note that in general it is not true that

   ```
   length ls = 1 + length (remelm y ls)
   ```

   What is different is that in the context of the `selsort` we are removing the *least* element, which is therefore guaranteed to be a *member* of the list. So we can reformulate our key property as:

   ```
   if (mem y ls) then length ls = 1+length(remelm y ls)
   Case i. ls = []
    if (mem y []) then length [] = 1+length(remelm y [])
    if false then 0 = 1
    (true)
   Case ii. ls = x::xs
    mem y ls = mem y x::xs = if x=y then true else mem y xs
    length x::xs = 1+length xs
    remelm y x::xs = if x=y then xs else remelm y xs
    length (remelm y x::xs) = length (if x=y then xs else x::remelm y xs).
   Case ii.a. x=y
    if true then 1+length xs = 1+length (if true then xs else x::remelm y xs)
    1+length xs = 1+length xs.
    (true)
   Case ii.b. x<>y
   ```

```
if (mem y xs) then 1+length xs = 1+length(x::remelm y xs), equivalently,
if (mem y xs) then length xs = 1+length(remelm y xs)
(true by structural induction)
```

11.4 Performance of quick-sort

Our program has two weaknesses:

1. It needs to traverse the list twice, first to get the smaller elements then to get the larger elements.
2. If the list is already sorted it's very inefficient because either the prefix will always be empty or the suffix will always be empty.

Here are two profiled executions of sort, one with random lists and one with pre-sorted lists of size 1,000:

```
let rec select prop = function
| [] -> (* 2000 *) []
| hd :: tl ->
    (* 22256 *) if prop hd
    then (* 11128 *) hd :: (select prop tl)
    else (* 11128 *) select prop tl

let gt x y = (* 11128 *) x > y
let select_gt y l = (* 1000 *) select (gt y) l

let leq x y  = (* 11128 *) x <= y
let select_leq y l = (* 1000 *) select (leq y) l

let rec sort = function
  | [] -> (* 1001 *) []
  | hd :: tl ->
      (* 1000 *) let prefix = sort (select_gt hd tl) in
      let suffix = sort (select_leq hd tl) in
      prefix @ (hd::suffix)

let rec gen_random_ls n =
  (* 1001 *) if n = 0 then (* 1 *) []
  else (* 1000 *) (Random.int max_int) :: (gen_random_ls (n-1))
;;
sort (gen_random_ls 1000)
;;
```

```
let rec select prop = function
| [] -> (* 2000 *) []
| hd :: tl ->
    (* 999000 *) if prop hd
    then (* 499500 *) hd :: (select prop tl)
    else (* 499500 *) select prop tl

let gt x y = (* 499500 *) x > y
let select_gt y l = (* 1000 *) select (gt y) l

let leq x y  = (* 499500 *) x <= y
let select_leq y l = (* 1000 *) select (leq y) l
```

```
let rec sort = function
  | [] -> (* 1001 *) []
  | hd :: tl ->
      (* 1000 *) let prefix = sort (select_gt hd tl) in
      let suffix = sort (select_leq hd tl) in
      prefix @ (hd::suffix)

let rec gen_sorted_ls n =
  (* 1001 *) if n = 0 then (* 1 *) []
  else (* 1000 *) n :: (gen_sorted_ls (n-1))
;;
sort (gen_sorted_ls 1000)
;;
```

You can see that the select function makes 22,256 comparison for the random list, but a whopping 999,000 comparisons for the pre-sorted list!

This algorithm is a simple functional version of the quicksort algorithm[9].

11.5 Mergesort

We need a better algorithm, to avoid the weaknesses of the algorithm above.

The `mergesort` algorithm is a *divide-and-conquer* algorithm: it breaks the main problem into separate smaller equal-sized sub-problems which are solved separately. The key aspect of divide-and-conquer is that the separate solving of the sub-problems can happen independently (and even *in parallel*), thus further speeding up the algorithm execution.

Sequentially, mergesort proceeds as follows, so that at any given step one list is split or two sorted lists are merged into a sorted list:

```
[8;7;6;5;4;3;2;1]
~~~~~~~~~~~~~~~~
[8;6;4;2] [7;5;3;1]
~~~~~~~~~
[8;4][6;2] [7;5;3;1]
~~~~~
[8] [4] [6;2] [7;5;3;1]
^^^^^^^
[4;8] [6;2] [7;5;3;1]
      ~~~~~
[4;8] [6] [2] [7;5;3;1]
      ^^^^^^^
[4;8] [2;6] [7;5;3;1]
^^^^^^^^^^^
[2;4;6;8] [7;5;3;1]
          ~~~~~~~~~
[2;4;6;8] [7;3] [5;1]
[2;4;6;8] [7] [3] [5;1]
          ^^^^^^^ ~~~~~
[2;4;6;8] [3;7] [5] [1]
                ^^^^^^^
[2;4;6;8] [3;7] [1;5]
          ^^^^^^^^^^^
[2;4;6;8] [1;3;5;7]
^^^^^^^^^^^^^^^^^^
[1;2;3;4;5;6;7;8]
```

[9] http://en.wikipedia.org/wiki/Quicksort

However, there is no reason, in principle, why the split lists cannot be processed in parallel!

```
[8;7;6;5;4;3;2;1]
[8;6;4;2] [7;5;3;1]
[8;4] [6;2] [7;3] [5;1]
[8] [4] [6] [2] [7] [3] [5] [1]
[4;8] [2;6] [3;7] [1;5]
[2;4;6;8] [1;3;5;7]
[1;2;3;4;5;6;7;8]
```

The program is:

```
let rec split = function
    | []     -> [], []
    | hd::[] -> [hd], []
    | hd1::hd2::tl ->
        let left, right = split tl in
        (hd1::left, hd2::right)

let rec merge = function
    | ls, [] -> ls
    | [], ls -> ls
    | hd1::tl1, hd2::tl2 ->
       if hd1 < hd2
       then hd1 :: merge (tl1, hd2::tl2)
       else hd2 :: merge (hd1::tl1, tl2)

let rec mergesort = function
    | []   -> []
    | [hd] -> [hd]
    | ls   ->
        let (left, right) = split ls in
        let left = mergesort left  in
        let right = mergesort right  in
        merge (left, right)
```

Here is a profiled execution, for 1,000 element random lists:

```
let rec split = function
    | []     -> (* 887 *) [], []
    | hd::[] -> (* 112 *) [hd], []
    | hd1::hd2::tl ->
        (* 4932 *) let left, right = split tl in
        (hd1::left, hd2::right)

let rec merge = function
    | ls, [] -> (* 503 *) ls
    | [], ls -> (* 496 *) ls
    | hd1::tl1, hd2::tl2 ->
       (* 8702 *) if hd1 < hd2
       then (* 4400 *) hd1 :: merge (tl1, hd2::tl2)
       else (* 4302 *) hd2 :: merge (hd1::tl1, tl2)

let rec mergesort = function
    | []   -> (* 0 *) []
    | [hd] -> (* 1000 *) [hd]
    | ls   ->
        (* 999 *) let (left, right) = split ls in
```

```
        let left = mergesort left  in
        let right = mergesort right  in
        merge (left, right)

let rec gen_random_ls n =
  (* 1001 *) if n = 0 then (* 1 *) []
  else (* 1000 *) (Random.int max_int) :: (gen_random_ls (n-1))

let rec gen_sorted_ls n =
  (* 0 *) if n = 0 then (* 0 *) []
  else (* 0 *) n :: (gen_sorted_ls (n-1))
;;
mergesort (gen_random_ls 1000)
;;
```

The execution required 8,702 comparisons (only 'merge' has any comparisons), better than the 22,256 of the previous algorithm. For pre-sorted lists:

```
let rec split = function
    | []     -> (* 1774 *) [], []
    | hd::[] -> (* 224 *) [hd], []
    | hd1::hd2::tl ->
        (* 9864 *) let left, right = split tl in
        (hd1::left, hd2::right)

let rec merge = function
    | ls, [] -> (* 1502 *) ls
    | [], ls -> (* 496 *) ls
    | hd1::tl1, hd2::tl2 ->
       (* 17679 *) if hd1 < hd2
       then (* 8445 *) hd1 :: merge (tl1, hd2::tl2)
       else (* 9234 *) hd2 :: merge (hd1::tl1, tl2)

let rec mergesort = function
    | []   -> (* 0 *) []
    | [hd] -> (* 2000 *) [hd]
    | ls   ->
        (* 1998 *) let (left, right) = split ls in
        let left = mergesort left  in
        let right = mergesort right  in
        merge (left, right)

let rec gen_random_ls n =
  (* 1001 *) if n = 0 then (* 1 *) []
  else (* 1000 *) (Random.int max_int) :: (gen_random_ls (n-1))

let rec gen_sorted_ls n =
  (* 1001 *) if n = 0 then (* 1 *) []
  else (* 1000 *) n :: (gen_sorted_ls (n-1))
;;
mergesort (gen_sorted_ls 1000)
;;
```

The number of comparisons (17,679) is larger, but much better than the almost 1M comparisons of the previous algorithm.

Parallelism. In the main function, `mergesort`, if we wrote it as

```
let rec mergesort = function
    | []   -> []
    | [hd] -> [hd]
    | ls   ->
        let (left, right) = split ls in
        let (left, right) = (mergesort left, mergesort right)  in
        merge (left, right)
```

it would suggest that the two recursive calls can be executed in parallel (however, this does not happen in the OCaml compiler, but it could happen in principle).

Observation. For comparison, here is the same algorithm working on arrays, in Java:[10]

```
public class Mergesort {
    private int[] numbers;
    private int number;
    public void sort(int[] values) {
        this.numbers = values;
        number = values.length;
        mergesort(0, number - 1);
    }

    private void mergesort(int low, int high) {
        // Check if low is smaller then high, if not then the array is sorted
        if (low < high) {
            // Get the index of the element which is in the middle
            int middle = (low + high) / 2;
            // Sort the left side of the array
            mergesort(low, middle);
            // Sort the right side of the array
            mergesort(middle + 1, high);
            // Combine them both
            merge(low, middle, high);
        }
    }

    private void merge(int low, int middle, int high) {
        // Helperarray
        int[] helper = new int[number];
        // Copy both parts into the helper array
        for (int i = low; i <= high; i++) {
            helper[i] = numbers[i];
        }
        int i = low;
        int j = middle + 1;
        int k = low;
        // Copy the smallest values from either the left or the right side back
        // to the original array
        while (i <= middle && j <= high) {
            if (helper[i] <= helper[j]) {
                numbers[k] = helper[i];
                i++;
            } else {
                numbers[k] = helper[j];
                j++;
```

[10] http://www.vogella.de/articles/JavaAlgorithmsMergesort/article.html

```
            }
            k++;
        }
        // Copy the rest of the left side of the array into the target array
        while (i <= middle) {
            numbers[k] = helper[i];
            k++;
            i++;
        }
        helper = null;
    }
}
```

11.6 Questions

1. (easy) Implement a universal quantifier `forall` such that for any predicate p, list l, `forall p l` is true if and only if all elements x of l satisfy p, i.e. `p x = true`.

2. (med) Show that for any list l and any predicate p

   ```
   select p (select p l) = select p l.
   ```

3. (med) Show that for any list l and any predicate p

   ```
   select p (select (fun x -> not p x) l) = [].
   ```

4. (med) Show that for any list l and any predicates p, q

   ```
   select p (select q l) = select (fun x -> p x && q x) l.
   ```

5. (hard) We say that two lists are equivalent if they have the same elements, i.e. for any x, `mem x l = mem x l'`. [11] If two lists are equivalent we write `l` $\sim$ `l'`. Show that for any list l, any predicate p, `l` $\sim$ `(select p l) @ (select (fun x -> not p x) l`.

6. (programming, easy) Write a function `sorted` which tests whether a list has elements in ascending order.

7. (programming, easy) Write a function `equiv` which tests whether two lists have the same elements.

8. (programming, medium) Implement `select` using only `fold_l`.

9. (programmin, hard) Implement `sort` using only `fold_l`.

[11] Here `mem` is the list membership function, the same as `find` in this section.

12 Correctness of sorting

Remember functional quick-sort:

```
# let rec sort = function
| [] -> []
| hd :: tl ->
    let prefix = sort (select_lt hd tl) in
    let suffix = sort (select_gte hd tl) in
    prefix @ (hd :: suffix);;
val sort : 'a list -> 'a list = <fun>
# sort [1;3;2;2;4;3;2;3;4;3;2;1];;
- : int list = [1; 1; 2; 2; 2; 2; 3; 3; 3; 3; 4; 4]
```

What does it mean for a sorting algorithm to be correct? If `sort l = l'` what properties does l' need to satisfy in order for it to be a *sorted* version of l? This bigger property is equivalent to two smaller properties:

1. The list l' should have the same elements as l. This can be easily formalised as follows: for any x, the membership test relative to l and l' should give the same value: `mem x l = mem x l'`. If two lists have the same elements we say that they are equivalent `l` $\sim$ `l'`. This is however not enough, because elements might repeat. If we want to keep track of repeating elements then we need to count the number of occurrences, which is more complicated. For now we will stick with the simpler version, which means that we will prove correctness of sort for lists where elemnts occur only once (*setified*).

2. The list l' should have its elements *in order*. The first simple thing we need to fix is whether it is supposed to be ascending or descending, and let's suppose we want *ascending*. This property is itself quite complex and it can be expressed in several ways. For example, if we use the nth element function we could say that `if i` $<$ `j` $<$ `length l, then nth i l < nth j l`. This is perhaps an intuitively obvious way to define orderness but it's very complicated to reason about because of the indices.

   Another alternative is to define it directly, but only look at consecutive elements:

   ```
   let rec ordered = function
   | []
   | [x] -> true
   | x::x'::tl -> (x <= x') && ordered (x'::tl)
   ```

   Empty and one-element lists are ordered, and for all other lists the head is smaller than the head of the tail, and the tail is ordered. This is also intuitively quite obvious, but if we try to work with it it will prove awkward because of the quite complicated induction scheme which involves the head and the head of the tail.

   It is best to have a simple definition, inductively speaking, even if it seems a bit less obvious:

   ```
   let rec ordered = function
   | [] -> true
   | hd::tl -> (forall (fun x -> hd <= x) tl) && (ordered tl)
   ```

   In words, a list is ordered if empty or if the head is smallest than every element in the tail, and the tail is ordered. This is a nice definition because it uses a clean structural induction scheme.

Lets now prove the correctness of (functional) quicksort. We take the two properties separately and we prove them by induction.

Proving that `l` $\sim$ `sort l`

For the base, the empty list `l = [ ]`, the property is immediate since `sort [ ] = [ ]` by definition.

For the inductive step we need to show that

```
hd::tl ~ sort (hd::tl), or equivalently
hd::tl ~ (sort (select_lt hd tl)) @ (hd::(sort (select_gte hd tl)))
             by def. of sort, or equivalently
[hd] @ tl ~ (sort (select_lt hd tl)) @ [hd] @ (sort (select_gte hd tl))
             by the definition of @
```

Here, in order to simplify we will use several properties of the interplay between @ and $\sim$ which we didn't prove, but which we can prove separately as *lemmas*:

```
(A) If l1 ~ l1' and l2 ~ l2' then l1 @ l2 ~ l1' @ l2'
(B) l ~ l1 @ l2 if and only if l ~ l2 @ l1
(C) l @ l1 ~ l @ l1' if and only if l1 ~ l1'
```

In words: if two pairs of lists have the same sets of elements then their pairwise concatenations have the same elements; if two lists have the same elements then any reorder of the lists have the same elements; if two lists are equivalent and have the same prefix, their suffix are equivalent. These are intuitively obvious. Using these properties

```
[hd] @ tl ~ (sort (select_lt hd tl)) @ [hd] @ (sort (select_gte hd tl))
             is equivalent to
[hd] @ tl ~  [hd] @ (sort (select_lt hd tl)) @ (sort (select_gte hd tl))
             by the property (B) above, which is equivalent to
tl ~  (sort (select_lt hd tl)) @ (sort (select_gte hd tl))
             by property (C).
```

This is good, because we managed to *reduce* a property expressed in terms of hd::tl in terms of tl only.

But this is not good enough because we cannot apply the IH, which is `tl` $\sim$ `sorted tl`, whereas in our property the tl is used after other functions are applied to it: `sort (select_lt hd tl)`.

This is a property that we will prove separately, and we will also strenghten it.

Proving the main Lemma. For our proof, we just look at where we got stuck and will try to prove, as a separate lemma, a stronger property that for any list `l` and element `x`

```
l ~  (sort (select_lt x l)) @ (sort (select_gte x l))
```

Observation: This stronger version of the property suggests that the *pivot* of the algorithm does not need to be the head. In fact, it is better (on average) for the pivot to be a random value withing the range of the list so that (on average) the two partitions are of equal size and the algorithm is efficient. It is often the case that attempting to formally prove an algorithm will not only find mistakes but also suggest ways of improving it!

We can go and prove this now, by induction on `l`. The base case, `l = [ ]` is obvious. LHS = `[ ]`, RHS = `[ ] @ [ ] = [ ]`, all by definition.

For the inductive step we need to prove that

```
hd::tl ~ (sort (select_lt x (hd::tl))) @ (sort (select_gte x (hd::tl)))
```

We consider two cases, `x >= hd` and `x < hd`. In the first case, by the definition of `select_lt` and `select_gte` this is equivalent to

```
hd::tl ~ (sort (select_lt x tl)) @ (sort hd::(select_gte x tl))
```

Let us focus on the suffix of the list because the prefix is already in a form that we can apply the IH.

```
sort hd::(select_gte x tl)
= (sort (select_lt  hd (select_gte x tl))) @ [hd]
@ (sort (select_gte hd (select_gte x tl)))
        by def. of sort and @.
```

The prefix and the suffix both have the same pattern: applying the select to the result of a select. This has been given as an assignment question in the previous section, so we know that:

```
select p (select q l) = select (fun x -> p x && q x) l.
select p (select p l) = select p l.
select (fun x -> false) l = [].
```

This means that, since `x >= hd`,

```
sort (select_lt  hd (select_gte x tl)) = sort [] = []
sort (select_gte hd (select_gte x tl)) = sort (select_gte x tl)
```

The RHS is therefore equal to

```
(sort (select_lt x tl)) @ [hd] @ [ ] @ (sort (select_gte x tl))
= (sort (select_lt x tl)) @ [hd] @ (sort (select_gte x tl))
  by properties of @.
```

LHS can be written as `[hd] @ tl`, so we need to prove that

```
[hd] @ tl ~ (sort (select_lt x tl)) @ [hd] @ (sort (select_gte x tl))
```

which using the lemmas developped in the failed attempt is equivalent to

```
tl ~ (sort (select_lt x tl)) @ (sort (select_gte x tl))
```

Which is true by IH.

The second case (`x < hd`) is perfectly similar.

This completes the main lemma.

The lemma can now be used immediately to complete the overall proof that `l` $\sim$ `sort l`.

Proving that the list is ordered

We now prove that the result of sort is ordered: for any list, `ordered (sort l)` is true. The proof is by induction on `l`. The base case (`l = [ ]`) is immediate by definition of sort and ordered.

For the inductive case, we need to show that

```
ordered (sort (hd::tl))
= ordered (sort (select_lt hd tl) @ hd :: sort (select_gte hd tl))
```

Here we are applying the ordered function to a concatenation of lists, so we can think how we can break this up. The idea is that a concatenation of lists is (ascendingly) ordered if the individual segments are ordered and if the head of the second list is larger than all elements in the first list (because that is the smallest element). We can take this up as a separate lemma, left as exercise:

```
(A) ordered (l @ l') =
    ordered l && (l' = [] || ordered l' && (forall (fun x -> x < hd l') l))
```

The interesting case (l' not empty) can be re-written as:

```
ordered (l@(hd::tl)) = ordered l && ordered (hd::tl) && forall (fun x -> x < hd) l
```

Applying this lemma (twice), it means that what we need to prove is:

```
ordered (sort (select_lt hd tl))
&& (ordered (hd :: sort (select_gte hd tl)) = []
    || (forall (fun x -> x < hd) (sort (select_lt hd tl))))
```

The first clause above is as simple as it can get, and it is where we should use the induction hypothesis (although for the same reason as above it doesn't quite match).

The second clause is clearly false because the list is not empty.

The third clause can be proved as a lemma, left also as an exercise: for any list l, predicate p, `forall p (select p l)`.

The main lemma. As in the previous situation, we need to take this property (the first clause) and prove it separately, in its own right and in a slightly generalised form: for any list l, element x, `ordered (sort (select_lt x l))`. The proof is by induction on l. The base case (empty list) is immediate.

For the inductive step we need to consider two cases, `x < hd vs.  x >= hd`. In the first case,

```
ordered (sort (select_lt x hd::tl)) = ordered (sort (hd :: (select_lt x tl)))
= ordered (select_lt hd (select_lt x tl) @ hd :: (select_gte hd (select_lt x tl)))
```

As in the previous section, we can simplify the nested selects:

```
= ordered ((select_lt x tl) @ hd :: [])
= ordered ((select_lt x tl) @ [hd])
= ordered (select_lt x tl) && (forall (fun y -> y < hd) (select_lt x tl))
```

The first clause is true by IH. The second clause is true by the exercise lemma.

The other case is similar.

The main lemma is then used to prove the main result.

This concludes the proof. We now know (almost) for sure that our sorting algorithm is correct for (setified) lists.

13 Number representations

13.1 Natural numbers

What is a natural number? Natural numbers are just another data type! The two *constructors* that let us create natural numbers are `Zero` and a successor operation (`Suc`): because any natural number is the successor of the successor of the successor of ... of the successor of zero. We can define this is OCaml:

```
type nat = Zero | Suc of nat;;
```

Note that the words `nat`, `Zero`, and `Suc` are all introduced in this definition; they are not OCaml keywords.

We can now define particular naturals, and name them:

```
# let zero = Zero;;
val zero : nat = Zero
# let one = Suc zero;;
val one : nat = Suc Zero
# let two = Suc one;;
val two : nat = Suc (Suc Zero)
# let three = Suc two;;
val three : nat = Suc (Suc (Suc Zero))
# let four = Suc three;;
val four : nat = Suc (Suc (Suc (Suc Zero)))
```

So instead of `Suc (Suc (Suc (Suc Zero)))` we can always say `four`. Of course, we cannot name all numbers like this, they are infinitely many!

Important: this is just a *representation* of the abstract concept of natural numbers. The same abstract concept can be represented differently:

```
type nat' = unit list;;
```

A number k is a list of k dummy values `()`.

We can name the numbers again, but using different representations:

```
# let zero' = [];;
val zero' : 'a list = []
# let one' = ()::zero';;
val one' : unit list = [()]
# let two' = ()::one';;
val two' : unit list = [(); ()]
# let three' = ()::two';;
val three' : unit list = [(); (); ()]
```

How can we tell these are representations of the same thing? These data types are *isomorphic*. I am showing below the two functions that take us from one representation to the other:

```
let rec f = function
  | Zero -> []
  | Suc n -> ()::(f n);;
- : nat -> unit list = <fun>

let rec g = function
  | [] -> Zero
  | x::xs -> Suc (g xs);;
- : 'a list -> nat = <fun>
```

In the implementation of `f : nat -> unit list` note that we are pattern-matching on the definition of the type `nat` just like we can pattern-match on the definition of pairs of lists.

We can experiment with these implementations:

```
# f three;;
- : unit list = [(); (); ()]
# g three' ;;
- : nat = Suc (Suc (Suc Zero))
# f three = three' ;;
- : bool = true
# g three' = three;;
- : bool = true
```

However, to really show that this is an isomorphism we need to *prove* that `f (g x) = x` and `g (f y) = y` for any `x, y` of the right type.

Now that we have natural numbers defined, we can implement operations, such as addition:

```
let rec add m n = match m with
    | Zero -> n
    | Suc m' -> Suc (add m' n);;
```

or make it prettier with an infix operator:

```
let rec (+) m n = match m with
    | Zero -> n
    | Suc m' -> Suc (m' + n);;
```

Overflow In Ocaml we have the built in data type `int` which gives a representation of integers, and implicitly one of natural numbers, which are just positive integers. However, we must be aware that Ocaml ints are *finite*, a limitation which strikes when trying to work with large numbers.

Here is the exponentiation function, which grows very quickly:

```
# let rec pow m n =
  if n = 0 then 1 else m * (pow m (n-1)) ;;
    val pow : int -> int -> int = <fun>
# pow 2 5;;
- : int = 32
# pow 2 10;;
- : int = 1024
# pow 2 15;;
- : int = 32768
# pow 2 20;;
- : int = 1048576
# pow 2 32;;
- : int = 0
# pow 2 40;;
- : int = 0
# pow 3 20;;
- : int = -808182895
#
```

You can see with the naked eye that the last three values are wrong.

In fact in Ocaml there is a special value denoting the maximum representable int, and you can see how computing with it leads to errors, called *overflow* errors.

```
# max_int;;
- : int = 1073741823
# max_int + 1;;
- : int = -1073741824
# max_int * max_int;;
- : int = 1
```

Arbitrary precision Numbers are conventionally represented as sequences of *digits*. In programming we can achieve arbitrary precision arithmetic (up to memory limitations, of course, which are very generous) by representing numbers as lists of digits. For example 2012 can be represented as $[2; 0; 1; 2]$. It will actually turn out to be more convenient to put the least significant digit first and the most significant last, so we use the *isomorphic* representation $[2; 1; 0; 2]$.

In Ocaml we can *define a type* using the syntax:

```
type bignat = int list
```

We can always convert from (positive) ints to bignats. What is the least significant digit of a number? The remainder of division by 10. What remains of a number when we remove the last significant digit? The result of division by 10. The conversion function therefore is:

```
# let rec int2bignat n =
  if n < 10 then [n] else (n mod 10)::(int2bignat (n/10)) ;;
val int2bignat : int -> int list = <fun>
# int2bignat 2012;;
- : int list = [2; 1; 0; 2]
```

Note that the reverse is not always possible, because a bignat might be too large for int. Also note that not any list of ints is a proper bignat, for example $[1; 23; 4]$, $[4; -4; 5]$, $[3; 5; 2; 0]$ etc.

We can print bignats nicely:

```
let print_bignat n =
let rec print_revbignat = function
[] -> ()
| hd::tl -> print_int hd; print_revbignat tl
in print_revbignat (rev n)
```

Arithmetic The existing arithmetic and comparison operations do not work on bignat, except for equality testing which is the same as that of lists of integers. We need to redefine them. Although they are not the most efficient we can use the algorithms of elementary arithmetic [12] as a starting point.

For example, the algorithm for addition is:

```
let rec bigadd n1 n2 carry =
  match n1, n2 with
  | [], [] -> if carry = 0 then [] else [1]
  | [], n2' -> bigadd [0] n2 carry
  | n1', [] -> bigadd n1 [0] carry
  | hd1::tl1, hd2::tl2 ->
      let hd' = (hd1 + hd2 + carry) mod 10 in
      let carry' = (hd1 + hd2 + carry) / 10 in
      hd' :: (bigadd tl1 tl2 carry')
```

In Ocaml we can define overloaded operators for bignats:

```
let (++) n1 n2 = bigadd n1 n2 0
# print_bignat ([1;2;3] ++ [9;9;9;9]);;
10320- : unit = ()
```

The other operators can be similarly defined.

Zero Notice that the number zero is a bit funny. We don't want bignats to start with a leading 0 because that would ruin equality, because $4 = 4$ as numbers but $[4; 0] \neq [4]$ as lists of ints. So it makes sense to define zero as the empty list

```
let zero = []
```

Otherwise our assumption of what is a legal bignat would be awkward: "a list of ints between 0 and 9 which does not end with a 0, *except if the list has only one element*".

[12] `http://en.wikipedia.org/wiki/Elementary_arithmetic`

Properties In order to prove their correctness we can show that they satisfy the properties of addition on natural numbers, for example:

For any bignat n,

```
n ++ zero = n
```

The proof is by structural induction.

For the base case note that we take $n = [\,]$.

```
bigadd [d] [] 0 = bigadd [d] [0] 0 = hd'::(bigadd [] [] carry')
                                     where hd' = d, carry' = 0
= [d].
```

The inductive case is $n = hd :: tl$, and we can assume the induction hypothesis

```
IH: bigadd tl [] 0 = tl.
```

We need to prove that

```
bigadd (hd::tl) [] 0 = hd:tl
```

We calculate the LHS from the definition:

```
bigadd (hd::tl) [] 0
= bigadd (hd::tl) [0] 0
= hd' :: (bigadd tl [] carry') where hd' = hd, carry' = 0
= hd :: (bigadd tl [] 0)
= hd::tl BY I.H.
```

13.2 Arbitrary numeral systems

A natural number can be represented as a list of digits. In the conventional practice of arithmetic we use *base 10* or *decimal*, which means we use 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. If we only use two digits it is called *binary*: 0, 1. If we use 8 digits it is called *octal*: 0, 1, 2, 3, 4, 5, 6, 7. If we use 16 digits it is calle *hex(adecimal)*: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Mathematically, the value of a number n written in the expanded version with digits $a_k, \ldots, a_0$ in some base b is

$$n = \overline{a_k \cdots a_1 a_0} = a_0 + a_1 b + a_2 b^2 + \cdots + a_k b^k = \sum_{0 \le i \le k} a_k b^k.$$

The representation and algorithms of the previous section work almost unchanged if we want to use a different base of representation:

```
type bignathex = int list

let rec int2bignathex n =
  if n < 16 then [n] else (n mod 16)::(int2bignathex (n/16))

let rec bigaddhex n1 n2 carry =
  match n1, n2 with
  | [], [] -> if carry = 0 then [] else [1]
  | [], n2' -> bigaddhex [0] n2 carry
  | n1', [] -> bigaddhex n1 [0] carry
  | hd1::tl1, hd2::tl2 ->
      let hd' = (hd1 + hd2 + carry) mod 16 in
      let carry' = (hd1 + hd2 + carry) / 16 in
      hd' :: (bigaddhex tl1 tl2 carry')

let (++) n1 n2 = bigaddhex n1 n2 0

# [10;10;10] ++ [11;11;11;17] ;;
- : int list = [5; 6; 6; 2; 1]
```

The binary representation is special because we can replace arithmetic on the digits with logical operations on the digits. For two-digit boolean addition with carry the representation is standard and can be figured out via case analysis [13].

```
type bignatbin = bool list

let rec int2bignatbin n =
  if n = 0 then [ ] else ((n mod 2)=1)::(int2bignatbin (n/2))

let rec bigaddbin n1 n2 carry =
  let xor x y = (x && not y) || (y && not x) in
  match n1, n2 with
  | [], [] -> if not carry then [] else [true]
  | [], n2' -> bigaddbin [false] n2 carry
  | n1', [] -> bigaddbin n1 [false] carry
  | hd1::tl1, hd2::tl2 ->
      let hd' = xor hd1 (xor hd2  carry) in
      let carry' = ((xor hd1 hd2) && carry) || (hd1 && hd2) in
      hd' :: (bigaddbin tl1 tl2 carry')

let (++) n1 n2 = bigaddbin n1 n2 false
```

For the purpose of testing it is convenient to write a reverse represenation function, but on the understanding that it might overflow:

```
let rec bignatbin2int = function
| [] -> 0
| hd::tl -> (if hd then 1 else 0) + 2*(bignatbin2int tl)

# bignatbin2int ((int2bignatbin 999) ++ (int2bignatbin 999));;
- : int = 1998
```

13.3 Canonical representation

The notion of (natural) number itself can be formalised in Ocaml in a way which is independent of its data-types int and bool [14]:

```
type nat = Zero | Succ of nat
```

You can see this is a recursive definition, as a natural number is either zero or the successor of a natural number. All arithmetic can be defined using this representation. Some common values would be:

```
#
let zero = Zero
let one = Succ (zero)
let two = Succ (one)
let three = Succ (two)
;;
val zero : nat = Zero
val one : nat = Succ Zero
val two : nat = Succ (Succ Zero)
val three : nat = Succ (Succ (Succ Zero))

let rec plus n1 n2 = match n1 with
| Zero -> n2
```

[13] http://en.wikipedia.org/wiki/Adder_(electronics)
[14] http://en.wikipedia.org/wiki/Peano_axioms

```
| Succ n1' -> Succ (plus n1' n2)

#
 plus one two = three;;
- : bool = true
#
```

This representation has its own induction principle, which says that if something is true for Zero and if something is true for any n is also true for Succ(n) then it is true for any natural.

We can prove various properties of addition:

zero `plus Zero n = plus n Zero = n` for any natural number n.

basis `plus Zero Zero = plus Zero Zero = Zero`.

induction
```
Assume: plus Zero n = plus n Zero = n
    Prove: plus Zero (Succ n) = plus (Succ n) Zero = Succ n
    LHS = Succ n, by def. of plus
    MID = Succ (plus n Zero), by def. of plus
    = Succ n, by ind. hyp.
```

associativity `plus m (plus n p) = plus (plus m n) p`.

zero `plus 0 (plus n p) = plus n p = plus (plus 0 n) p`, from the zero property above.

induction
```
Assume: plus m (plus n p) = plus (plus m n) p
    Prove: plus (Succ m) (plus n p) = plus (plus (Succ m) n) p
    LHS = Succ (plus m (plus n p)) by def. of plus
    RHS = plus (Succ (plus m n) p) by def. of plus
    = Succ (plus (plus m n) p) by def. of plus
    = LHS by Ind. Hyp.
```

13.4 Questions

1. **(written, hard)** Show that the types of hex digit representation `bignathex` and octal digit representation `bignatoct` are isomorphic by defining the appropriate functions and showing they are mutual inverses.

2. **(programming, easy)** Define a *subtraction* function `sub m n` for the canonical type of integers `nat`. If m is less than n the function will return Zero.

3. **(written, easy)** Prove that for the function `sub`, for any natural number `sub n n = Zero`.

4. **(programming, easy)** Define a *multiplication* function `mul m n` for the canonical type of integers `nat`.

5. **(programming, easy)** Define a *multiplication* function `dmulbignat n d` between decimal numbers n `bignat` and one-digit numbers d.

6. **(programming, medium)** Define a *multiplication* function `mulbignat m n` between decimal numbers `bignat`.

7. **(programming, medium)** Define a *division* function `ddivbignat n d` between decimal numbers n `bignat` and one-digit numbers d.

8. **(programming, hard)** Define a *division* function `ddivbignat m n` between decimal numbers.

9. **(written, medium)** Show that `plus m n = plus n m` for any m, n, canonical natural numbers.

10. **(written, medium)** Show that for canonical natural numbers `mul m (plus n p) = plus (mul m n) (mul m p)` (addition distributes over multiplication to the left). You may use the result of the previous question(s) without proof.

11. **(written, hard)** Show that for canonical numbers `mul m n = mul n m`.

12. **(written, hard)** Show that for canonical natural numbers `mul (plus m n) p = plus (mul m p) (mul n p)`.

14 Data types

14.1 Syntax of Ocaml types

In Ocaml we can give names to types. For example

```
(* PAIRS *)
type pint = int * int       (* pair of ints *)
type ppint = pint * pint    (* pair of pair of ints *)
type 'a pair = ('a * 'a)    (* pair where components have the same type *)

(* LISTS *)
type lint = int list        (* list of ints *)
type llint = lint list      (* list of list of ints *)
type 'a plist=('a*'a) list  (* list of pairs where components have the same type *)
```

We can also create *variant types* where different types can be assigned to the same variable:

```
type binary = Zero | One                          (* binary values *)
type listint = Empty | Cons of (int * listint)  (* list of ints *)
```

Note that names of types which are not variant types are 'abbreviations' so we can use either the name of the type or the definition, interchangeably:

```
# (fun p -> p) (3,4);;
- : int * int = (3, 4)
# type pint = int * int ;;
type pint = int * int
# (3,4);;
- : int * int = (3, 4)
# (fun (p:pint) -> p) (3,4);; (* some new Ocaml syntax to force a type declaration *)
- : pint = (3, 4)
```

However, names of variant types are 'definitions' and cannot be used intercheangeably:

```
# type binary = Zero | One;;
type binary = Zero | One
# type binary' = Zero | One;;
type binary' = Zero | One
# Zero;;
- : binary' = Zero
# (fun p -> p) Zero;;
- : binary' = Zero
# (fun (p:binary) -> p) Zero;;
Characters 22-26:
  (fun (p:binary) -> p) Zero;;
                        ^^^^
Error: This expression has type binary'
       but an expression was expected of type binary
```

Avoid using such definitions, they are usually wrong:

```
type nat = Zero | Succ of nat
type bin = Zero | One
# Zero;;
- : bin = Zero
```

Because the bin Zero, as a name, will hide the nat Zero. This is just like redefining a function name, the old function is still there but the name is hidden!

14.2 Handling errors through data types

It is often the case that we need to write functions that are *partial,* which means that they are only defined for some inputs. For example, the function computing the second element of a list is defined only if the list has at least two elements. If the list is empty or a singleton (one element) we don't say what should happen and we simply allow the program to *fail*:

```
let second = function
| []
| [_] -> failwith "List too short"
| x::x'::xs -> x'
```

Or just `let second = function x::x'::xs -> x'` which will give an error message but will otherwise behave similarly at run-time, i.e. failing when the list is empty or singleton.

There are two problems with this approach:

- It is harsh and it allows no recourse if an error occurs. The user of this function needs to check whether the function is correctly used before it is used, which can be expensive. It is often more efficient to attempt execution and if some assumptions are not met to return an error code, allowing the user to take remedial action after the fact.
- It is not functional! Mathematically speaking, a function should always compute a resulting value but failure is not a value. This actually makes reasoning about such functions technically more complicated, complications which we have elided.

What is an appropriate error value though? Let us examine the *type* of the function:

```
val second : 'a list -> 'a = <fun>
```

This function takes a list of *whatever* and returns an element of *whatever*. An arbitrary type `'a` does not have a distinguished special element to symbolise error.

Fortunately, OCaml gives us a way to define such a type! From any data type we can define a new data type with a special error value

```
type 'a maybe = Nothing | Just of 'a
```

The syntactic elements in this *type definition* are:

type indicates that we are defining a new type

'a indicates that the new type is parameterised by another type (is itself a polymorphic construction)

maybe is the name of the new type

= separates the name from the definition

Nothing is a *type constructor* and it is, in this case, only a special (symbolic) value

Just is another type constructor and it is, in this case, only a tag

of indicates that the tag is to be applied to some other type

'a this is the same `'a` in the definition.

The definition of the function becomes:

```
let second = function
| []
| [_] -> Nothing
| x::x'::xs -> Just x'
```

Its type now is:

```
val second : 'a list -> 'a maybe = <fun>
```

Note that the type reflects the fact that the function `may` return a value of type `'a` but it also *may* return (literally) `Nothing`.

```
second [] = Nothing
second [1] = Nothing
second [1;2;3] = Just 2
```

Note that errors must still be handled! Consider a program that *takes two lists and returns the sum of their second element.*

With the old implementation, which failed, we can simply write

```
let sumsec ls1 ls2 = second ls1 + second ls2
```

This program would fail if any of the two lists is empty or singleton, because one of the two calls to `second` would fail.

However, in the error-handling implementation this implementation no longer compiles!

```
 let sumsec ls1 ls2 = second ls1 + second ls2;;
  Characters 21-31:
  let sumsec ls1 ls2 = second ls1 + second ls2
                       ^^^^^^^^^^
Error: This expression has type 'a maybe
       but an expression was expected of type int
```

Why is that? Because addition is only defined for the type `int` and not for `maybe int`. This time around we need to *handle* the error.

```
let sumsec ls1 ls2 =
    match second ls1, second ls2 with
    | Nothing, _
    | _, Nothing -> Nothing
    | Just n1, Just n2 -> Just (n1 + n2)
```

The type and some test runs are:

```
val sumsec : int list -> int list -> int maybe = <fun>
# sumsec [] [];;
- : int maybe = Nothing
# sumsec [1;2;3] [];;
- : int maybe = Nothing
# sumsec [1;2;3] [1;2;3];;
- : int maybe = Just 4
```

In conclusion, error checking and handling is nice to have, but it will add a layer of complexity to your code. This can be made a lot nicer by using special mechanism called *monad*[15].

14.3 Integers

For integers we need an extra bit of book-keeping for the sign, which can be positive or negative (two values), so we can represent them for example as:

```
type bigint = BIZero | BIPos of bignat | BINeg of bignat
```

A slight technical problem is that zero does not have a unique representation in the presence of signs, because `BIZero <> BIPos zero <> BINeg zero` although $0 = +0 = -0$. We can either redefine equality (which is unpleasant) or we must make sure that we never use `BIPos zero`, `BINeg zero` just like we don't use `[0]` in `bignat`. These conventions which must be preserved by all the operations are called *invariants*.

If we have the right operations on `bignat`, implementing the equivalent on `bigint` is a snap. For example

[15] `http://en.wikipedia.org/wiki/Monad_(functional_programming)`

```
let mul_bi m n = match (m, n) with
| BIZero, _
| _, BIZero -> BIZero
| BIPos m', BIPos n'
| BINeg m', BINeg n' -> BIPos (mul_bn m' n')
| BINeg m', BIPos n'
| BIPos m', BINeg n' -> BIPos (mul_bn m' n')
```

Question: Is the invariant preserved, i.e. if `m,n <> BIPos zero, BINeg zero` then `mul_bi m n <> BIPos zero, BINeg zero`?

14.4 Rationals

Rational numbers (fractions) can be represented as pairs of naturals with a sign:

```
type bigfrac = BFZero | BFPos of (bignat * bignat) | BFNeg of (bignat * bignat)
```

The algorithms of elementary arithmetics can be used on fractions, but now we must bear in mind that each rational can be represented in an arbitrary number of ways. Also, we need to have at least one invariant, that the *denominator* is never zero.

We either re-implement equality, which is annoying:

```
let bf_eq m n = match m, n with
| BFZero, BFZero -> true
| BFPos (m1, m2), BFPos (n1, n2)
| BFNeg (m1, m2), BFNeg (n1, n2) -> mul_bn m1 n2 = mul_bn m2 n1
| _ -> false
```

because

$$\frac{n_1}{m_1} = \frac{n_2}{m_2} \iff n_1 m_2 = n_2 m_1$$

Another option is to use an extra invariant, so that our fractions are always *irreducible*. Then we can use the Ocaml equality but we also must reduce the fractions after each operation, which can be inefficient for very large numbers as it requires calculating the *greatest common denominator*[16].

14.5 Real numbers

Real numbers are very different than naturals, integers or rationals: they do not have a finite representation! A real number is represetable by a potentially *infinite* sequence of digits. This means they are not representable by lists, which must be finite. In fact representing them is very challenging.

A possibly infinite sequence can be represented by a function from the naturals to the sequence domain, returning the nth element in the sequence. Consider this sequence of even numbers and its representation:

```
evens = 0; 2; 4; 6; 8; 10; ...

nth evens 0 = 0
nth evens 1 = 2
nth evens 2 = 4
...
nth evens k = 2*k
```

In fact any function from the naturals can be seen as an infinite sequence of its values.

For simplicity let us work only with the real numbers in the interval $(0, 1)$, which has as many points as the entire real line[17]. The real line can be built from this using a bit of bureaucracy. And let us work with a decimal representation, for convenience.

[16] http://en.wikipedia.org/wiki/Greatest_common_divisor
[17] http://en.wikipedia.org/wiki/Cardinality_of_the_continuum

```
type unitinterval = bignat -> int
```

Because `bignat` is already implemented by Ocaml as part of `Big_int` we will use that instead, so that we don't have to define all the operations.

```
open Big_int
type unitinterval = big_int -> int
```

The value $0.5 = 0.500000000...$ for example is represented as

```
let half = fun n -> if n = 0 then 5 else 0
```

The value $1/3 = 0.3333333...$ for example is represented as

```
let third = fun n -> 3
```

It may seem that we are on our way, but serious difficulties appear. The elementary addition algorithm works from the least significant digit, and a real number does not have one.

Here is a simple argument why we cannot write a program to perform arithmetic on infinite streams of digits.

First an intuitive principle: in order for an algorithm to be useful, whenever it is asked to produce a finite approximation of the result it must require only a finite approximation of the input. A finite approximation means a finite number of digits.

Suppose that we have a function to perform such multiplication `mul x y`. Obviously, it must work correctly if `y` is a representation of 3. And obviously it must be able to produce the first digit while consulting a finite number of digits of `x`. Suppose that this finite number of digits is n and suppose that we are feeding an `x` which is $0.3\ldots3$, n times. If the $n+1$th digit is 3 or smaller then the first digit of `y` will be 9. Otherwise it will be 1. So looking at any finite n digits is not enough.

If we cannot implement multiplication it also means that we cannot implement addition, because repeat addition could implement multiplication, which we know it's impossible.

So no arithmetic on real numbers?

This is actually possible if we choose digits which overlap! Read more about it if you are interested.[18][19]

14.6 Floating point

The common alternative is to represent real numbers as an *approximation*, using a pair of ints. This definition is called *floating point* [20]. A floating point number is a pair, where the first number are the significant digits (*the significand*, also called *the mantissa*) and the second number represents the exponent, so if $x = (s, e) = s \times 10^e$. The base does not need to be 10, obviously, and on a conventional computer it is 2. In order to efficiently represent numbers with a maximum degree of precision the *invariant* of floating point is that the significand must always have the same number of digits (cannot start with a 0).

So let's say that we are working with base 10 numbers (computer works with base 2) and the significand uses 4 digits and the exponent 4 digits. Then the representation of 1 is:

$$1 =_{fp} 1000 \times 10^{-3}.$$

The common formats are:

single precision where the significand is between $\pm 2^{23}$ and the exponent between -126 and 127;

double precision where the significand is between $\pm 2^{52}$ and the exponent between -1022 and 1023;

Floating point also has certain special values:

[18] `http://en.wikipedia.org/wiki/Golden_ratio_base`
[19] `http://www.cs.bham.ac.uk/~mhe/papers/fun2011.lhs`
[20] `http://en.wikipedia.org/wiki/Floating_point`

signed zeros In floating point there are separate representations for pozitive and negative zero, but they are taken to be equal:

```
# +0. ;;
- : float = 0.
# -0. ;;
- : float = -0.
# +0. = -0. ;;
- : bool = true
```

infinities However, signed zeros produce infinities of different signs via division:

```
# 1. /. 0. ;;
- : float = infinity
# 1. /. -0. ;;
- : float = neg_infinity
```

which infinities behave as you might expect informally:

```
# infinity +. 1. ;;
- : float = infinity
# infinity -. 1. ;;
- : float = infinity
# infinity +. infinity ;;
- : float = infinity
# infinity -. infinity ;;
- : float = nan
# infinity /. infinity ;;
- : float = nan
# 1. /. infinity ;;
- : float = 0.
```

It is very important to point out that the rules of algebra no longer apply for floating point, for example $x - x = 0$ or $x/x = 1$ because of the presence of these special values.

not-a-number We also have a special value for operations that cannot be carried out, as you have seen above. This value has very poor properties because it breaks even reflexivity of equality $x \neq x$:

```
# nan = nan;;
- : bool = false
# nan <> nan;;
- : bool = true
# nan *. 0. ;;
- : float = nan
```

Because of approximation errors and special values reasoning about floating point is a huge mess. This happens even in the absence of special values.

Here is an example of failure of associativity $x + y + z \neq x + (y + z)$:

```
# let bigf = 10. ** 100. ;;
val bigf : float = 1e+100
# let bignegf = ~-.bigf;;
val bignegf : float = -1e+100
# bigf +. bignegf +. 100.;;
- : float = 100.
# bigf +. (bignegf +. 100.);;
- : float = 0.
```

Question. What do you think theses functions will eventually produce when applied to initial value 0?

```
let rec inc n = if n > 1000 then 0 else inc (n + 1)

let rec inc' x = if n > (10. ** 20.) then 0 else inc' (x +. 1.)
```

Note that even constants are not always correctly represented, since decimal notation cannot be converted to binary without some loss of precision.

```
# 1.2 +. 0.1;;
- : float = 1.3
# 1.12 +. 0.11;;
- : float = 1.2300000000000002
```

Arithmetic on floating point In arithmetic on floating point we need to make sure that the invariant is preserved, i.e. the significand has the same number of digits.

addition We first must ensure that the exponents are equal, then we can perform addition then we must re-normalise. For example if both significand and exponent are on 4 digital digits:

$$\begin{aligned} 1 + 99 &= 1000 \times 10^{-3} + 9900 \times 10^{-2} \\ &= 1000 \times 10^{-3} + 99000 \times 10^{-3} \\ &= (1000 + 99000) \times 10^{-3} \\ &= 100000 \times 10^{-3} \\ &= 1000 \times 10^{-5}. \end{aligned}$$

multiplication Here we only need to normalise at the end:

$$\begin{aligned} 7 \times 8 &= 7000 \times 10^{-3} \times 8000 \times 10^{-3} \\ &= 56000000 \times 10^{-3-3} \\ &= 5600 \times 10^{-6+4} \\ &= 5600 \times 10^{-2}. \end{aligned}$$

Note that in the execution of floating point arithmetic we need temporary resources where we can store data with higher precision (usually double) than that of the arguments and the result.

14.7 An application: logistic map

The logistic map[21] occurs in nature as an equation predicting population growth in time:

$$x_{n+1} = r x_n (1 - x_n),$$

where $0 < x_n < 1$ and represents ratio from maximum population. Its behaviour is heavily r-dependent:

- $0 < r < 1$ makes the population die off;
- $1 < r < 3$ makes the population stable around $(r - 1)/r$
- $3 < r < 3.57$ makes the population oscilate around a finite number of values
- $3.58 < r < 4$ makes the population change *chaotically*
- $4 < r$ usually breaks the equation as the values exits $[0, 1]$.

[21] `http://en.wikipedia.org/wiki/Logistic_map`

These can be nicely visualised using Wolfram Alpha[22].

This map is very sensitive to numerical errors. Lets look at three implementations:

```
let one = num_of_int 1
let a = num_of_int 4
let a' = 4.
let f x = a */ x */ (one -/ x)       (* arbitrary precision *)
let f' x = a' *. x *. (1. -. x)      (* floating point *)
let f'' x = a' *. x -. a' *. x *. x  (* floating point, expanded *)
```

And let us consider a function which iterates the application of a function

```
let rec iter f n x0 = if n = 0 then x0 else f(iter f (n-1) x0)
```

These are some results we get for $x_0 = 0.671875$.

```
# float_of_num (iter f 10 x0);;
- : float = 0.313036768961
# iter f' 10 x0';;
- : float = 0.31303676896100124
# iter f'' 10 x0';;
- : float = 0.31303676896100319
```

10 iterations is not too bad, how about 15:

```
# float_of_num (iter f' 15 x0');;
- : float = 0.0227358250013
# (iter f' 15 x0');;
- : float = 0.022735825001293637
# (iter f'' 15 x0');;
- : float = 0.022735825001313405
```

Note that the arbitrary precision implementation is starting to get very slow. Here is why:

```
# string_of_num (iter f 15 x0);;
- : string =
"365058427291961659754775973375896901316320692597136033824097748692
772617108316522957657405686978079043024159242856111907657626755166 0
718113960101729964308371544538794772608758810022743453946935090251 9
478534156613094597281771401075033985593327488892856856680665974464 2
892830507225421903971707929421530057847865258198550330984383772596 9
494232412486623865974018823007851962044832119549032171456142577226 1
408676065270955144137690871626118817671678042337929279108881316765 8
374980551945895039588006989424361888822376218286356374623638927950 6
890299961032711619091158859974970326740959846580217061502319481766 8
107567368578403389533286103059889621692417751359287422141298159359
813300203201938853856527592862832581549689486230775161798449193942 5
348109336928018137628375942270971695244536734951933159689830854966 3
...
```

This number goes on for 20 pages (78,920 digits)!

For 20, we arbitrary precision will be too slow and

```
# iter f'' 20 x0' ;;
- : float = 0.9828919791505567
# iter f'' 20 x0' ;;
- : float = 0.98289197915000637
```

For 25, we still have agreement between the two floating point implementations:

[22] http://www.wolframalpha.com/input/?i=logistic+map

```
# iter f'' 25 x0' ;;
- : float = 0.75754867350944055
# iter f' 25 x0' ;;
- : float = 0.75754867345123833
```

However, the actual result is 0.652837... and this can be calculated using arbitrary precision for reals as streams of digits (see previous section). The floating point value is completely wrong. We can see disagreement beetween them for large values of n:

```
# iter f'' 250 x0' ;;
- : float = 0.90424601171683061
# iter f' 250 x0' ;;
- : float = 0.3553051386138697
```

There is no agreement even on the first digit, and obviously they are both wrong.

In practice, floating point is used because it gives the answer fast, even though it's wrong.

Questions

In the question below you may use the List Ocaml module. These questions are both programming and written. For the written part, explain why your implementation avoids/minimises overflow and rounding errors.

1. (easy). Implement addition `add44` and multiplication `mul44` for this custom *positive* floating point data type which uses 4 decimal digits for the significant and 4 decimal digits for the exponent:

   ```
   type fp44 = NaN | Fp of (int * int * int * int * int * int * int * int)
   ```

   All the special values are mapped into `NaN`.

2. (easy). Write a function `avg2 : int -> int -> int` which calculates the average of two `int` numbers, paying attention to avoiding overflow and minimising rounding errors.

3. (medium). Write a function `sum3 : float -> float -> float -> float` which calculates the sum of three `float` numbers, paying attention to avoiding overflow and minimising rounding errors.

4. (medium). Write a function `avgl : int list -> int` which calculates the average of a list of `int` numbers, paying attention to avoiding overflow and minimising rounding errors.

5. (hard). Write a function `suml : float list -> float` which calculates the sum of a list of `float` numbers, paying attention to avoiding overflow and minimising rounding errors.

15 Introduction to complexity

Up to this point we didn't mention *efficiency* of computation but we focused on *effectiveness*, which means computing the right result eventually.

Here we will consider two alternative implementation of list reversal. We will notice, by counting steps in manual computation of some examples, that one seems to be more *efficient* than the other (it takes less writing). We also have an intuitive notion of efficienty as we can see a program taking more or less time compared to another which does the same thing, or a program using more or less memory compared to another which does the same thing. On a mobile phone we may notice another form of efficiency: an app may use the battery more or less than another. In general, efficieny is meaningful as a concept only in relation to a predefined notion of *resource* which is consumed in the process of computation, such as time, memory (*space*) or energy. But there can be any other useful resources, for example network bandwidth or anything else.

Example. Consider our two versions of reverse:

```
(* less efficient *)
let rec append ls1 ls2 = match ls1 with
  | [] -> ls2
  | hd :: tl -> hd :: (append tl ls2)

let rec rev ls = match ls with
  | [] -> []
  | hd :: tl -> append (rev tl) [hd]

(* more efficient *)
let rec revap ls sl = match ls with
| [] -> sl
| hd :: tl -> revap tl (hd :: sl)
```

Here are the example reductions:

```
1  rev [1; 2; 3] = append (rev [2; 3]) [1]
2                = append (append (rev [3]) [2]) [1]
3                = append (append (append (rev []) [3]) [2]) [1]
4                = append (append (append [] [3]) [2]) [1]
5                = append (append [3] [2]) [1]
6                = append (3 :: (append [] [2]) [1]
7                = append (3 :: [2]) [1]
8                = append ([3; 2]) [1]
9                = 3 :: (append [2] [1])
10               = 3 :: (2 :: append [] [1])
11               = 3 :: (2 :: [1])
12               = [3; 2; 1]
```

It took 12 computational steps to revert a list of 3 elements. This seems inefficient because it is!

```
1  revap [1; 2; 3] [] = revap [2; 3] (1 :: [])
2                     = revap [3] (2::(1::[]))
3                     = revap [] (3::(2::(1::[])))
4                     = (3::(2::(1::[])))
5                     = [3; 2; 1]
```

We will *instrument* the functions to count operations which we want to keep track of. We can track the use of any of all operations we are interested in, but in a functional implementation it is common to track the number of function calls. This instrumentation of the program is quite similar to the debugging instrumentation we did with the print functions. In fact we can do the same here, adding a print statement to track each function call:

```
(* less efficient *)
let rec append ls1 ls2 =
  print_string "#";
  match ls1 with
  | [] -> ls2
  | hd :: tl -> hd :: (append tl ls2)

let rec rev ls =
  print_string "#";
  match ls with
  | [] -> []
  | hd :: tl -> append (rev tl) [hd]

(* more efficient *)
let rec revap ls sl =
print_string "#";
match ls with
| [] -> sl
| hd :: tl -> revap tl (hd :: sl)

# rev [1;2;3;4;5];;
#####################- : int list = [5; 4; 3; 2; 1]
# revap [1;2;3;4;5] [];;
######- : int list = [5; 4; 3; 2; 1]
# rev [1;2;3;4;5;6;7;8;9;0];;
##################################################################- : int list = [0; 9; 8; 7; 6
# revap [1;2;3;4;5;6;7;8;9;0] [];;
###########- : int list = [0; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

Two things are worth noting here: `rev` results in more function calls than `revap` and the number of function calls generated by `rev` seems to *grow faster* with the input compared to `revap`.

Lets make a more interesting experiment. We generate and reverse lists from size 0 to size 100. The content of the lists does not matter in the two algorithms, so we generate just list of 0's.

```
  let rec genlist n = match n with
| 0 -> []
| _ -> 0 :: genlist (n-1)

let rec revto rev n = match n with
| 0 -> ()
| _ -> rev (genlist n);
    print_newline ();
    revto rev (n-1)
```

The output for the two programs are:

```
# revto rev 15;;
########################################################################################################################################
########################################################################################################################
#########################################################################################################
###########################################################################################
##############################################################################
##################################################################
#######################################################
#############################################
####################################
############################
#####################
###############
##########
######
###
- : unit = ()
# revto (fun ls -> revap ls []) 15;;
################
###############
##############
#############
############
###########
```

```
#########
########
########
#######
######
#####
####
###
##
- : unit = ()
#
```

The second one looks clearly *linear* whereas the first looks like a *parabola*, the graph of a quadratic function.

Let us sketch out structural induction proofs of how many function calls it takes to reverse a list of n elements with the two programs. For `rev` the relation between the size of the input and that of the output is

```
0 1
1 3
2 6
3 10
4 15
5 21
```

Can you guess the sequence? I don't think I can, but WolframAlpha is very good at guessing:

$$a_n = \frac{n(n+1)}{2}.$$

We use structural induction to show that:

Fact. If the size of the list `ls` is `n` then the number of calls made by `revap ls ls'` is `n+1`.

Proof. We take the two cases of `ls`:

- `ls=[]`. In this case we only have the one function call to `revap [] ls'`.
- `ls=hd::tl`. In this case we have

```
revap hd::tl ls'            (* call revap *)
    = revap tl (hd::ls')    (* n+1 calls by SI *)
```

  Note that we don't really care what the result is, just how many calls are made. And indeed we have `(n+1)+1=n+2` calls.

Fact. If the size of the list `ls` is `n` then the number of calls made by `rev ls` is `n * (n+1) / 2`.

Proof. We take the two cases of `ls`:

- `ls = []`. In this case we have only the one function call to `rev`.
- `ls = hd::tl`. In this case we have

```
rev hd::tl =                        call rev
    = append (rev tl) [hd]          n*(n+1)/2 calls, by SI
                                    because length tl = length(rev(tl)),
                                    see below the proof.
    = append ls' [hd]               n calls by append
```

  So in total we have `1+n*(n+1)/2+n=(n+1)*(n+2)/2=n*n/2+(3 n)/2+1`.

Fact. `length l = length (rev l)`, where `length = function [] -> 0 | _ ::  tl -> 1 + (length tl)`.

Proof. By structural induction:

- `ls = []`.

  ```
  RHS: length (rev []) = length [] (from def. or rev).
                       = LHS
  ```

- `ls = hd::tl`

  ```
  IH: length tl = length (rev tl)
  WTP: length hd::tl = length (rev hd::tl)
  LHS = 1 + length tl
  RHS = length (append tl [hd]) (def. of rev)
      = length (rev tl) + length [hd] (prop. of append)
      = length (rev tl) + 1 (def. length)
      = length tl + 1 (IH)
      = LHS (commutativity of +)
  ```

15.1 Profiling

OCaml (and many other languages) have a special system to instrument for efficiency: *profiling*. In the case of the functional fragment of OCaml, the profiler may track function calls, if statements and pattern matches. Other potentially interesting aspects (arithmetic operations, memory usage) cannot be so easily tracked. This is how we use profiling for our simple example (I have removed the hand-instrumentation):

```
(* less efficient *)
let rec append ls1 ls2 = match ls1 with
  | [] -> ls2
  | hd :: tl -> hd :: (append tl ls2)

let rec rev ls = match ls with
  | [] -> []
  | hd :: tl -> append (rev tl) [hd]

(* more efficient *)
let rec revap ls sl = match ls with
| [] -> sl
| hd :: tl -> revap tl (hd :: sl)

let rec genlist n = match n with
| 0 -> []
| _ -> 0 :: genlist (n-1)

;;
let l = genlist 100 in
rev l;
revap l []
;;
```

Using the profiler:

```
$ ocamlcp -P a rev.ml -o rev
File "F:\cygwin\tmp\ocamlppa540a5", line 22, characters 0-5:
Warning 10: this expression should have type unit.
$ ./rev
$ ocamlprof rev.ml
```

```
(* less efficient *)
let rec append ls1 ls2 = (* 10100 *) match ls1 with
  | [] -> (* 200 *) ls2
  | hd :: tl -> (* 9900 *) hd :: (append tl ls2)

let rec rev ls = (* 202 *) match ls with
  | [] -> (* 2 *) []
  | hd :: tl -> (* 200 *) append (rev tl) [hd]

(* more efficient *)
let rec revap ls sl = (* 101 *) match ls with
| [] -> (* 1 *) sl
| hd :: tl -> (* 100 *) revap tl (hd :: sl)

let rec genlist n = (* 202 *) match n with
| 0 -> (* 2 *) []
| _ -> (* 200 *) 0 :: genlist (n-1)

;;
let l = genlist 100 in
rev l;
revap l []
;;
```

Note how comments of the form `(* 202 *)` have been inserted at certain points in the program. They show how many time that execution point has been reached for reversing a list of 100 elements.

15.2 Tail recursion

Consider two implementations of a function that sums all the elements in a list:

```
let rec sum = function
  | [] -> 0
  | hd::tl -> hd + (sum tl)

sum [1;2;3] = 6

let rec sum' acc = function
  | [] -> acc
  | hd::tl -> sum' (acc+hd) tl

sum' 0 [1;2;3] = 6
```

These functions are equivalent both as result and as complexity (number of function calls). But here is how the step-by-step evaluation proceeds for the two equivalent functions

```
1 sum [1;2;3]
2 = 1 + sum [2;3]
3 = 1 + (2 + sum [3])
4 = 1 + (2 + (3 + sum []))
5 = 1 + (2 + (3 + 0))
6 = 1 + (2 + 3)
7 = 1 + 5
8 = 6
```

versus

```
1 sum' 0 [1;2;3]
2 = sum' (0+1) [2;3]
```

```
= sum' 1 [2;3]
= sum' (1+2) [3]
= sum' 3 [3]
= sum' (3+3) []
= sum' 6 []
= 6
```

Both functions use 8 elementary steps of computation but `sum` needs to create the intermediate expression `1 + (2 + (3 + 0))`, whereas `sum'` works in *constant space*.

Tail-recursive functions are much more efficient but they are uglier because of the accumulator. Consider these two functions:

```
let rec sumn n = if n = 0 then 0 else n + sumn (n-1)
let rec sumn' acc n = if n = 0 then acc else sumn' (n+acc) (n-1)

# sumn 500000;;
Stack overflow during evaluation (looping recursion?).
# sumn' 0 500000;;
- : int = 446198416
```

Observation. A tail-recursive function is a function in which the recursive call is the last function call that happens in the body of the function. Superficially it may seem that this is a tail-recursive call:

```
let rec sum = function
  | [] -> 0
  | hd::tl -> hd + (sum tl)
                   ^^^^^^^^
```

but this is only because + is an infix operator. The function is actually:

```
let rec sum = function
  | [] -> 0
  | hd::tl -> add hd (sum tl)
              ^^^
```

so the last call is to *add*.

In fact there is a certain similarity between tail-recursion and imperative programming. Consider how you would write the sum function imperatively:

```
s = 0;
for i = 0 to n do
    s = s + a[i];
done;
return s
```

The local variables (sum) become arguments to the function, assignment becomes 'let' and the return statement is the base case:

```
let rec sum s =
    | a::tl ->
        let s = s + a in
        sum s tl
    | [] -> s
```

16 Arrays and imperative programming

This section assumes you are already familiar with the basics of imperative programming (e.g. from Java). We will first look at OCaml syntax for imperative constructs.

assignable variables Functional variables cannot be modified, but assignable variables can. The *type* of assignable variables of type `'a` in Ocaml is `ref 'a` where `'a` is (almost) any type.

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# let j = ref i;;
val j : int ref ref = {contents = {contents = 1}}
# let l = ref [1;2;3];;
val l : int list ref = {contents = [1; 2; 3]}
# let f = ref (fun x -> x + x);;
val f : (int -> int) ref = {contents = <fun>}
# let id = ref (fun x -> x);;
val id : ('_a -> '_a) ref = {contents = <fun>}
```

Note the `'_a` type: it means "unknown type" rather than "any type". The difference will be seen in action shortly.

assignment and dereferencing Assignable variables (*references*) can be passed around like any other functional variables but they can also be *read from* ("dereferenced") or *changed* ("assigned to"). The syntax is as follows:

```
(* Assign to a reference its value plus 1 *)
# i := !i + 1;;
- : unit = ()
(* The updated reference and its contents *)
# i;;
- : int ref = {contents = 2}
```

The next example shows the difference between unknown and polymorphic types. The first time we use the reference to a function (id) the type becomes known: it is int. Once it becomes known it is fixed in place and cannot be subsequently changed, unlike that of a polymorphic function.

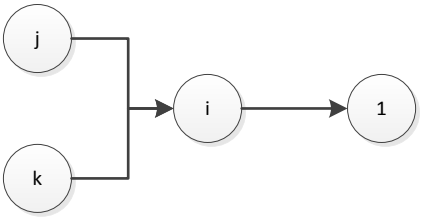
```
# (!id) 3;;
- : int = 3
# (!id) 'a';;
Characters 6-9:
  (!id) 'a';;
        ^^^
Error: This expression has type char but an expression was expected of type
         int
#
```

An important phenomenon in imperative programming is that of *aliasing*, two variables referencing the same memory location. Changing the value of one also changes the value reached from the other:

```
# let i = ref 0;;
val i : int ref = {contents = 0}
# let j = ref i;;
val j : int ref ref = {contents = {contents = 0}}
# let k = ref i;;
```

```
val k : int ref ref = {contents = {contents = 0}}
# i := 1;;
- : unit = ()
# !j;;
- : int ref = {contents = 1}
# !k;;
- : int ref = {contents = 1}
```

The references are structured like in this diagram:



This situation makes reasoning about imperative programs extremely difficult.

arrays Arrays (or vectors) collect a fixed number of elements of the same type. They can be created in two ways:

```
# let a = [| 1; 2; 3; 4; 5 |];;
```

```
val a : int array = [|1; 2; 3; 4; 5|]
# let b = Array.create 10 0;;
val b : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Dereferencing and assigning to arrays has a special syntax:

```
# a.(1) <- a.(1) + a.(2) + a.(3);;
- : unit = ()
# a;;
- : int array = [|1; 9; 3; 4; 5|]
```

For processing arrays it is important that we know their size:

```
# Array.length a;;
- : int = 5
```

loops The main iterative control structure is the loop, which can be a for-loop or a while-loop. Here is printing the integers from 1 to 10 using both structures:

```
# for i = 1 to 10 do
  print_int i;
  print_newline ();
done;;
1
2
3
4
5
6
7
8
9
10
- : unit = ()
# let i = ref 1 in
while !i <= 10 do
  print_int !i;
  print_newline ();
  i := !i + 1;
done;;
1
2
3
4
5
6
7
8
9
10
- : unit = ()
```

16.1 Searching and sorting

The main advantage of imperative programming is that it can process data in place, with no need for copying (much like tail-recursion). The main disadvantage of imperative programming is that it's fiddly and easy to get wrong because of working with integer indices and because of aliasing.

As on any data structure, very common operations are searching and sorting. We will re-implement the code of Section 11 imperatively, using arrays.

```
# let find x a =
  let i = ref 0 in
  let n = Array.length a in
  let found = ref false in
  while not !found && !i < n do
    if a.(!i) = x then found := true;
    i := !i + 1;
  done;
  !found;;
val find : 'a -> 'a array -> bool = <fun>
# find 3 a;;
- : bool = true
# find 9 a;;
- : bool = true
# find 34 a;;
- : bool = false
```

For comparison, this was the functional version:

```
let rec find x = function
| [] -> false
| hd::tl -> x = hd || find x tl
```

The imperative version is more complicated (in fact we will not attempt to reason about its correctness) but it uses a constant amount of memory, whereas the functional version is simpler but it uses an amount of memory proportional with the size of the list.

The find function can be easily generalised to the 'exists' function (left as an exercise).

The 'select' function is inherently functional, because it returns a new list. In imperative programming what we want to avoid is using too much memory, so we will slightly change the spec of the 'select' function so that it will change the way the argument array is organised: it will put all the elements that satisfy the predicate at the beginning and all the other elements at the end, in the same array and it will return the index of the pivot element (the first element that doesn't satisfy the predicate). And we will call this function, therefore 'partition'.

```
# let partition a p =
  let i = ref 0 in
  let pivot = ref (Array.length a) in
  while !i < !pivot do
    if not (p (a.(!i))) then begin
      pivot := !pivot - 1;
      swap a !i !pivot
    end else
      i := !i + 1;
  done;
  !pivot;;
val partition : 'a array -> ('a -> bool) -> int = <fun>
# a;;
- : int array = [|1; 9; 3; 4; 5|]
# partition a (fun x -> false);;
- : int = 0
# a;;
- : int array = [|9; 3; 4; 5; 1|]
# partition a (fun x -> x > 4) ;;
- : int = 2
# a;;
- : int array = [|9; 5; 4; 1; 3|]
```

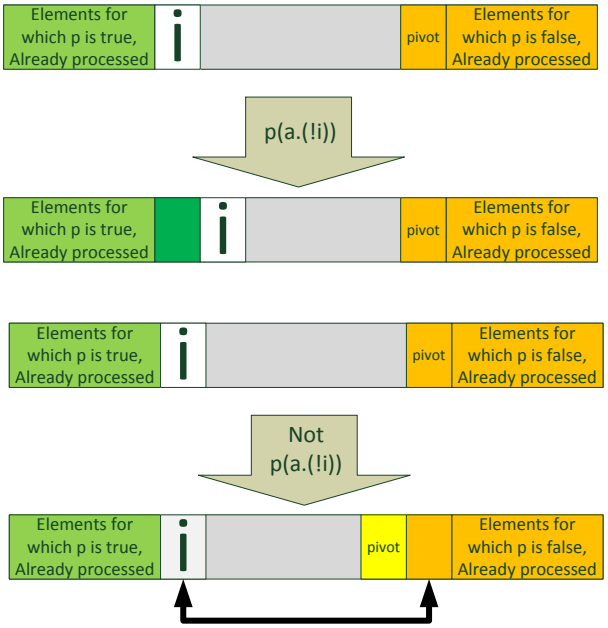


Figure 2: Partitioning an array with predicate p

'Swap' does that to two locations in the array:

```
let swap a i j =
let tmp = a.(i) in
a.(i) <- a.(j);
a.(j) <- tmp
```

The function is best illustrated diagrammatically in Figure 2. This function is clearly more subtle than 'select', but also much more memory-efficient.

If we know how to partition an array then sorting is just one step away, functionally. Imperatively, it is two-steps away. The first step is to slightly generalize partition to work on an array segment rather than a whole array:

```
let partition a p m n =
  let i = ref m in
  let pivot = ref n in
  while !i < !pivot do
```

```
    if not (p (a.(!i))) then begin
      pivot := !pivot - 1;
      swap a !i !pivot
    end else
      i := !i + 1;
  done;
  !pivot
```

Now we are ready to implement in-place quicksort (with some debug printing in place):

```
let rec qs a m n =
  print_string "Sorting : ";
  Array.iter print_int a;
  print_string "\nfrom "; print_int m;
  print_string " to "; print_int n;
  print_newline ();
  let pv = a.(m) in
  let pivot = partition a (fun x -> x < pv) m n in
  if pivot - m > 1 then qs a m pivot;
  if n - pivot > 1 then qs a pivot n

let quicksort a = qs a 0 (Array.length a)
# quicksort a;;
Sorting : 918273645
from 0 to 9
Sorting : 518273649
from 0 to 8
Sorting : 413276859
from 0 to 4
Sorting : 213476859
from 0 to 3
Sorting : 132476859
from 1 to 3
Sorting : 123476859
from 4 to 8
Sorting : 123456879
from 4 to 6
Sorting : 123465879
from 4 to 6
Sorting : 123456879
from 6 to 8
- : unit = ()
```

Beware! Note that this is correct only for set-like arrays (all elements are unique). If elements are repeating this program might not terminate. Because it is tail-recursive it will not bust the stack but it will just run forever! Writing a version that works for repeated elements is left as an exercise.

Questions (only programming)

- (**Easy**) Implement the 'exists' function over arrays. For example:

  ```
  exits (fun x -> x = 7) [|1; 4; 6; 2; 8; 7; 9 |] = true
  exits (fun x -> x > 10) [|1; 4; 6; 2; 8; 7; 9 |] = false
  ```

- (**Easy**) Implement the 'forall' function over arrays. For example:

```
forall (fun x -> x = 7) [|1; 4; 6; 2; 8; 7; 9 |] = false
forall (fun x -> x < 10) [|1; 4; 6; 2; 8; 7; 9 |] = true
```

- (**Easy**) Implement a 'fold' operation overa an array. For example:

  ```
  fold (fun x y -> max x y) [|1; 4; 6; 2; 8; 7; 9 |] = 9
  ```

- (**Medium**) Implement a version of imperative quicksort which correctly sorts arrays with duplicate values.
- (**Hard**) Implement an imperative version of mergesort (see Sec. 11.5) which sorts an array in place.

17 Binary trees as a data types

The `maybe 'a` data type is perhaps one of the simplest data types you can have. These types are extremely useful in programming. They can, for example, allow you to have functions that sometimes return integers and sometimes string by defining a data type such as

```
type intstring = Integer of int | String of string
```

The consistent and systematic use of the `Integer` and `String` tags ensures that no runtime errors are possible by confusing an integer for a string or vice versa.

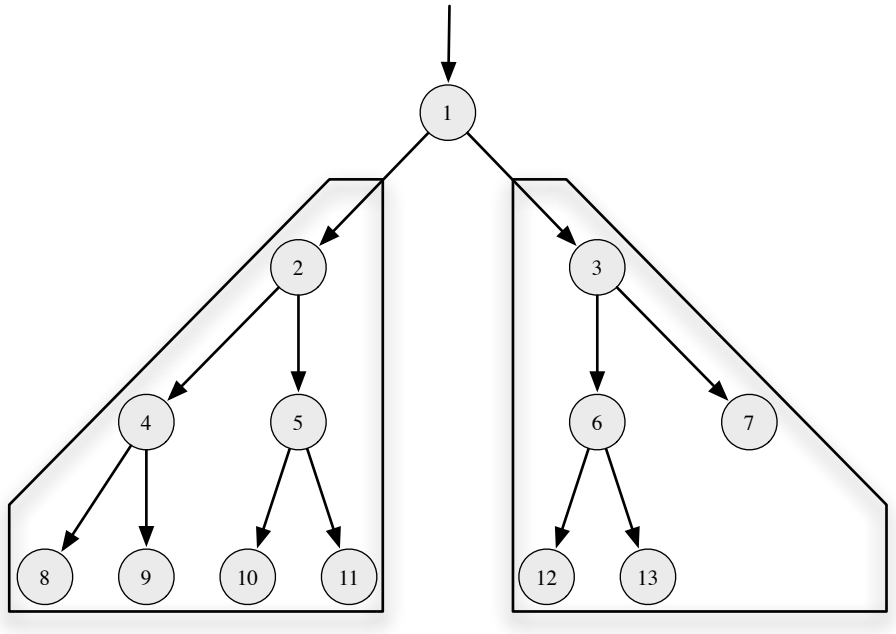
```
# Integer 5;;
- : intstring = Integer 5
# String "asdf";;
- : intstring = String "asdf"
# Integer "asfd";;
Characters 8-14:
  Integer "asfd";;
          ^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

Also, the type system, through pattern matching ensures that a function always considers all the cases of its input.

```
# let sumsec ls1 ls2 =
    match second ls1, second ls2 with Just n1, Just n2 -> Just (n1 + n2) ;;
  Characters 26-94:
      match second ls1, second ls2 with Just n1, Just n2 -> Just (n1 + n2) ;;
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(Nothing, _)
val sumsec : int list -> int list -> int maybe = <fun>
```

This is very nice.

However, the most useful application of data types is in the definition of tree-like structures. Consider, for example, binary trees:



A *tree* is either empty, or a *node* and the branching of a *pair of trees*. We can write this as a (recursively defined) data-type:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

We can represent the tree in the diagram as:

```
let t' = Node (1,
     Node (2,
          Node (4,
               Node (8, Empty, Empty),
               Node (9, Empty, Empty)),
          Node (5,
               Node (10, Empty, Empty),
               Node (11, Empty, Empty))),
     Node (3,
          Node (6,
               Node (12, Empty, Empty),
               Node (13, Empty, Empty)),
          Node (7, Empty, Empty)))
```

Note that this can be immediately generalised to trees with three nodes (ternary) and so on.

Programs on trees, and on data types in general, are written in a very similar way to programs written for lists: we use the definition of the data type as a the general scheme providing the cases that must be handled for the program.

Some basic tree programs are discussed below.

Counting the number of nodes The number of nodes in an empty tree is zero, and in a non-empty tree it is one plus the number of nodes on the left plus the number of nodes on the right:

```
let rec count = function
| Empty -> 0
| Node (_, l, r) -> 1 + count l + count r
```

An example trace of this function:

```
count Node (1, Node (2, Empty, Empty), Node (3, Empty, Empty)) =
1 + count (Node (2, Empty, Empty)) + count (Node (3, Empty, Empty)) =
1 + 1 + count Empty + count Empty + 1 + count Empty + count Empty =
1 + 1 + 0 + 0 + 1 + 0 + 0 =
3.
```

Counting the longest path The longest path in the empty tree is zero. In a non-empty tree it is one plus the maximum between the longest path on the left and the longest path on the right:

```
let rec depth = function
| Empty -> 0
| Node (_, l, r) -> 1 + max (depth l) (depth r)
```

An example trace of this function:

```
depth Node (1, Node (2, Empty, Empty), Empty) =
1 + max (depth (Node (2, Empty, Empty)), depth Empty) =
1 + max (1 + max (depth Empty, depth Empty), depth Empty) =
1 + max (1 + max (0, 0), 0) =
1 + 1 =
2
```

17.1 Tree traversals

Another essential operation on trees is the *traversal,* which is the enumeration of all its elements in a list of the same type. Traversal of a tree can happen in several ways, some of the more common being:

pre-order first list the node, then its children on the left, then its children on the right

in-order first the children on the left, then the node, then the children on the right

post-order first the children on the left, then the children on the right, then the node

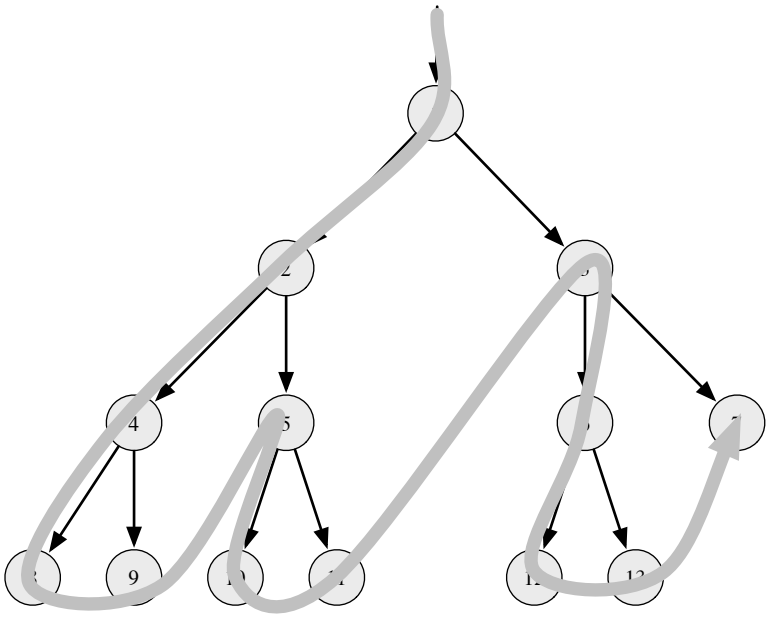
This corresponds to three distinct functions:

```
let rec preorder = function
| Empty -> []
| Node (x, l, r) -> x :: (preorder l) @ (preorder r)

let rec inorder = function
| Empty -> []
| Node (x, l, r) -> (inorder l) @ ([x] @ (inorder r))

let rec postorder = function
| Empty -> []
| Node (x, l, r) -> (postorder l) @ (postorder r) @ [x]
```

For the example tree we have:

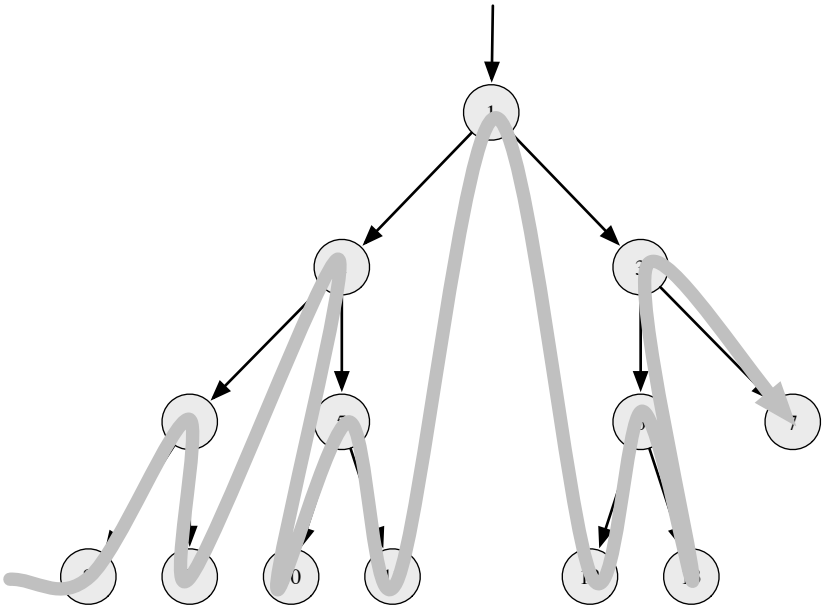
```
# preorder t';;
- : int list = [1; 2; 4; 8; 9; 5; 10; 11; 3; 6; 12; 13; 7]
```

Diagrammatically,



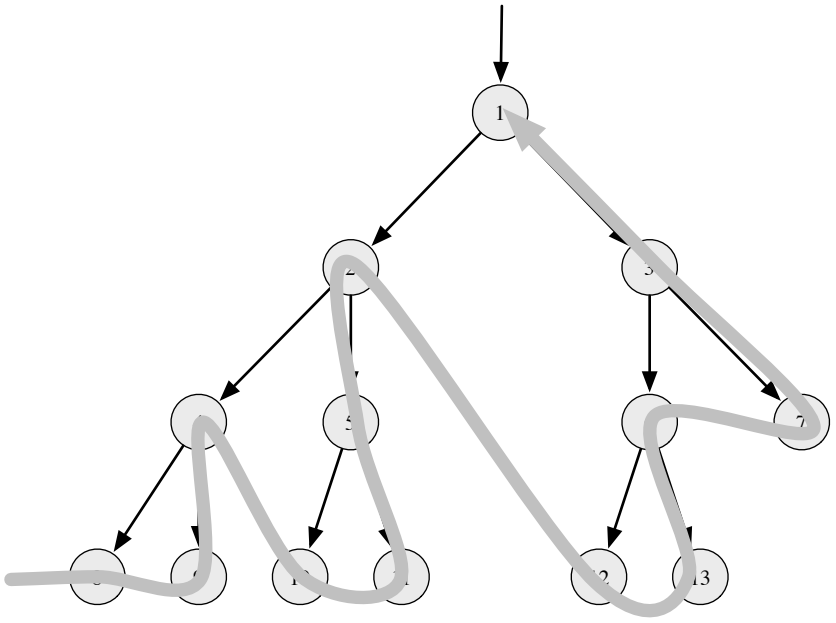
```
# inorder t';;
- : int list = [8; 4; 9; 2; 10; 5; 11; 1; 12; 6; 13; 3; 7]
```

Diagrammatically,



```
# postorder t';;
- : int list = [8; 9; 4; 10; 11; 5; 2; 12; 13; 6; 7; 3; 1]
```

Diagrammatically,



17.1.1 Complexity of traversal

Trees are a data structure that leads to efficient computation in the average case. Unlike a list, which can be split in two sub-lists in *linear* time, a tree can be split in two sub-trees in *constant* time. In order for this computational advantage to materialize, however, it is important for the trees to be *balanced,* that is each sub-tree to have roughly equal numbers of nodes. The opposite situation is of an *inbalanced tree,* where most of the nodes are in one sub-tree. The very worst situation is when the tree is *degenerate,* which is essentially a list, with one of the sub-trees of any node being empty.

We know that the complexity of the append operation is linear, so the worst-case complexity of each of the traversal functions, in the case of the degenerate tree, is described by the recurrence relation

$$T(n+1) = 1 + T(n) + T(0) + n$$

which is *quadratic*. The quadratic behaviour is down to the list append being linear and, in the worst case, the tree being possibly imbalanced.

The efficient, linear version of the traversal uses an extra accumulator argument:

```
let rec preord vs = function
| Empty -> vs
| Node (x, l, r) ->
       let prelist = preord vs r in
       x :: preord prelist l

let rec inord vs = function
| Empty -> vs
| Node (x, l, r) ->
        let inlist = inord vs r in
        inord (x::inlist) l

let rec postord vs = function
| Empty -> vs
| Node (x, l, r) ->
        let postlist = postord (x::vs) r in
        postord postlist l
```

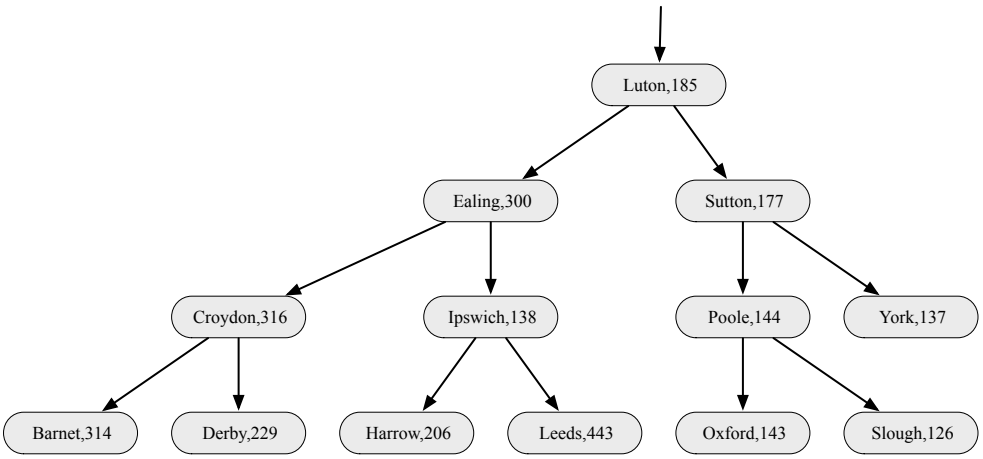
The reason these functions are more efficient is because the list is computed incrementally both on desceding along the subtrees and on returning up to the parent node. See the lecture slides for a diagrammatic step-by-step representation.

17.2 Dictionaries as binary search trees

A dictionary is a data type that manages a collection of pairs of *keys* and *values*. Keys only appear once, and they can be *searched*. The value associated with a key can be *retrieved*. One of the typical efficient implementations of dictionaries is as a particular kind of binary trees called *binary search trees* (BST).

In a BST, at any node, all the keys situated to in the left sub-tree of a node are smaller than the key at the node and all the keys situated in the right sub-tree are greater than the key at the node. This makes searching for a key fast, $\mathcal{O}(n \log n)$ if the tree is *balanced*. In a balanced tree, at any node, the depth of the left sub-tree and the depth of the right sub-tree do not differ by more than one. Note that if the tree is degenerate, i.e. a list, then search performance is no better than that of a list.

Below we have an example tree where the key is a string representing the name of a city and the value is a number representing its population (in thousands)



In OCaml we can define the dictionary data type building on the definition of the binary tree:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

by filling out the definition of the entry:

```
type ('k, 'v) entry = ('k * 'v)
type ('k, 'v) dict = (('k, 'v) entry) tree
```

This can also be done directly, as

```
type ('k, 'v) dict = ('k * 'v) tree
```

A tree such as the one in the example would have type `(string, int) dict`.

Note. In OCaml, unlike Java, the *names* of types do not matter, so the two definitions above are perfectly equivalent. If we defined, for example

```
type ('a, 'b) pair1 = 'a * 'b
type ('a, 'b) pair2 = 'a * 'b
```

then any function that accepts a `pair1` as argument can take a `pair2` and vice-versa. In Java, the names of classes is important so two classes

```
public class Pair1 { public Object o1, o2; }
public class Pair2 { public Object o1, o2; }
```

are distinct, and methods which work on a `Pair1` do not work on a `Pair2`, and vice-versa, even though the classes are essentially the same.

The OCaml representation for the example tree is:

```
let st =
  Node ( ("Luton", 185),
         Node ( ("Ealing", 300),
                Node ( ("Croydon", 316),
                       Node ( ("Barnet", 314), Empty, Empty),
                       Node ( ("Derby", 229), Empty, Empty)),
                Node (("Ipswich", 138),
                       Node (("Harrow", 206), Empty, Empty),
                       Node (("Leeds", 443), Empty, Empty))),
         Node (("Sutton", 177),
                Node (("Poole", 144),
                       Node (("Oxford", 143), Empty, Empty),
                       Node (("Slough", 126), Empty, Empty)),
                Node (("York", 137), Empty, Empty)))
```

Observation: In order for a dictionary to be correctly implemented it is essential that the underlying tree is a proper search tree, which means the keys to the left are always smaller than the keys to the right. However, there is nothing in the type system which would actually enforce this! This is an *assumption* or an *invariant* which needs to be enforced and preserved by the various operations on trees, because checking it at run-time is expensive (linear). Note that there are advanced programming languages (e.g. Agda) or systems (e.g. Coq) in which complex invariants can be formulated and checked at compile-time, but not in OCaml (or in most other current programming languages).

17.2.1 Search

Search is efficient because the ordering on the keys allows us to only search one the sub-trees. The searching operation is part of the action of retrieving the value associated with a key

```
let rec get k = function
| Empty -> Nothing
| Node ((k', v), l, r) ->
       if k = k' then Just v
       else if k < k' then get k l
       else get k r

val get : 'a -> ('a * 'b) tree -> 'b maybe = <fun>
```

Some examples:

```
# get "Poole" st;;
- : int maybe = Just 144
# get "Leeds" st;;
- : int maybe = Just 443
# get "Bucharest" st;;
- : int maybe = Nothing
```

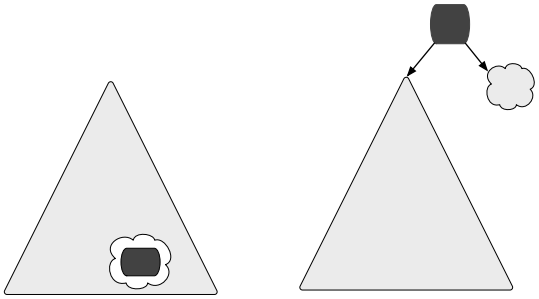
For a balance tree the number of calls needed to retrieve the value of a key is logarithmic, which is very efficient.

Example. We illustrate BST search (parallel version) in the lecture.

17.3 Search tree manipulation

17.3.1 Insertion

In a binary tree a new element can be inserted, without changing the shape of the tree, instead of any of the `Empty` leaves. It can also be inserted at the very top as a new root.



In the case of a search tree the choice of the insertion spot is decided by the value of the key, so that the tree remains a search tree (for correctness) and no re-shuffling of data is required (for efficiency):

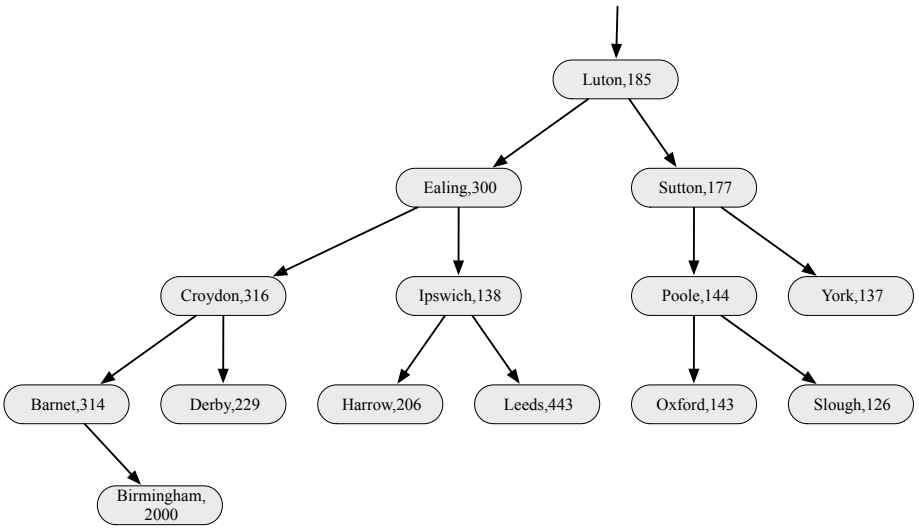
- if the tree is Empty we create a new node with the key, value pair
- if we are at a given node
  - if the keys are the same we have an error, since keys must be unique
  - if the key in the node is greater than the new key we insert on the left
  - otherwise we insert on the right

```
let rec insert k v = function
| Empty -> Node ((k, v), Empty, Empty)
| Node ((k', v'), l, r) ->
       if k = k' then failwith "Duplicate key"
       else if k < k' then Node ((k', v'), insert k v l, r)
       else Node ((k', v'), l, insert k v r)
```

Example:

```
# insert "Birmingham" 2000 st;;
- : (string * int) tree =
Node (("Luton", 185),
 Node (("Ealing", 300),
  Node (("Croydon", 316),
   Node (("Barnet", 314), Empty, Node (("Birmingham", 2000), Empty, Empty)),
   Node (("Derby", 229), Empty, Empty)),
  Node (("Ipswich", 138), Node (("Harrow", 206), Empty, Empty),
   Node (("Leeds", 443), Empty, Empty))),
 Node (("Sutton", 177),
  Node (("Poole", 144), Node (("Oxford", 143), Empty, Empty),
   Node (("Slough", 126), Empty, Empty)),
  Node (("York", 137), Empty, Empty)))
```

which corresponds to the tree



17.3.2 Deletion

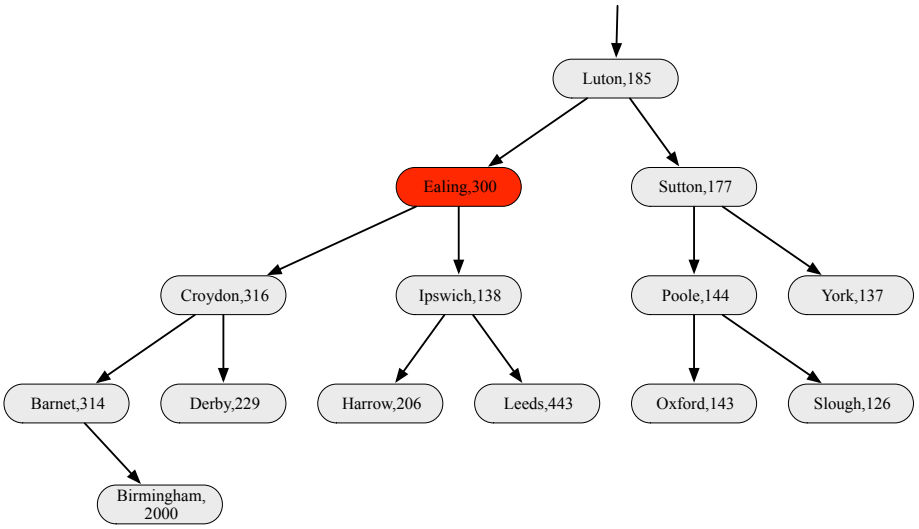
The converse operation of inserting an element is removing an element. Removing from a node with no children is easy as it simply consists of replacing the node with an empty node. Removing a node with one child is also easy because it simply consists of replacing the node with its child. Removing a node with two children requires some re-arrangement of the tree:

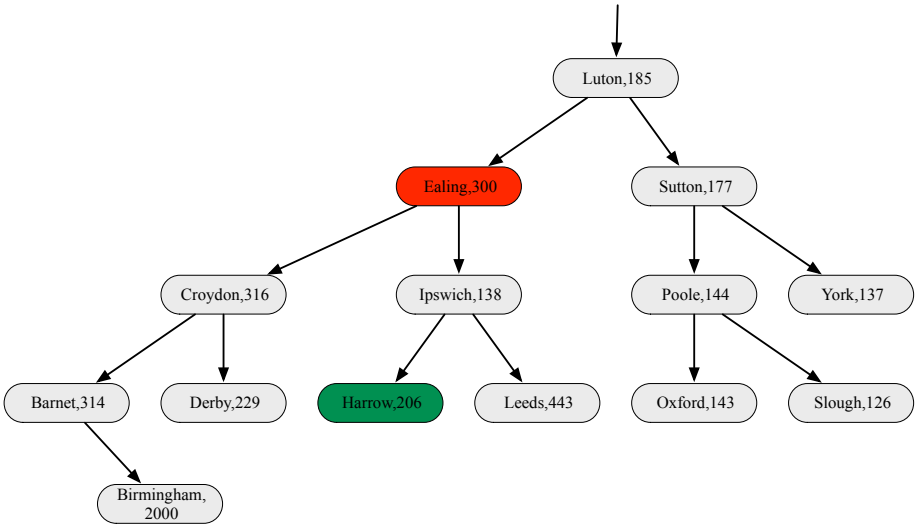
- we find the entry e with the "next" key in the tree
- we replace the node entry with it
- we remove e's node

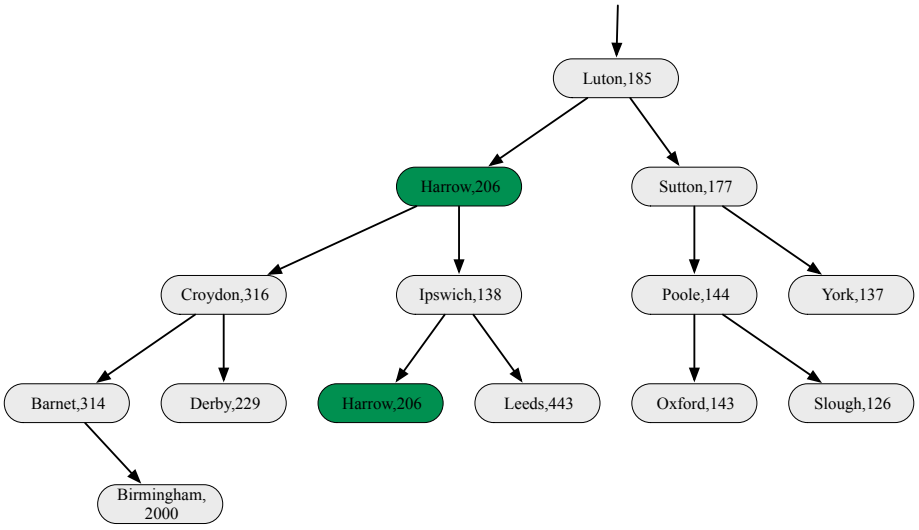
To find e we need to first go right in the tree (to find a *greater* key) then always left (to find the smallest such key). This means that, necessarily, this key has either no children or one child (to the right), so it is easy to remove it.

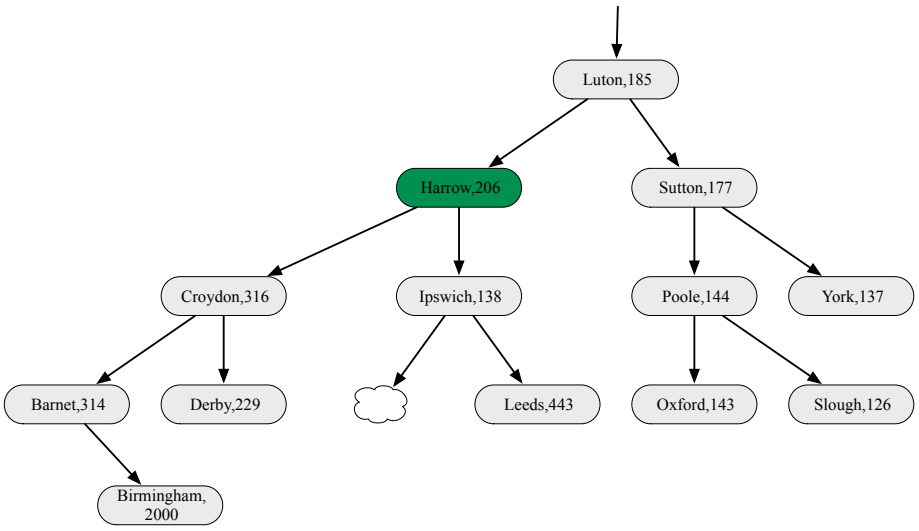
```
let rec first_inorder = function
  | Empty -> failwith "Empty tree"
  | Node (a, Empty, _) -> a
  | Node (_, l, _) -> first_inorder l

let rec remove k = function
  | Empty -> Empty
  | Node ((k', v), l, r) ->
      if k < k' then Node ((k', v), remove k l, r)
      else if k > k' then Node ((k', v), l, remove k r)
      else if l = Empty then r
      else if r = Empty then l
      else let (k'', v'') = first_inorder r in
      Node ((k'', v''), l, remove k'' r)
```

The operation on an example tree is shown in the following sequence of diagrams:


Luton,185
Ealing,300
Sutton,177
Croydon,316
Ipswich,138
Poole,144
York,137
Barnet,314
Derby,229
Harrow,206
Leeds,443
Oxford,143
Slough,126
Birmingham, 2000


Luton,185
Ealing,300
Sutton,177
Croydon,316
Ipswich,138
Poole,144
York,137
Barnet,314
Derby,229
Harrow,206
Leeds,443
Oxford,143
Slough,126
Birmingham, 2000


Luton,185
Harrow,206
Sutton,177
Croydon,316
Ipswich,138
Poole,144
York,137
Barnet,314
Derby,229
Harrow,206
Leeds,443
Oxford,143
Slough,126
Birmingham, 2000



Note that this is not the most efficient implementation of this operation, as it requires traversing the right sub-tree twice. A more efficient implementation which avoids this is left as an exercise.

Sorting Because insertion in a BST is (average case) logarithmic and in-order traversal is linear, the combination of the two can give us a simple and (average) efficient (quasi-linear) sorting algorithm.

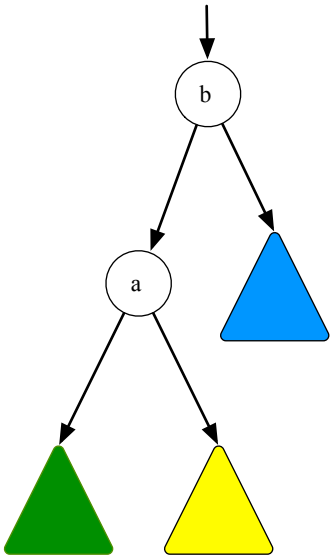
However, the worst-case complexity of the algorithm is quadratic, if the input list is pre-sorted (because the BST is degenerate).

17.3.3 Balancing

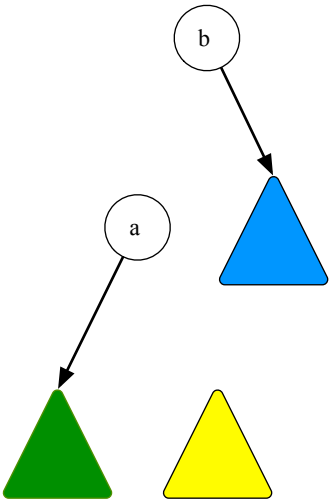
A tree is imbalanced if the depths of any of its sub-trees differ by more than one. Tree algorithms (search) have good performance only if the trees are balanced. Therefore, an important operation is *tree balancing*, which means obtaining a (more) balanced tree from a given tree.

One operation that plays a key role in the process of rebalancing is the so-called *tree rotation*, which increases the depth of one sub-tree and decreases the depth of the other, while preserving the BST nature of the tree.

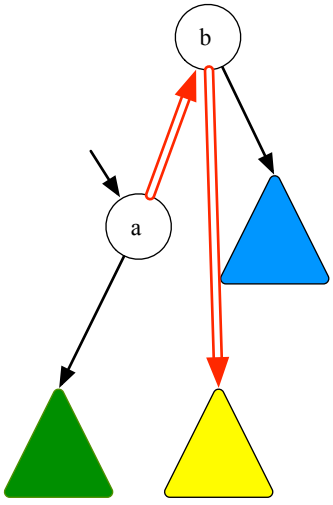
Diagrammatically, a rotation to the right looks like this:



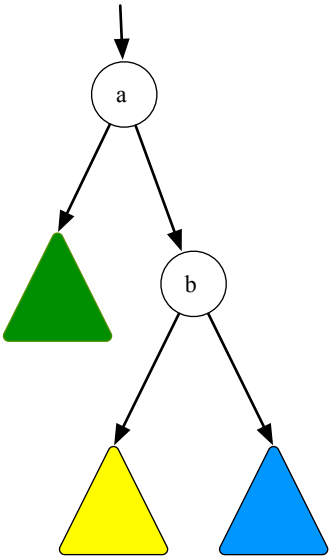
(1)



(2)

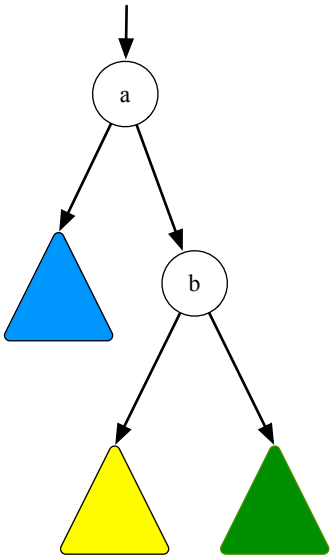


(3)

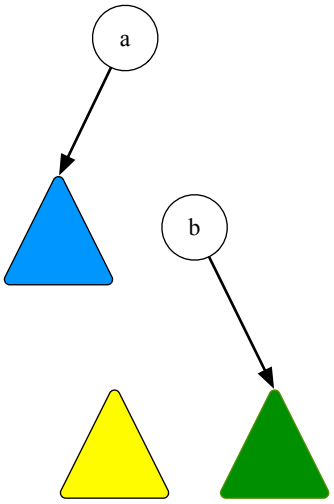


(4)

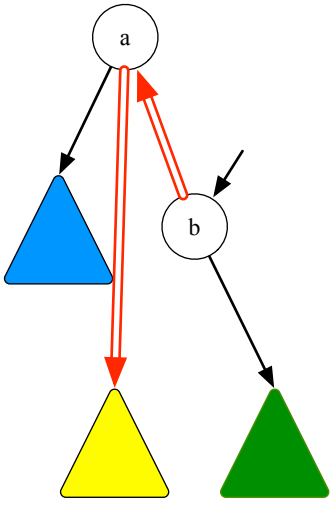
Diagrammatically, a rotation to the left looks like this:



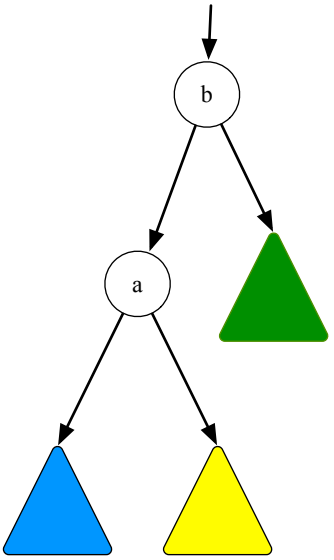
(1)



(2)



(3)



(4)

Note how in both cases a BST remains a proper BST and the three sub-trees do not need to be changed. Also note that the two rotations are inverses to each other, so a left rotation followed by a right rotations produces the same tree (and a right rotations followed by a left).

The implementation is (following the notation from the diagrams):

```
let rotright = function
  | Empty -> Empty
  | Node (b, l, blue) ->
      match l with
      | Empty -> failwith "Cannot rotate right"
      | Node (a, green, yellow) ->
  Node (a, green, Node (b, yellow, blue))

let rotleft = function
  | Empty -> Empty
  | Node (a, blue, r) ->
      match r with
      | Empty -> failwith "Cannot rotate left"
      | Node (b, yellow, green) ->
  Node (b, Node (a, blue, yellow), green)
```

Note that both these operations work in constant time.

Balancing a tree is then simply a matter of checking the depth of sub-trees and rotating until it becomes balanced!

Note that there are self-balancing implementations of BSTs, such as *AVL trees*[23] or *red-black trees*[24] where the insertion operation is combined with rebalancing, for maximum efficiency.

This is a good idea because in a balanced tree an insertion operation can either preserve the balance (if it is placed in a sub-tree which is smaller by 1, resulting in a perfectly balanced node) or create an imbalance which can be repaired by a single rotation – which acts in constant time. So the overhead of dynamically maintaining the balance can be negligible!

17.4 Expression evaluation

One of the nicest applications of trees is in expression evaluation. This is mainly because the most popular formalism for language representation is the tree. In OCaml we can use the type system to represent precisely the structure of the expression language then evaluation can proceed in a straightforward fashion on the tree.

17.4.1 Logic

Lets consider a simple language for logic consisting of boolean constants (true, false) and the standard connectives (conjunction, disjunction, implication, negation).

This can be easily implemented as an OCaml data type:

```
type prop =
    True
  | False
  | And of prop * prop
  | Or of prop * prop
  | Imply of prop * prop
  | Not of prop
```

Note that `True, False` are special values and the other connectives are represented as tree nodes. The node structure respects the arity of the connector, so that `And` must always take two propositions but `Not` must always take one.

An expression such as $(t \wedge f) \rightarrow (t \vee f)$ is represented as

```
Imply (And (True, False), Or (True, False))
```

[23] `http://en.wikipedia.org/wiki/AVL_tree`

[24] `http://en.wikipedia.org/wiki/Red-black_tree`

The evaluation of the expression is a direct representation of the rules of propositional logic:

```
let rec eval = function
  | True -> true
  | False -> false
  | And (p, q) -> eval p && eval q
  | Or (p, q) -> eval p || eval q
  | Imply (p, q) -> (not (eval p)) || eval q
  | Not p -> not (eval p)
```

17.4.2 Arithmetic

A simple language of arithmetic can be similarly defined:

```
type arith_exp =
  | Const of int
  | Add of arith_exp * arith_exp
  | Sub of arith_exp * arith_exp
  | Mul of arith_exp * arith_exp
  | Div of arith_exp * arith_exp
  | Neg of arith_exp
```

An expression such as $(1 + 2) * 4$ is written as

```
Mul (Add (Const 1, Const 2), Const 4)
```

The evaluation is similarly straightforward:

```
let rec eval_arith = function
  | Const k -> k
  | Add (k, k') -> eval_arith k + eval_arith k'
  | Sub (k, k') -> eval_arith k - eval_arith k'
  | Mul (k, k') -> eval_arith k * eval_arith k'
  | Div (k, k') -> eval_arith k / eval_arith k'
  | Neg k -> - eval_arith k
```

17.5 Arithmetic-logic expressions

Arithmetic-logic expressions are expressions which take integer arguments and produce logical values, for example comparison operators. Building on the definitions before, we can add these immediately:

```
type arithlog_exp =
  | Lt of arith_exp * arith_exp
  | Gt of arith_exp * arith_exp
  | Eq of arith_exp * arith_exp

let rec eval_arithlog = function
  | Lt (a, a') -> eval_arith a < eval_arith a'
  | Gt (a, a') -> eval_arith a > eval_arith a'
  | Eq (a, a') -> eval_arith a = eval_arith a'
```

We can then incorporate arithmetic-logic expressions into our language of propositions by adding a new case to the type definition and to the evaluation:

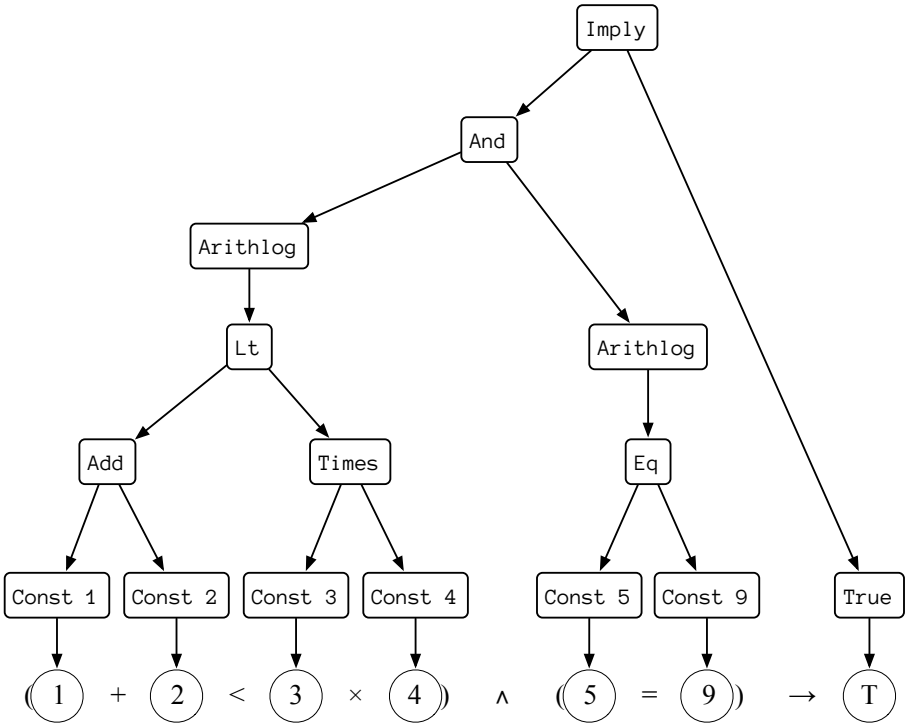
```
type prop =
  ...
  | Arithlog of arithlog_exp

let rec eval = function
  ...
  | Arithlog e -> eval_arithlog e
```

We can now evaluate complex arithmetic-logic expressions such as

$$(1 + 2 < 3 \times 4) \wedge (5 = 9) \rightarrow true$$

```
# let p = Imply (
  And (
    Arithlog (Lt (
      (Add (
        (Const 1),
        (Const 2))),
      (Mul (
        (Const 3),
        (Const 4))))),
    Arithlog (Eq (
      (Const 5),
      (Const 9)))),
  True) in
eval p;;
- : bool = true
```

The tree for this expression is:



18 Queues

A *queue* is a data structures in which the elements are kept in order and (efficient) operations for adding at one end and removing at the other are provided. Some other (auxiliary) operations could also be provided. This makes the queue a "first-in-first-out" data structure so that the first element to be added is the first to be removed and the last to be added is the last to be removed. Contrast this with a list, which is a "last-in-first-out" data structure: the last element to be added (the head) is the first element that can be removed.

Queues are extremely useful in the implementation of operating systems (to store events waiting to be processed), in simulations (particularly operations research and transportation) and in tree traversal (breadth-first search).

The *signature* we want for a queue should include, at least, an *enqueue* operation which takes an element and a queue and produces a new queue (with the new element in the last position), and a *dequeue* operation which takes a queue and produces the last element a a new queue (with the last element removed). The first one is the *constructor* and the second one the *destructor* of the queue:

```
enq : 'a * 'a queue -> 'a queue
deq : 'a queue -> ('a * 'a queue)
```

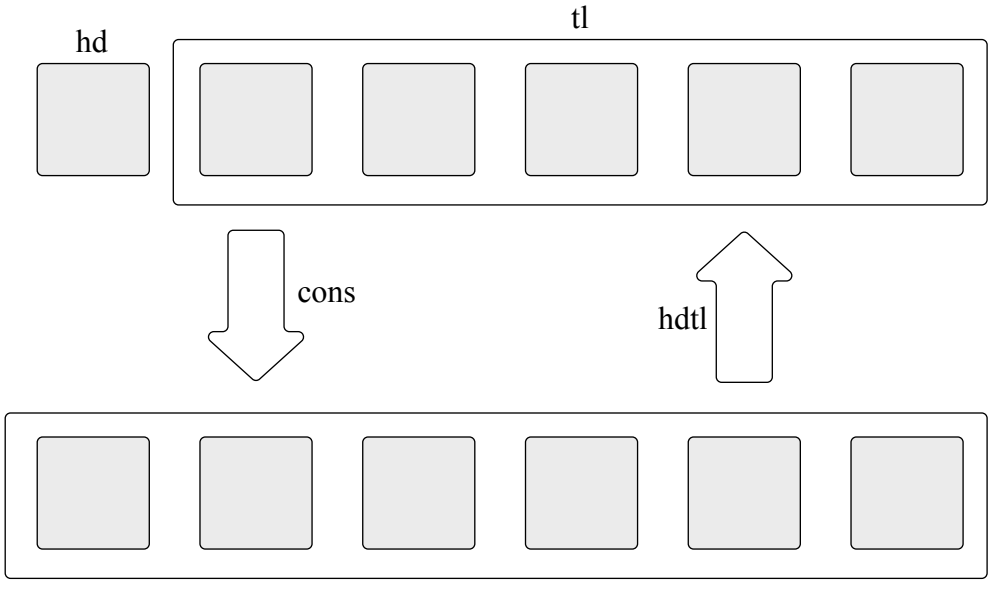
Compare this with the constructor and the destructor for a list:

```
let cons (x, xs) = x::xs : 'a * 'a list -> 'a list
let hdtl l = (hd l, tl l) : 'a list -> 'a * 'a list = <fun>
```

The destructor is what makes the pattern-matching work. Note that in both cases the destructor may fail because of an empty queue or list, so the destructors are *partial*

In the case of the list the constructor and the destructor are (almost) inverses:

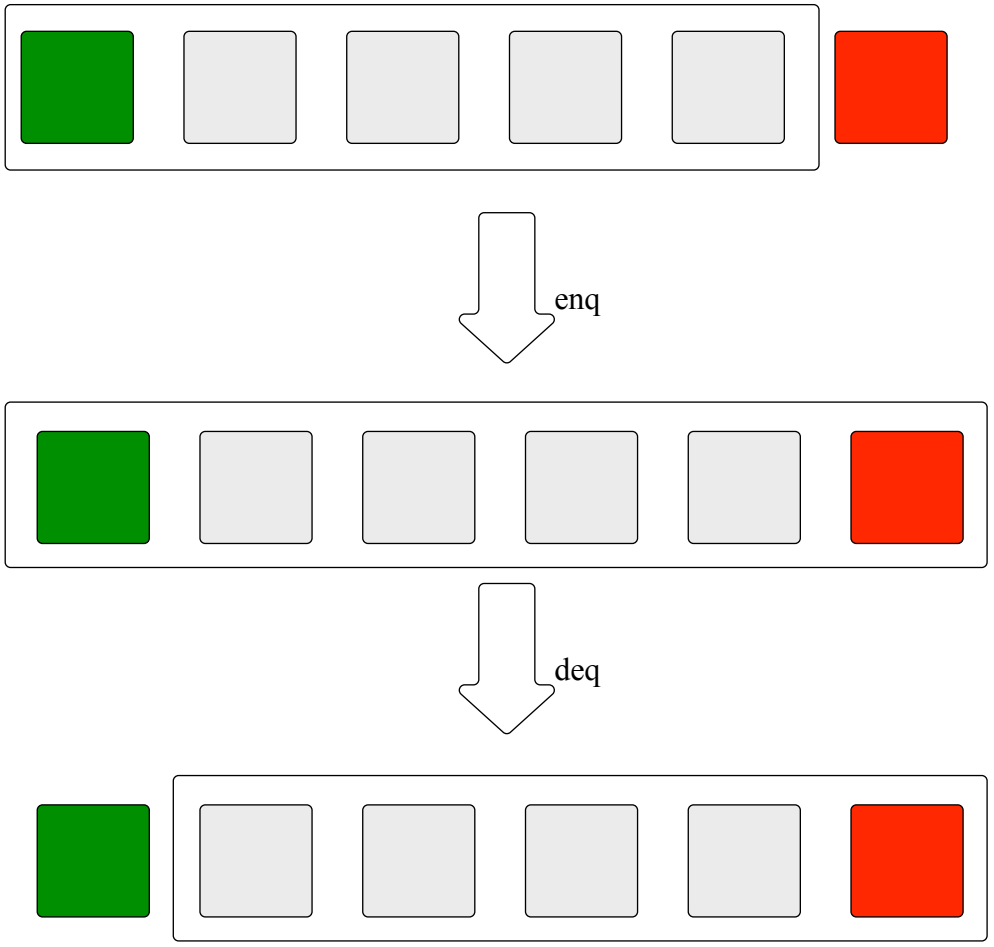
```
hdtl (cons (x, xs))
= hdtl x::xs
= (hd x::xs, tl x::xs)
= (x, xs)

cons (hdtl x::xs)
= cons (x, xs)
= x::xs
```



Whereas for queues the constructor and destructor are not inverses at all:

```
deq (enq (x, xs)) <> (x, xs)
enq (deq xs) <> xs
```

because the enqueuing and dequeuing happen at different ends of the queue.



Lists and trees are *inductive data structures*: they can be defined inductively and can be processed recursively, as we have seen in the previous lectures. Queues, however, are not, and we can see that throught the fact that the constructor and destructor do not cancel each other out. This means that queues do not lend themselves easily to implementation in programming languages such as ML which are based on *inductive* data types. This is an interesting challenge!

18.1 Naive implementation

Let us first see why the naive approach of using a list as a queue does not work:

```
open List
type 'a queue = 'a list

let enq (x, xs) = xs @ [x]
let deq xs = (hd xs, tl xs)
```

The operations are correctly implemented, in the sense that:

```
# let q = enq (1, enq (2, enq (3, (enq (4, [])))));;
val q : int list = [4; 3; 2; 1]
# deq q;;
- : int * int list = (4, [3; 2; 1])
```

However, the complexity of the enq operation is *linear* rather than constant time because it relies on appending at the end of the list. This is not good: construction and destruction should work in constant time.

If we try to store the queue in reverse order:

```
open List
type 'a queue = 'a list

let enq (x, xs) = x::xs
let rec deq = function
    | [] -> failwith "Empty queue"
    | [x] -> (x, [])
    | x::xs -> let (x', xs') = deq xs in (x', x::xs')
```

then enq works in constant time but deq works in linear time. Not much better.

18.2 Banker's queues

The most common implementation of queues in a functional setting is due to David Gries (1981) and popularised by Okasaki. It relies on using a pair of lists, `(f, r)` where `f` holds the *front elements*, in the original order, and `r` the *rear elements*, in the reversed order. For example the queue with elements 1–6 can be held in lists

```
f = [1;2;3]
r = [6;5;4]
```

The representation of a list is not unique.

When processing the queue elements are added to the rear (in constant time) and removed from the front (in constant time).

```
type 'a queue = 'a list * 'a list

let enq (x, (f, r)) = (x::f, r)
let deq (f, x::r) = (x, (f, r))
```

This is not quite complete, as the OCaml compiler will complain:

```
let deq (f, x::r) = (x, (f, r))  ;;
        ^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(_, [])
```

Obviously, we also need to migrate elements from the rear to the front so that the front list does not become empty. In the course of this migration `f` is reversed and it is then set to `[]`. It is also convenient to define the empty queue as a pair of empty lists.

The full implementation is:

```
type 'a queue = 'a list * 'a list

let nil = ([], [])

let enq (x, (f, r)) = (f, x::r)

let deq (f, r) =
    let (f', r') = if f = [] then (rev r, []) else (f, r) in
    (hd f', (tl f', r'))
```

Let us see some examples of queue behaviour:

```
# let q = nil;;
val q : 'a list * 'b list = ([], [])
# let q = enq (1, q);;
val q : 'a list * int list = ([], [1])
# let q = enq (2, q);;
val q : 'a list * int list = ([], [2; 1])
# let q = enq (3, q);;
val q : 'a list * int list = ([], [3; 2; 1])
# let q = enq (4, q);;
val q : 'a list * int list = ([], [4; 3; 2; 1])
# let x,q = deq q;;
val x : int = 1
val q : int list * int list = ([2; 3; 4], [])
# let q = enq (5, q);;
val q : int list * int list = ([2; 3; 4], [5])
# let q = enq (6, q);;
val q : int list * int list = ([2; 3; 4], [6; 5])
# let x, q = deq q;;
val x : int = 2
val q : int list * int list = ([3; 4], [6; 5])
# let x, q = deq q;;
val x : int = 3
val q : int list * int list = ([4], [6; 5])
# let x, q = deq q;;
val x : int = 4
val q : int list * int list = ([], [6; 5])
# let x, q = deq q;;
val x : int = 5
val q : int list * int list = ([6], [])
# let x, q = deq q;;
val x : int = 6
val q : int list * int list = ([], [])
# let x, q = deq q;;
Exception: Failure "tl".
```

The underlined calls are those which involve reversing the rear list.

The extensional (input-output) behaviour of this implementation is the right one, but how about the complexity.

Let us evaluate step-by-step enq and deq as applied to the queue `([6; 5], [3; 4])`:

```
enq (7, ([3; 4], [5; 6]))
= ([3; 4], 7::[5; 6]) = ([3; 4]; [7; 6; 5])


deq ([3; 4], [5; 6])
= let (f', r') = if [3; 4] = [] then (rev [5; 6], []) else ([3; 4], [5; 6]) in
   (hd f', (tl f', r'))
= let (f', r') = false then (rev [5; 6], []) else ([3; 4], [5; 6]) in
   (hd f', (tl f', r'))
= let (f', r') = ([3; 4], [5; 6]) in
   (hd f', (tl f', r'))
= (hd [3; 4], (tl [3; 4], [5; 6]))
= (3, ([4], [5; 6]))
```

Both of them require a *constant* number of function calls. However, dequeing from `[], [4; 3; 2; 1]` works differently:

```
deq ([], [4; 3; 2; 1])
= let (f', r') = if [] = [] then (rev [4; 3; 2; 1], []) else ([], [4; 3; 2; 1]) in
```

```
   (hd f', (tl f', r'))
= let (f', r') = if true then (rev [4; 3; 2; 1], []) else ([], [4; 3; 2; 1]) in
   (hd f', (tl f', r'))
= let (f', r') =  (rev [4; 3; 2; 1], []) in
   (hd f', (tl f', r'))
= let (f', r') =  ([1; 2; 3; 4], []) in
   (hd f', (tl f', r'))
```

Note that the underlined operation (rev) required a linear number of calls!

```
= (hd [1; 2; 3; 4], (tl [1; 2; 3; 4], []))
= (1, ([2; 3; 4], []))
```

So the overall execution of the deq in this case takes a *linear* number of function calls!

18.2.1 Amortised complexity

It looks like our implementation is still not quite efficient enough. Or is it?

Let us compare the total number of calls in some arbitrary scenario of usage between the naive implementation and the "Banker's" implementation:

	Naive	Banker's
`let q = nil`	0	0
`let q = enq (1, q)`	1	1
`let q = enq (2, q)`	2	1
`let q = enq (3, q)`	3	1
⋮		
`let q = enq (`n`, q)`	n	1
`let (q, x) = deq q`	1	n
`let (q, x) = deq q`	1	1
`let (q, x) = deq q`	1	1
`let (q, x) = deq q`	1	1
⋮		
`let (q, x) = deq q`	1	1
`let (q, x) = deq q`	1	1
totals	$(1+2+3+\cdots+n)+n$	$n+n+n$
	$= n \times (n+1)/2 + n$	$= 3 \times n$
average	$n + 1.5$	3

So in the course of normal usage enqueuing then dequeing n times takes:

- a *quadratic* amount of function calls with the naive implementation
- a *linear* amount

If we look at the *average* amount of calls that each operation takes we can see that the naive implementation is on average *linear* whereas the Banker's algorithm uses a *constant* number of calls. So the naive implementation is *always* making a linear number of calls, whereas the Banker's implementation makes a linear number of calls only once in a while so that, on average, it uses a constant number of calls. We only looked at an arbitrary scenario, but a more rigorous proof involving any long enough sequence of enq and deq can show that, asymptotically, the amortised complexity is always constant time.

18.3 Interaction nets

An alternative idea is to represent a queue `[1; 2; 3; 4; 5]` as a function `fun z -> [1; 2; 3; 4; 5; z]`. The variable `z` gives immediate access to the end of the queue, whereas access to the head of the queue is done by normal pattern matching.

We always (informally) think of the queue as being of form `fun z' -> L::z'` where `L` is the list segment holding the elements of the queue and `z'` a "pointer" to the end of the queue. This is informal because it is just shortcut for a list written as, for example, `fun z -> 1::2::3::4::5::z`. The notation does not denote a language operation, it is only a short-cut.

Note that the following are perfectly equal:

```
fun z -> L :: z   vs.   fun z' -> L :: z'
```

So we sometimes rename the variable bound by `fun` to avoid clashes.

The implementation of the queue is:

```
let nil = fun z -> z
let enq (x, q) = fun z -> q (x::z)
let deq q = (hd (q []), fun z -> tl (q z))
```

nil The empty queue only has the pointer to the end, and no elements.

enq Here is how it operates on a generic queue:

```
enq x (fun z' -> L::z')
= fun z -> (fun z' -> L::z') (x::z)
= fun z -> L::x::z
```

As you see, `x` ends at the bottom of the queue, just before the dummy variable, without any extra function calls.

deq Here is how it operates on a generic queue:

```
deq (fun z' -> L::z')
= (hd ((fun z' -> L::z') []), fun z -> tl ((fun z' -> L::z') z))

First component
= hd (L::[])
= hd L

Second component
= fun z -> tl (L::z)
```

Note that both `hd` and `tl` above require a constant number of operations.

However, this is not how an OCaml implementation would behave! Let us look at some examples:

```
let q = nil

let q = enq (1, nil)
= fun z -> nil (1::z)
= fun z -> (fun z' -> z') (1::z)
```

This does equal, further,

```
=  fun z -> 1::z
```

but OCaml cannot carry out this computation! The reason is that if OCaml finds a term of the form `fun x -> M` then it stops computing, even if `M` could have some computations "inside". So we are actually stuck with

```
let q = fun z -> (fun z' -> z') (1::z)
```

Let us perform another enqueuing:

```
let q = enq (2, q)
= fun z'' -> (fun z -> (fun z' -> z') (1::z)) (2::z'')
```

As before, we are stuck here.

So although the computation is carried out in constant time, it is only *thunked*. It is dequeuing which forces some of the computation to be carried out:

```
let (x, q) = deq q
= (hd ((fun z'' -> (fun z -> (fun z' -> z') (1::z)) (2::z'')) []),
   fun z''' -> tl ((fun z'' -> (fun z -> (fun z' -> z') (1::z)) (2::z'')) z'''))
```

The second component is stuck, but the first component is evaluated

```
hd ((fun z'' -> (fun z -> (fun z' -> z') (1::z)) (2::z'')) [])
= hd ((fun z' -> z') (1::2::[]))
= hd (1::2::[])
= 1
```

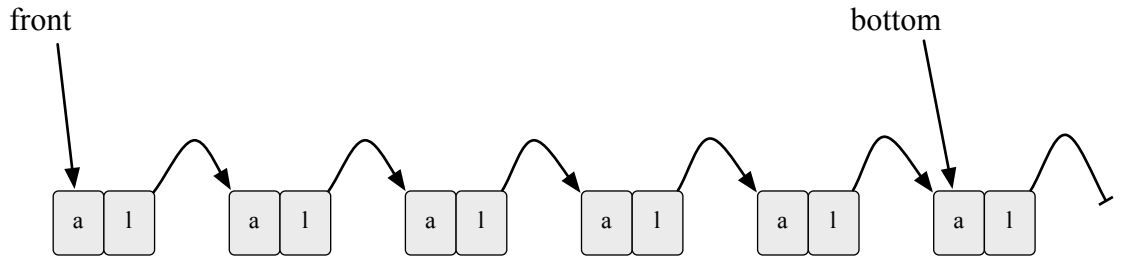
Unfortunately, now the evaluation takes a linear number of operations. Meanwhile, the dequeue operation is just thunked on the end of the second component, resulting in an ever-growing representation. This makes the performance of this representation, in OCaml, worse than the naive one.

However, in a language implementation that can reduce inside functions this implementation would be more efficient than Banker's queues! Such languages are theoretically possible but not implemented yet. Such experimental languages are based on a theoretical framework of "interaction nets", hence the name.

There is a similar, older, version of this algorithm in Prolog, called *difference lists*.

18.4 Imperative queues

In imperative languages with pointers, such as C, C++ or Java, queues are commonly implemented as linked structures with a pointer both to the front and the bottom of the queue:



As it happens, OCaml also allows pointers which can store mutable data. The content of a mutable cell can be anything of a legal type!

create `let x = ref 1` creates a mutable reference storing the value 1:

```
# let x = ref 1;;
val x : int ref = {contents = 1}
```

dereference `!x` reads the value of a mutable reference:

```
# !x;;
- : int = 1
```

assignment `x := 2` changes the value of the mutable reference:

```
# x := 2;;
- : unit = ()
# !x;;
- : int = 2
```

The assigned and the stored value must have matching types.

Note that we can store anything, including functions:

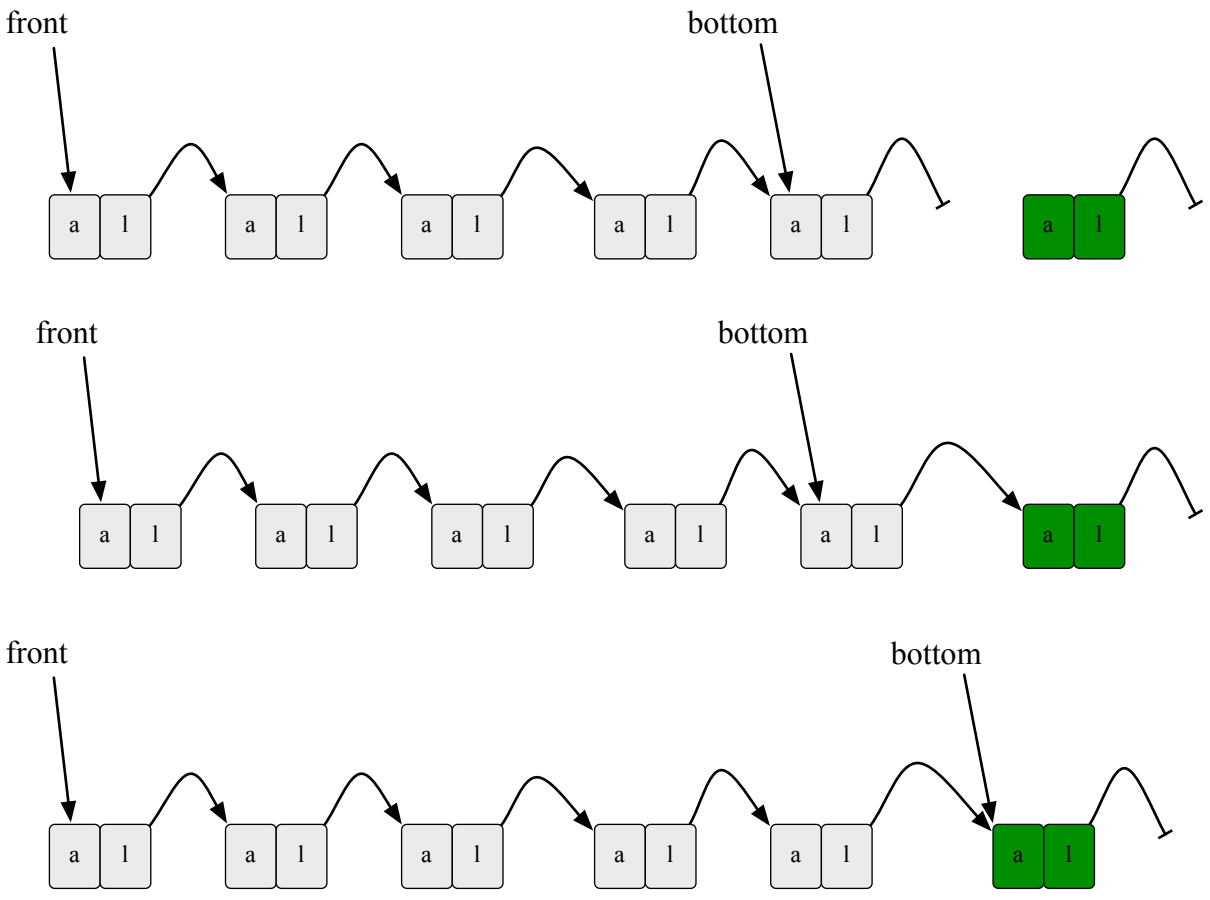
```
# let y = ref (fun x -> x + 2);;
val y : (int -> int) ref = {contents = <fun>}
# (!y)3;;
- : int = 5
# y := (fun x -> x + 3);;
- : unit = ()
# (!y)3;;
- : int = 6
```

Note that *proper* pointers are mutable cells that stores addresses of mutable cells, so that two distinct pointers can point to the same cell. For example:

```
# let c = ref 1;;                                    (* set up a mutable cell storing 1 *)
val c : int ref = {contents = 1}
# let p = ref c;;                                    (* make p a pointer to it *)
val p : int ref ref = {contents = {contents = 1}}
# let p' = ref c;;                                   (* make p' a different pointer to it *)
val p' : int ref ref = {contents = {contents = 1}}
# !p := 2;;                                          (* change the value of c to 2 via p *)
- : unit = ()
# !(!p');;                                           (* accessing c via p' also reads a 2 *)
- : int ref = {contents = 2}
```

We can defined a linked data structure for queues much like we would define it in C, C++ or Java. By pointer manipulation, enq and deq can be easily implemented in constant time.

Enqueuing, diagrammatically looks like this:



Dequeuing, diagrammatically looks like this: