Foundations of Computer Science (2024-2025)

Anil Madhavapeddy, Jonathan Ludlam

University of Cambridge Computer Laboratory
Version 1.6
September 20, 2024

Contents

1 Lecture 1: Introduction to Programming **5**
1.1 Basic Concepts in Computer Science . . . 5
1.2 Goals of Programming . . . 6
1.3 Why Program in OCaml? . . . 7
1.4 A first session with OCaml . . . 8
1.5 Raising a Number to a Power . . . 9
1.6 *Efficiently* Raising a Number to a Power . . . 11

2 Lecture 2: Recursion and Efficiency **13**
2.1 Expression Evaluation . . . 13
2.2 Summing the first n integers . . . 13
2.3 Iteratively summing the first `n` integers . . . 14
2.4 Recursion *vs* Iteration . . . 14
2.5 Silly Summing the First n Integers . . . 15
2.6 Comparing Algorithms: O Notation . . . 16
2.7 Simple Facts About O Notation . . . 17
2.8 Common Complexity Classes . . . 17
2.9 Sample costs in O notation . . . 18
2.10 Some Simple Recurrence Relations . . . 18

3 Lecture 3: Lists **21**
3.1 The List Primitives . . . 22
3.2 Getting at the Head and Tail . . . 22
3.3 Computing the Length of a List . . . 24
3.4 Efficiently Computing the Length of a List . . . 25
3.5 Append: List Concatenation . . . 26
3.6 Reversing a List in $O(n^2)$ . . . 26

3.7 Reversing a List in $O(n)$ . . . 27
3.8 Lists, Strings and Characters . . . 28

4 Lecture 4: More on Lists **30**
4.1 List Utilities: take and drop . . . 30
4.2 Linear Search . . . 31
4.3 Equality Tests . . . 31
4.4 Building a List of Pairs . . . 32
4.5 Building a Pair of Results . . . 33
4.6 An Application: Making Change . . . 33
4.7 All Ways of Making Change . . . 34
4.8 All Ways of Making Change — Faster! . . . 35

5 Lecture 5: Sorting **37**
5.1 How Fast Can We Sort? . . . 37
5.2 Insertion Sort . . . 39
5.3 Quicksort: The Idea . . . 40
5.4 Quicksort: The Code . . . 40
5.5 Append-Free Quicksort . . . 41
5.6 Merging Two Lists . . . 42
5.7 Top-down Merge sort . . . 42
5.8 Summary of Sorting Algorithms . . . 43

6 Lecture 6: Datatypes and Trees **45**
6.1 An Enumeration Type . . . 45
6.2 Declaring a Function on Vehicles . . . 46
6.3 A Datatype whose Constructors have Arguments . . . 46
6.4 A Finer Wheel Computation . . . 47
6.5 Error Handling: Exceptions . . . 47
6.6 Exceptions in OCaml . . . 48
6.7 Making Change with Exceptions . . . 49
6.8 Making Change: A Trace . . . 51
6.9 Binary Trees, a Recursive Datatype . . . 52
6.10 Basic Properties of Binary Trees . . . 53

7 Lecture 7: Dictionaries and Functional Arrays **55**
7.1 Dictionaries . . . 55
7.2 Binary Search Trees . . . 56
7.3 Lookup: Seeks Left or Right . . . 56
7.4 Update . . . 57
7.5 Aside: Traversing Trees (3 Methods) . . . 58
7.6 Efficiently Traversing Trees . . . 59
7.7 Arrays . . . 59
7.8 Functional Arrays as Binary Trees . . . 60
7.9 The Lookup Function . . . 61
7.10 The Update Function . . . 62

8 Lecture 8: Functions as Values **64**

8.1 Functions Without Names . . . 64
8.2 Curried Functions . . . 65
8.3 Shorthand for Curried Functions . . . 67
8.4 Partial Application: A Curried Insertion Sort . . . 68
8.5 map: the "Apply to All" Function . . . 69
8.6 Example: Matrix Transpose . . . 69
8.7 Review of Matrix Multiplication . . . 70
8.8 Matrix Multiplication in OCaml . . . 71
8.9 List Functionals for Predicates . . . 71
8.10 Applications of the Predicate Functionals . . . 72

9 Lecture 9: Sequences, or Lazy Lists 75
9.1 A Pipeline . . . 75
9.2 Lazy Lists (or Streams) . . . 75
9.3 Lazy Lists in OCaml . . . 76
9.4 The Infinite Sequence: k, $k+1$, $k+2$, ... 77
9.5 Consuming a Sequence . . . 77
9.6 Sample Evaluation . . . 78
9.7 Joining Two Sequences . . . 78
9.8 Functionals for Lazy Lists . . . 79
9.9 Numerical Computations on Infinite Sequences . . . 79

10 Lecture 10: Queues and Search Strategies 82
10.1 Breadth-First v Depth-First Tree Traversal . . . 82
10.2 Breadth-First Tree Traversal — Using Append . . . 83
10.3 An Abstract Data Type: Queues . . . 83
10.4 Efficient Functional Queues: Idea . . . 83
10.5 Efficient Functional Queues: Code . . . 84
10.6 Breadth-First Tree Traversal — Using Queues . . . 85
10.7 Iterative deepening: Another Exhaustive Search . . . 86
10.8 Another Abstract Data Type: Stacks . . . 86
10.9 A Survey of Search Methods . . . 87

11 Lecture 11: Elements of Procedural Programming 89
11.1 Procedural Programming . . . 89
11.2 OCaml Primitives for References . . . 89
11.3 Trying Out References . . . 90
11.4 Commands: Expressions with Effects . . . 91
11.5 Iteration: the `while` command . . . 92
11.6 Private, Persistent References . . . 92
11.7 Two Bank Accounts . . . 93
11.8 OCaml Primitives for Arrays . . . 94
11.9 Array Examples . . . 95
11.10 References: OCaml *vs* conventional languages . . . 96

This course has two aims. The first is to teach programming. The second is to present some fundamental principles of computer science, especially algorithm design. Most students will have some programming experience already, but there are few people whose programming cannot be improved through greater knowledge of basic principles. Please bear this point in mind if you have extensive experience and find parts of the course rather slow.

The programming in this course is based on the language OCaml and mostly concerns the functional programming style. Functional programs tend to be shorter and easier to understand than their counterparts in conventional languages such as C. In the space of a few weeks, we shall cover many fundamental data structures and learn basic methods for estimating efficiency.

The first thing you will notice about this course is that there is an *interactive* version hosted online at `https://hub.cl.cam.ac.uk/`, where you can login with your Cambridge Raven identity and edit the code fragments in your browser. You are encouraged to do so – such edits will only persist in your session, and will help you to explore the world of functional programming. If you are using the web-based version, then you need to know a few concepts:

- The notebook consists of a sequence of textual and code snippets.
- The code snippets can be executed individually, and will "remember" the results of the previous snippets.
- To begin with, click on `Cell` / `Run All` in the menu to execute the entire notebook.
- You can later double click on any cell and modify its contents, and press `Shift+Enter` to reevaluate its contents. This will only modify the current cell, so you will have to `Run All` again to see the effects on the whole notebook.
- While editing longer snippets, you can also press `Shift+Tab` while typing to get more documentation hints about the code you are writing.

This course is lectured by Anil Madhavapeddy, with the practical exercises managed by Jonathan Ludlam. These notes are translated from Lawrence C. Paulson's earlier course on Standard ML, which had credits to David Allsopp, Stuart Becker, Gavin Bierman, Chloë Brown, Silas Brown, Qi Chen, David Cottingham, William Denman, Robert Harle, Daniel Hulme, Frank King, Jack Lawrence-Jones, Joseph Lord, Dimitrios Los, Farhan Mannan, James Margetson, David Morgan, Alan Mycroft, Sridhar Prabhu, Frank Stajano, Alex Trifanov, Thomas Tuerk, Xincheng Wang, Philip Withnall and Assel Zhiyenbayeva for pointing out errors. The current notes were ported to OCaml in 2019 by Anil Madhavapeddy, David Allsopp, and Jon Ludlam and subsequently edited by Jeremy Yallop. We thank Richard Sharp, Srinivasan Keshav, Ambroise Lafont, Vojtěch Tvrdík and Jeremy Yallop for further feedback and corrections since 2020.

Some books that are complementary to this course are:

- *OCaml from the Very Beginning* by John Whitington.

Chapter 1

Lecture 1: Introduction to Programming

1.1 Basic Concepts in Computer Science

- Computers: a child can use them; **nobody** can fully understand them!
- We can master complexity through levels of abstraction.
- Focus on 2 or 3 levels at most!

Recurring issues:

- *what services* to provide at each level
- *how to implement* them using lower-level services
- *the interface* that defines how the two levels should communicate

A basic concept in computer science is that large systems can only be understood in levels, with each level further subdivided into functions or services of some sort. The interface to the higher level should supply the advertised services. Just as important, it should block access to the means by which those services are implemented. This *abstraction barrier* allows one level to be changed without affecting levels above. For example, when a manufacturer designs a faster version of a processor, it is essential that existing programs continue to run on it. Any differences between the old and new processors should be invisible to the program.

Modern processors have elaborate specifications, which still sometimes leave out important details. In the old days, you then had to consult the circuit diagrams.

1.1.1 Example 1: Dates

- Abstract level: dates over a certain interval
- Concrete level: typically 6 characters: `YYMMDD` (where each character is represented by 8 bits)
- Date crises caused by **inadequate** internal formats:
  - Digital's PDP-10: using 12-bit dates (good for at most 11 years)
  - 2000 crisis: 48 bits could be good for lifetime of universe!

Digital Equipment Corporation's date crisis occurred in 1975. The PDP-10 was a 36-bit mainframe computer. It represented dates using a 12-bit format designed for the tiny PDP-8. With 12 bits,

one can distinguish $2^{12} = 4096$ days or 11 years.

The most common industry format for dates uses six characters: two for the year, two for the month and two for the day. The most common "solution" to the year 2000 crisis is to add two further characters, thereby altering file sizes. Others have noticed that the existing six characters consist of 48 bits, already sufficient to represent all dates over the projected lifetime of the universe: $2^{48} = 2.8 \times 10^{14}$ days $= 7.7 \times 10^{11}$ years!

Mathematicians think in terms of unbounded ranges, but the representation we choose for the computer usually imposes hard limits. A good programming language like OCaml lets one easily change the representation used in the program. But if files in the old representation exist all over the place, there will still be conversion problems. The need for compatibility with older systems causes problems across the computer industry.

1.1.2 Example II: Floating Point Numbers

Computers have integers like `1066` and floats like 1.066×10^3. A floating-point number is represented by two integers. The concept of *data type* involves:

- how a value is represented inside the computer
- the suite of operations given to programmers
- valid and invalid (or exceptional) results, such as "infinity"

Computer arithmetic can yield *incorrect answers!*

In science, numbers written with finite precision and a decimal exponent are said to be in *standard form*. The computational equivalent is the *floating point number*. These are familiar to anybody who has used a scientific calculator. Internally, a float consists of two integers.

Because of its finite precision, floating-point computations are potentially inaccurate. To see an example, use your nearest electronic calculator to compute $(2^{1/10000})^{10000}$. I get 1.99999959! With certain computations, the errors spiral out of control. Many programming languages fail to check whether even integer computations fall within the allowed range: you can add two positive integers and get a negative one!

Most computers give us a choice of precisions. In 32-bit precision, integers typically range from $2^{31} - 1$ (namely 2 147 483 647) to -2^{31}; floats are accurate to about six decimal places and can get as large as 10^{35} or so. For floats, 64-bit precision is often preferred. Early languages like Fortran required variables to be declared as `INTEGER`, `REAL` or `COMPLEX` and barred programmers from mixing numbers in a computation. Nowadays, programs handle many different kinds of data, including text and symbols. The concept of a *data type* can ensure that different types of data are not combined in a senseless way.

Inside the computer, all data are stored as bits. In most programming languages, the compiler uses types to generate correct machine code, and types are not stored during program execution. In this course, we focus almost entirely on programming in a high-level language: OCaml.

1.2 Goals of Programming

- to describe a computation so that it can be done **mechanically**:
  - Expressions compute values.
  - Commands cause effects.

- to do so efficiently and **correctly**, giving the right answers quickly
- to allow easy modification as needs change
  - Through an orderly **structure** based on abstraction principles
  - Such as modules or classes

Programming *in-the-small* concerns the writing of code to do simple, clearly defined tasks. Programs provide expressions for describing mathematical formulae and so forth. This was the original contribution of FORTRAN, the FORmula TRANslator. Commands describe how control should flow from one part of the program to the next.

As we code layer upon layer, we eventually find ourselves programming *in the large* : joining large modules to solve some messy task. Programming languages have used various mechanisms to allow one part of the program to provide interfaces to other parts. Modules encapsulate a body of code, allowing outside access only through a programmer-defined interface. *Abstract Data Types* are a simpler version of this concept, which implement a single concept such as dates or floating-point numbers.

Object-oriented programming is the most complicated approach to modularity. *Classes* define concepts, and they can be built upon other classes. Operations can be defined that work in appropriately specialised ways on a family of related classes. *Objects* are instances of classes and hold the data that is being manipulated.

This course does not cover OCaml's sophisticated module system, which can do many of the same things as classes. You will learn all about objects when you study Java. OCaml includes a powerful object system, although this is not used as much as its module system.

1.3 Why Program in OCaml?

Why program in OCaml at all?

- It is interactive.
- It has a flexible notion of *data type*.
- It hides the underlying hardware: *no crashes*.
- Programs can easily be understood mathematically.
- It distinguishes naming something from *updating memory*.
- It manages storage for us.

Programming languages matter. They affect the reliability, security, and efficiency of the code you write, as well as how easy it is to read, refactor, and extend. The languages you know can also change how you think, influencing the way you design software even when you're not using them.

What makes OCaml special is that it occupies a sweet spot in the space of programming language designs. It provides a combination of efficiency, expressiveness and practicality that is difficult to find matched by any other language. "ML" was originally the meta language of the LCF (Logic for Computable Functions) proof assistant released by Robin Milner in 1972 (at Stanford, and later at Cambridge). ML was turned into a compiler in order to make it easier to use LCF on different machines, and it was gradually turned into a full-fledged system of its own by the 1980s.

The modern OCaml emerged in 1996, and the past twenty five years have seen OCaml attract a significant user base with language improvements being steadily added to support the growing commercial and academic codebases. OCaml is therefore the outcome of years of research into

programming languages, and a good base to begin our journey into learning the foundations of computer science.

Because of its connection to mathematics, OCaml programs can be designed and understood without thinking in detail about how the computer will run them. Although a program can abort, it cannot crash: it remains under the control of the OCaml system. It still achieves respectable efficiency and provides lower-level primitives for those who need them. Most other languages allow direct access to the underlying machine and even try to execute illegal operations, causing crashes.

The only way to learn programming is by writing and running programs. This web notebook provides an interactive environment where you can modify the example fragments and see the results for yourself. You should also consider installing OCaml on your own computer so that you try more advanced programs locally.

1.4 A first session with OCaml

```
In [1]: let pi = 3.14159265358979
```

```
Out[1]: val pi : float = 3.14159265358979
```

The first line of this simple session is a *value declaration*. It makes the name `pi` stand for the floating point number `3.14159`. (Such names are called *identifiers*.) OCaml echoes the name (`pi`) and type (`float`) of the declared identifier.

```
In [2]: pi *. 1.5 *. 1.5
```

```
Out[2]: - : float = 7.06858347057702829
```

The second line computes the area of the circle with radius `1.5` using the formula $A = \pi r^2$. We use `pi` as an abbreviation for `3.14159`. Multiplication is expressed using `*.`, which is called an *infix operator* because it is written between its two operands.

OCaml replies with the computed value (about `7.07`) and its type (again `float`).

```
In [3]: let area r = pi *. r *. r
```

```
Out[3]: val area : float -> float = <fun>
```

To work abstractly, we should provide the service "compute the area of a circle," so that we no longer need to remember the formula. This sort of encapsulated computation is called a *function*. The third line declares the function `area`. Given any floating point number `r`, it returns another floating point number computed using the `area` formula; note that the function has type `float -> float`.

```
In [4]: area 2.0
```

```
Out[4]: - : float = 12.56637061435916
```

The fourth line calls the function `area` supplying `2.0` as the argument. A circle of radius `2` has an area of about `12.6`. Note that brackets around a function argument are not necessary.

The function uses `pi` to stand for `3.14159`. Unlike what you may have seen in other programming languages, `pi` cannot be "assigned to" or otherwise updated. Its meaning within `area` will persist even if we issue a new `let` declaration for `pi` afterwards.

1.5 Raising a Number to a Power

```
In [5]: let rec npower x n =
          if n = 0 then 1.0
          else x *. npower x (n - 1)
```

```
Out[5]: val npower : float -> int -> float = <fun>
```

Our new `npower` definition can now take additional arguments, reflected in the arrows present in the type of `npower`; these represent *parameters* that can be passed to the new value being defined, with the final segment being the resulting type. Thus our `npower` type can be read as "pass in a float and integer to return a float".

Mathematical Justification (for $x \neq 0$):

$$x^0 = 1$$
$$x^{n+1} = x \times x^n.$$

The function `npower` raises its float argument `x` to the power `n`, a non-negative integer. The function is **recursive**: it calls itself. You can spot a recursive function due to the `rec` keyword in the definition: this indicates that any invocation of the function name within the function body should call itself. This concept should be familiar from mathematics, since exponentiation is defined by the rules shown above. You may also have seen recursion in the product rule for differentiation: $(u \cdot v)' = u \cdot v' + u' \cdot v$. In finding the derivative of $u \cdot v$, we recursively find the derivatives of u and v, combining them to obtain the desired result. The recursion is meaningful because it terminates: we reduce the problem to two smaller problems, and this cannot go on forever. The OCaml programmer uses recursion heavily. For $n \geq 0$, the equation $x^{n+1} = x \times x^n$ yields an obvious computation:

$$x^3 = x \times x^2 = x \times x \times x^1 = x \times x \times x \times x^0 = x \times x \times x$$

The equation clearly holds even for negative n. However, the corresponding computation runs forever:

$$x^{-1} = x \times x^{-2} = x \times x \times x^{-3} = \cdots$$

Note that the function `npower` contains both an integer constant (`0`) and a floating point constant (`1.0`). The decimal point makes all the difference. OCaml will notice and ascribe different meaning to each type of constant.

```
In [6]: let square x = x *. x
```

```
Out[6]: val square : float -> float = <fun>
```

Now for a tiresome but necessary aside. In most languages, the types of arguments and results must always be specified. OCaml is unusual that it normally infers the types itself. However, sometimes it is useful to supply a hint to help you debug and develop your program. OCaml will still infer the types even if you don't specify them, but in some cases it will use a more inefficient function than a specialised one. Some languages have just one type of number, converting automatically between different formats; this is slow and could lead to unexpected rounding errors. Type constraints are allowed almost anywhere. We can put one on any occurrence of x in the function.

```
In [7]: let square (x : float) = x *. x

Out[7]: val square : float -> float = <fun>
```

Or we can constrain the type of the function's result:

```
In [8]: let square x : float = x *. x

Out[8]: val square : float -> float = <fun>
```

OCaml treats the equality and comparison test specially. Expressions like `if x = y then` … are allowed provided `x` and `y` have the same type and equality testing is possible for that type. (We discuss equality further in a later lecture.) Note that `x <> y` is OCaml for $x \neq y$.

A characteristic feature of the computer is its ability to test for conditions and act accordingly. In the early days, a program might jump to a given address depending on the sign of some number. Later, John McCarthy defined the *conditional expression* to satisfy `if true then x else y = x` and `if false then x else y = y`.

OCaml evaluates the expression `if` B `then` E_1 `else` E_2 by first evaluating B. If the result is `true` then OCaml evaluates E_1 and otherwise E_2. Only one of the two expressions E_1 and E_2 is evaluated! If both were evaluated, then recursive functions like `npower` above would run forever.

The `if`-expression is governed by an expression of type `bool`, whose two values are `true` and `false`. In modern programming languages, tests are not built into "conditional branch" constructs but can just be part of normal expressions. Tests, or *Boolean expressions,* can be expressed using relational operators such as `<` and `=`. They can be combined using the Boolean operators for negation (`not`), conjunction (written as `&&`) and disjunction (written as `||`). New properties can be declared as functions: here, to test whether an integer is even, for example:

```
In [9]: let even n = n mod 2 = 0

Out[9]: val even : int -> bool = <fun>
```

1.6 *Efficiently* Raising a Number to a Power

```
In [10]: let rec power x n =
           if n = 1 then x
           else if even n then
             power (x *. x) (n / 2)
           else
             x *. power (x *. x) (n / 2)
```

```
Out[10]: val power : float -> int -> float = <fun>
```

Mathematical Justification

$$\begin{aligned} x^1 &= x \\ x^{2n} &= (x^2)^n \\ x^{2n+1} &= x \times (x^2)^n. \end{aligned}$$

For large `n`, computing powers using $x^{n+1} = x \times x^n$ is too slow to be practical. The equations above are much faster. Example:

$$2^{12} = 4^6 = 16^3 = 16 \times 256^1 = 16 \times 256 = 4096.$$

Instead of `n` multiplications, we need at most $2 \lg n$ multiplications, where $\lg n$ is the logarithm of n to the base 2.

We use the function `even`, declared previously, to test whether the exponent is even. Integer division (`/`) truncates its result to an integer: dividing $2n+1$ by 2 yields n.

A recurrence is a useful computation rule only if it is bound to terminate. If $n > 0$ then n is smaller than both $2n$ and $2n+1$. After enough recursive calls, the exponent will be reduced to 1. The equations also hold if $n \leq 0$, but the corresponding computation runs forever.

Our reasoning assumes arithmetic to be *exact*. Fortunately, the calculation is well-behaved using floating-point.

Computer numbers have a finite range, which if exceeded results in the integer wrapping around. You will understand this behaviour more as you learn about computer architecture and how modern systems represent numbers in memory.

If integers and floats must be combined in a calculation, OCaml provides functions to convert between them:

```
In [11]: int_of_float 3.14159
```

```
Out[11]: - : int = 3
```

```
In [12]: float_of_int 3
```

```
Out[12]: - : float = 3.
```

OCaml's libraries are organised using "modules", so we may use compound identifiers such as `Float.of_int` to refer to library functions. There are many thousands of library functions available in the OCaml ecosystem, including text-processing and operating systems functions in addition to the usual numerical ones.

1.6.1 Exercise 1.1

One solution to the year 2000 bug involves storing years as two digits, but interpreting them such that 50 means 1950 and 49 means 2049. Comment on the merits and demerits of this approach.

1.6.2 Exercise 1.2

Using the date representation of the previous exercise, code OCaml functions to (a) compare two years (b) add/subtract some given number of years from another year.

1.6.3 Exercise 1.3

Why would no experienced programmer write an expression of the form `if … then true else false`? What about expressions of the form `if … then false else true`?

1.6.4 Exercise 1.4

Functions `npower` and `power` both return a `float`. The definition of `npower` returns the float value `1.0` in its base case. The definition of `power` does not, so how does the OCaml type checker know that `power` returns a `float`?

1.6.5 Exercise 1.5

Because computer arithmetic is based on binary numbers, simple decimals such as 0.1 often cannot be represented exactly. Write a function `mul` that performs the computation m

$$\underbrace{x + x + \cdots + x}_{n}$$

where x has type `float`. (It is essential to use repeated addition rather than multiplication!)

The value computed with `n = 10000` and `x = 0.1` may print as `1000.0`, which looks exact. If that happens, then evaluate the expression `mul 0.1 10000 - 1000.0`

An error of this type has been blamed for the failure of an American Patriot Missile battery to intercept an incoming Iraqi missile during the first Gulf War. The missile hit an American Army barracks, killing 28.

1.6.6 Exercise 1.6

Another example of the inaccuracy of floating-point arithmetic takes the golden ratio $\phi \approx 1.618\ldots$ as its starting point:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \gamma_{n+1} = \frac{1}{\gamma_n - 1}.$$

In theory, it is easy to prove that $\gamma_n = \cdots = \gamma_1 = \gamma_0$ for all $n > 0$. Code this computation in OCaml and report the value of γ_{50}. *Hint:* in OCaml, $\sqrt{5}$ is expressed as `sqrt 5.0`.

Chapter 2

Lecture 2: Recursion and Efficiency

2.1 Expression Evaluation

Expression evaluation concerns expressions and the values they return. This view of computation may seem to be too narrow. It is certainly far removed from computer hardware, but that can be seen as an advantage. For the traditional concept of computing solutions to problems, expression evaluation is entirely adequate.

Starting with E_0, the expression E_i is reduced to E_{i+1} until this process concludes with a value v. A *value* is something like a number that cannot be further reduced.

We write $E \to E'$ to say that E is *reduced* to E'. Mathematically, they are equal: $E = E'$, but the computation goes from E to E' and never the other way around.

Computers also interact with the outside world. For a start, they need some means of accepting problems and delivering solutions. Many computer systems monitor and control industrial processes. This role of computers is familiar now, but was never envisaged in the early days. Computer pioneers focused on mathematical calculations. Modelling interaction and control requires a notion of *states* that can be observed and changed. Then we can consider updating the state by assigning to variables or performing input/output, finally arriving at conventional programs as coded in C, for instance.

For now, we remain at the level of expressions, which is usually termed *functional programming*.

2.2 Summing the first n integers

```
In [13]: let rec nsum n =
           if n = 0 then
             0
           else
             n + nsum (n - 1)

Out[13]: val nsum : int -> int = <fun>
```

The function call `nsum n` computes the sum `1 + … + nz` rather naively, hence the initial `n` in its name:

$$\begin{aligned}\text{nsum } 3 &\Rightarrow 3 + (\text{nsum } 2)\\ &\Rightarrow 3 + (2 + (\text{nsum } 1)\\ &\Rightarrow 3 + (2 + (1 + \text{nsum } 0))\\ &\Rightarrow 3 + (2 + (1 + 0))\end{aligned}$$

The nesting of parentheses is not just an artifact of our notation; it indicates a real problem. The function gathers up a collection of numbers, but none of the additions can be performed until `nsum 0` is reached. Meanwhile, the computer must store the numbers in an internal data structure, typically the *stack*. For large `n`, say `nsum 10000`, the computation might fail due to stack overflow.

We all know that the additions can be performed as we go along. How do we make the computer do that?

2.3 Iteratively summing the first `n` integers

```
In [14]: let rec summing n total =
           if n = 0 then
             total
           else
             summing (n - 1) (n + total)
```

```
Out[14]: val summing : int -> int -> int = <fun>
```

Function `summing` takes an additional argument: a running total. If `n` is zero then it returns the running total; otherwise, `summing` adds to it and continues. The recursive calls do not nest; the additions are done immediately.

A recursive function whose computation does not nest is called *iterative* or *tail-recursive*. Many functions can be made iterative by introducing an argument analogous to `total`, which is often called an *accumulator*.

The gain in efficiency is sometimes worthwhile and sometimes not. The function `power` is not iterative because nesting occurs whenever the exponent is odd. Adding a third argument makes it iterative, but the change complicates the function and the gain in efficiency is minute; for 32-bit integers, the maximum possible nesting is 30 for the exponent $2^{31} - 1$.

2.4 Recursion *vs* Iteration

- "Iterative" normally refers to a loop, coded using `while` for example (see the final lecture)
- Tail-recursion is only efficient if the compiler detects it
- Mainly it saves space (memory), though iterative code can also run faster
- Do not make programs iterative unless the gain is worth it

A classic book by Abelson and Sussman, which describes the Lisp dialect known as Scheme, used *iterative* to mean *tail-recursive*. Iterative functions produce computations resembling those that

can be done using while-loops in conventional languages.

Many algorithms can be expressed naturally using recursion, but only awkwardly using iteration. There is a story that Dijkstra sneaked recursion into Algol-60 by inserting the words "any other occurrence of the procedure name denotes execution of the procedure." By not using the word "recursion", he managed to slip this amendment past sceptical colleagues.

Obsession with tail recursion leads to a coding style in which functions have many more arguments than necessary. Write straightforward code first, avoiding only gross inefficiency. If the program turns out to be too slow, tools are available for pinpointing the cause. Always remember KISS (Keep It Simple, Stupid).

I hope you have all noticed by now that the summation can be done even more efficiently using the arithmetic progression formula:

$$1 + \cdots + n = n(n+1)/2$$

2.5 Silly Summing the First n Integers

```
In [15]: let rec sillySum n =
           if n = 0 then
             0
           else
             n + (sillySum (n - 1) + sillySum (n - 1)) / 2
```

```
Out[15]: val sillySum : int -> int = <fun>
```

The function calls itself 2^n times! Bigger inputs mean higher costs—but what's the growth rate?

Now let us consider how to estimate various costs associated with a program. *Asymptotic complexity* refers to how costs—usually time or space—grow with increasing inputs. Space complexity can never exceed time complexity, for it takes time to do anything with the space. Time complexity often greatly exceeds space complexity.

The function `sillySum` calls itself twice in each recursive step. This function is contrived, but many mathematical formulas refer to a particular quantity more than once. In OCaml, we can create a local binding to a computed value using the *local declaration* syntax. In the following expression, `y` is computed once and used twice:

```
In [16]: let x = 2.0 in
         let y = Float.pow x 20.0 in
         y *. (x /. y)
```

```
Out[16]: - : float = 2.
```

You can read `let x = e1 in e2` as assigning (or "binding") the name `x` with the value of `e1` into `e2`. Any use of `x` within `e2` will have the value of `e1`, and `x` will only be visible in subexpressions into which it has been bound.

Why do we need let bindings? Fast hardware does not make good algorithms unnecessary. On the contrary, faster hardware magnifies the superiority of better algorithms. Typically, we want to handle the largest inputs possible. If we double our processing power, what do we gain? How much can we increase n, the input to our function?

With `sillySum`, we can only go from n to $n+1$. We are limited to this modest increase because the function's running time is proportional to 2^n. With the function `npower` defined in the previous section, we can go from n to $2n$: we can handle problems twice as big. With `power` we can do much better still, going from n to n^2.

The following table (excerpted from a 50-year-old book!) illustrates the effect of various time complexities. The left-hand column (dubbed "complexity") is defind as how many milliseconds are required to process an input of size n. The other entries show the maximum size of n that can be processed in the given time (one second, minute or hour).

complexity	1 second	1 minute	1 hour	gain
n	1000	60 000	3 600 000	$\times 60$
$n \log n$	140	4 895	204 095	$\times 41$
n^2	31	244	1 897	$\times 8$
n^3	10	39	153	$\times 4$
2^n	9	15	21	$+6$

The table illustrates how large an input can be processed as a function of time. As we increase the computer time per input from one second to one minute and then to one hour, the size of the input increases accordingly.

The top two rows (complexities n and $n \lg n$) increase rapidly: for n, by a factor of 60 per column. The bottom two start out close together, but n^3 (which grows by a factor of 3.9) pulls well away from 2^n (whose growth is only additive). If an algorithm's complexity is exponential then it can never handle large inputs, even if it is given huge resources. On the other hand, suppose the complexity has the form n^c, where c is a constant. (We say the complexity is *polynomial*.) Doubling the argument then increases the cost by a constant factor. That is much better, though if $c > 3$ the algorithm may not be considered practical.

2.6 Comparing Algorithms: O Notation

- Formally, define $f(n) = O(g(n))$ provided $|f(n)| \leq c|g(n)|$ as $n \to \infty$
- $|f(n)|$ is bounded for some constant c and all *sufficiently large* n.
- Intuitively, look at the *most significant* term.
- Ignore *constant factors* as they seldom dominate and are often transitory

For example: consider n^2 instead of $3n^2 + 34n + 433$.

The cost of a program is usually a complicated formula. Often we should consider only the most significant term. If the cost is $n^2 + 99n + 900$ for an input of size n, then the n^2 term will eventually dominate, even though $99n$ is bigger for $n < 99$. The constant term 900 may look big, but it is soon dominated by n^2.

Constant factors in costs can be ignored unless they are large. For one thing, they seldom make a difference: $100n^2$ will be better than n^3 in the long run: or *asymptotically* to use the jargon. Moreover, constant factors are seldom stable. They depend upon details such as which hardware, operating system or programming language is being used. By ignoring constant factors, we can make comparisons between algorithms that remain valid in a broad range of circumstances.

The "Big O" notation is commonly used to describe efficiency—to be precise, *asymptotic complexity*. It concerns the limit of a function as its argument tends to infinity. It is an abstraction that meets the informal criteria that we have just discussed. In the definition, *sufficiently large* means there is some constant n_0 such that $|f(n)| \leq c|g(n)|$ for all n greater than n_0. The role of n_0 is to ignore finitely many exceptions to the bound, such as the cases when $99n$ exceeds n^2.

2.7 Simple Facts About O Notation

$$
\begin{aligned}
O(2g(n)) &\text{ is the same as } O(g(n)) \\
O(\log_{10} n) &\text{ is the same as } O(\ln n) \\
O(n^2 + 50n + 36) &\text{ is the same as } O(n^2)
\end{aligned}
$$

$$
\begin{aligned}
O(n^2) &\text{ is contained in } O(n^3) \\
O(2^n) &\text{ is contained in } O(3^n) \\
O(\log n) &\text{ is contained in } O(\sqrt{n})
\end{aligned}
$$

O notation lets us reason about the costs of algorithms easily.

- Constant factors such as the 2 in $O(2g(n))$ drop out: we can use $O(g(n))$ with twice the value of c in the definition.
- Because constant factors drop out, the base of logarithms is irrelevant.
- Insignificant terms drop out. To see that $O(n^2 + 50n + 36)$ is the same as $O(n^2)$, consider that $n^2 + 50n + 36/n^2$ converges to 1 for increasing n. In fact, $n^2 + 50n + 36 \leq 2n^2$ for $n \geq 51$, so can double the constant factor

If c and d are constants (that is, they are independent of n) with $0 < c < d$ then - $O(n^c)$ is contained in $O(n^d)$ - $O(c^n)$ is contained in $O(d^n)$ - $O(\log n)$ is contained $inO(n^c)$

To say that $O(c^n)$ *is contained in* $O(d^n)$ means that the former gives a tighter bound than the latter. For example, if $f(n) = O(2^n)$ then $f(n) = O(3^n)$ trivially, but the converse does not hold.

2.8 Common Complexity Classes

- $O(1)$ is *constant*
- $O(\log n)$ is *logarithmic*
- $O(n)$ is *linear*
- $O(n \log n)$ is *quasi-linear*
- $O(n^2)$ is *quadratic*
- $O(n^3)$ is *cubic*
- $O(a^n)$ is *exponential* (for fixed a)

Logarithms grow very slowly, so $O(\log n)$ complexity is excellent. Because O notation ignores constant factors, the base of the logarithm is irrelevant!

Under linear we might mention $O(n \log n)$, which occasionally is called *quasilinear* and which scales up well for large n.

An example of quadratic complexity is matrix addition: forming the sum of two $n \times n$ matrices obviously takes n^2 additions. Matrix multiplication is of cubic complexity, which limits the size of matrices that we can multiply in reasonable time. An $O(n^{2.81})$ algorithm exists, but it is too complicated to be of much use, even though it is theoretically better.

An exponential growth rate such as 2^n restricts us to small values of n. Already with $n = 20$ the cost exceeds one million. However, the worst case might not arise in normal practice. OCaml type-checking is exponential in the worst case, but not for ordinary programs.

2.9 Sample costs in O notation

Recall that `npower` computes x^n by repeated multiplication while `nsum` naively computes the sum $1 + \cdots + n$. Each obviously performs $O(n)$ arithmetic operations. Because they are not tail recursive, their use of space is also $O(n)$. The function `summing` is a version of `nsum` with an accumulating argument; its iterative behaviour lets it work in constant space. O notation spares us from having to specify the units used to measure space.

Function	Time	Space
npower, nsum	O(n)	O(n)
summing	O(n)	O(1)
$n(n+1)/2$	O(1)	O(1)
power	O(log n)	O(log n)
sillySum	O(2^n)	O(n)

Even ignoring constant factors, the units chosen can influence the result. Multiplication may be regarded as a single unit of cost. However, the cost of multiplying two n-digit numbers for large n is itself an important question, especially now that public-key cryptography uses numbers hundreds of digits long.

Few things can *really* be done in constant time or stored in constant space. Merely to store the number n requires $O(\log n)$ bits. If a program cost is $O(1)$, then we have probably assumed that certain operations it performs are also $O(1)$—typically because we expect never to exceed the capacity of the standard hardware arithmetic.

With `power`, the precise number of operations depends upon n in a complicated way, depending on how many odd numbers arise, so it is convenient that we can just write $O(\log n)$. An accumulating argument could reduce its space cost to $O(1)$.

2.10 Some Simple Recurrence Relations

Consider a function $T(n)$ that has a cost we want to bound using O notation. A typical *base case* is $T(1) = 1$. Some *recurrences* are:

Equation	Complexity
$T(n+1) = T(n) + 1$	$O(n)$
$T(n+1) = T(n) + n$	$O(n^2)$
$T(n) = T(n/2) + 1$	$O(\log n)$
$T(n) = 2T(n/2) + n$	$O(n \log n)$

To analyse a function, inspect its OCaml declaration. Recurrence equations for the cost function $T(n)$ can usually be read off. Since we ignore constant factors, we can give the base case a cost of one unit. Constant work done in the recursive step can also be given unit cost; since we only need an upper bound, this unit represents the larger of the two actual costs. We could use other constants if it simplifies the algebra.

For example, recall our function `nsum`:

```
In [17]: let rec nsum n =
           if n = 0 then
             0
           else
             n + nsum (n - 1)
```

```
Out[17]: val nsum : int -> int = <fun>
```

Given $n+1$, it performs a constant amount of work (an addition and subtraction) and calls itself recursively with argument n. We get the recurrence equations $T(0) = 1$ and $T(n+1) = T(n) + 1$. The closed form is clearly $T(n) = n + 1$, as we can easily verify by substitution. The cost is *linear*.

This function, given $n+1$, calls `nsum`, performing $O(n)$ work. Again ignoring constant factors, we can say that this call takes exactly n units.

```
In [18]: let rec nsumsum n =
           if n = 0 then
             0
           else
             nsum n + nsumsum (n - 1)
```

```
Out[18]: val nsumsum : int -> int = <fun>
```

We get the recurrence equations $T(0) = 1$ and $T(n+1) = T(n) + n$. It is easy to see that $T(n) = (n-1) + \cdots + 1 = n(n-1)/2 = O(n^2)$. The cost is *quadratic*.

The function `power` divides its input n into two, with the recurrence equation $T(n) = T(n/2) + 1$. Clearly $T(2^n) = n + 1$, so $T(n) = O(\log n)$.

2.10.1 Exercise 2.1

Code an iterative version of the function `power`.

2.10.2 Exercise 2.2

Add a column to the table of complexities from *The Design and Analysis of Computer Algorithms* with the heading *60 hours:*

complexity	1 second	1 minute	1 hour	60 hours
n	1000	60 000	3 600 000	
$n \log n$	140	4 895	204 095	
n^2	31	244	1 897	
n^3	10	39	153	
2^n	9	15	21	

2.10.3 Exercise 2.3

Let g_1, ..., g_k be functions such that $g_i(n) \geq 0$ for $i = 1$, ..., k and all sufficiently large n.

Show that if $f(n) = O(a_1 g_1(n) + \cdots + a_k g_k(n))$ then $f(n) = O(g_1(n) + \cdots + g_k(n))$.

2.10.4 Exercise 2.4

Find an upper bound for the recurrence given by $T(1) = 1$ and $T(n) = 2T(n/2) + 1$. You should be able to find a tighter bound than $O(n \log n)$.

Chapter 3

Lecture 3: Lists

```
In [19]: let x = [3; 5; 9]

Out[19]: val x : int list = [3; 5; 9]

In [20]: let y = [(1, "one"); (2, "two")]

Out[20]: val y : (int * string) list = [(1, "one"); (2, "two")]
```

A *list* is an ordered series of elements; repetitions are significant. So `[3; 5; 9]` differs from `[5; 3; 9]` and from `[3; 3; 5; 9]`. Elements in the list are separated with `;` when constructed, as opposed to the `,` syntax used for fixed-length tuples.

All elements of a list must have the same type. Above we see a list of integers and a list of `(integer, string)` pairs. One can also have lists of lists, such as `[[3]; []; [5; 6]]`, which has type `int list list`.

In the general case, if $x_1; \ldots; x_n$ all have the same type (say τ) then the list $[x_1; \ldots; x_n]$ has type (τ)`list`.

Lists are the simplest data structure that can be used to process collections of items. Conventional languages use *arrays* whose elements are accessed using subscripting: for example, $A[i]$ yields the ith element of the array A. Subscripting errors are a known cause of programmer grief, however, so arrays should be replaced by higher-level data structures whenever possible.

```
In [21]: x @ [2; 10]

Out[21]: - : int list = [3; 5; 9; 2; 10]

In [22]: List.rev [(1, "one"); (2, "two")]

Out[22]: - : (int * string) list = [(2, "two"); (1, "one")]
```

The infix operator `@` (also called `List.append`) concatenates two lists. Also built-in is `List.rev`, which reverses a list. These are demonstrated in the session above.

3.1 The List Primitives

There are two kinds of lists:

- `[]` represents the empty list
- `x :: l` is the list with head x and tail l

```
In [23]: let nil = []
```

```
Out[23]: val nil : 'a list = []
```

```
In [24]: 1 :: nil
```

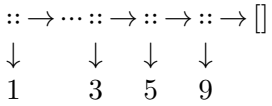
```
Out[24]: - : int list = [1]
```

```
In [25]: 1 :: 2 :: nil
```

```
Out[25]: - : int list = [1; 2]
```

The operator `::` (also called `List.cons` for "construct"), puts a new element on to the head of an existing list. While we should not be too preoccupied with implementation details, it is essential to know that `::` is an $O(1)$ operation. It uses constant time and space, regardless of the length of the resulting list. Lists are represented internally with a linked structure; adding a new element to a list merely hooks the new element to the front of the existing structure. Moreover, that structure continues to denote the same list as it did before; to see the new list, one must look at the new `::` node (or "cons cell") just created. We will explain the `'a` notation in the next section.

Here we see the element `1` being consed to the front of the list `[3; 5; 9]`:



Given a list, taking its first element (its "head") or its list of remaining elements (its "tail") also takes constant time. Each operation just follows a link. In the diagram above, the first down arrow leads to the head and the leftmost right arrow leads to the tail. Once we have the tail, its head is the second element of the original list, etc.

The tail is *not* the last element; it is the *list* of all elements other than the head!

3.2 Getting at the Head and Tail

```
In [26]: let null = function
           | [] -> true
           | x :: l -> false
```

```
Out[26]: val null : 'a list -> bool = <fun>
```

```
In [27]: null []
```

```
Out[27]: - : bool = true

In [28]: null [1; 2; 3]

Out[28]: - : bool = false

In [29]: let hd (x::l) = x

Out[29]: val hd : 'a list -> 'a = <fun>

In [30]: hd [1; 2; 3]

Out[30]: - : int = 1

In [31]: let tl (x::l) = l

Out[31]: val tl : 'a list -> 'a list = <fun>

In [32]: tl [7; 6; 5]

Out[32]: - : int list = [6; 5]
```

The empty list has neither head nor tail. Applying `hd` or `tl` to `[]` is an error—strictly speaking, an "exception". The function `null` can be used to check for the empty list beforehand. Taking a list apart using combinations of `hd` and `tl` is hard to get right. Fortunately, it is seldom necessary because of *pattern-matching*.

The declaration of `null` introduces a new concept known as "pattern matching", which we will explore more in subsequent lectures. For now, it is sufficient to observe that `let null = function` allows for matching on the two possible values that might be passed in as argument to `null` here: one for the empty list (for which it returns `true`) and one for non-empty lists (for which it returns `false`).

The declaration of `hd` above has only one clause, for non-empty lists. They have the form `x::l` and the function returns `x`, which is the head. If you compile this program, OCaml also prints a warning to tell us that calling the function could raise an exception because not all possible inputs are handled, including a counter-example (in this case, the empty list `[]`). The declaration of `tl` is similar to `hd`.

These three primitive functions are *polymorphic* and allow flexibility in the types of their arguments and results. Note their types!

```
In [33]: null

Out[33]: - : 'a list -> bool = <fun>

In [34]: hd

Out[34]: - : 'a list -> 'a = <fun>
```

```
In [35]: tl
```

```
Out[35]: - : 'a list -> 'a list = <fun>
```

Symbols `'a` and `'b` are called *type variables* and stand for any types. Code written using these functions is checked for type correctness at compile time. And this guarantees strong properties at run time, for example that the elements of any list all have the same type. They are usually read as their corresponding greek characters; `'a` is "alpha", `'b` is "beta", and so on.

3.3 Computing the Length of a List

```
In [36]: let rec nlength = function
         | [] -> 0
         | x :: xs -> 1 + nlength xs
```

```
Out[36]: val nlength : 'a list -> int = <fun>
```

```
In [37]: nlength []
```

```
Out[37]: - : int = 0
```

```
In [38]: nlength [5; 6; 7]
```

```
Out[38]: - : int = 3
```

$$\begin{aligned}\text{nlength } [a;b;c] &\Rightarrow 1 + \text{nlength } [b;c] \\ &\Rightarrow 1 + (1 + \text{nlength } [c]) \\ &\Rightarrow 1 + (1 + (1 + \text{nlength } [])) \\ &\Rightarrow 1 + (1 + (1 + 0)) \\ &\Rightarrow \ \ldots \ \ 3\end{aligned}$$

Most list processing involves recursion. This is a simple example; patterns can be more complex. Observe the use of a vertical bar `|` to separate the function's clauses. We have *one* function declaration that handles two cases. To understand its role, consider the following faulty code:

```
In [39]: let rec nlength [] = 0
```

```
Out[39]: val nlength : 'a list -> int = <fun>
```

```
In [40]: let rec nlength (x::xs) = 1 + nlength xs
```

```
Out[40]: val nlength : 'a list -> int = <fun>
```

These are two declarations, not one. First we declare `nlength` to be a function that handles only empty lists. Then we redeclare it to be a function that handles only non-empty lists; it can

never deliver a result. We see that a second `let` declaration replaces any previous one rather than extending it to cover new cases.

Now, let us return to our original declaration of `nlength`. The length function is *polymorphic* and applies to *all* lists regardless of element type! Most programming languages lack such flexibility.

Unfortunately, this length computation is naive and wasteful. Like `nsum` earlier, it is not tail-recursive. It uses $O(n)$ space, where n is the length of its input. As usual, the solution is to add an accumulating argument.

3.4 Efficiently Computing the Length of a List

```
In [41]: let rec addlen n = function
         | []    -> n
         | x::xs -> addlen (n + 1) xs

Out[41]: val addlen : int -> 'a list -> int = <fun>


In [42]: addlen 0 [5; 6; 7]

Out[42]: - : int = 3
```

Recall that the use of `function` introduces an extra (unnamed) argument that is pattern matched in the subsequent clauses; in this case, to break open the list.

$$\begin{aligned}\text{addlen } 0[a;b;c] &\Rightarrow \text{addlen } 1\ [b;c] \\ &\Rightarrow \text{addlen } 2\ [c] \\ &\Rightarrow \text{addlen } 3\ [] \\ &\Rightarrow 3\end{aligned}$$

Function `addlen` is again polymorphic. Its type mentions the integer accumulator.

Now we may declare an efficient length function. It is simply a wrapper for `addlen`, supplying zero as the initial value of n.

```
In [43]: let length xs = addlen 0 xs

Out[43]: val length : 'a list -> int = <fun>


In [44]: length [5; 6; 7; 8]

Out[44]: - : int = 4
```

The recursive calls do not nest: this version is iterative. It takes $O(1)$ space. Obviously its time requirement is $O(n)$ because it takes at least n steps to find the length of an n-element list.

3.5 Append: List Concatenation

```
In [45]: let rec append xs ys =
           match xs, ys with
           | [], ys    -> ys
           | x::xs, ys -> x :: append xs ys
```

```
Out[45]: val append : 'a list -> 'a list -> 'a list = <fun>
```

```
In [46]: append [1; 2; 3] [4]
```

```
Out[46]: - : int list = [1; 2; 3; 4]
```

```
In [47]: let (@) = append
```

```
Out[47]: val ( @ ) : 'a list -> 'a list -> 'a list = <fun>
```

```
In [48]: [1; 2; 3] @ [4]
```

```
Out[48]: - : int list = [1; 2; 3; 4]
```

Patterns can be as complicated as we like. Here, the two patterns are `[], ys` and `x::xs, ys`.

$$\begin{aligned}\text{append } [1;2;3][4] &\Rightarrow 1 :: \text{append } [2;3]\ [4] \\ &\Rightarrow 1 :: (2 :: \text{append } [3]\ [4]) \\ &\Rightarrow 1 :: (2 :: (3 :: \text{append } []\ [4])) \\ &\Rightarrow 1 :: (2 :: (3 :: [4]))\ [1;2;3;4]\end{aligned}$$

Here is how append might be declared, also noting that we have defined `@` as an infix operator that is a more convenient way to call `append` on two lists. However, this function is also not iterative. It scans its first argument, sets up a string of `cons` operations (`::`) and finally does them.

It uses $O(n)$ space and time, where n is the length of its first argument. *Its costs are independent of its second argument.*

An accumulating argument could make it iterative, but with considerable complication. The iterative version would still require $O(n)$ space and time because concatenation requires copying all the elements of the first list. Therefore, we cannot hope for asymptotic gains; at best we can decrease the constant factor involved in $O(n)$, but complicating the code is likely to increase that factor. Never add an accumulator merely out of habit.

Note append's polymorphic type. It tells us that two lists can be joined if their element types agree.

3.6 Reversing a List in $O(n^2)$

Let us consider one way to reverse a list.

```
In [49]: let rec nrev = function
         | [] -> []
         | x::xs -> (nrev xs) @ [x]
```

```
Out[49]: val nrev : 'a list -> 'a list = <fun>
```

```
In [50]: nrev [1; 2; 3]
```

```
Out[50]: - : int list = [3; 2; 1]
```

$$\begin{aligned}\text{nrev } [a;b;c] &\Rightarrow \text{nrev } [b;c] \mathbin{@} [a] \\ &\Rightarrow (\text{nrev } [c] \mathbin{@} [b]) \mathbin{@} [a] \\ &\Rightarrow ((\text{nrev } [] \mathbin{@} [c]) \mathbin{@} [b]) \mathbin{@} [a] \\ &\Rightarrow (([] \mathbin{@} [c]) \mathbin{@} [b]) \mathbin{@} [a] \;\ldots\; [c;b;a]\end{aligned}$$

This reverse function is grossly inefficient due to poor usage of `append`, which copies its first argument. If `nrev` is given a list of length $n > 0$, then append makes $n-1$ conses to copy the reversed tail. Constructing the list `[x]` calls `cons` again, for a total of n calls. Reversing the tail requires $n-1$ more conses, and so forth. The total number of conses is:

$$0 + 1 + 2 + \cdots + n = n(n+1)/2$$

The time complexity is therefore $O(n^2)$. Space complexity is only $O(n)$ because the copies don't all exist at the same time.

3.7 Reversing a List in $O(n)$

```
In [51]: let rec rev_app xs ys =
           match xs, ys with
           | [], ys    -> ys
           | x::xs, ys -> rev_app xs (x::ys)
```

```
Out[51]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

$$\begin{aligned}\text{rev\_app } [a;b;c]\ [] &\Rightarrow \text{rev\_app } [b;c]\ [a] \\ &\Rightarrow \text{rev\_app } [c]\ [b;a] \\ &\Rightarrow \text{rev\_app } []\ [c;b;a] \\ &\Rightarrow [c;b;a]\end{aligned}$$

Calling `rev_app xs ys` reverses the elements of `xs` and prepends them to `ys`. Now we may declare

```
In [52]: let rev xs = rev_app xs []
```

```
Out[52]: val rev : 'a list -> 'a list = <fun>
```

```
In [53]: rev [1; 2; 3]

Out[53]: - : int list = [3; 2; 1]
```

It is easy to see that this reverse function performs just n conses, given an n-element list. For both reverse functions, we could count the number of conses precisely—not just up to a constant factor. O notation is still useful to describe the overall running time: the time taken by a cons varies from one system to another.

The accumulator y makes the function iterative. But the gain in complexity arises from the removal of `append`. Replacing an expensive operation (append) by a series of cheap operations (cons) is called *reduction in strength* and is a common technique in computer science. It originated when many computers did not have a hardware multiply instruction; the series of products $i \times r$ for $i = 0, \ldots, n$ could more efficiently be computed by repeated addition. Reduction in strength can be done in various ways; we shall see many instances of removing append.

Consing to an accumulator produces the result in reverse. If that forces the use of an extra list reversal then the iterative function may be much slower than the recursive one.

3.8 Lists, Strings and Characters

Strings are provided in most programming languages to allow text processing. Strings are essential for communication with users. Even a purely numerical program formats its results ultimately as strings.

```
In [54]: 'a'   (* a character constant *)

Out[54]: - : char = 'a'

In [55]: "a"   (* a string constant of length 1 *)

Out[55]: - : string = "a"

In [56]: "abc" (* a string constant of length 3 *)

Out[56]: - : string = "abc"

In [57]: String.length "abc"

Out[57]: - : int = 3

In [58]: "abc" ^ "def"  (* concatenate two strings *)

Out[58]: - : string = "abcdef"
```

In a few programming languages, strings simply are lists of characters. In OCaml they are a separate type, unrelated to lists, reflecting the fact that strings are an abstract concept in themselves.

Similarly, characters are not strings of size one, but are a primitive concept. Character constants in OCaml have the form `'c'`, where *c* is any character. For example, the comma character is `','`.

Special characters are coded in strings using *escape sequences* involving the backslash character; among many others, a double quote is written `"\\"` and the newline character is written `"\n"`. For example, the string `"I\nLIKE\nCHEESE\n"` represents three text lines.

In addition to the operators described above, the relations `<`, `<=`, `>`, and `>=` work for strings and yield alphabetic order (more precisely, lexicographic order with respect to ASCII character codes).

3.8.1 Exercise 3.1

Code a recursive function to compute the sum of a list's elements. Then code an iterative version and comment on the improvement in efficiency.

3.8.2 Exercise 3.2

Code a function to return the last element of a non-empty list. How efficiently can this be done?

3.8.3 Exercise 3.3

Code a function to return the list consisting of the even-numbered elements of the list given as its argument. For example, given `[a; b; c; d]` it should return `[b; d]`.

3.8.4 Exercise 3.4

Consider the polymorphic types in these two function declarations:

```
In [59]: let id x = x

Out[59]: val id : 'a -> 'a = <fun>


In [60]: let rec loop x = loop x

Out[60]: val loop : 'a -> 'b = <fun>
```

Explain why these types make logical sense, preventing run time type errors, even for expressions like `id [id [id 0]]` or `loop true / loop 3`. (`/` is the integer division operator in OCaml)

3.8.5 Exercise 3.5

Code a function `tails` to return the list of the tails of its argument. For example, given `[1; 2; 3]` it should return `[[1; 2; 3]; [2; 3]; [3]; []]`.

Chapter 4

Lecture 4: More on Lists

4.1 List Utilities: take and drop

This lecture examines more list utilities, illustrating more patterns of recursion, and concludes with a small program for making change.

The functions `take` and `drop` divide a list into parts, returning or discarding the first i elements.

$$xs = [\underbrace{x_0, \ldots, x_{i-1}}_{\text{take i xs}}, \underbrace{x_i, \ldots, x_{n-1}}_{\text{drop i xs}}]$$

They can be implemented in OCaml as follows:

```
In [61]: let rec take i = function
           | [] -> []
           | x::xs ->
               if i > 0 then x :: take (i - 1) xs
               else []
```

```
Out[61]: val take : int -> 'a list -> 'a list = <fun>
```

```
In [62]: let rec drop i = function
           | [] -> []
           | x::xs ->
               if i > 0 then drop (i-1) xs
               else x::xs
```

```
Out[62]: val drop : int -> 'a list -> 'a list = <fun>
```

Applications of `take` and `drop` will appear in future lectures. Typically, they divide a collection of items into equal parts for recursive processing.

The `take` function is not iterative, but making it so would not improve its efficiency. The task requires copying up to i list elements, which must take $O(i)$ space and time.

Function `drop` simply skips over i list elements. This requires $O(i)$ time but only constant space. It is iterative and much faster than `take`. Both functions use $O(i)$ time, but skipping elements is faster than copying them: `drop`'s constant factor is smaller.

Both functions take an integer and a list, returning a list of the same type. So their type is `int -> 'a list -> 'a list`.

4.2 Linear Search

- find x in list $[x_1, \ldots, x_n]$ by comparing with each element
- obviously $O(n)$ time
- simple & general
- ordered searching needs only $O(\log n)$
- indexed lookup needs only $O(1)$

Linear search is the obvious way to find a desired item in a collection: simply look through all the items, one at a time. If x is in the list, then it will be found in $n/2$ steps on average, and even the worst case is obviously $O(n)$.

Large collections of data are usually ordered or indexed so that items can be found in $O(\log n)$ time, which is exponentially better than $O(n)$. Even $O(1)$ is achievable (using a hash table), though subject to the usual proviso that machine limits are not exceeded.

Efficient indexing methods are of prime importance: consider Web search engines. Nevertheless, linear search is often used to search small collections because it is so simple and general, and it is the starting point for better algorithms.

4.3 Equality Tests

```
In [63]: let rec member x = function
           | [] -> false
           | y::l ->
             if x = y then true
             else member x l
```

```
Out[63]: val member : 'a -> 'a list -> bool = <fun>
```

All the list functions we have encountered up to now have been "polymorphic", working for lists of any type. Function `member` uses linear search to report whether or not `x` occurs in `l`.

To do this generically, it uses a special feature of OCaml known as "polymorphic equality", which manifests itself via the `=`, `>=`, `<=`, `>` and `<` operators. These operators inspect the *structure* of the values using a consistent order. Types you can legitimately compare this way include integers, strings, booleans, and tuples or lists of primitive types.

More complex types can be compared this way within careful limits: recursive structures or function values will not work (we will cover function values in the Currying lecture later). For now, it is sufficient to use these magic polymorphic equality operators. As you get more familiar with OCaml and the use of higher order functions (also covered in a later lecture), you will encounter the use of explicit `compare` functions that are used to provide more complex equality tests.

The presence of polymorphic equality is a contentious feature in OCaml. While it provides a great ease of use in smaller codebases, it starts to become more dangerous when building larger OCaml-based systems. Most large-scale users of OCaml tend towards not using it in important code, but it is just fine for our purposes while learning the beginning steps of computer science.

4.4 Building a List of Pairs

```
In [64]: let rec zip xs ys =
           match xs, ys with
           | (x::xs, y::ys) -> (x, y) :: zip xs ys
           | _ -> []
```

```
Out[64]: val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

$$\left.\begin{matrix}[x_1, \ldots, x_n] \\ [y_1, \ldots, y_n]\end{matrix}\right\} \longmapsto [(x_1, y_1), \ldots, (x_n, y_n)]$$

The *wildcard* pattern `_` matches *anything*. We could have written a variable such as `p` instead, but the wildcard reminds us that the relevant clause ignores this argument.

The patterns are also tested in order of their definitions: first `(x::xs, y::ys)`, then `_`.

A list of pairs of the form $[(x_1, y_1), \ldots, (x_n, y_n)]$ associates each x_i with y_i. Conceptually, a telephone directory could be regarded as such a list, where x_i ranges over names and y_i over the corresponding telephone number. Linear search in such a list can find the y_i associated with a given x_i, or vice versa—very slowly.

In other cases, the (x_i, y_i) pairs might have been generated by applying a function to the elements of another list $[z_1, \ldots, z_n]$.

```
In [65]: let rec unzip = function
           | [] -> ([], [])
           | (x, y)::pairs ->
               let xs, ys = unzip pairs in
               (x::xs, y::ys)
```

```
Out[65]: val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
```

Given a list of pairs, `unzip` has to build *two* lists of results, which is awkward using recursion. The version shown above uses the *local binding* `let p =` E_1 `in` E_2, where the value of E_1 is bound to the pattern `P` within E_2. The let-construct counts as an expression and can be used (perhaps wrapped within parentheses) wherever an expression is expected.

Note especially the phrase `let xs, ys = unzip pairs` which binds `xs` and `ys` to the results of the recursive call. In general, the phrase `let P = E` matches the pattern P against the value of expression E. It binds all the variables in P to the corresponding values.

The functions `zip` and `unzip` build and take apart lists of pairs: `zip` pairs up corresponding list elements and `unzip` inverts this operation. Their types reflect what they do:

```
In [66]: zip

Out[66]: - : 'a list -> 'b list -> ('a * 'b) list = <fun>


In [67]: unzip

Out[67]: - : ('a * 'b) list -> 'a list * 'b list = <fun>
```

If the lists are of unequal length, `zip` discards surplus items at the end of the longer list. Its first pattern only matches a pair of non-empty lists. The second pattern is just a wildcard and could match anything. OCaml tries the clauses in the order given, so the first pattern is tried first. The second only gets arguments where at least one of the lists is empty.

4.5 Building a Pair of Results

Here is a version of `unzip` that replaces the local declaration by a function `conspair` for taking apart the pair of lists in the recursive call. It defines the same computation as the previous version of `unzip` and is possibly clearer, but not every local binding can be eliminated as easily.

```
In [68]: let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)

Out[68]: val conspair : ('a * 'b) * ('a list * 'b list) -> 'a list * 'b list = <fun>


In [69]: let rec unzip = function
           | [] -> ([], [])
           | xy :: pairs -> conspair (xy, unzip pairs)

Out[69]: val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
```

Making the function iterative yields `revUnzip` below, which is very simple. Iteration can construct many results at once in different argument positions. Both output lists are built in reverse order, which can be corrected by reversing the input to `revUnzip`. The total costs will probably exceed those of `unzip` despite the advantages of iteration.

```
In [70]: let rec revUnzip = function
           | ([], xs, ys) -> (xs, ys)
           | ((x, y)::pairs, xs, ys) ->
               revUnzip (pairs, x::xs, y::ys)

Out[70]: val revUnzip : ('a * 'b) list * 'a list * 'b list -> 'a list * 'b list =
           <fun>
```

4.6 An Application: Making Change

Consider a till that has unlimited supplies of coins. The largest coins should be tried first, to avoid giving change all in pennies. The list of legal coin values, called `till`, is given in descending order,

such as 50, 20, 10, 5, 2 and 1. (Recall that the head of a list is the element most easily reached.) The code for `change` is based on simple observations:

- Change for zero consists of no coins at all. (Note the pattern of `0` in the first clause.)
- For a nonzero amount, try the largest available coin. If it is small enough, use it and decrease the amount accordingly.
- Exclude from consideration any coins that are too large.

```
In [71]: let rec change till amt =
           match till, amt with
           | _, 0         -> []
           | [], _        -> raise (Failure "no more coins!")
           | c::till, amt -> if amt < c then change till amt
                             else c :: change (c::till) (amt - c)

Out[71]: val change : int list -> int -> int list = <fun>
```

Although nobody considers making change for zero, this is the simplest way to make the algorithm terminate. Most iterative procedures become simplest if, in their base case, they do nothing. A base case of one instead of zero is often a sign of a novice programmer.

- The recursion *terminates* when `amt = 0`.
- Tries the *largest coin first* to use large coins.
- The algorithm is *greedy* and can fail!

The function can terminate either with success or failure. It fails by raising exception `Failure` namely if `till` becomes empty while `amt` is still nonzero. (Exceptions will be discussed later.)

Unfortunately, failure can occur even when change can be made. The greedy "largest coin first" approach is to blame. Suppose we have coins of values 5 and 2, and must make change for 6; the only way is $6 = 2 + 2 + 2$, ignoring the 5. *Greedy algorithms* are often effective, but not here.

4.7 All Ways of Making Change

Now we generalise the problem to return the list of *all possible ways* of making change, and write a new `change` function.

```
In [72]: let rec change till amt =
           match till, amt with
           | _       , 0   -> [ [] ]
           | []      , _   -> []
           | c::till , amt -> if amt < c then change till amt
                              else let rec allc = function
                                     | [] -> []
                                     | cs :: css -> (c::cs) :: allc css
                                   in
                                     allc (change (c::till) (amt - c)) @
                                          change till amt

Out[72]: val change : int list -> int -> int list list = <fun>
```

Look at the type: the result is now a list of lists. The code will also never raise exceptions. It expresses failure by returning an empty list of solutions: it returns `[]` if the till is empty and the amount is nonzero.

If the amount is zero, then there is only one way of making change; the result should be `[[]]`. This is success in the base case.

In nontrivial cases, there are two sources of solutions: to use a coin (if possible) and decrease the amount accordingly, or to remove the current coin value from consideration.

The function `allc` is declared locally in order to make use of `c`, the current coin. It adds an extra `c` to all the solutions returned by the recursive call to make change for `amt - c`.

Observe the naming convention: `cs` is a list of coins, while `css` is a list of such lists. The trailing 's' is suggestive of a plural.

This complicated program, and the even trickier one on the next slide, are included as challenges. Are you enthusiastic enough to work them out? We shall revisit the "making change" task later to illustrate exception-handling.

4.8 All Ways of Making Change — Faster!

```
In [73]: let rec change till amt chg chgs =
           match till, amt with
           | _       , 0   -> chg::chgs
           | []      , _   -> chgs
           | c::till , amt -> if amt < 0 then chgs
                              else change (c::till) (amt - c) (c::chg)
                                          (change till amt chg chgs)

Out[73]: val change : int list -> int -> int list -> int list list -> int list list =
           <fun>
```

We've added *another* accumulating parameter! Repeatedly improving simple code is called *stepwise refinement*.

Two extra arguments eliminate many `::` and append operations from the previous slide's `change` function. The first, `chg`, accumulates the coins chosen so far; one evaluation of c::chg} replaces many evaluations of `allc`. The second, `chgs`, accumulates the list of solutions so far; it avoids the need for append. This version runs several times faster than the previous one.

Making change is still extremely slow for an obvious reason: the number of solutions grows rapidly in the amount being changed. Using 50, 20, 10, 5, 2 and 1, there are 4366 ways of expressing 99.

Our three change functions illustrate a basic technique: program development by stepwise refinement. Begin by writing a very simple program and add requirements individually. Add efficiency refinements last of all. Even if the simpler program cannot be included in the next version and has to be discarded, one has learned about the task by writing it.

4.8.1 Exercise 4.1

Sets can be represented in OCaml using lists containing no duplicated items (i.e. where no item is equal to another using polymorphic comparison).

Using the `member` function defined above, code a function to implement set union. It should avoid introducing repetitions, for example the union of the lists `[4; 7; 1]` and `[6; 4; 7]` should be `[1; 6; 4; 7]` (though the order does not matter).

4.8.2 Exercise 4.2

Code a function that takes a list of integers and returns two lists, the first consisting of all non-negative numbers found in the input and the second consisting of all the negative numbers.

4.8.3 Exercise 4.3

How does this version of `zip` differ from the one above?

```
In [74]: let rec zip xs ys =
           match xs, ys with
           | (x::xs, y::ys) -> (x, y) :: zip xs ys
           | ([], [])   -> []
```

```
Out[74]: val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

4.8.4 Exercise 4.4

What assumptions do the 'making change' functions make about the variables `till` and `amt`? Describe what could happen if these assumptions were violated.

4.8.5 Exercise 4.5

Show that the number of ways of making change for n (ignoring order) is $O(n)$ if there are two legal coin values. What if there are three, four, … coin values?

4.8.6 Exercise 4.6

We know nothing about the functions `f` and `g` other than their polymorphic types: `val f : 'a * 'b -> 'b * 'a` and `val g : 'a -> 'a list`. Suppose that `f (1, true)` and `g 0` are evaluated and return their results. State, with reasons, what you think the resulting values will be.

Chapter 5

Lecture 5: Sorting

A few applications for sorting and arranging items into order are:

- search
- merging
- duplicates
- inverting tables
- graphics algorithms

Sorting is perhaps the most deeply studied aspect of algorithm design. Knuth's series *The Art of Computer Programming* devotes an entire volume to sorting and searching! Sedgewick also covers sorting. Sorting has countless applications.

Sorting a collection allows items to be found quickly. Recall that linear search requires $O(n)$ steps to search among n items. A sorted collection admits *binary search* which requires only $O(\log n)$ time. The idea of binary search is to compare the item being sought with the middle item (in position $n/2$) and then to discard either the left half or the right, depending on the result of the comparison. Binary search needs arrays or trees, not lists; we shall come to binary search trees later.

Two sorted files can quickly be *merged* to form a larger sorted file. Other applications include finding *duplicates* that, after sorting, are adjacent.

A telephone directory is sorted alphabetically by name. The same information can instead be sorted by telephone number (useful to the police) or by street address (useful to junk-mail firms). Sorting information in different ways gives it different applications.

Common sorting algorithms include insertion sort, quicksort, mergesort and heapsort. We shall consider the first three of these. Each algorithm has its advantages.

As a concrete basis for comparison, runtimes are quoted for DECstation computers. These were based on the MIPS chip (an early RISC design) and are really old now, but still useful to look at relative to each other.

5.1 How Fast Can We Sort?

- typically count *comparisons* $C(n)$

- there are $n!$ permutations of n elements
- each comparison eliminates *half* of the permutations $2^{C(n)} \geq n!$
- therefore $C(n) \geq \log(n!) \approx n \log n - 1.44n$

The usual measure of efficiency for sorting algorithms is the number of comparison operations required. Mergesort requires only $O(n \log n)$ comparisons to sort an input of n items. It is straightforward to prove that this complexity is the best possible. There are $n!$ permutations of n elements and each comparison distinguishes two permutations. The lower bound on the number of comparisons, $C(n)$, is obtained by solving $2^{C(n)} \geq n!$; therefore $C(n) \geq \log(n!) \approx n \log n - 1.44n$.

In order to compare the sorting algorithms, we use the following source of pseudo-random numbers. Never mind how this works: generating statistically good random numbers is hard. Much effort has gone into those few lines of code.

```
In [75]: let nextrandom seed =
           let a = 16807.0 in
           let m = 2147483647.0 in
           let t = a *. seed in
           t -. m *. (floor (t /. m))
```

```
Out[75]: val nextrandom : float -> float = <fun>
```

```
In [76]: let rec randlist (seed, seeds) = function
           | 0 -> (seed, seeds)
           | n -> randlist (nextrandom seed, seed::seeds) (n-1)
```

```
Out[76]: val randlist : float * float list -> int -> float * float list = <fun>
```

We can now bind the identifier `rs` to a list of 10,000 random numbers.

```
In [77]: let seed, rs = randlist (1.0, []) 10000
```

```
Out[77]: val seed : float = 1043618065.
         val rs : float list =
           [1484786315.; 925166085.; 1614852353.; 721631166.; 173942219.; 1229443779.;
            789328014.; 570809709.; 1760109362.; 270600523.; 2108528931.; 16480421.;
            519782231.; 162430624.; 372212905.; 1954184989.; 898872741.; 1651521688.;
            1114791388.; 1325968501.; 1469981427.; 465437343.; 1732504088.;
            280054095.; 1924919450.; 1244369648.; 1524535715.; 706293012.;
            1372325856.; 1302473561.; 941382430.; 2137445578.; 1937168414.;
            1852570660.; 495231255.; 1092873378.; 140232191.; 328129841.; 632752255.;
            227857208.; 1616471915.; 719842438.; 1402481130.; 745001020.; 791471334.;
            2131048000.; 312659966.; 1389551813.; 443838892.; 854190041.; 741774068.;
            267473377.; 1372555293.; 1539748349.; 697860888.; 1261546017.; 734770781.;
            1512111397.; 813238415.; 1034499961.; 602256496.; 462191385.; 250718457.;
            246489360.; 295426232.; 468306241.; 877829533.; 1130589227.; 1914364883.;
            1479854970.; 878528585.; 1268712064.; 115837978.; 1803525169.; 689954646.;
            1174020926.; 651968560.; 391152461.; 1776325865.; 2015344107.; 246977673.;
            1381242649.; 1115030853.; 190703911.; 316761032.; 464218769.; 1537522160.;
            1958981931.; 390463588.; 224009597.; 235243732.; 620352731.; 1374109567.;
```

```
832140633.; 675075162.; 1296171190.; 2009054653.; 1534419747.; 145880482.;
1649432515.; 403989126.; 1112417244.; 1290575192.; 896661113.; 218545469.;
1002393512.; 2131316096.; 551979127.; 932010335.; 665881436.; 1975412808.;
639877791.; 1781707137.; 894518191.; 568004958.; 1331430214.; 629489848.;
183264178.; 162027282.; 464592882.; 93302056.; 1178713033.; 1401486247.;
1846150129.; 1646978216.; 1104441491.; 111995009.; 66193165.; 2038880392.;
79340676.; 871801051.; 967550305.; 2067810758.; 1600354198.; 1746626663.;
1516388116.; 1308870791.; 173082747.; 189881227.; 478010722.; 739707315.;
255334803.; 164203714.; 1893097038.; 1587694259.; 292950569.; 918323194.;
41453146.; 1217297445.; 256768724.; 586494122.; 586258194.; 660494391.;
507554325.; 699716071.; 672895139.; 76065072.; 1594869218.; 1439459639.;
641123634.; 1650611940.; 177447368.; 301427463.; 525804524.; 553672425.;
926899509.; 794676486.; 690277940.; 2115070333.; 1062048650.; 1653192448.;
1808855340.; 126475289.; 1028198214.; 1739565096.; 1515748830.;
427491435.; 319330584.; 666483848.; 854842154.; 1853528448.; 1975611245.;
1905343266.; 1229802342.; 1416055428.; 2091603253.; 1068308139.;
198239748.; 982076370.; 1094563396.; 44402415.; 889814989.; 290736902.;
417580014.; 1935788352.; 595665917.; 367638848.; 894945148.; 1868608068.;
317883051.; 941451621.; 1595942893.; 789094274.; 1150772108.; 422742112.;
1444245279.; 1273601104.; 256005435.; 1742330161.; 1514599036.;
956344512.; 2113041793.; 293237373.; 1386995194.; 1509339194.; 891946522.;
1020832915.; 592544922.; 1746311153.; 1471539715.; 143832370.;
2041568248.; 1039556199.; 1608726047.; 1205124472.; 2123533995.;
1560620058.; 1837598795.; 1028172251.; 98318742.; 1405510706.;
1047695837.; 59221314.; 1822176683.; 1096018886.; 1528104537.;
1270922857.; 812074106.; 291115596.; 795788616.; 638657646.; 2034314619.;
1527649272.; 156357479.; 1010056202.; 1139413443.; 1110927723.;
1216083346.; 846825145.; 2100385733.; 315213605.; 1629637749.;
1139833627.; 895118866.; 296359237.; 1361440746.; 1188627020.;
1964199872.; 166733080.; 54185744.; 575493576.; 1810324496.; 1765549585.;
53514233.; 747348448.; 61758907.; 1710119765.; 188311628.; 8827553.;
67975851.; 1808633248.; 1290488843.; 1264775607.; 1711469075.;
1537468597.; 706677101.; 518290019.; 190285086.; 157683412.; 985907152.;
1571668636.; 632570698.; 791081325.; 1773794197.; 1787141077.;
1727982894.; 794213057.; 633163306.; 682601940.; 1573439414.; 1041956036.;
1169697582.; 758914445.; 2096291761.; 1502226099.; 1665995955.;
948048264.; 1596326605.; 1816773893.; …]
```

5.2 Insertion Sort

An insert operation does $n/2$ comparisons on average.

```
In [78]: let rec ins x = function
           | [] -> [x]
           | y::ys -> if x <= y then x :: y :: ys
                      else y :: ins x ys
```

```
Out[78]: val ins : 'a -> 'a list -> 'a list = <fun>
```

Insertion sort takes $O(n^2)$ comparisons on average:

```
In [79]: let rec insort = function
             | [] -> []
             | x::xs -> ins x (insort xs)

Out[79]: val insort : 'a list -> 'a list = <fun>
```

Items from the input are copied one at a time to the output. Each new item is inserted into the right place so that the output is always in order.

We could easily write iterative versions of these functions, but to no purpose. Insertion sort is slow because it does $O(n^2)$ comparisons (and a lot of list copying), not because it is recursive. Its quadratic runtime makes it nearly useless: it takes 174 seconds for our example while the next-worst figure is 1.4 seconds.

Insertion sort is worth considering because it is easy to code and illustrates the concepts. Two efficient sorting algorithms, mergesort and heapsort, can be regarded as refinements of insertion sort.

5.3 Quicksort: The Idea

The Quicksort algorithm has the following flow:

- Choose a *pivot* element, a
- Divide to partition the input into two sublists:
  - those *at most* a in value
  - those *exceeding* a
- Conquer using recursive calls to sort the sublists
- Combine the sorted lists by appending one to the other

Quicksort was invented by Sir Anthony Hoare, who works at Microsoft Research, Cambridge. Quicksort works by *divide and conquer,* a basic algorithm design principle. Quicksort chooses from the input some value a, called the *pivot*. It partitions the remaining items into two parts: those $\leq a$, and those $> a$. It sorts each part recursively, then puts the smaller part before the greater.

The cleverest feature of Hoare's algorithm was that the partition could be done *in place* by exchanging array elements. Quicksort was invented before recursion was well known, and people found it extremely hard to understand. As usual, we shall consider a list version based on functional programming.

5.4 Quicksort: The Code

```
In [80]: let rec quick = function
             | [] -> []
             | [x] -> [x]
             | a::bs ->
```

```
    let rec part l r = function
      | [] -> (quick l) @ (a :: quick r)
      | x::xs ->
          if (x <= a) then
            part (x::l) r xs
          else
            part l (x::r) xs
    in
    part [] [] bs
```

```
Out[80]: val quick : 'a list -> 'a list = <fun>
```

Our OCaml quicksort copies the items. It is still pretty fast, and it is much easier to understand. It takes roughly 0.74 seconds to sort our list of random numbers.

The function declaration consists of three clauses. The first handles the empty list; the second handles singleton lists (those of the form `[x]`); the third handles lists of two or more elements. Often, lists of length up to five or so are treated as special cases to boost speed.

The locally declared function `part` partitions the input using `a` as the pivot. The arguments `l` and `r` accumulate items for the left ($\leq a$) and right ($> a$) parts of the input, respectively.

It is not hard to prove that quicksort does $n \log n$ comparisons, *in the average case* (see page 94 of Aho). With random data, the pivot usually has an average value that divides the input in two approximately equal parts. We have the recurrence $T(1) = 1$ and $T(n) = 2T(n/2) + n$, which is $O(n \log n)$. In our example, it is about 235 times faster than insertion sort.

In the worst case, quicksort's running time is quadratic! An example is when its input is almost sorted or reverse sorted. Nearly all of the items end up in one partition; work is not divided evenly. We have the recurrence $T(1) = 1$ and $T(n+1) = T(n) + n$, which is $O(n^2)$. Randomising the input makes the worst case highly unlikely.

5.5 Append-Free Quicksort

```
In [81]: let rec quik = function
           | ([], sorted) -> sorted
           | ([x], sorted) -> x::sorted
           | a::bs, sorted ->
             let rec part = function
               | l, r, [] -> quik (l, a :: quik (r, sorted))
               | l, r, x::xs ->
                   if x <= a then
                     part (x::l, r, xs)
                   else
                     part (l, x::r, xs)
             in
             part ([], [], bs)
```

```
Out[81]: val quik : 'a list * 'a list -> 'a list = <fun>
```

The list `sorted` accumulates the result in the *combine* stage of the quicksort algorithm. We have again used the standard technique for eliminating append. Calling `quik(xs, sorted)` reverses the elements of `xs` and prepends them to the list `sorted`.

Looking closely at `part`, observe that `quik(r, sorted)` is performed first. Then `a` is consed to this sorted list. Finally, `quik` is called again to sort the elements of `l`.

The speedup is significant. An imperative quicksort coded in Pascal (taken from Sedgewick) is just slightly faster than function `quik`. The near-agreement is surprising because the computational overheads of lists exceed those of arrays. In realistic applications, comparisons are the dominant cost and the overheads matter even less.

5.6 Merging Two Lists

Merge joins two sorted lists.

```
In [82]: let rec merge = function
           | [], ys -> ys
           | xs, [] -> xs
           | x::xs, y::ys ->
               if x <= y then
                 x :: merge (xs, y::ys)
               else
                 y :: merge (x::xs, ys)
```

```
Out[82]: val merge : 'a list * 'a list -> 'a list = <fun>
```

Generalises insert to two lists, and does at most $m+n-1$ comparisons.

Merging means combining two sorted lists to form a larger sorted list. It does at most $m+n$ comparisons, where m and n are the lengths of the input lists. If m and n are roughly equal then we have a fast way of constructing sorted lists; if $n = 1$ then merging degenerates to insertion, doing much work for little gain.

Merging is the basis of several sorting algorithms; we look at a divide-and-conquer one. Mergesort is seldom found in conventional programming because it is hard to code for arrays; it works nicely with lists. It divides the input (if non-trivial) into two roughly equal parts, sorts them recursively, then merges them.

Function `merge` is not iterative; the recursion is deep. An iterative version is of little benefit for the same reasons that apply to `append` in the earlier lecture on Lists.

5.7 Top-down Merge sort

```
In [83]: let rec tmergesort = function
           | [] -> []
           | [x] -> [x]
           | xs ->
               let k = List.length xs / 2 in
               let l = tmergesort (take k xs) in
```

```
        let r = tmergesort (drop k xs) in
        merge (l, r)
```

```
Out[83]: val tmergesort : 'a list -> 'a list = <fun>
```

$O(n \log n)$ comparisons in worst case

Mergesort's *divide* stage divides the input not by choosing a pivot (as in quicksort) but by simply counting out half of the elements. The *conquer* stage again involves recursive calls, and the *combine* stage involves merging. Function `tmergesort` takes roughly 1.4 seconds to sort the list `rs`.

In the worst case, mergesort does $O(n \log n)$ comparisons, with the same recurrence equation as in quicksort's average case. Because `take` and `drop` divide the input in two equal parts (they differ at most by one element), we always have $T(n) = 2T(n/2) + n$.

Quicksort is nearly 3 times as fast in the example. But it risks a quadratic worst case! Merge sort is safe but slow. So which algorithm is best?

We have seen a *top-down* mergesort. *Bottom-up* algorithms also exist. They start with a list of one-element lists and repeatedly merge adjacent lists until only one is left. A refinement, which exploits any initial order among the input, is to start with a list of increasing or decreasing runs of input items.

5.8 Summary of Sorting Algorithms

- Optimal is $O(n \log n)$ comparisons
- Insertion sort: simple to code; too slow (quadratic) [174 secs]
- Quicksort: fast on average; quadratic in worst case [0.53 secs]
- Mergesort: optimal in theory; often slower than quicksort [1.4 secs]
- *Match the algorithm to the application*

Quicksort's worst case cannot be ignored. For large n, a complexity of $O(n^2)$ is catastrophic. Mergesort has an $O(n \log n)$ worst case running time, which is optimal, but it is typically slower than quicksort for random data.

Non-comparison sorting deserves mentioning. We can sort a large number of small integers using their radix representation in $O(n)$ time. This result does not contradict the comparison-counting argument because comparisons are not used at all. Linear time is achievable only if the greatest integer is fixed in advance; as n goes to infinity, increasingly many of the items are the same. It is a simple special case.

Many other sorting algorithms exist. A few are outlined in the exercises.

5.8.1 Exercise 5.1

Another sorting algorithm (selection sort) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements. State, with justification, the time complexity of this approach.

5.8.2 Exercise 5.2

Implement selection sort (see previous exercise) using OCaml.

5.8.3 Exercise 5.3

Another sorting algorithm (bubble sort) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible. State, with justification, the time complexity of this approach.

5.8.4 Exercise 5.4

Implement bubble sort (see previous exercise) using OCaml.

Chapter 6

Lecture 6: Datatypes and Trees

6.1 An Enumeration Type

We will now learn how to define more expressive types than the basic ones supplied with the core OCaml language.

```
In [84]: type vehicle = Bike
                      | Motorbike
                      | Car
                      | Lorry

Out[84]: type vehicle = Bike | Motorbike | Car | Lorry
```

- We have declared a *new type* named `vehicle`.
- ... along with four new constants.
- They are the *constructors* of the datatype.

The `type` declaration adds a new type to our OCaml session. Type `vehicle` is as good as any built-in type and even admits pattern-matching (as we used with the built-in list types earlier). The four new identifiers of type `vehicle` are called *constructors*.

We could represent the various vehicles by the numbers 0–3. However, the code would be hard to read and even harder to maintain. Consider adding `Tricycle` as a new vehicle. If we wanted to add it before `Bike`, then all the numbers would have to be changed. Using `type`, such additions are trivial and the compiler can (at least sometimes) warn us when it encounters a function declaration that doesn't yet have a case for `Tricycle`.

Representing vehicles by strings like `"Bike"`, `"Car"`, etc., is also bad. Comparing string values is slow and the compiler can't warn us of misspellings like `"MOtorbike"`: they will make our code fail.

Most programming languages allow the declaration of types like `vehicle`. Because they consist of a series of identifiers, they are called *enumeration types*. Other common examples are days of the week or colours. The compiler chooses the integers for us; type-checking prevents us from confusing `Bike` with `Red` or `Sunday`.

6.2 Declaring a Function on Vehicles

```
In [85]: let wheels = function
         | Bike -> 2
         | Motorbike -> 2
         | Car -> 4
         | Lorry -> 18
```

```
Out[85]: val wheels : vehicle -> int = <fun>
```

- Datatype constructors can be used in patterns.
- Pattern-matching is fast, even complicated nested patterns.
- Notice the type of the argument is `vehicle`, which we defined earlier.

The beauty of datatype declarations is that the new types behave as if they were built into OCaml. Type-checking catches common errors, such as mixing up different datatypes in a function like `wheels`, as well as missing and redundant patterns.

6.3 A Datatype whose Constructors have Arguments

```
In [86]: type vehicle = Bike
                      | Motorbike of int
                      | Car       of bool
                      | Lorry     of int
```

```
Out[86]: type vehicle = Bike | Motorbike of int | Car of bool | Lorry of int
```

- Constructors with arguments (like `Lorry`) are *distinct values*. (So `Car true` is distinct from `Car false`).
- Different kinds of `vehicle` can belong to one list: `[Bike, Car true, Motorbike 450]`

OCaml generalises the notion of enumeration type to allow data to be associated with each constructor. The constructor `Bike` is a vehicle all by itself, but the other three constructors create vehicles from arguments.

Since we might find it hard to remember what the various `int` and `bool` components are for, it is wise to include *comments* in complex declarations. In OCaml, comments are enclosed in the brackets `(*` and `*)`. Programmers should comment their code to explain design decisions and key features of the algorithms (sometimes by citing a reference work).

```
In [87]: type vehicle = Bike
                      | Motorbike of int  (* engine size in CCs *)
                      | Car       of bool (* true if a Reliant Robin *)
                      | Lorry     of int  (* number of wheels *)
```

```
Out[87]: type vehicle = Bike | Motorbike of int | Car of bool | Lorry of int
```

The list shown on the slide represents a bicycle, a Reliant Robin and a large motorbike. It can be almost seen as a mixed-type list containing integers and booleans. It is actually a list of vehicles;

datatypes lessen the impact of the restriction that all list elements must have the same type.

6.4 A Finer Wheel Computation

We now define a `wheels` function to calculate the number of wheels in any `vehicle`. This requires pattern matching to retrieve the constructors and their associated arguments, around which we build the logic:

```
In [88]: let wheels = function
         | Bike -> 2
         | Motorbike _ -> 2
         | Car robin -> if robin then 3 else 4
         | Lorry w -> w
```

```
Out[88]: val wheels : vehicle -> int = <fun>
```

This function consists of four clauses:

- A Bike has two wheels. This is a constant result.
- A Motorbike has two wheels. The `_` signifies a "wildcard" pattern match that we discard, since the engine size of the bike is not relevant to the number of wheels.
- A Reliant Robin has three wheels; all other cars have four. We bind `robin` to the `bool` argument and then use it in the right hand side of the pattern match, much like a `let` binding in normal code.
- A Lorry has the number of wheels stored with its constructor, and we simply return that.

There is no overlap between the `Motorbike` and `Lorry` cases. Although `Motorbike` and `Lorry` both hold an integer, OCaml takes the constructor into account and keeps any `Motorbike` distinct from any `Lorry`.

Vehicles are one example of a concept consisting of several varieties with distinct features. Most programming languages can represent such concepts using something analogous to datatypes. (They are sometimes called *union types* or *variant records* whose *tag fields* play the role of the constructors.)

A pattern may be built from the constructors of several datatypes, including lists. A pattern may also contain integer and string constants. There is no limit to the size of patterns or the number of clauses in a function declaration. OCaml performs pattern-matching efficiently (you do not need to understand the details of how it optimises pattern-matching at this stage).

6.5 Error Handling: Exceptions

During a computation, what happens if something goes *wrong?*

- Division by zero
- Pattern matching failure

Exception-handling lets us recover gracefully.

- Raising an exception abandons the current computation.
- Handling the exception attempts an alternative computation.

- The raising and handling can be far apart in the code.
- Errors of *different sorts* can be handled separately.

Exceptions are necessary because it is not always possible to tell in advance whether or not a search will lead to a dead end or whether a numerical calculation will encounter errors such as overflow or divide by zero. Rather than just crashing, programs should check whether things have gone wrong, and perhaps attempt an alternative computation (perhaps using a different algorithm or higher precision). A number of modern languages provide exception handling.

6.6 Exceptions in OCaml

```
In [89]: exception Failure
```

```
Out[89]: exception Failure
```

```
In [90]: exception NoChange of int
```

```
Out[90]: exception NoChange of int
```

```
In [91]: raise Failure
```

```
Exception: Failure.
```

Each `exception` declaration introduces a distinct sort of exception, which can be handled separately from others. If E raises an exception, then its evaluation has failed; *handling* an exception means evaluating another expression and returning its value instead. One exception handler can specify separate expressions for different sorts of exceptions.

Exception names are *constructors* of the special datatype `exn`. This is a peculiarity of OCaml that lets exception-handlers use pattern-matching. Note that exception `Failure` is just an error indication, while `NoChange n` carries further information: the integer n.

```
In [92]: try
           print_endline "pre exception";
           raise (NoChange 1);
           print_endline "post exception";
         with
           | NoChange _ ->
               print_endline "handled a NoChange exception"
```

```
pre exception


handled a NoChange exception


Out[92]: - : unit = ()
```

The effect of `raise <expr>` is to jump to the most recently-encountered handler that matches `<expr>`. The matching handler can only be found *dynamically* (during execution); contrast with how OCaml associates occurrences of identifiers with their matching declarations, which does not require running the program. A handler is introduced via the `try` keyword, which executes the subexpression and dispatches any exceptions encountered to the corresponding pattern match for exceptions defined in the `with` block.

This is also the first time that we have encountered the `unit` type. This represents a type that has no values, and is used to indicate that a block has no meaningful return value. We will come back to this when learning more about *imperative* programming later on. For now, it is sufficient to understand that `print_endline` will print out the argument to the console output, and return a `unit` type. The semicolon at the end of the expression is a convenient way to execute sequential statements that return the `unit` type.

One criticism of OCaml's exceptions is that—unlike the Java language—nothing in a function declaration indicates which exceptions it might raise. One alternative to exceptions is to instead return a value of datatype `option`.

```
In [93]: let x = Some 1
```

```
Out[93]: val x : int option = Some 1
```

```
In [94]: let y = None
```

```
Out[94]: val y : 'a option = None
```

```
In [95]: type 'a option = None | Some of 'a
```

```
Out[95]: type 'a option = None | Some of 'a
```

`None` signifies an error, while `Some x` returns the solution x. This approach looks clean, but the drawback is that many places in the code would have to check for `None`. Despite this, there is a builtin `option` type in OCaml as it is so useful. We will see in later lectures how to define our own version of `option` using polymorphic datatype definitions.

6.7 Making Change with Exceptions

```
In [96]: exception Change
         let rec change till amt =
           match till, amt with
           | _, 0         -> []
           | [], _        -> raise Change
           | c::till, amt -> if amt < 0 then raise Change
                             else try c :: change (c::till) (amt - c)
                                  with Change -> change till amt
```

```
Out[96]: exception Change
```

```
Out[96]: val change : int list -> int -> int list = <fun>
```

In the Lists lectures, we considered the problem of making change. The greedy algorithm presented there could not express "6 using 5 and 2" because it always took the largest coin. Returning the list of all possible solutions avoids that problem rather expensively: we only need one solution.

Using exceptions, we can code a *backtracking* algorithm: one that can undo past decisions if it comes to a dead end. The exception `Change` is raised if we run out of coins (with a non-zero amount) or if the amount goes negative. We always try the largest coin, but enclose the recursive call in an exception handler, which undoes the choice if it goes wrong.

Carefully observe how exceptions interact with recursion. The exception handler always undoes the *most recent* choice, leaving others possibly to be undone later. If making change really is impossible, then eventually `exception Change` will be raised with no handler to catch it, and it will be reported at top level.

6.8 Making Change: A Trace

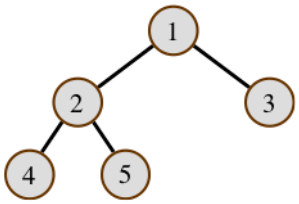
Here is the full execution. Observe how the exception handlers nest and how they drop away once the given expression has returned a value.

```
change [5; 2] 6 ⇒ try 5::change [5; 2] 1
                     with Change -> change [2] 6
                ⇒ try 5::(try 5::change [5; 2] (-4)
                     with Change -> change [2] 1)
                     with Change -> change [2] 6
                ⇒ 5::(change [2] 1)
                     with Change -> change [2] 6
                ⇒ try 5::(try 2::change [2] (-1)
                     with Change -> change [] 1)
                     with Change -> change [2] 6
                ⇒ try 5::(change [] 1)
                     with Change -> change [2] 6
                ⇒ change [2] 6
                ⇒ try 2::change [2] 4
                     with Change -> change [] 6
                ⇒ try 2::(try 2::change [2] 2
                     with Change -> change [] 4)
                     with Change -> change [] 6
                ⇒ try 2::(try 2::(try 2::change [2] 0
                     with Change -> change [] 2)
                     with Change -> change [] 4)
                     with Change -> change [] 6
                ⇒ try 2::(try 2::[2]
                     with Change -> change [] 4)
                     with Change -> change [] 6
                ⇒ try 2::[2; 2]
                     with Change -> change [] 6
                ⇒ [2; 2; 2]
```

6.9 Binary Trees, a Recursive Datatype

```
In [97]: type 'a tree =
           Lf
         | Br of 'a * 'a tree * 'a tree
```

```
Out[97]: type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```




```
In [98]: Br(1, Br(2, Br(4, Lf, Lf),
                     Br(5, Lf, Lf)),
                Br(3, Lf, Lf))
```

```
Out[98]: - : int tree = Br (1, Br (2, Br (4, Lf, Lf), Br (5, Lf, Lf)), Br (3, Lf, Lf))
```

A data structure with multiple branching is called a "tree". Trees can represent mathematical expressions, logical formulae, computer programs, the phrase structure of English sentences, etc.

Binary trees are nearly as fundamental as lists. They can provide efficient storage and retrieval of information. In a binary tree, each node is empty (Lf), or is a branch (Br) with a label and two subtrees.

OCaml lists are a datatype and could be declared as follows:

```
In [99]: type 'a mylist =
         | Nil
         | Cons of 'a * 'a mylist
```

```
Out[99]: type 'a mylist = Nil | Cons of 'a * 'a mylist
```

We could even declare `::` as an infix constructor. The only thing we could not define is the `[...]` notation, which is part of the OCaml grammar (although there does exist a mechanism to use a *similar* syntax for custom indexed datatypes).

A recursive type does not have to be polymorphic. For example, here is a simple datatype of tree shapes with no attached data that is recursive but not polymorphic.

```
In [100]: type shape =
          | Null
          | Join of shape * shape
```

```
Out[100]: type shape = Null | Join of shape * shape
```

The datatype `'a option` (mentioned above) is the opposite – it is polymorphic, but not recursive.

6.10 Basic Properties of Binary Trees

```
In [101]: let rec count = function
          | Lf -> 0  (* number of branch nodes *)
          | Br (v, t1, t2) -> 1 + count t1 + count t2
```

```
Out[101]: val count : 'a tree -> int = <fun>
```

```
In [102]: let rec depth = function
          | Lf -> 0  (* length of longest path *)
          | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)
```

```
Out[102]: val depth : 'a tree -> int = <fun>
```

The invariant $\texttt{count}(t) \leq 2^{\texttt{depth}(t)} - 1$ holds in the functions above.

Functions on trees are expressed recursively using pattern-matching. Both functions above are analogous to `length` on lists. Here is a third measure of a tree's size:

```
In [103]: let rec leaves = function
          | Lf -> 1
          | Br (v, t1, t2) -> leaves t1 + leaves t2
```

```
Out[103]: val leaves : 'a tree -> int = <fun>
```

This function is redundant because of a basic fact about trees, which can be proved by induction: for every tree t, we have $\texttt{leaves}(t) = \texttt{count}(t) + 1$. The inequality shown on the slide also has an elementary proof by induction.

A tree of depth 20 can store $2^{20} - 1$ or approximately one million elements. The access paths to these elements are short, particularly when compared with a million-element list!

6.10.1 Exercise 6.1

Give the declaration of an OCaml type for the days of the week. Comment on the practicality of such a type in a calendar application.

6.10.2 Exercise 6.2

Write an OCaml function taking a binary tree labelled with integers and returning their sum.

6.10.3 Exercise 6.3

Using the definition of `'a tree` from before:

```
In [104]: type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

```
Out[104]: type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

Examine the following function declaration. What does `ftree (1, n)` accomplish?

```
In [105]: let rec ftree k n =
            if n = 0 then Lf
            else Br (k, ftree (2 * k) (n - 1), ftree (2 * k + 1) (n - 1))
```

```
Out[105]: val ftree : int -> int -> int tree = <fun>
```

6.10.4 Exercise 6.4

Give the declaration of an OCaml type for arithmetic expressions that have the following possibilities: floating-point numbers, variables (represented by strings), or expressions of the form $-E$, $E + E$, $E \times E$.

6.10.5 Exercise 6.5

Continuing the previous exercise, write a function that evaluates an expression. If the expression contains any variables, your function should raise an exception indicating the variable name.

Chapter 7

Lecture 7: Dictionaries and Functional Arrays

7.1 Dictionaries

- lookup: find an item in the dictionary
- update (insert): replace (store) an item in the dictionary
- delete: remove an item from the dictionary
- empty: the null dictionary
- Missing: exception for errors in `lookup` and `delete`

Ideally, an *abstract type* should provide these operations and hide the internal data structures.

A dictionary attaches values to identifiers, called "keys". Before choosing the internal representation for a data structure, you need to specify the full set of operations. In fact, here we only consider `update` (associating a value with an identifier) and `lookup` (retrieving such a value). Deletion is more difficult and would limit our choices. Some applications may need additional operations, such as `merge` (combining two dictionaries). We shall see that update can be done efficiently in a functional style, without excessive copying.

An *abstract type* provides specified operations while hiding low-level details, such as the data structure used to represent dictionaries. Abstract types can be declared in any modern programming language. Java's *objects* serve this role, as do OCaml's modules. This course does not cover modules, and we simply declare the dictionary operations individually.

An *association list* (a list of pairs) is the simplest dictionary representation. Lookup is by linear search, and therefore slow: $O(n)$. Association lists are only usable if there are few keys in use. However, they are general in that the keys do not need a concept of ordering, only equality.

```
In [106]: exception Missing

Out[106]: exception Missing


In [107]: let rec lookup a = function
          | [] -> raise Missing
          | (x, y) :: pairs ->
```

```
            if a = x then y
            else lookup a pairs
```

```
Out[107]: val lookup : 'a -> ('a * 'b) list -> 'b = <fun>
```

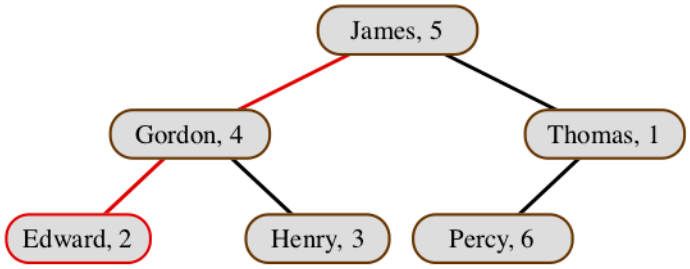
```
In [108]: let update (l, b, y) = (b, y) :: l
```

```
Out[108]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list = <fun>
```

To enter a new `(key, value)` pair, simply "cons" it to the list with `update`. This takes constant time, which is the best we could hope for. But the space requirement is huge: linear in the number of updates, not in the number of distinct keys. Obsolete entries are never deleted: that would require first finding them, increasing the update time from $O(1)$ to $O(n)$.

7.2 Binary Search Trees

A *dictionary* associates *values* (here, numbers) with *keys*.



Binary search trees are an important application of binary trees. They work for keys that have a total ordering, such as strings. Each branch of the tree carries a $(key, value)$ pair; its left subtree holds smaller keys; the right subtree holds greater keys. If the tree remains reasonably balanced, then update and lookup both take $O(\log n)$ for a tree of size n. These times hold in the average case; given random data, the tree is likely to remain balanced.

At a given node, all keys in the left subtree are smaller (or equal) while all trees in the right subtree are greater.

An unbalanced tree has a linear access time in the worst case. Examples include building a tree by repeated insertions of elements in increasing or decreasing order; there is a close resemblance to quicksort. Building a binary search tree, then converting it to inorder, yields a sorting algorithm called *treesort*.

Self-balancing trees, such as Red-Black trees, attain $O(\log n)$ in the worst case. They are complicated to implement.

7.3 Lookup: Seeks Left or Right

```
In [109]: exception Missing of string
```

```
Out[109]: exception Missing of string
```

```
In [110]: let rec lookup b = function
          | Br ((a, x), t1, t2) ->
              if b < a then
                lookup b t1
              else if a < b then
                lookup b t2
              else
                x
          | Lf -> raise (Missing b)
```

```
Out[110]: val lookup : string -> (string * 'a) tree -> 'a = <fun>
```

This has guaranteed $O(\log n)$ access time *if* the tree is balanced!

Lookup in the binary search tree goes to the left subtree if the desired key is smaller than the current one and to the right if it is greater. It raises `Missing` if it encounters an empty tree.

Since an ordering is involved, we have to declare the functions for a specific type, here `string`. Now exception `Missing` mentions that type: if lookup fails, the exception returns the missing key. The exception could be eliminated using type `option` of our earlier Datatypes lecture, using the constructor `None` for failure.

7.4 Update

```
In [111]: let rec update k v = function
          | Lf -> Br ((k, v), Lf, Lf)
          | Br ((a, x), t1, t2) ->
              if k < a then
                Br ((a, x), update k v t1, t2)
              else if a < k then
                Br ((a, x), t1, update k v t2)
              else (* a = k *)
                Br ((a, v), t1, t2)
```

```
Out[111]: val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
```

This is also $O(\log n)$ as it copies the path only, and *not whole subtrees!*

If you are familiar with the usual update operation for this sort of tree, you may wonder whether it can be implemented in OCaml, where there is no direct way to replace part of a data structure by something else.

The update operation is a nice piece of functional programming. It searches in the same manner as `lookup`, but the recursive calls reconstruct a new tree around the result of the update. One subtree is updated and the other left unchanged. The internal representation of trees ensures that unchanged parts of the tree are not copied, but *shared*. Therefore, update copies only the path from the root to the new node. Its time and space requirements, for a reasonably balanced tree, are both $O(\log n)$.

The comparison between b and a allows three cases: - smaller: update the left subtree; share the right - greater: update the right subtree; share the left - equal: update the label and share both subtrees

Note: in the function definition, `(* a = b*)` is a comment. Comments in OCaml are enclosed in the brackets `(*` and `*)`.

7.5 Aside: Traversing Trees (3 Methods)

```
In [112]: let rec preorder = function
          | Lf -> []
          | Br (v, t1, t2) ->
              [v] @ preorder t1 @ preorder t2
```

```
Out[112]: val preorder : 'a tree -> 'a list = <fun>
```

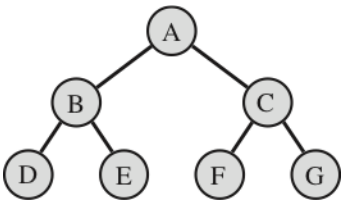
```
In [113]: let rec inorder = function
          | Lf -> []
          | Br (v, t1, t2) ->
              inorder t1 @ [v] @ inorder t2
```

```
Out[113]: val inorder : 'a tree -> 'a list = <fun>
```

```
In [114]: let rec postorder = function
          | Lf -> []
          | Br (v, t1, t2) ->
              postorder t1 @ postorder t2 @ [v]
```

```
Out[114]: val postorder : 'a tree -> 'a list = <fun>
```

Tree traversal means examining each node of a tree in some order. D. E. Knuth has identified three forms of tree traversal: preorder, inorder and postorder. We can code these "visiting orders" as functions that convert trees into lists of labels. Algorithms based on these notions typically perform some action at each node; the functions above simply copy the nodes into lists. Consider the tree:



- `preorder` visits the label first ("Polish notation"), yielding `ABDECFG`
- `inorder` visits the label midway, yielding `DBEAFCG`
- `postorder` visits the label last ("Reverse Polish"), yielding `DEBFGCA`. You might be familiar with this concept if you own an ancient RPN calculator!

What is the use of `inorder`? Consider applying it to a binary search tree: the result is a sorted list of pairs. We could use this, for example, to merge two binary search trees. It is not difficult to transform a sorted list of pairs into a binary search tree.

7.6 Efficiently Traversing Trees

Unfortunately, the functions shown on the previous slide are quadratic in the worst case: the appends in the recursive calls are inefficient. To correct that problem, we (as usual) add an accumulating argument. Observe how each function constructs its result list and compare with how appends were eliminated from `quicksort` in the Sorting lecture.

```
In [115]: let rec preord = function
          | Lf, vs -> vs
          | Br (v, t1, t2), vs ->
              v :: preord (t1, preord (t2, vs))

Out[115]: val preord : 'a tree * 'a list -> 'a list = <fun>
```

```
In [116]: let rec inord = function
          | Lf, vs -> vs
          | Br (v, t1, t2), vs ->
              inord (t1, v::inord (t2, vs))

Out[116]: val inord : 'a tree * 'a list -> 'a list = <fun>
```

```
In [117]: let rec postord = function
          | Lf, vs -> vs
          | Br (v, t1, t2), vs ->
              postord (t1, postord (t2, v::vs))

Out[117]: val postord : 'a tree * 'a list -> 'a list = <fun>
```

One can prove equations relating each of these functions to its counterpart on the previous section. For example:

$$\texttt{inord}(t, vs) = \texttt{inorder}(t)@vs$$

These three types of tree traversal are related in that all are depth-first. They each traverse the left subtree in full before traversing the right subtree. Breadth-first search (from the Queues lecture) is another possibility. That involves going through the levels of a tree one at a time.

7.7 Arrays

- A conventional array is an indexed storage area.
  - It is updated *in place* by the command `a.(k) <- x`
  - The concept is inherently *imperative*.
- A *functional array* is a finite map from integers to data.
  - Updating implies *copying* to return `update(A, k, x)`
  - The new array equals `A` except that `A.(k) = x`.
- Can we do updates efficiently?

The elements of a list can only be reached by counting from the front. Elements of a tree are reached by following a path from the root. An *array* hides such structural matters; its elements are uniformly designated by number. Immediate access to arbitrary parts of a data structure is called *random access*.

Arrays are the dominant data structure in conventional programming languages. The ingenious use of arrays is the key to many of the great classical algorithms, such as Hoare's original quicksort (the partition step) and Warshall's transitive-closure algorithm.

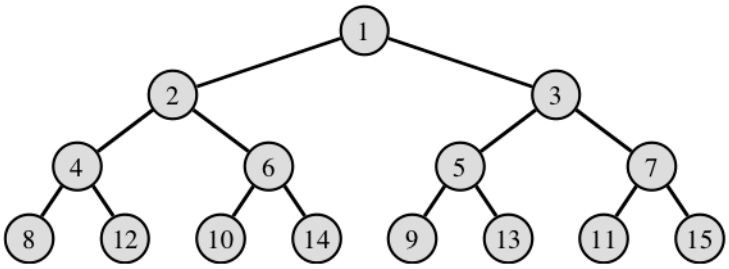
The drawback is that subscripting is a chief cause of programmer error. That is why arrays play little role in this introductory course.

Functional arrays are described below in order to illustrate another way of using trees to organise data. Here is a summary of basic dictionary data structures in order of decreasing generality and increasing efficiency:

- Linear search: Most general, needing only equality on keys, but inefficient: linear time.
- Binary search: Needs an ordering on keys. Logarithmic access time in the average case, but our binary search trees are linear in the worst case.
- Array subscripting: Least general, requiring keys to be integers, but even worst-case time is logarithmic.

7.8 Functional Arrays as Binary Trees

The path to element i follows the *binary code* for i (its "subscript").



This simple representation (credited to W. Braun) ensures that the tree is balanced. Complexity of access is always $O(\log n)$, which is optimal. For actual running time, access to conventional arrays is much faster: it requires only a few hardware instructions. Array access is often taken to be $O(1)$, which (as always) presumes that hardware limits are never exceeded.

The lower bound for array subscripts (or "indices' ') is one. The upper bound starts at zero (which signifies the empty array) and can grow without limit. Inspection of the diagram above should make it clear that these trees are always balanced: the left subtree can have at most one node more than the right subtree, recursively all the way down. (This assumes that the array is defined for subscripts $1 \ldots n$ with no gaps; an array defined only for odd numbers, for example, would obviously be unbalanced.)

The numbers in the diagram above are not the labels of branch nodes, but indicate the positions of array elements. For example, the label corresponding to $A[2]$ is at the position shown. The nodes of a functional array are labelled with the data we want to store, not with these integers.

7.9 The Lookup Function

```
In [118]: exception Subscript
          let rec sub = function
          | Lf, _ -> raise Subscript  (* Not found *)
          | Br (v, t1, t2), k ->
              if k = 1 then v
              else if k mod 2 = 0 then
                sub (t1, k / 2)
              else
                sub (t2, k / 2)
```

```
Out[118]: exception Subscript
```

```
Out[118]: val sub : 'a tree * int -> 'a = <fun>
```

```
In [119]: let rec sub = function (* Alternative implementation *)
          | Lf, _ -> raise Subscript
          | Br (v, t1, t2), 1 -> v
          | Br (v, t1, t2), k when k mod 2 = 0 -> sub (t1, k / 2)
          | Br (v, t1, t2), k -> sub (t2, k / 2)
```

```
Out[119]: val sub : 'a tree * int -> 'a = <fun>
```

Notice that we have used a new keyword `when` above, which changes pattern clauses to be only matched if the expression evalutes to true. This can be equivalently expressed by moving the corresponding checks into an `if` clause on the right hand side of the pattern match, but is often more readable using `when` (as above).

The lookup function `sub`, divides the subscript by 2 until 1 is reached. If the remainder is 0 then the function follows the left subtree, otherwise the right. If it reaches a leaf, it signals error by raising exception `Subscript`.

Array access can also be understood in terms of the subscript's binary code. Because the subscript must be a positive integer, in binary it has a leading one. Discard this one and reverse the remaining bits. Interpreting zero as *left* and one as *right* yields the path from the root to the subscript.

Popular literature often explains the importance of binary as being led by hardware: because a circuit is either on or off. The truth is almost the opposite. Designers of digital electronics go to a lot of trouble to suppress the continuous behaviour that would naturally arise. The real reason why binary is important is its role in algorithms: an `if-then-else` decision leads to binary branching.

Data structures, such as trees, and algorithms, such as mergesort, use binary branching in order to reduce a cost from $O(n)$ to $O(\log n)$. Two is the smallest integer divisor that achieves this reduction. (Larger divisors are only occasionally helpful, as in the case of B-trees, where they reduce the constant factor.) The simplicity of binary arithmetic compared with decimal arithmetic is just another instance of the simplicity of algorithms based on binary choices.

7.10 The Update Function

```
In [120]: let rec update = function
          | Lf, k, w ->
              if k = 1 then
                Br (w, Lf, Lf)
              else
                raise Subscript  (* Gap in tree *)
          | Br (v, t1, t2), k, w ->
              if k = 1 then
                Br (w, t1, t2)
              else if k mod 2 = 0 then
                Br (v, update (t1, k / 2, w), t2)
              else
                Br (v, t1, update (t2, k / 2, w))
```

```
Out[120]: val update : 'a tree * int * 'a -> 'a tree = <fun>
```

The `update` function also divides the subscript repeatedly by two. When it reaches a value of one, it has identified the element position. Then it replaces the branch node by another branch with the new label.

A leaf may be replaced by a branch, extending the array, provided no intervening nodes have to be generated. This suffices for arrays without gaps in their subscripting. (The data structure can be modified to allow *sparse* arrays, where most subscript positions are undefined.) Exception `Subscript` indicates that the subscript position does not exist and cannot be created. This use of exceptions is not easily replaced by `None` and `Some`.

Note that there are two tests involving $k = 1$. If we have reached a leaf, it returns a branch, extending the array by one. If we are still at a branch node, then the effect is to update an existing array element.

A similar function can *shrink* an array by one.

7.10.1 Exercise 7.1

Draw the binary search tree that arises from successively inserting the following pairs into the empty tree: `("Alice", 6)`, `("Tobias", 2)`, `("Gerald", 8)`, `("Lucy", 9)`. Then repeat this task using the order `("Gerald", 8)`, `("Alice", 6)`, `("Lucy", 9)`, `("Tobias", 2)`. Why are results different?

7.10.2 Exercise 7.2

Code an insertion function for binary search trees. It should resemble the existing `update` function except that it should raise the exception `Collision` if the item to be inserted is already present.

7.10.3 Exercise 7.3

Continuing the previous exercise, it would be natural for exceptional `Collision` to return the value previously stored in the dictionary. Why is that goal difficult to achieve?

7.10.4 Exercise 7.4

Describe an algorithm for deleting an entry from a binary search tree. Comment on the suitability of your approach.

7.10.5 Exercise 7.5

Code the delete function outlined in the previous exercise.

7.10.6 Exercise 7.6

Show that the functions `preorder`, `inorder` and `postorder` all require $O(n^2)$ time in the worst case, where n is the size of the tree.

7.10.7 Exercise 7.7

Show that the functions `preord`, `inord` and `postord` all take linear time in the size of the tree.

7.10.8 Exercise 7.8

Write a function to remove the first element from a functional array. All the other elements are to have their subscripts reduced by one. The cost of this operation should be linear in the size of the array.

Chapter 8

Lecture 8: Functions as Values

In OCaml, functions can be

- passed as arguments to other functions,
- returned as results,
- put into lists, trees, etc.,
- but *not* tested for equality.

```
In [121]: [(fun n -> n * 2);
           (fun n -> n * 3);
           (fun n -> n + 1)]
```

```
Out[121]: - : (int -> int) list = [<fun>; <fun>; <fun>]
```

Progress in programming languages can be measured by what abstractions they admit. Conditional expressions (descended from conditional jumps based on the sign of some numeric variable) and parametric types such as α `list` are examples. The idea that functions could be used as values in a computation arose early, but it took some time before the idea was fully realised. Many programming languages let functions be passed as arguments to other functions, but few take the trouble needed to allow functions to be returned as results.

In mathematics, a *functional* or *higher-order function* is a function that operates on other functions. Many functionals are familiar from mathematics: for example, the differential operator maps functions to their derivatives, which are also functions. To a mathematician, a function is typically an infinite, uncomputable object. We use OCaml functions to represent algorithms. Sometimes they represent infinite collections of data given by computation rules.

Functions cannot be compared for equality. We could compare the machine addresses of the compiled code, but that would merely be a test of identity: it would regard any two separate functions as unequal even if they were compiled from identical pieces of source code. Such a low-level feature has no place in a principled language.

8.1 Functions Without Names

If functions are to be regarded as computational values, then we need a notation for them. The `fun` notation expresses a non-recursive function value without giving the function a name.

`fun x -> E` is the function f such that $f(x) = E$. The function `fun n -> n*2` is a *doubling function*.

```
In [122]: fun n -> n * 2

Out[122]: - : int -> int = <fun>
```

```
In [123]: (fun n -> n * 2) 17

Out[123]: - : int = 34
```

The main purpose of `fun`-notation is to package up small expressions that are to be applied repeatedly using some other function. The expression `fun n -> n*2` has the same value as the identifier `double`, declared as follows:

```
In [124]: let double n = n * 2

Out[124]: val double : int -> int = <fun>
```

The `fun` notation can also do pattern matching, and the `function` keyword adds an anonymous variable name to pattern match against. The following functions are all equivalent, with the latter definitions bound to the `is_zero` value and the earlier ones anonymous:

```
In [125]: fun x -> match x with 0 -> true | _ -> false

Out[125]: - : int -> bool = <fun>
```

```
In [126]: function 0 -> true | _ -> false

Out[126]: - : int -> bool = <fun>
```

```
In [127]: let is_zero = fun x -> match x with 0 -> true | _ -> false

Out[127]: val is_zero : int -> bool = <fun>
```

```
In [128]: let is_zero = function 0 -> true | _ -> false

Out[128]: val is_zero : int -> bool = <fun>
```

8.2 Curried Functions

A *curried function* returns another function as its result. We use the string concetenation operator `(^)` to illustrate how this works.

```
In [129]: (^)

Out[129]: - : string -> string -> string = <fun>
```

```
In [130]: let prefix = fun a -> fun b -> a ^ b
```

```
Out[130]: val prefix : string -> string -> string = <fun>

In [131]: let promote = prefix "Senior "

Out[131]: val promote : string -> string = <fun>

In [132]: prefix "Junior " "Professor"

Out[132]: - : string = "Junior Professor"

In [133]: promote "Professor"

Out[133]: - : string = "Senior Professor"
```

A short form for the definition of `prefix` is simply to pass multiple arguments to the function definition. The following two definitions are equivalent in OCaml:

```
In [134]: let prefix = fun a -> fun b -> a ^ b

Out[134]: val prefix : string -> string -> string = <fun>

In [135]: let prefix a b = a ^ b

Out[135]: val prefix : string -> string -> string = <fun>
```

Currying is the technique of expressing a function taking multiple arguments as nested functions, each taking a single argument. The `fun`-notation lets us package `n*2` as the function `fun n -> n * 2`, but what if there are several variables, as in `fun n -> n * 2 + k`? A function of two arguments could be coded using pattern-matching on pairs, writing `fun (n, k) -> n * 2 + k`.

Currying is an alternative, where we *nest* the `fun`-notation:

```
In [136]: fun k -> fun n -> n * 2 + k

Out[136]: - : int -> int -> int = <fun>
```

Applying this curried function to the argument 1 yields another function, in which `k` has been replaced by 1:

```
In [137]: let fn = fun k -> fun n -> n * 2 + k

Out[137]: val fn : int -> int -> int = <fun>

In [138]: let fn' = fn 1 (* n * 2 + 1 *)

Out[138]: val fn' : int -> int = <fun>

In [139]: fn' 3  (* 3 * 2 + 1 *)
```

```
Out[139]: - : int = 7
```

And this function, when applied to 3, yields the result 7. The two arguments are supplied one after another.

The example on the slide is similar but refers to the expression `a^b`, where `^` is the infix operator for string concatenation. Function `promote` binds the first argument of `prefix` to `"Professor"`; the resulting function prefixes that title to any string to which it is applied.

8.3 Shorthand for Curried Functions

A function-returning function is just a function of two arguments.

This curried function syntax is nicer than nested `fun` binders:

```
In [140]: let prefix a b = a ^ b
Out[140]: val prefix : string -> string -> string = <fun>

In [141]: let dub = prefix "Sir "
Out[141]: val dub : string -> string = <fun>
```

Curried functions allows *partial application* (to the first argument).

In OCaml, an n-argument curried function `f` can be declared using the syntax:

$$\texttt{let f } \texttt{x}_1 \ \ldots \ \texttt{x}_n \ = \ \texttt{E}$$

and applied using the syntax:

$$\texttt{E}_1 \ \ldots \ \texttt{E}_n$$

If `f` is not recursive, then it is equivalent to the function expressed via nesting as follows:

$$\texttt{fun } \texttt{x}_1 \ \rightarrow \cdots \rightarrow \texttt{fun } \texttt{x}_n \rightarrow \texttt{E}$$

We now have two ways of expressing functions of multiple arguments: either by passing a pair of arguments or by currying. Currying allows *partial application* which is useful when fixing the first argument yields a function that is interesting in its own right. An example from mathematics is the function x^y, where fixing $y = 2$ yields a function in x alone, namely squaring. Similarly, $y = 3$ yields cubing, while $y = 1$ yields the identity function.

Though the function `hd` (which returns the head of a list) is not curried, it may be used with the curried application syntax in some expressions:

```
In [142]: List.hd [dub; promote] "Hamilton"
```

```
Out[142]: - : string = "Sir Hamilton"
```

Here `List.hd` is applied to a list of functions, and the resulting function `dub` is then applied to the string `"Hamilton"`. The idea of executing code stored in data structures reaches its full development in *object-oriented* programming, like in Java.

8.4 Partial Application: A Curried Insertion Sort

```
In [143]: let insort lessequal =
            let rec ins x = function
              | [] -> [x]
              | y::ys -> if lessequal x y then x :: y :: ys
                         else y :: ins x ys
            in
            let rec sort = function
              | [] -> []
              | x::xs -> ins x (sort xs)
            in
            sort

Out[143]: val insort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

The sorting functions we discussed in earlier lectures are coded to sort floating-point numbers. They can be generalised to an arbitrary ordered type by passing the ordering predicate `lessequal` as an argument.

Functions `ins` and `sort` are declared locally, referring to `lessequal`. Though it may not be obvious, `insort` is a curried function. Given its first argument, a predicate for comparing some particular type of items, it returns the function `sort` for sorting lists of that type of items.

Some examples of its use:

```
In [144]: insort (<=) [5; 3; 9; 8]

Out[144]: - : int list = [3; 5; 8; 9]

In [145]: insort (<=) ["bitten"; "on"; "a"; "bee"]

Out[145]: - : string list = ["a"; "bee"; "bitten"; "on"]

In [146]: insort (>=) [5; 3; 9; 8]

Out[146]: - : int list = [9; 8; 5; 3]
```

An obscure point: the syntax `(<=)` denotes the comparison operator as a function, which is then given to `insort`. Passing the relation $\geq$ for `lessequal` gives a decreasing sort. This is no coding trick; it is justified in mathematics, since if $\leq$ is a partial ordering then so is $\geq$.

8.5 map: the "Apply to All" Function

```
In [147]: let rec map f = function
            | [] -> []
            | x::xs -> (f x) :: map f xs

Out[147]: val map : ('a -> 'b) -> 'a list -> 'b list = <fun>


In [148]: map (fun s -> s ^ "ppy") ["Hi"; "Ho"]

Out[148]: - : string list = ["Hippy"; "Hoppy"]


In [149]: map (map double) [[1]; [2; 3]]

Out[149]: - : int list list = [[2]; [4; 6]]
```

The functional `map` applies a function to every element of a list, returning a list of the function's results. "Apply to all" is a fundamental operation and we shall see several applications of it below. We again see the advantages of `fun`-notation, currying and `map`. If we did not have them, the first use of `map` in the above code block would require a preliminary function declaration:

```
In [150]: let rec sillylist = function
            | [] -> []
            | s::ss -> (s ^ "ppy") :: sillylist ss

Out[150]: val sillylist : string list -> string list = <fun>
```

An expression containing several applications of functionals—such as our second example—can abbreviate a long series of declarations. Sometimes this coding style is cryptic, but it can be clear as crystal. Treating functions as values lets us capture common program structures once and for all.

In the second example, `double` is the obvious integer doubling function we defined earlier. Note that `map` is a built-in OCaml function in the form of `List.map`. OCaml's standard library includes, among much else, many list functions.

8.6 Example: Matrix Transpose

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}^T = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

```
In [151]: let rec transp = function
            | []::_ -> []
            | rows -> (map List.hd rows) ::
                      (transp (map List.tl rows))

Out[151]: val transp : 'a list list -> 'a list list = <fun>
```

A matrix can be viewed as a list of rows, each row a list of matrix elements. This representation is not especially efficient compared with the conventional one (using arrays). Lists of lists turn up often, though, and we can see how to deal with them by taking familiar matrix operations as examples. *ML for the Working Programmer* goes as far as Gaussian elimination, which presents surprisingly few difficulties.

The transpose of the matrix $\left(\begin{smallmatrix} a & b & c \\ d & e & f \end{smallmatrix}\right)$ is $\left(\begin{smallmatrix} a & d \\ b & e \\ c & f \end{smallmatrix}\right)$, which in OCaml corresponds to the following transformation on lists of lists:

$$\text{[[a; b; c]; [d; e; f]]} \Rightarrow \text{[[a; d]; [b; e]; [c; f]]}$$

The workings of function `transp` are simple. If `rows` is the matrix to be transposed, then `map hd` extracts its first column and `map tl` extracts its second column:

$$\begin{aligned} \text{map hd rows} &\Rightarrow \text{[a; d]} \\ \text{map tl rows} &\Rightarrow \text{[[b; c]; [e; f]]} \end{aligned}$$

A recursive call transposes the latter matrix, which is then given the column `[a; d]` as its first row. The two functions expressed using `map` would otherwise have to be declared separately.

8.7 Review of Matrix Multiplication

$$\begin{pmatrix} A_1 & \cdots & A_k \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ \vdots \\ B_k \end{pmatrix} = \begin{pmatrix} A_1 B_1 + \cdots + A_k B_k \end{pmatrix}$$

The right side is the *vector dot product* $\vec{A} \cdot \vec{B}$. Repeat for each *row* of A and *column* of B.

The *dot product* of two vectors is

$$(a_1, \ldots, a_k) \cdot (b_1, \ldots, b_k) = a_1 b_1 + \cdots + a_k b_k$$

A simple case of matrix multiplication is when A consists of a single row and B consists of a single column. Provided A and B contain the same number k of elements, multiplying them yields a 1×1 matrix whose single element is the dot product shown above.

If A is an $m \times k$ matrix and B is a $k \times n$ matrix then $A \times B$ is an $m \times n$ matrix. For each i and j, the (i, j) element of $A \times B$ is the dot product of row i of A with column j of B.

$$\begin{pmatrix} 2 & 0 \\ 3 & -1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 2 \\ 4 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 \\ -1 & 1 & 6 \\ 4 & -1 & 0 \\ 5 & -1 & 2 \end{pmatrix}$$

The (1, 1) element above is computed by

$$(2, 0) \cdot (1, 4) = 2 \times 1 + 0 \times 4 = 2.$$

Coding matrix multiplication in a conventional programming language usually involves three nested loops. It is hard to avoid mistakes in the subscripting, which often runs slowly due to redundant internal calculations.

8.8 Matrix Multiplication in OCaml

Dot product of two vectors—a *curried function*

```
In [152]: let rec dotprod xs ys =
            match xs, ys with
            | [], [] -> 0.0
            | x::xs, y::ys ->  (x *. y) +. (dotprod xs ys)
```

```
Out[152]: val dotprod : float list -> float list -> float = <fun>
```

Matrix product

```
In [153]: let rec matprod arows brows =
            let cols = transp brows in
            map (fun row -> map (dotprod row) cols) arows
```

```
Out[153]: val matprod : float list list -> float list list -> float list list = <fun>
```

The `transp brows` converts B into a list of columns, yielding a list whose elements are the columns of B. Each row of $A \times B$ is obtained by multiplying a row of A by the columns of B.

Because `dotprod` is curried, it can be applied to a row of A. The resulting function is applied to all the columns of B. We have another example of currying and partial application.

The outer `map` applies `dotprod` to each row of A. The inner `map`, using `fun`-notation, applies `dotprod row` to each column of B. Compare with the version in *ML for the Working Programmer* (page 89) which does not use `map` and requires two additional function declarations.

In the dot product function, the two vectors must have the same length. Otherwise, exception `Match_failure` is raised.

8.9 List Functionals for Predicates

```
In [154]: let rec exists p = function
          | [] -> false
          | x::xs -> (p x) || (exists p xs)
```

```
Out[154]: val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

```
In [155]: let rec filter p = function
          | [] -> []
          | x::xs ->
              if p x then
                x :: filter p xs
```

```
            else
              filter p xs
```

```
Out[155]: val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

A *predicate* is a *boolean-valued* function.

The functional `exists` transforms a predicate into a predicate over lists. Given a list, `exists p` tests whether or not some list element satisfies `p` (making it return `true`). If it finds one, it stops searching immediately, thanks to the behaviour of the lazy `||` operator.

Dually, we have a functional to test whether all list elements satisfy the predicate. If it finds a counterexample then it, too, stops searching.

```
In [156]: let rec all p = function
          | [] -> true
          | x::xs -> (p x) && all p xs
```

```
Out[156]: val all : ('a -> bool) -> 'a list -> bool = <fun>
```

The `filter` functional, like `map`, transforms lists. It applies a predicate to all the list elements, but instead of returning the resulting values (which could only be `true` or `false`), it returns the list of elements satisfying the predicate.

8.10 Applications of the Predicate Functionals

```
In [157]: let member y xs =
            exists (fun x -> x=y) xs
```

```
Out[157]: val member : 'a -> 'a list -> bool = <fun>
```

```
In [158]: let inter xs ys =
            filter (fun x -> member x ys) xs
```

```
Out[158]: val inter : 'a list -> 'a list -> 'a list = <fun>
```

Testing whether two lists have no common elements

```
In [159]: let disjoint xs ys =
            all (fun x -> all (fun y -> x<>y) ys) xs
```

```
Out[159]: val disjoint : 'a list -> 'a list -> bool = <fun>
```

The Lists lecture presented the function `member`, which tests whether a specified value can be found as a list element, and `inter`, which returns the "intersection" of two lists: the list of elements they have in common.

But remember: the purpose of list functionals is not to replace the declarations of popular functions, which probably are available already. It is to eliminate the need for separate declarations of ad-hoc

functions. When they are nested, like the calls to `all` in `disjoint` above, the inner functions are almost certainly one-offs, not worth declaring separately.

Our primitives themselves can be seen as a programming language. Part of the task of programming is to extend our programming language with notation for solving the problem at hand. The levels of notation that we define should correspond to natural levels of abstraction in the problem domain.

Historical Note: Alonzo Church's λ-calculus gave a simple syntax, λ-notation, for expressing functions. It is the direct precursor of OCaml's `fun`-notation. It was soon shown that his system was equivalent in computational power to Turing machines, and *Church's thesis* states that this defines precisely the set of functions that can be computed effectively.

The λ-calculus had a tremendous influence on the design of functional programming languages. McCarthy's Lisp was something of a false start; it interpreted variable binding incorrectly, an error that stood for some 20 years. But in 1966, Peter Landin (of Queen Mary College, University of London) sketched out the main features of functional languages.

8.10.1 Exercise 8.1

What does the following function do, and what are its uses?

```
In [160]: let sw f x y = f y x
```

```
Out[160]: val sw : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

8.10.2 Exercise 8.2

There are many ways of combining orderings. The `lexicographic ordering` uses two keys for comparisons. It is specified by

$$(x', y') < (x, y) \iff x' < x \vee (x' = x \wedge y' < y).$$

Write an OCaml function to lexicographically combine two orderings, supplied as functions. Explain how it allows function `insort` to sort a list of pairs.

8.10.3 Exercise 8.3

Without using `map` write a function `map2` such that `map2 f` is equivalent to `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with only one by exploiting nested pattern-matching.

8.10.4 Exercise 8.4

The type `'a option`, declared below, can be viewed as a type of lists having at most one element. (It is typically used as an alternative to exceptions.) Declare an analogue of the function `map` for type `'a option`.

```
In [161]: type 'a option = None | Some of 'a
```

```
Out[161]: type 'a option = None | Some of 'a
```

8.10.5 Exercise 8.5

Recall the making change function of Lecture 4:

```
In [162]: let rec change till amt =
            match till, amt with
            | _       , 0   -> [ [] ]
            | []      , _   -> []
            | c::till , amt -> if amt < c then change till amt
                               else let rec allc = function
                                      | [] -> []
                                      | cs :: css -> (c::cs) :: allc css
                                    in
                                      allc (change (c::till) (amt - c)) @
                                           change till amt
```

```
Out[162]: val change : int list -> int -> int list list = <fun>
```

Function `allc` applies the function 'cons a `c`' to every element of a list. Eliminate it by declaring a curried cons function and applying `map`.

Chapter 9

Lecture 9: Sequences, or Lazy Lists

9.1 A Pipeline

$$\boxed{\textit{Producer}} \to \boxed{\textit{Filter}} \to \cdots \to \boxed{\textit{Filter}} \to \boxed{\textit{Consumer}}$$

- Produce sequence of items
- Filter sequence in stages
- Consume results as needed
- *Lazy lists* join the stages together

Two types of program can be distinguished. A sequential program accepts a problem to solve, processes for a while, and finally terminates with its result. A typical example is the huge numerical simulations that are run on supercomputers. Most of our OCaml functions also fit this model.

At the other extreme are *reactive* programs, whose job is to interact with the environment. They communicate constantly during their operation and run for as long as is necessary. A typical example is the software that controls many modern aircraft. Reactive programs often consist of *concurrent processes* running at the same time and communicating with one another.

Concurrency is too difficult to consider in this course, but we can model simple pipelines such as that shown above. The *Producer* represents one or more sources of data, which it outputs as a stream. The *Filter* stages convert the input stream to an output stream, perhaps consuming several input items to yield a single output item. The *Consumer* takes as many elements as necessary.

The Consumer drives the pipeline: nothing is computed except in response to its demand for an additional datum. Execution of the Filter stages is interleaved as required for the computation to go through. The programmer sets up the data dependencies but has no clear idea of what happens when. We have the illusion of concurrent computation.

The Unix operating system provides similar ideas through its *pipes* that link processes together. In OCaml, we can model pipelines using *lazy lists*.

9.2 Lazy Lists (or Streams)

- Lists of possibly *infinite* length
- Elements *computed upon demand*

- *Avoids waste* if there are many solutions
- *Infinite* values are a useful abstraction

In OCaml, we can implement laziness by *delaying evaluation* of the tail of the list.

Lazy lists have practical uses. Some algorithms, like making change, can yield many solutions when only a few are required. Sometimes the original problem concerns infinite series: with lazy lists, we can pretend they really exist!

We are now dealing with *infinite* (or at least unbounded) computations. A potentially infinite source of data is processed one element at a time, upon demand. Such programs are harder to understand than terminating ones and have more ways of going wrong.

Some purely functional languages, such as Haskell, use lazy evaluation everywhere. Even the if-then-else construct can be a function, and all lists are lazy. In OCaml, we can declare a type of lists such that evaluation of the tail does not occur until demanded. *Delayed* evaluation is weaker than *lazy* evaluation, but it is good enough for our purposes and often the best compromise for performance and memory usage.

The traditional word "stream" is reserved in OCaml parlance for input/output channels. Let us call lazy lists *sequences* instead.

9.3 Lazy Lists in OCaml

- The empty tuple `()` and its *type* `unit`
- Delayed version of E is `fun () -> E`

```
In [163]: type 'a seq =
          | Nil
          | Cons of 'a * (unit -> 'a seq)
```

```
Out[163]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In [164]: let head (Cons (x, _)) = x
```

```
Out[164]: val head : 'a seq -> 'a = <fun>
```

```
In [165]: let tail (Cons (_, xf)) = xf ()
```

```
Out[165]: val tail : 'a seq -> 'a seq = <fun>
```

`Cons(x,xf)` has *head* x and *tail function* xf

The primitive OCaml type `unit` has one element, which is written `()`. This element may be regarded as a 0-tuple, and `unit` as the nullary Cartesian product. (Think of the connection between multiplication and the number 1.)

The empty tuple serves as a placeholder in situations where no information is required. It may:

- appear in a data structure. For example, a `unit`-valued dictionary represents a set of keys.
- be the argument of a function, where its effect is to *delay evaluation*.
- be the argument or result of a procedure. (see the Procedural Programming section)

The empty tuple, like all tuples, is a constructor and is allowed in patterns; for example: `let f () = ...`

In particular `fun` () → `E` is the function that takes an argument of type `unit` and returns the value of E as its result. Expression E is not evaluated until the function is called, even though the only possible argument is `()`. The function simply delays the evaluation of E.

9.4 The Infinite Sequence: k, $k+1$, $k+2$, ...

```
In [166]: let rec from k = Cons (k, fun () -> from (k+1))

Out[166]: val from : int -> int seq = <fun>


In [167]: let it = from 1

Out[167]: val it : int seq = Cons (1, <fun>)


In [168]: let it = tail it

Out[168]: val it : int seq = Cons (2, <fun>)


In [169]: let it = tail it

Out[169]: val it : int seq = Cons (3, <fun>)
```

Function `from` constructs the infinite sequence of integers starting from k. Execution terminates because of the `fun` enclosing the recursive call. OCaml displays the tail of a sequence as `fun`, which stands for some function value. Each call to `tail` generates the next sequence element. We could do this forever.

This example is of little practical value because the cost of computing a sequence element will be dominated by that of creating the dummy function. Lazy lists tend to have high overheads.

9.5 Consuming a Sequence

```
In [170]: let rec get n s =
            match n, s with
            | 0, _             -> []
            | n, Nil           -> []
            | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[170]: val get : int -> 'a seq -> 'a list = <fun>
```

The above code gets the first n elements as a list. `xf ()` *forces* evaluation.

The function `get` converts a sequence to a list. It takes the first n elements; it takes all of them if $n < 0$, which can terminate only if the sequence is finite.

In the last line of `get`, the expression `xf()` calls the tail function, demanding evaluation of the next element. This operation is called *forcing* the sequence.

9.6 Sample Evaluation

$$\begin{aligned}
&\texttt{get}(2, \texttt{from}\ 6)\\
&\texttt{get}(2, \texttt{Cons}(6, \texttt{fun}\ () \to \texttt{from}\ (6+1)))\\
&6 :: \texttt{get}(1, \texttt{from}\ (6+1))\\
&6 :: \texttt{get}(1, \texttt{Cons}\ (7, \texttt{fun}\ () \to \texttt{from}\ (7+1)))\\
&6 :: 7 :: \texttt{get}(0, \texttt{Cons}\ (8, \texttt{fun}\ () \to \texttt{from}\ (8+1)))\\
&6 :: 7 :: []\\
&[6; 7]
\end{aligned}$$

Here we ask for two elements of the infinite sequence. In fact, three elements are computed: 6, 7 and 8. Our implementation is slightly too eager. A more complicated `type` declaration could avoid this problem. Another problem is that if one repeatedly examines some particular list element using forcing, that element is repeatedly evaluated. In a lazy programming language, the result of the first evaluation would be stored for later reference. To get the same effect in OCaml requires the use of references.

We should be grateful that the potentially infinite computation is kept finite. The tail of the original sequence even contains the unevaluated expression 6+1.

9.7 Joining Two Sequences

```
In [171]: let rec appendq xq yq =
            match xq with
            | Nil -> yq
            | Cons (x, xf) -> Cons(x, fun () -> appendq (xf ()) yq)
```

```
Out[171]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>
```

A more fair alternative:

```
In [172]: let rec interleave xq yq =
            match xq with
            | Nil -> yq
            | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))
```

```
Out[172]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>
```

Most list functions and functionals have analogues on sequences, but strange things can happen. Can an infinite list be reversed?

Function `appendq` is precisely the same idea as `append` from the Lists lecture; it concatenates two sequences. If the first argument is infinite, then `appendq` never gets to its second argument, which is lost. Concatenation of infinite sequences is not terribly interesting.

The function `interleave` avoids this problem by exchanging the two arguments in each recursive call. It combines the two lazy lists, losing no elements. Interleaving is the right way to combine two potentially infinite information sources into one.

In both function declarations, observe that each `xf ()` is enclosed within a `fun`() → .... Each *force* is enclosed within a *delay*. This practice makes the functions lazy. A force not enclosed in a delay, as in `get` above, runs the risk of evaluating the sequence in full.

9.8 Functionals for Lazy Lists

Filtering lazy lists:

```
In [173]: let rec filterq p = function
          | Nil -> Nil
          | Cons (x, xf) ->
              if p x then
                Cons (x, fun () -> filterq p (xf ()))
              else
                filterq p (xf ())
```

```
Out[173]: val filterq : ('a -> bool) -> 'a seq -> 'a seq = <fun>
```

The infinite sequence x, $f(x)$, $f(f(x))$, ...

```
In [174]: let rec iterates f x =
            Cons (x, fun () -> iterates f (f x))
```

```
Out[174]: val iterates : ('a -> 'a) -> 'a -> 'a seq = <fun>
```

The functional `filterq` demands elements of `xq` until it finds one satisfying `p`. (Recall `filter`, the analogous operation for ordinary lists.) It contains a *force* not protected by a *delay*. If `xq` is infinite and contains no satisfactory element, then `filtering` runs forever.

The functional `iterates` generalises `from`. It creates the next element not by adding one but by calling the function `f`.

9.9 Numerical Computations on Infinite Sequences

```
In [175]: let next a x = (a /. x +. x) /. 2.0
```

```
Out[175]: val next : float -> float -> float = <fun>
```

Close enough?

```
In [176]: let rec within eps = function
          | Cons (x, xf) ->
              match xf () with
              | Cons (y, yf) ->
```

```
                if abs_float (x -. y) <= eps then y
                else within eps (Cons (y, yf))
```

```
Out[176]: val within : float -> float seq -> float = <fun>
```

Square Roots:

```
In [177]: let root a = within 1e-6 (iterates (next a) 1.0)
```

```
Out[177]: val root : float -> float = <fun>
```

The *Newton-Raphson method* is widely used for computing square roots. The infinite series $x_0, (a/x_0 + x_0)/2, \ldots$ converges rapidly to $\sqrt{a}$. The initial approximation, x_0, is typically retrieved from a table, and is accurate enough that only a few iterations of the method are necessary. Calling `iterates (next a) x0` generates the *infinite series* of approximations to the square root of a using the Newton-Raphson method. To compute $\sqrt{2}$, the resulting series begins 1, 1.5, 1.41667, 1.4142157, 1.414213562, ..., and this last figure is already accurate to 10 significant digits!

Function `within` searches down the lazy list for two points whose difference is less than `eps`. It tests their absolute difference. Relative difference and other "close enough" tests can be coded. Such components can be used to implement other numerical functions directly as functions over sequences. The point is to build programs from small, interchangeable parts.

Function `root` uses `within`, `iterates` and `next` to to apply Newton-Raphson with a tolerance of 10^{-6} and a (poor) initial approximation of 1.0.

```
In [178]: root 2.0;;
```

```
Out[178]: - : float = 1.41421356237309492
```

This treatment of numerical computation has received some attention in the research literature; a recurring example is Richardson extrapolation.

9.9.1 Exercise 9.1

Code an analogue of `map` for sequences.

9.9.2 Exercise 9.2

Consider the list function `concat`, which concatenates a list of lists to form a single list. Can it be generalised to concatenate a sequence of sequences? What can go wrong?

```
In [179]: let rec concat = function
          | [] -> []
          | l::ls -> l @ concat ls
```

```
Out[179]: val concat : 'a list list -> 'a list = <fun>
```

9.9.3 Exercise 9.3

Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint*: to benefit from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

9.9.4 Exercise 9.4

A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an OCaml datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. (Taken from the exam question 2008 Paper 1 Question 5.)

9.9.5 Exercise 9.5

Code the lazy list whose elements are all ordinary lists of zeroes and ones, namely `[]; [0]; [1]; [0; 0]; [0; 1]; [1; 0]; [1; 1]; [0; 0; 0];`…. (Taken from the exam question 2003 Paper 1 Question 5.)

9.9.6 Exercise 9.6

(Continuing the previous exercise.) A *palindrome* is a list that equals its own reverse. Code the lazy list whose elements are all palindromes of 0s and 1s, namely `[]; [0]; [1]; [0; 0]; [0; 0; 0]; [0; 1; 0]; [1; 1]; [1; 0; 1]; [1; 1; 1]; [0; 0; 0; 0];`, …. You may take the reversal function `List.rev` as given.

Chapter 10

Lecture 10: Queues and Search Strategies

10.1 Breadth-First v Depth-First Tree Traversal

- binary trees as *decision trees*
- look for *solution nodes*
  - Depth-first: search one subtree in full before moving on
  - Breadth-first: search all nodes at level k before moving to $k+1$
- finds *all* solutions — nearest first!

Preorder, inorder and postorder tree traversals all have something in common: they are depth-first. At each node, the left subtree is entirely traversed before the right subtree. Depth-first traversals are easy to code and can be efficient, but they are ill-suited for some problems.

Suppose the tree represents the possible moves in a puzzle, and the purpose of the traversal is to search for a node containing a solution. Then a depth-first traversal may find one solution node deep in the left subtree, when another solution is at the very top of the right subtree. Often we want the shortest path to a solution.

Suppose the tree is *infinite* or simply extremely large. Depth-first search is almost useless with such trees, for if the left subtree is infinite then the search will never reach the right subtree. OCaml can represent infinite trees by the means discussed in the lecture on laziness. Another tree representation (suitable for solving solitaire, for example) is by a function `next : pos -> pos list`, which maps a board position to a list of the positions possible after the next move. For simplicity, the examples below use the OCaml datatype `tree`, which has only finite trees.

A *breadth-first* traversal explores the nodes horizontally rather than vertically. When visiting a node, it does not traverse the subtrees until it has visited all other nodes at the current depth. This is easily implemented by keeping a list of trees to visit. Initially, this list consists of one element: the entire tree. Each iteration removes a tree from the head of the list and adds its subtrees after the end of the list.

10.2 Breadth-First Tree Traversal — Using Append

```
In [180]: let rec nbreadth = function
          | [] -> []
          | Lf :: ts -> nbreadth ts
          | Br (v, t, u) :: ts ->
              v :: nbreadth (ts @ [t; u])

Out[180]: val nbreadth : 'a tree list -> 'a list = <fun>
```

Keeps an *enormous queue* of nodes of search, and is a wasteful use of `append`.

Breadth-first search can be inefficient, this naive implementation especially so. When the search is at depth d of the tree, the list contains all the remaining trees at depth d, followed by the subtrees (all at depth $d+1$) of the trees that have already been visited. At depth 10, the list could already contain 1024 elements. It requires a lot of space, and aggravates this with a gross misuse of append. Evaluating `ts@[t, u]` copies the long list `ts` just to insert two elements.

10.3 An Abstract Data Type: Queues

- `qempty` is the *empty queue*
- `qnull` *tests* whether a queue is empty
- `qhd` *returns* the element at the *head* of a queue
- `deq` *discards* the element at the *head* of a queue
- `enq` *adds* an element at the *end* of a queue

Breadth-first search becomes much faster if we replace the lists by *queues*. A queue represents a sequence, allowing elements to be taken from the head and added to the tail. This is a First-In-First-Out (FIFO) discipline: the item next to be removed is the one that has been in the queue for the longest time. Lists can implement queues, but append is a poor means of adding elements to the tail.

Our functional arrays are suitable, provided we augment them with a function to delete the first array element. (See *ML for the Working Programmer* page 156.) Each operation would take $O(\log n)$ time for a queue of length n.

We shall describe a representation of queues that is purely functional, based upon lists, and efficient. Operations take $O(1)$ time when "amortized": averaged over the lifetime of a queue.

A conventional programming technique is to represent a queue by an array. Two indices point to the front and back of the queue, which may wrap around the end of the array. The coding is somewhat tricky. Worse, the length of the queue must be given a fixed upper bound.

10.4 Efficient Functional Queues: Idea

- Represent the queue $x_1\ x_2\ \ldots\ x_m\ y_n\ \ldots\ y_1$ by any *pair of lists*

$$([x_1, x_2, \ldots, x_m],\ [y_1, y_2, \ldots, y_n])$$

- Add new items to the *rear list*

- Remove items from *front list* and if empty move *rear* to *front*
- *Amortized* time per operation is $O(1)$

Queues require efficient access at both ends: at the front, for removal, and at the back, for insertion. Ideally, access should take constant time, $O(1)$. It may appear that lists cannot provide such access. If `enq(q, x)` performs `q@[x]`, then this operation will be $O(n)$. We could represent queues by reversed lists, implementing `enq(q, x)` by `x::q`, but then the `deq` and `qhd` operations would be $O(n)$. Linear time is intolerable: a series of n queue operations could then require $O(n^2)$ time.

The solution is to represent a queue by a pair of lists, where

$$([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_n])$$

represents the queue $x_1 x_2 \ldots x_m y_n \ldots y_1$.

The front part of the queue is stored in order, and the rear part is stored in reverse order. The `enq` operation adds elements to the rear part using cons, since this list is reversed; thus, `enq` takes constant time. The `deq` and `qhd` operations look at the front part, which normally takes constant time, since this list is stored in order. But sometimes `deq` removes the last element from the front part; when this happens, it reverses the rear part, which becomes the new front part.

Amortized time refers to the cost per operation averaged over the lifetime of any complete execution. Even for the worst possible execution, the average cost per operation turns out to be constant; see the analysis below.

10.5 Efficient Functional Queues: Code

```
In [181]: type 'a queue =
          | Q of 'a list * 'a list
```

```
Out[181]: type 'a queue = Q of 'a list * 'a list
```

```
In [182]: let norm = function
          | Q ([], tls) -> Q (List.rev tls, [])
          | q -> q
```

```
Out[182]: val norm : 'a queue -> 'a queue = <fun>
```

```
In [183]: let qnull = function
          | Q ([], []) -> true
          | _ -> false
```

```
Out[183]: val qnull : 'a queue -> bool = <fun>
```

```
In [184]: let enq (Q (hds, tls)) x = norm (Q (hds, x::tls))
```

```
Out[184]: val enq : 'a queue -> 'a -> 'a queue = <fun>
```

```
In [185]: exception Empty
```

```
Out[185]: exception Empty
```

```
In [186]: let deq = function
          | Q (x::hds, tls) -> norm (Q (hds, tls))
          | _ -> raise Empty
Out[186]: val deq : 'a queue -> 'a queue = <fun>
```

```
In [187]: let qempty = Q ([], [])
Out[187]: val qempty : 'a queue = Q ([], [])
```

```
In [188]: let qhd = function
          | Q (x::_, _) -> x
          | _ -> raise Empty
Out[188]: val qhd : 'a queue -> 'a = <fun>
```

The datatype of queues prevents confusion with other pairs of lists. The empty queue has both parts empty.

The function `norm` puts a queue into normal form, ensuring that the front part is never empty unless the entire queue is empty. Functions `deq` and `enq` call `norm` to normalise their result.

Because queues are in normal form, their head is certain to be in their front part, so `qhd` looks there.

Let us analyse the cost of an execution comprising (in any possible order) n `enq` operations and n `deq` operations, starting with an empty queue. Each `enq` operation will perform one cons, adding an element to the rear part. Since the final queue must be empty, each element of the rear part gets transferred to the front part. The corresponding reversals perform one cons per element. Thus, the total cost of the series of queue operations is $2n$ cons operations, an average of 2 per operation. The amortized time is $O(1)$.

There is a catch. The conses need not be distributed evenly; reversing a long list could take up to $n-1$ of them. Unpredictable delays make the approach unsuitable for *real-time programming* where deadlines must be met.

10.6 Breadth-First Tree Traversal — Using Queues

```
In [189]: let rec breadth q =
            if qnull q then []
            else
              match qhd q with
              | Lf -> breadth (deq q)
              | Br (v, t, u) -> v :: breadth (enq (enq (deq q) t) u)
Out[189]: val breadth : 'a tree queue -> 'a list = <fun>
```

This function implements the same algorithm as `nbreadth` but uses a different data structure. It represents queues using type `queue` instead of type `list`.

To compare their efficiency, I applied both functions to the full binary tree of depth 12, which contains 4095 labels. The function `nbreadth` took 30 seconds while `breadth` took only 0.15 seconds: faster by a factor of 200.

For larger trees, the speedup would be greater. Choosing the right data structure pays handsomely.

10.7 Iterative deepening: Another Exhaustive Search

- Breadth-first search examines $O(b^d)$ nodes:

$$1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} \qquad \begin{array}{l} b = \text{branching factor} \\ d = \text{depth} \end{array}$$

- Recompute nodes at depth d instead of storing them
- Time factor is $b/(b-1)$ if $b > 1$; complexity is still $O(b^d)$
- Space required at depth d drops from b^d to d

Breadth-first search is not practical for infinite trees: it uses too much space. Large parts of the tree have to be stored. Consider the slightly more general problem of searching trees whose branching factor is b (for binary trees, $b = 2$). Then breadth-first search to depth d examines $(b^{d+1} - 1)/(b-1)$ nodes, which is $O(b^d)$, ignoring the constant factor of $b/(b-1)$. Since all nodes that are examined are also stored, the space and time requirements are both $O(b^d)$.

Depth-first iterative deepening combines the space efficiency of depth-first with the "nearest-first" property of breadth-first search. It performs repeated depth-first searches with increasing depth bounds, each time discarding the result of the previous search. Thus it searches to depth 1, then to depth 2, and so on until it finds a solution. We can afford to discard previous results because the number of nodes is growing exponentially. There are b^{d+1} nodes at level $d+1$; if $b \geq 2$, this number actually exceeds the total number of nodes of all previous levels put together, namely $(b^{d+1} - 1)/(b-1)$.

Korf shows that the time needed for iterative deepening to reach depth d is only $b/(b-1)$ times that for breadth-first search, if $b > 1$. This is a constant factor; both algorithms have the same time complexity, $O(b^d)$. In typical applications where $b \geq 2$ the extra factor of $b/(b-1)$ is quite tolerable. The reduction in the space requirement is exponential, from $O(b^d)$ for breadth-first to $O(d)$ for iterative deepening. Of course, this assumes that the tree itself is not stored in memory.

10.8 Another Abstract Data Type: Stacks

- `empty` is the *empty stack*
- `null` tests whether a stack is empty
- `top` returns the element at the *top* of a stack
- `pop` discards the element at the *top* of a stack
- `push` adds an element at the *top* of a stack

A *stack* is a sequence such that items can be added or removed from the head only. A stack obeys a Last-In-First-Out (LIFO) discipline: the item next to be removed is the one that has been in the queue for the *shortest* time. Lists can easily implement stacks because both `cons` and `hd` affect the

head. But unlike lists, stacks are often regarded as an imperative data structure: the effect of `push` or `pop` is to change an existing stack, not return a new one.

In conventional programming languages, a stack is often implemented by storing the elements in an array, using a variable (the "stack pointer") to count them. Most language processors keep track of recursive function calls using an internal stack.

10.9 A Survey of Search Methods

- Depth-first: use a *stack* (efficient but incomplete)
- Breadth-first: use a *queue* (uses too much space!)
- Iterative deepening: use depth-first to get benefits of breadth-first (trades time for space)
- Best-first: use a *priority queue* (heuristic search)

The data structure determines the search!

Search procedures can be classified by the data structure used to store pending subtrees. Depth-first search stores them on a stack, which is implicit in functions like `inorder`, but can be made explicit. Breadth-first search stores such nodes in a queue.

An important variation is to store the nodes in a priority queue, which is an ordered sequence. The priority queue applies some sort of ranking function to the nodes, placing higher-ranked nodes before lower-ranked ones. The ranking function typically estimates the distance from the node to a solution. If the estimate is good, the solution is located swiftly. This method is called best-first search.

The priority queue can be kept as a sorted list, although this is slow. Binary search trees would be much better on average, and fancier data structures improve matters further.

10.9.1 Exercise 10.1

Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented above.

10.9.2 Exercise 10.2

The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

10.9.3 Exercise 10.3

Write a version of the function `breadth` using a nested `let` construction rather than `match`.

10.9.4 Exercise 10.4

Iterative deepening is inappropriate if $b \approx 1$, where b is the branching factor. What search strategy is appropriate in this case?

10.9.5 Exercise 10.5

Consider the following OCaml function.

```
In [190]: let next n = [2 * n; 2 * n + 1]
```

```
Out[190]: val next : int -> int list = <fun>
```

If we regard it as representing a tree, where the subtrees are computed from the current label, what tree does `next 1` represent?

Chapter 11

Lecture 11: Elements of Procedural Programming

11.1 Procedural Programming

- Procedural programs can change the machine state.
- They can interact with its environment
- They use control structures like branching, iteration and procedures.

They use data abstractions of the computer's memory: - *references* to memory cells - *arrays* that are blocks of memory cells - *linked structures* such as *linked lists*

Procedural programming is programming in the traditional sense of the word. A program *state* is repeatedly transformed by the execution of *commands* or *statements*. A state change might be local to the machine and consist of updating a variable or array. A state change might consist of sending data to the outside world. Even reading data counts as a state change, since this act normally removes the data from the environment.

Procedural programming languages provide primitive commands and control structures for combining them. The primitive commands include *assignment* for updating variables, and various *input/output* commands for communication. Control structures include `if` and `match` constructs for conditional execution, and repetitive constructs such as `while`. Programmers can package up their own commands as *procedures* taking arguments. The need for such "subroutines" was evident from the earliest days; they represent one of the first examples of abstraction in programming languages.

OCaml makes no distinction between commands and expressions. OCaml provides built-in 'functions' to perform assignment and communication, and these can be used in the traditional (procedural) style. OCaml programmers often follow a functional style for most internal computations and use imperative features mainly for communication with the outside world.

11.2 OCaml Primitives for References

Syntax	Effect
`ref E`	*create* a reference with *initial contents* = value of `E`
`!P`	return *current contents* of reference `P`
`P := E`	*update* contents of `P` to value of `E`

The above text presents the OCaml primitives, but most languages have analogues of them, often heavily disguised. We need a means of creating references (or allocating storage), getting at the current contents of a reference cell, and updating that cell.

The function `ref` creates references (also called "locations"). Calling `ref` allocates a new location in memory. Initially, this location holds the value given by expression E.

The function `!`, when applied to a reference, returns its contents. This operation is called *dereferencing*. Clearly `!` is not a mathematical function; its result depends upon the store.

The assignment `P:=E` evaluates expression P, which must return a reference p, and E. It stores at address p the value of E. Syntactically, `:=` is a function and `P:=E` is an expression, even though it updates the store. Like many functions that change the state, it returns the value `()` of type `unit`.

If τ is some OCaml type, then τ `ref` is the type of references to cells that can hold values of τ. Please do not confuse the type `ref` with the function `ref`. This table of the primitive functions and their types might be useful:

Syntax	OCaml Type
`ref`	`'a -> 'a ref`
`!`	`'a ref -> 'a`
`:=`	`'a ref -> 'a -> unit`

11.3 Trying Out References

```
In [191]: let p = ref 5 (* create a reference *)
Out[191]: val p : int ref = {contents = 5}

In [192]: p := !p + 1    (* p now holds value 6 *)
Out[192]: - : unit = ()

In [193]: let ps = [ ref 77; p ]
Out[193]: val ps : int ref list = [{contents = 77}; {contents = 6}]

In [194]: List.hd ps := 3
Out[194]: - : unit = ()
```

```
In [195]: ps

Out[195]: - : int ref list = [{contents = 3}; {contents = 6}]
```

The first line declares `p` to hold a reference to an integer, initially 5. Its type is `int ref`, not just `int`, so it admits assignment. Assignment never changes `let` bindings: they are *immutable*. The identifier `p` will always denote the reference mentioned in its declaration unless superseded by a new usage of `p`. Only the *contents* of the reference is mutable.

OCaml displays a reference value as `{contents=v}`, where value v is the contents. This notation is readable but gives us no way of telling whether two references holding the same value are actually the same reference. To display a reference as a machine address has obvious drawbacks!

In the first assignment, the expression `!p` yields the reference's current contents, namely 5. The assignment changes the contents of `p` to 6. Most languages do not have an explicit dereferencing operator (like `!`) because of its inconvenience. Instead, by convention, occurrences of the reference on the *left-hand* side of the `:=` denote locations and those on the *right-hand* side denote the contents. A special 'address of' operator may be available to override the convention and make a reference on the right-hand side to denote a location. Logically this is a mess, but it makes programs shorter.

The list `ps` is declared to hold a new reference (initially containing 77) as well as `p`. Then the new reference is updated to hold 3. The assignment to `hd ps` does *not* update `ps`, only the contents of a reference in that list.

11.4 Commands: Expressions with Effects

- Basic commands update references, write to files, etc.
- $C_1; \ldots; C_n$ causes a series of expressions to be evaluated and returns the value of C_n.
- A typical command returns the empty tuple: `()`
- `if` B `then` C_1 `else` C_2 behaves like the traditional control structure if C_1 and C_2 have effects.
- Other OCaml constructs behave naturally with commands, including `match` expressions and recursive functions.

We use the term *command* informally to refer to an expression that has an effect on the state. All expressions denote some value, but they can return `()`, which conveys no actual information.

We need a way to execute one command after another. The construct $C_1; \ldots; C_n$ evaluates the expressions C_1 to C_n in the order given and returns the value of C_n. The values of the other expressions are discarded; their only purpose is to change the state.

Commands may be used with `if` and `match` much as in conventional languages. OCaml functions play the role of procedures.

Other languages that combine the functional and imperative programming paradigms include Lisp (and its dialect Scheme), Scala, and even a systems programming language, BLISS (now long extinct).

11.5 Iteration: the `while` command

```
In [196]: let tlopt = function
          | [] -> None
          | _::xs -> Some xs
```

```
Out[196]: val tlopt : 'a list -> 'a list option = <fun>
```

```
In [197]: let length xs =
            let lp  = ref xs in (* list of uncounted elements *)
            let np  = ref 0  in (* accumulated count *)
            let fin = ref false in
            while not !fin do
              match tlopt !lp with
              | None -> fin := true
              | Some xs ->
                  lp := xs;
                  np := 1 + !np
            done;
            !np (* the final count is returned *)
```

```
Out[197]: val length : 'a list -> int = <fun>
```

Once we can change the state, we need to do so repeatedly. Recursion can serve this purpose, but having to declare a procedure for every loop is clumsy, and compilers for conventional languages seldom exploit tail-recursion.

Early programming languages provided little support for repetition. The programmer had to set up loops using goto commands, exiting the loop using another goto controlled by an `if`. Modern languages provide a confusing jumble of looping constructs, the most fundamental of which is `while B do C`. The boolean expression B is evaluated, and if true, command C is executed and the command repeats. If B evaluates to false then the `while` command terminates, perhaps without executing C even once.

OCaml's main looping construct is `while`, which returns the value `()`. The function `length` declares references to hold the list under examination (`lp`) and number of elements counted so far (`np`) as well as whether the end of the list has been reached (the boolean reference `fin`). While the list is non-empty, we skip over one more element (by setting it to its tail) and count that element.

The body of the `while` loop first checks to see if the end of the list has been reached, in which case it sets the `fin` variable to true. If there is a tail value, then two assignments are executed in sequence. The `lp` reference is set to the tail of the list, and the `np` reference integer is incremented by one. When the while loop terminates due to the `fin` variable being set to true, the expression `!np` returns the computed length as the function's result.

11.6 Private, Persistent References

```
In [198]: exception TooMuch of int
```

```
Out[198]: exception TooMuch of int
```

```
In [199]: let makeAccount initBalance =
            let balance = ref initBalance in
            let withdraw amt =
               if amt > !balance then
                 raise (TooMuch (amt - !balance))
               else begin
                 balance := !balance - amt;
                 !balance
               end
            in
            withdraw
```

```
Out[199]: val makeAccount : int -> int -> int = <fun>
```

As you may have noticed, OCaml's programming style looks clumsy compared with that of languages like C. OCaml omits the defaults and abbreviations they provide to shorten programs. However, OCaml's explicitness makes it ideal for teaching the fine points of references and arrays. OCaml's references are more flexible than those found in other languages.

The function `makeAccount` models a bank. Calling the function with a specified initial balance creates a new reference `balance`) to maintain the account balance and returns a function (`withdraw`) having sole access to that reference. Calling `withdraw` reduces the balance by the specified amount and returns the new balance. You can pay money in by withdrawing a negative amount. The `if`-construct prevents the account from going overdrawn, raising an exception.

Look at the $(\mathtt{E}_1;\mathtt{E}_2)$ construct in the *else* part above. The first expression updates the account balance and returns the trivial value (). The second expression, `!balance`, returns the current balance but does not return the reference itself: that would allow unauthorised updates.

This example is based on one by Dr A C Norman.

11.7 Two Bank Accounts

```
In [200]: let student = makeAccount 500
```

```
Out[200]: val student : int -> int = <fun>
```

```
In [201]: let director = makeAccount 4000000;
```

```
Out[201]: val director : int -> int = <fun>
```

```
In [202]: student 5        (* coach fare *)
```

```
Out[202]: - : int = 495
```

```
In [203]: director 150000  (* Tesla *)
```

```
Out[203]: - : int = 3850000

In [204]: student 500      (* oh oh *)

Exception: TooMuch 5.
```

Each call to `makeAccount` returns a copy of `withdraw` holding a *fresh* instance of the reference `balance`. As with a real bank pass-book, there is no access to the account balance except via the corresponding `withdraw` function. If that function is discarded, the reference cell becomes unreachable; the computer will eventually reclaim it, just as banks close down dormant accounts.

Here we see two people managing their accounts. For better or worse, neither can take money from the other.

We could generalise `makeAccount` to return several functions that jointly manage information held in shared references. The functions might be packaged using OCaml records, which are not discussed in this course. Most procedural languages do not properly support the concept of private references, although *object-oriented* languages take them as a basic theme.

11.8 OCaml Primitives for Arrays

```
In [205]: [|"a"; "b"; "c"|] (* allocate a fresh string array *)
Out[205]: - : string array = [|"a"; "b"; "c"|]

In [206]: Array.make 3 'a'   (* array[3] with cell containing 'a' *)
Out[206]: - : char array = [|'a'; 'a'; 'a'|]

In [207]: let aa = Array.init 5 (fun i -> i * 10) (* array[5] initialised to (fun i) *)
Out[207]: val aa : int array = [|0; 10; 20; 30; 40|]

In [208]: Array.get aa 3  (* retrieve the 4th cell in the array *)
Out[208]: - : int = 30

In [209]: Array.set aa 3 42 (* set the 4th cell's value to 42 *)
Out[209]: - : unit = ()
```

There are many other array operations in the `Array` module in the OCaml standard library.

```
In [210]: Array.make
Out[210]: - : int -> 'a -> 'a array = <fun>

In [211]: Array.init
```

```
Out[211]: - : int -> (int -> 'a) -> 'a array = <fun>
```

```
In [212]: Array.get
```

```
Out[212]: - : 'a array -> int -> 'a = <fun>
```

```
In [213]: Array.set
```

```
Out[213]: - : 'a array -> int -> 'a -> unit = <fun>
```

OCaml arrays are like references that hold several elements instead of one. The elements of an n-element array are designated by the integers from 0 to $n-1$. The ith array element is usually written $A.(i)$. If τ is a type then τ `array` is the type of arrays (of any size) with elements from τ.

Calling `Array.init n f` creates an array of the size specified in n by function f. Initially, element $A.(i)$ holds the value of $f(i)$ for $i = 0$, ...,~$n-1$. Like `ref`, it allocates mutable storage to hold the specified values.

Calling `Array.get A i` returns the contents of $A.(i)$.

Calling `Array.set A i E` modifies the array A by storing the value of E as the new contents of $A[i]$; it returns `()` as its value.

OCaml's arrays are much safer than C's. In C, an array is nothing more than an address indicating the start of a storage area. Nothing indicates the size of the area. Therefore C programs are vulnerable to *buffer overrun attacks:* an attacker sends more data than the receiving program expects, overrunning the area of storage set aside to hold it. The attack eventually overwrites the program itself, replacing it with code controlled by the attacker.

11.9 Array Examples

In the following session, the identifier `ar` is bound to an array of 20 elements, which are initially set to the squares of their subscripts. The array's third element (which actually has subscript 2) is inspected and found to be four. The second call to `Array.get` supplies a subscript that is out of range, so OCaml rejects it.

```
In [214]: let ar = Array.init 20 (fun i -> i * i)
```

```
Out[214]: val ar : int array =
            [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81; 100; 121; 144; 169; 196; 225; 256;
              289; 324; 361|]
```

```
In [215]: Array.get ar 2
```

```
Out[215]: - : int = 4
```

```
In [216]: Array.get ar 20
```

```
Exception: Invalid_argument "index out of bounds".
```

```
In [217]: Array.set ar 2 33; ar

Out[217]: - : int array =
          [|0; 1; 33; 9; 16; 25; 36; 49; 64; 81; 100; 121; 144; 169; 196; 225; 256;
            289; 324; 361|]
```

By calling `Array.set`, we then modify the element with subscript 2. Note however that we cannot modify the array's length. If we outgrow the array, we have to create a new one, copy the data into it, and then forget the old array. Typically the new array would be double the size of the old one, so that the cost of copying is insignificant.

OCaml provides numerous operators for modifying, computing over and searching in arrays. Many are analogous to functions on lists. For example, `Array.exists` takes a boolean-valued function and returns `true` if an array element satisfies it.

```
In [218]: Array.exists (fun i -> i > 200) ar

Out[218]: - : bool = true
```

```
In [219]: Array.exists (fun i -> i < 0) ar

Out[219]: - : bool = false
```

11.10 References: OCaml *vs* conventional languages

- We must write `!p` to get the *contents* of `p`
- We write just `p` for the *address* of `p`
- We can store private reference cells (like `balance`) in functions—analogous to elements of *object-oriented programming*
- OCaml's assignment syntax is `V := E` instead of $V = E$
- OCaml has few control structures: `while`, `match`, `if` and `for` (the latter is not covered in this course)
- OCaml has syntax for updating an array via the `a.(i) <- v` syntax which is the same as `Array.set a i v`.

Conventional syntax for variables and assignments has hardly changed since Fortran, the first high-level language. In conventional languages, virtually all variables can be updated. We declare something like `p: int`, mentioning no reference type even if the language provides them. If we do not specify an initial value, we may get whatever bits were previously at that address. Illegal values arising from uninitialised variables can cause errors that are almost impossible to diagnose.

Dereferencing operators (like OCaml's `!`) are especially unpopular, because they clutter the program text. Virtually all programming languages make dereferencing implicit (that is, automatic).

It is generally accepted these days that a two-dimensional array A is nothing but an array of arrays. An assignment to such an array is typically written something like $A[i,j]{:=}x$; in C, the syntax is

`A[i][j] = x`. Higher dimensions are treated analogously. The corresponding OCaml code can either declare an array of arrays, or use the `A.(i)` syntax to calculate the linear offset into a single array.

You can use the constructs we have learnt to easily create linked (mutable) lists as an alternative to arrays.

```
In [220]: type 'a mlist =
          | Nil
          | Cons of 'a * 'a mlist ref
```

```
Out[220]: type 'a mlist = Nil | Cons of 'a * 'a mlist ref
```

It is worth mentioning that OCaml's references fully suffice for coding the sort of linked data structures taught in algorithms courses, and is illustrated in the figure above. The programming style is a little different from the usual, but the principles are the same. OCaml also provides comprehensive input/output primitives for various types of file and operating system.

OCaml's system of modules include *structures,* which can be seen as encapsulated groups of declarations, and *signatures,* which are specifications of structures listing the name and type of each component. Finally, there are *functors,* which are analogous to functions that combine a number of argument structures, and which can be used to plug program components together. These primitives are useful for managing large programming projects.

11.10.1 Exercise 11.1

Comment, with examples, on the differences between an `int ref list` and an `int list ref`.

11.10.2 Exercise 11.2

Write a version of function `power` (Lecture 1) using `while` instead of recursion.

11.10.3 Exercise 11.3

What is the effect of `while` C_1; `B do` C_2 `done` ?

11.10.4 Exercise 11.4

Write a function to exchange the values of two references, `xr` and `yr`.

11.10.5 Exercise 11.5

Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given n, and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$