Foundations of Computer Science

Computer Science Tripos Part IA

Lawrence C Paulson
Computer Laboratory
University of Cambridge

lp15@cam.ac.uk

Copyright © 2006 by Lawrence C. Paulson

Contents

1 Introduction **1**

2 Recursive Functions **14**

3 O Notation: Estimating Costs in the Limit **25**

4 Lists **36**

5 More on Lists **46**

6 Sorting **56**

7 Datatypes and Trees **65**

8 Dictionaries and Functional Arrays **78**

9 Queues and Search Strategies **87**

10 Functions as Values **97**

11 List Functionals **108**

12 Polynomial Arithmetic **118**

13 Sequences, or Lazy Lists **128**

14 Elements of Procedural Programming **138**

15 Linked Data Structures **150**

This course has two objectives. First (and obvious) is to teach programming. 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 ML 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 be able to cover most of the forms of data structures seen in programming. The course also covers basic methods for estimating efficiency.

Learning Guide. Suggestions for further reading, discussion topics, exercises and past exam questions appear at the end of each lecture. Extra reading is mostly drawn from my book *ML for the Working Programmer* (second edition), which also contains many exercises. You can find relevant exam questions in the Part IA papers from 1998 onwards. (Earlier papers pertain to a predecessor of this course.)

Thanks to David Allsopp, Stuart Becker, Gavin Bierman, Silas Brown, David Cottingham, Daniel Hulme, Frank King, Joseph Lord, James Margetson, David Morgan and Frank Stajano for pointing out errors in these notes. Please inform me of further errors and of passages that are particularly hard to understand. If I use your suggestion, I'll acknowledge it in the next printing.

Reading List

My own book is not based on these notes, but there is some overlap. The Hansen/Rischel and Ullman books are good alternatives. *The Little MLer* is a rather quirky tutorial on recursion and types. See *Introduction to Algorithms* for O-notation.

- Paulson, Lawrence C. (1996). *ML for the Working Programmer.* Cambridge University Press (2nd ed.).
- Hansen, Michael and Rischel, Hans (1999) *Introduction to Programming Using SML.* Addison-Wesley.
- Ullman, Jeffrey D. (1998) *Elements of ML97 Programming.* Prentice Hall.
- Mattias Felleisen and Daniel P. Friedman (1998). *The Little MLer.* MIT Press.
- Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest (1990). *Introduction to Algorithms.* MIT Press.

Slide 101

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*: 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.

Slide 102

Example I: Dates

Abstract level: names for dates over a certain range

Concrete level: typically 6 characters: YYMMDD

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!

Lessons:

- Information can be represented in many ways.
- Get it wrong, and you will be sorry!

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:

$$2^{48} = 2.8 \times 10^{14} \text{ days} = 7.7 \times 10^{11} \text{ years!}$$

Mathematicians think in terms of unbounded ranges, but the representation we choose for the computer usually imposes hard limits. A good programming language like ML 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.

Slide 103

Example II: Floating-Point Numbers

Computers have *integers* like 1066 and *reals* like 1.066×10^3.

A *floating-point number* is represented by two integers.

Both types come in different *precisions*: their range and accuracy.

The concept of DATA TYPE comprises

- how a value is *represented* inside the computer, and
- the suite of *operations* given to programmers.

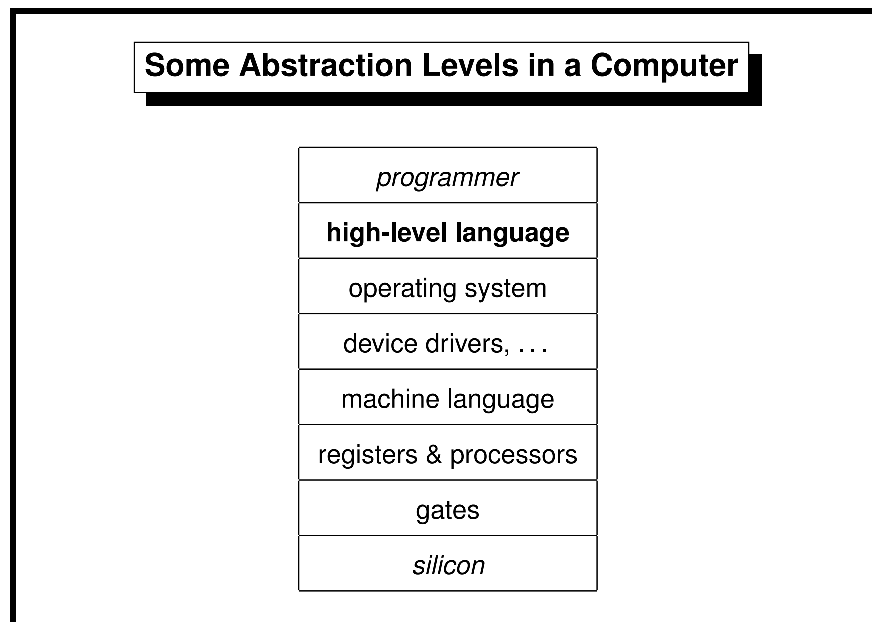
Floating point numbers are what you get on any pocket calculator. Internally, a float consists of two integers: the *mantissa* (fractional part) and the *exponent*. Complex numbers, consisting of two reals, might be provided. We have three levels of numbers already!

Most computers give us a choice of precisions, too. In 32-bit precision, integers typically range from $2^{31} - 1$ (namely 2,147,483,647) to -2^{31}; reals are accurate to about six decimal places and can get as large as 10^{35} or so. For reals, 64-bit precision is often preferred. How do we keep track of so many kinds of numbers? If we apply floating-point arithmetic to an integer, the result is undefined and might even vary from one version of a chip to another.

Early languages like Fortran required variables to be declared as INTEGER or REAL and prevented programmers from mixing both kinds of number in a computation. Nowadays, programs handle many different kinds of data, including text and symbols. Modern languages use the concept of *data type* to ensure that a datum undergoes only those operations that are meaningful for it.

Inside the computer, all data are stored as bits. Determining which type a particular bit pattern belongs to is impossible unless some bits have been set aside for that very purpose (as in languages like Lisp and Prolog). In most languages, the compiler uses types to generate correct machine code, and types are not stored during program execution.

Slide 104



These are just some of the levels that might be identified in a computer. Most large-scale systems are themselves divided into levels. For example, a management information system may consist of several database systems bolted together more-or-less elegantly.

This is the programmer's view. The user sees a different hierarchy, determined by the separate application programs, the operating system, and the visible hardware such as the screen and DVD-writer. Home computers are possible because the user's view can be so much simpler than the programmer's.

Communications protocols used on the Internet encompass several layers. Each layer has a different task, such as making unreliable links reliable (by trying again if a transmission is not acknowledged) and making insecure links secure (using cryptography). It sounds complicated, but the necessary software can be found on many personal computers.

In this course, we focus almost entirely on programming in a *high-level language*: ML.

Slide 105

What is Programming?

- to describe a computation so that it can be done *mechanically*:
  —*Expressions* compute *values*.
  —*Commands* cause *effects*.
- to do so **efficiently**, in both coding & execution
- to do so CORRECTLY, solving the right problem
- to allow easy *modification* as needs change
- to give LARGE programs a rational *structure* that others can comprehend and extend

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 in the usual way, we eventually find ourselves programming *in-the-large*: joining large modules to solve some possibly ill-defined task. It becomes a challenge if the modules were never intended to work together in the first place.

Programmers need a variety of skills:

- to *communicate requirements*, so they solve the right problem
- to *analyze problems*, breaking them down into smaller parts
- to *organize solutions* sensibly, so that they can be understood and modified
- to *estimate costs*, knowing in advance whether a given approach is feasible
- to *use mathematics* to arrive at correct and simple solutions

We shall look at all these points during the course, though programs will be too simple to have much risk of getting the requirements wrong.

Slide 106

Abstraction in Programming

- *Application Programming Interfaces*, or APIs **(Windows, Mac)**:
  —controlled access to operating systems facilities, etc.
- *Modules* **(ML and many post-1980 languages)**:
  —controlled access to a body of code
- *Abstract Data Types*, or ADTs **(various early languages)**:
  —controlled access to a hidden data structure
- *Classes and objects*: **(Java, C++, Smalltalk, ...)**
  —intermixed data structures and code
  —built into elaborate hierarchies

Each of these mechanisms has a common purpose: to provide services to programmers while hiding the details of how those services are implemented. Forcing programmers to use a standard interface helps to ensure that the system will still work if the implementation changes.

Software that refers to the Windows APIs can be expected to run on many different versions of Windows that have major internal differences. An API might provide routines to delete a file or to draw a menu.

Programming languages have used various mechanisms to allow one part of the program to provide APIs 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 latest and most complicated approach to modularity. *Classes* define concepts, and they can be built upon other classes. Operations can be defined that work in appropriately specialized 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 Standard ML's sophisticated module system, which can do many of the same things as classes. You will learn all about objects in the Java course.

Slide 107

Floating-Point, Revisited

Results are ALWAYS wrong—do we know how wrong?

Von Neumann doubted whether its **benefits** outweighed its COSTS!

Lessons:

- Innovations are often derided as luxuries for *lazy people*.
- Their HIDDEN COSTS can be worse than the obvious ones.
- *luxuries* often become *necessities*.

Floating-point is the basis for numerical computation: indispensable for science and engineering. Now read this [4, page 97]

> It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.

Von Neumann was one of the greatest figures in the early days of computing. How could he get it so wrong? It happens again and again:

- Time-sharing (supporting multiple interactive sessions, as on thor) was for people too lazy to queue up holding decks of punched cards.
- Automatic storage management (usually called *garbage collection*) was for people too lazy to do the job themselves.
- Screen editors were for people too lazy to use line-oriented editors.

To be fair, some innovations became established only after hardware advances reduced their costs.

Floating-point arithmetic is used, for example, to design aircraft—but would you fly in one? Code can be correct assuming exact arithmetic but deliver, under floating-point, wildly inaccurate results. The risk of error outweighs the increased complexity of the circuits: a hidden cost!

As it happens, there are methods for determining how accurate our answers are. A professional programmer will use them.

Slide 108

Why Program in ML?

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.

ML is the outcome of years of research into programming languages. It is unique among languages to be defined using a mathematical formalism (an *operational semantics*) that is both precise and comprehensible. Several commercially supported compilers are available, and thanks to the formal definition, there are remarkably few incompatibilities among them.

Because of its connection to mathematics, ML 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 ML 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. If you have a computer, install ML on it. I recommend Moscow ML,[1] which runs on PCs, Macintoshes and Unix and is fast and small. It comes with extensive libraries and supports the full language except for some aspects of modules, which are not covered in this course. Moscow ML is also available under PWF.

Cambridge ML is an alternative. It provides a Windows-based interface (due to Arthur Norman), but the compiler itself is the old Edinburgh ML, which is slow and buggy. It supports an out-of-date version of ML: many of the examples in my book [13] will not work.

[1] `http://www.dina.kvl.dk/~sestoft/mosml.html`

Slide 109

The Area of a Circle: $A = \pi r^2$

```
val pi = 3.14159;
> val pi = 3.14159 : real

pi * 1.5 * 1.5;
> val it = 7.0685775 : real

fun area (r) = pi*r*r;
> val area = fn : real -> real

area 2.0;
> val it = 12.56636 : real
```

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

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.

ML replies with the computed value (about 7.07) and its type (again `real`). Strictly speaking, we have declared the identifier `it`, which ML provides to let us refer to the value of the last expression entered at top level.

To work abstractly, we should provide the service "compute the area of a circle," so that we no longer need to remember the formula. So, the third line declares the function `area`. Given any real number `r`, it returns another real number, computed using the area formula; note that the function has type `real->real`.

The fourth line calls function `area` supplying 2.0 as the argument. A circle of radius 2 has an area of about 12.6. Note that the brackets around a function's argument are optional, both in declaration and in use.

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 `val` declaration for `pi` afterwards.

Slide 110

Integers; Multiple Arguments & Results

```
fun toSeconds (mins, secs) = secs + 60*mins;
> val toSeconds = fn : int * int -> int

fun fromSeconds s = (s div 60, s mod 60);
> val fromSeconds = fn : int -> int * int

toSeconds (5,7);
> val it = 307 : int

fromSeconds it;
> val it = (5, 7) : int * int
```

Given that there are 60 seconds in a minute, how many seconds are there in m minutes and s seconds? Function `toSeconds` performs the trivial calculation. It takes *a pair* of arguments, enclosed in brackets.

We are now using integers. The integer sixty is written `60`; the real sixty would be written `60.0`. The multiplication operator, `*`, is used for type `int` as well as `real`: it is *overloaded*. The addition operator, `+`, is also overloaded. As in most programming languages, multiplication (and division) have precedence over addition (and subtraction): we may write

`secs+60*mins` instead of `secs+(60*mins)`

The inverse of `toSeconds` demonstrates the infix operators `div` and `mod`, which express integer division and remainder. Function `fromSeconds` returns a pair of results, again enclosed in brackets.

Carefully observe the types of the two functions:

```
toSeconds   : int * int -> int
fromSeconds : int -> int * int
```

They tell us that `toSeconds` maps a pair of integers to an integer, while `fromSeconds` maps an integer to a pair of integers. In a similar fashion, an ML function may take any number of arguments and return any number of results, possibly of different types.

Slide 111

Summary of ML's numeric types

`int`: *the integers*

- constants `0 1 ~1 2 ~2 0032`...
- infixes `+ - * div mod`

`real`: *the floating-point numbers*

- constants `0.0 ~1.414 3.94e~7`...
- infixes `+ - * /`
- functions `Math.sqrt Math.sin Math.ln`...

The underlined symbols `val` and `fun` are *keywords*: they may not be used as identifiers. Here is a complete list of ML's keywords.

```
abstype and andalso as case datatype do else end eqtype exception
fn fun functor handle if in include infix infixr let local
nonfix of op open orelse raise rec
sharing sig signature struct structure
then type val where while with withtype
```

The negation of `x` is written `~x` rather than `-x`, please note. Most languages use the same symbol for minus and subtraction, but ML regards all operators, whether infix or not, as functions. Subtraction takes a pair of numbers, but minus takes a single number; they are distinct functions and must have distinct names. Similarly, we may not write `+x`.

Computer numbers have a finite range, which if exceeded gives rise to an *Overflow* error. Some ML systems can represent integers of arbitrary size.

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

`real  : int -> real`	convert an integer to the corresponding real
`floor : real -> int`	convert a real to the greatest integer not exceeding it

ML's libraries are organized using modules, so we use compound identifiers such as `Math.sqrt` to refer to library functions. In Moscow ML, library units are loaded by commands such as `load"Math";`. There are thousands of library functions, including text-processing and operating systems functions in addition to the usual numerical ones.

For more details on ML's syntax, please consult a textbook. Mine [13] and Wikström's [16] may be found in many College libraries. Ullman [15], in the Computer Lab library, is also worth a look.

Learning guide. Related material is in *ML for the Working Programmer*, pages 1–47, and especially 17–32.

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.

Exercise 1.2 Using the date representation of the previous exercise, code ML functions to (a) compare two years (b) add/subtract some given number of years from another year. (You may need to look ahead to the next lecture for ML's comparison operators.)

Slide 201

Raising a Number to a Power

```
fun npower(x,n) : real =
    if n=0
    then 1.0
    else x * npower(x, n-1);
> val npower = fn : real * int -> real
```

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

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

The function `npower` raises its real argument `x` to the power `n`, a non-negative integer. The function is *recursive*: it calls itself. This concept should be familiar from mathematics, since exponentiation is defined by the rules shown above. The ML 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:

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

Now for a tiresome but necessary aside. In most languages, the types of arguments and results must always be specified. ML is unusual in providing *type inference*: it normally works out the types for itself. However, sometimes ML needs a hint; function `npower` has a *type constraint* to say its result is `real`. Such constraints are required when overloading would otherwise make a function's type ambiguous. ML chooses type `int` by default or, in earlier versions, prints an error message.

Despite the best efforts of language designers, all programming languages have trouble points such as these. Typically, they are compromises caused by trying to get the best of both worlds, here type inference and overloading.

An Aside: Overloading

Slide 202

Functions defined for both `int` and `real`:

- operators ~ + − ⋆
- relations < <= > >=

The type checker requires help! — a *type constraint*

```
fun square (x) = x * x;          AMBIGUOUS
fun square (x:real) = x * x;     Clear
```

Nearly all programming languages overload the arithmetic operators. We don't want to have different operators for each type of number! 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. We can constrain the function's result:

```
fun square x = x * x : real;
fun square x : real = x * x;
```

ML treats the equality test specially. Expressions like

```
if x=y then ...
```

are fine provided x and y have the same type and equality testing is possible for that type.[1]

Note that `x <> y` is ML for $x \neq y$.

[1] All the types that we shall see for some time admit equality testing. Moscow ML allows even equality testing of reals, which is forbidden in the latest version of the ML library. Some compilers may insist that you write `Real.==(x,y)`.

Slide 203

Conditional Expressions and Type bool

$$\underline{\texttt{if}}\ b\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ y$$

$\texttt{not}(b)$ **negation of** b

$$p\ \underline{\texttt{andalso}}\ q \equiv \underline{\texttt{if}}\ p\ \underline{\texttt{then}}\ q\ \underline{\texttt{else}}\ \texttt{false}$$
$$p\ \underline{\texttt{orelse}}\ q \equiv \underline{\texttt{if}}\ p\ \underline{\texttt{then}}\ \texttt{true}\ \underline{\texttt{else}}\ q$$

A Boolean-valued function!

```
fun even n = (n mod 2 = 0);
> val even = fn : int -> bool
```

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

$$(\underline{\texttt{if}}\ \texttt{true}\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ y) = x$$
$$(\underline{\texttt{if}}\ \texttt{false}\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ y) = y$$

ML evaluates the expression if B then E_1 else E_2 by first evaluating B. If the result is `true` then ML 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 have an independent status.

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 (`andalso`) and disjunction (`orelse`). New properties can be declared as functions, e.g. to test whether an integer is even.

Note. The `andalso` and `orelse` operators evaluate their second operand only if necessary. They *cannot* be defined as functions: ML functions evaluate all their arguments. (In ML, any two-argument function can be turned into an infix operator.)

Raising a Number to a Power, Revisited

Slide 204

```
fun power(x,n) : real =
  if n=1 then x
  else if even n then      power(x*x, n div 2)
                 else x * power(x*x, n div 2)
```

Mathematical Justification:

$$x^1 = x$$
$$x^{2n} = (x^2)^n$$
$$x^{2n+1} = x \times (x^2)^n.$$

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:

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

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 (`div`) 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.

Slide 205

Expression Evaluation

$$E_0 \Rightarrow E_1 \Rightarrow \cdots \Rightarrow E_n \Rightarrow v$$

Sample evaluation for `power`:

$$\begin{aligned}\texttt{power}(2,12) &\Rightarrow \texttt{power}(4,6)\\ &\Rightarrow \texttt{power}(16,3)\\ &\Rightarrow 16 \times \texttt{power}(256,1)\\ &\Rightarrow 16 \times 256 \Rightarrow 4096.\end{aligned}$$

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 $E \Rightarrow E'$ 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.

Evaluation concerns only 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.

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 at first. Modelling it 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 (familiar to those of you who know C, for instance) that consist of commands.

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

Slide 206

Example: Summing the First n Integers

```
fun nsum n =
    if n=0 then 0
           else n + nsum (n-1);
> val nsum = fn: int -> int
```

$$
\begin{aligned}
\texttt{nsum}\,3 &\Rightarrow 3 + \texttt{nsum}\,2 \\
&\Rightarrow 3 + (2 + \texttt{nsum}\,1) \\
&\Rightarrow 3 + (2 + (1 + \texttt{nsum}\,0)) \\
&\Rightarrow 3 + (2 + (1 + 0)) \Rightarrow \ldots \Rightarrow 6
\end{aligned}
$$

The function call `nsum` n computes the sum $1 + \cdots + n$ rather naïvely, hence the initial `n` in its name. 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?

Slide 207

Iteratively Summing the First n Integers

```
fun summing (n,total) =
    if n=0 then total
           else summing (n-1, n + total);
> val summing = fn : int * int -> int
```

$$\texttt{summing}\,(3,0) \Rightarrow \texttt{summing}\,(2,3) \Rightarrow \texttt{summing}\,(1,5) \Rightarrow \texttt{summing}\,(0,6) \Rightarrow 6$$

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 $2^{31} - 1$.

Slide 208

Recursion Versus Iteration: Some Comments

Iterative normally refers to a *loop*—coded using `while`, for instance.

Tail-recursion is efficient only if the compiler detects it.

Mainly it saves *space*, though iterative code can run faster.

DON'T make programs iterative unless the gain is significant.

- People fought to give recursion to programmers: first, in *Algol-60*.
- Your code may turn into a mess!

A classic book by Abelson and Sussman [1] used *iterative* to mean *tail-recursive*. It describes the Lisp dialect known as Scheme. 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

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

Slide 209

Computing Square Roots: Newton-Raphson

$$x_{i+1} = \frac{a/x_i + x_i}{2}$$

```
fun nextApprox (a,x) = (a/x + x) / 2.0;
> nextApprox = fn : real * real -> real
nextApprox (2.0, 1.5);
> val it = 1.41666666667 : real
nextApprox (2.0, it);
> val it = 1.41421568627 : real
nextApprox (2.0, it);
> val it = 1.41421356237 : real
```

Now, let us look at a different sort of algorithm. The Newton-Raphson method is a highly effective means of finding roots of equations. It is used in numerical libraries to compute many standard functions, and in hardware, to compute reciprocals.

Starting with an approximation x_0, compute new ones x_1, x_2, ..., using a formula obtained from the equation to be solved. Provided the initial guess is sufficiently close to the root, the new approximations will converge to it rapidly.

The formula shown above computes the square root of a. The ML session demonstrates the computation of $\sqrt{2}$. Starting with the guess $x_0 = 1.5$, we reach by x_3 the square root in full machine precision. Continuing the session a bit longer reveals that the convergence has occurred, with $x_4 = x_3$:

```
nextApprox (2.0, it);
> val it = 1.41421356237 : real
it*it;
> val it = 2.0 : real
```

A Square Root Function

Slide 210

```
fun findRoot (a, x, epsilon) =
    let val nextx = (a/x + x) / 2.0
    in
       if abs(x-nextx) < epsilon*x  then nextx
       else findRoot (a, nextx, epsilon)
    end;

fun sqrt a = findRoot (a, 1.0, 1.0E~10);
> sqrt = fn : real -> real

sqrt 64.0;
> val it = 8.0 : real
```

The function `findRoot` applies Newton-Raphson to compute the square root of a, starting with the initial guess x, with relative accuracy ϵ. It terminates when successive approximations are within the tolerance ϵx, more precisely, when $|x_i - x_{i+1}| < \epsilon x$.

This recursive function differs fundamentally from previous ones like `power` and `summing`. For those, we can easily put a bound on the number of steps they will take, and their result is exact. For `findRoot`, determining how many steps are required for convergence is hard. It might oscillate between two approximations that differ in their last bit.

Observe how `nextx` is declared as the next approximation. This value is used three times but computed only once. In general, `let` D `in` E `end` declares the items in D but makes them visible only in the expression E. (Recall that identifiers declared using `val` cannot be assigned to.)

Function `sqrt` makes an initial guess of 1.0. A practical application of Newton-Raphson gets the initial approximation from a table. Indexed by say eight bits taken from a, the table would have only 256 entries. A good initial guess ensures convergence within a predetermined number of steps, typically two or three. The loop becomes straight-line code with no convergence test.

Learning guide. Related material is in *ML for the Working Programmer*, pages 48–58. The material on type checking (pages 63–67) may interest the more enthusiastic student.

Exercise 2.1 Code an iterative version of the function `power`.

Exercise 2.2 Try using $x_{i+1} = x_i(2 - x_i a)$ to compute $1/a$. Unless the initial approximation x_0 is good, it might not converge at all. (Pierre Jouet and Stefan Renold note that if $a > 0$ then the sequence converges if and only if $0 < x_0 < 2/a$.)

Exercise 2.3 Functions `npower` and `power` both have type constraints, but only one of them actually needs it. Try to work out which function does not needs its type constraint merely by looking at its declaration.

Slide 301

A Silly Square Root Function

```
fun nthApprox (a,x,n) =
    if n=0
    then x
    else (a / nthApprox(a,x,n-1) +
          nthApprox(a,x,n-1))
         / 2.0;
```

The function calls itself 2^n *times!*

Bigger inputs mean higher costs—but what's the *growth rate?*

The purpose of `nthApprox` is to compute x_n from the initial approximation x_0 using the Newton-Raphson formula $x_{i+1} = (a/x_i + x_i)/2$. Repeating the recursive call—and therefore the computation—is obviously wasteful. The repetition can be eliminated using `let val...in` E `end`. Better still is to call the function `nextApprox`, utilizing an existing abstraction.

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 buy a machine that is twice as powerful as our old one, how much can the input to our function be increased? With `nthApprox`, 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 Lect. 2, 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.

Asymptotic complexity refers to how costs grow with increasing inputs. Costs usually refer to time or space. Space complexity can never exceed time complexity, for it takes time to do anything with the space. Time complexity often greatly exceeds space complexity.

This lecture considers how to estimate various costs associated with a program. A brief introduction to a difficult subject, it draws upon the excellent texts *Concrete Mathematics* [6] and *Introduction to Algorithms* [5].

Slide 302

Some Illustrative Figures

complexity	*1 second*	*1 minute*	*1 hour*	**gain**
n	1000	60,000	3,600,000	$\times 60$
$n \lg n$	140	4,893	200,000	$\times 41$
n^2	31	244	1,897	$\times 8$
n^3	10	39	153	$\times 4$
2^n	9	15	21	$+6$

complexity = milliseconds needed for an input of size n

This table (excerpted from Aho et al. [2, page 3]) illustrates the effect of various time complexities. The left-hand column indicates 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).

The table illustrates how 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. 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.

Exercise 3.1 Add a column to the table with the heading *60 hours*.

Slide 303

Comparing Algorithms

Look at the *most significant* term.

Ignore *constant factors*:

- They are seldom important.
- They depend on ephemeral details such as computer brand.

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 as n increases it rapidly becomes insignificant.

Constant factors in costs are often ignored. For one thing, they seldom make a difference: $100n^2$ will be better than n^3 in the long run. Only if the leading terms are otherwise identical do constant factors become important. But there is a second difficulty: constant factors are seldom reliable. 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.

In practice, constant factors sometimes matter. If an algorithm is too complicated, its costs will include a large constant factor. In the case of multiplication, the theoretically fastest algorithm catches up with the standard one only for enormous values of n.

Slide 304

O Notation (And Friends)

$f(n) = O(g(n))$ provided $|f(n)| \leq c|g(n)|$

- for some *constant* c
- and all *sufficiently large* n.

$f(n) = O(g(n))$ means g is an **upper** bound on f

$f(n) = \Omega(g(n))$ means g is an **lower** bound on f

$f(n) = \Theta(g(n))$ means g gives **exact** bounds on f

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. The notation also ignores constant factors such as c. We may use a different c and n_0 with each f.

The standard notation $f(n) = O(g(n))$ is misleading: this is no equation. Please use common sense. From $f(n) = O(n)$ and $f'(n) = O(n)$ we cannot infer $f(n) = f'(n)$.

Note that $f(n) = O(g(n))$ gives an upper bound on f in terms of g. To specify a lower bound, we have the dual notation

$$f(n) = \Omega(g(n)) \iff |f(n)| \geq c|g(n))|$$

for some constant c and all sufficiently large n. The conjunction of $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ is written $f(n) = \Theta(g(n))$.

People often use $O(g(n))$ as if it gave a tight bound, confusing it with $\Theta(g(n))$. Since $O(g(n))$ gives an upper bound, if $f(n) = O(n)$ then also $f(n) = O(n^2)$. Tricky examination questions exploit this fact.

Slide 305

Simple Facts About O Notation

$O(2g(n))$ *is the same as* $O(g(n))$

$O(\log_{10} n)$ *is the same as* $O(\ln n)$

$O(n^2 + 50n + 36)$ *is the same as* $O(n^2)$

$O(n^2)$ **is contained in** $O(n^3)$

$O(2^n)$ **is contained in** $O(3^n)$

$O(\log n)$ **is contained in** $O(\sqrt{n})$

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 the value of n_0 needed in $f(n) = O(n^2 + 50n + 36)$. Using the law $(n + k)^2 = n^2 + 2nk + k^2$, it is easy to check that using $n_0 + 25$ for n_0 and keeping the same value of c gives $f(n) = O(n^2)$.

If c and d are constants (that is, they are independent of n) with $0 < c < d$ then

$$O(n^c) \text{ is contained in } O(n^d)$$
$$O(c^n) \text{ is contained in } O(d^n)$$
$$O(\log n) \text{ is contained in } O(n^c)$$

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.

Slide 306

Common Complexity Classes

$O(1)$	*constant*
$O(\log n)$	*logarithmic*
$O(n)$	*linear*
$O(n \log n)$	*quasi-linear*
$O(n^2)$	*quadratic*
$O(n^3)$	*cubic*
$O(a^n)$	*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 *quasi-linear*, 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. ML type-checking is exponential in the worst case, but not for ordinary programs.

Slide 307

Sample Costs in O Notation

function	*time*	*space*
`npower`, `nsum`	$O(n)$	$O(n)$
`summing`	$O(n)$	$O(1)$
$n(n+1)/2$	$O(1)$	$O(1)$
`power`	$O(\log n)$	$O(\log n)$
`nthApprox`	$O(2^n)$	$O(n)$

Recall (Lect. 2) that `npower` computes x^n by repeated multiplication while `nsum` naïvely 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.

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 $O(1)$.

Slide 308

Solving Simple Recurrence Relations

$T(n)$: a cost we want to bound using O notation

Typical *base case*: $T(1) = 1$

Some *recurrences*:

$$T(n+1) = T(n) + 1 \qquad \textbf{linear}$$
$$T(n+1) = T(n) + n \qquad \textbf{quadratic}$$
$$T(n) = T(n/2) + 1 \qquad \textbf{logarithmic}$$

To analyze a function, inspect its ML 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`:

```
fun nsum n =
    if n=0 then 0 else n + nsum (n-1);
```

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.

```
fun nsumsum n =
    if n=0 then 0 else nsum n + nsumsum (n-1);
```

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 $T(n) = O(\log n)$.

Slide 309

Recurrence for `nthApprox`: $O(2^n)$

$$T(0) = 1$$
$$T(n+1) = 2T(n) + 1$$

Explicit solution: $T(n) = 2^{n+1} - 1$

$$\begin{aligned} T(n+1) &= 2T(n) + 1 \\ &= 2(2^{n+1} - 1) + 1 \qquad \textbf{induction hypothesis} \\ &= 2^{n+2} - 1 \end{aligned}$$

Now we analyze the function `nthApprox` given at the start of the lecture. The two recursive calls are reflected in the term $2T(n)$ of the recurrence. As for the constant effort, although the recursive case does more work than the base case, we can choose units such that both constants are one. (Remember, we seek an upper bound rather than the exact cost.)

Given the recurrence equations for $T(n)$, let us solve them. It helps if we can guess the closed form, which in this case obviously is something like 2^n. Evaluating $T(n)$ for $n = 0, 1, 2, 3, \ldots$, we get $1, 3, 7, 15, \ldots$. Obviously $T(n) = 2^{n+1} - 1$, which we can easily prove by induction on n. We must check the base case:

$$T(0) = 2^1 - 1 = 1$$

In the inductive step, for $T(n+1)$, we may assume our equation in order to replace $T(n)$ by $2^{n+1} - 1$. The rest is easy.

We have proved $T(n) = O(2^{n+1} - 1)$, but obviously 2^n is also an upper bound: we may choose the constant factor to be two. Hence $T(n) = O(2^n)$.

The proof above is rather informal. The orthodox way of proving $f(n) = O(g(n))$ is to follow the definition of O notation. But an inductive proof of $T(n) \leq c2^n$, using the definition of $T(n)$, runs into difficulties: this bound is too loose. Tightening the bound to $T(n) \leq c2^n - 1$ lets the proof go through.

Exercise 3.2 Try the proof suggested above. What does it say about c?

Slide 310

An $O(n \log n)$ Recurrence

$$T(1) = 0$$
$$T(n) = 2T(n/2) + n$$

Proof that $T(n) \leq n \lg n$:

$$\begin{aligned} T(n) &\leq 2(n/2)\lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n \lg n - n + n \\ &= n \lg n \end{aligned}$$

This recurrence equation arises when a function divides its input into two equal parts, does $O(n)$ work and also calls itself recursively on each. Such *balancing* is beneficial. Instead dividing the input into unequal parts of sizes 1 and $n-1$ gives the recurrence $T(n+1) = T(n) + n$, which has quadratic complexity.

Shown on the slide is the result of substituting the closed form $T(n) = cn \lg n$ into the original equations. This is another proof by induction. The last step holds provided $c \geq 1$.

Something is wrong, however. The base case fails: if $n = 1$ then $cn \lg n = 0$, which is not an upper bound for $T(1)$. We could look for a precise closed form for $T(n)$, but it is simpler to recall that O notation lets us ignore a finite number of awkward cases. Choosing $n = 2$ and $n = 3$ as base cases eliminates $n = 1$ entirely from consideration. The constraints $T(2) \leq 2c \lg 2$ and $T(3) \leq 3c \lg 3$ can be satisfied for $c \geq 2$. So $T(n) = O(n \log n)$.

Incidentally, in these recurrences $n/2$ stands for integer division. To be precise, we should indicate truncation to the next smaller integer by writing $\lfloor n/2 \rfloor$. One-half of an odd number is given by $\lfloor (2n+1)/2 \rfloor = n$. For example, $\lfloor 2.9 \rfloor = 2$, and $\lfloor n \rfloor = n$ if n is an integer.

Learning guide. For a deeper treatment of complexity, you might look at Chapter 2 of *Introduction to Algorithms* [5].

Exercise 3.3 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 $O(n \log n)$.

Exercise 3.4 Prove that the recurrence

$$T(n) = \begin{cases} 1 & \text{if } 1 \leq n < 4 \\ T(\lceil n/4 \rceil) + T(\lfloor 3n/4 \rfloor) + n & \text{if } n \geq 4 \end{cases}$$

is $O(n \log n)$. The notation $\lceil x \rceil$ means truncation to the next larger integer; for example, $\lceil 3.1 \rceil = 4$.

Lists

Slide 401

```
[3,5,9];
> [3, 5, 9] : int list

it @ [2,10];
> [3, 5, 9, 2, 10] : int list

rev [(1,"one"), (2,"two")];
> [(2, "two"), (1, "one")] : (int*string) list
```

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].

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.

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

Slide 402

The List Primitives

The two kinds of list

`nil` or `[]` is the empty list

$x :: l$ is the list with head x and tail l

List notation

$$[x_1, x_2, \ldots, x_n] \equiv \underbrace{x_1}_{\text{head}} :: \underbrace{(x_2 :: \cdots (x_n :: nil))}_{\text{tail}}$$

The operator ::, called *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.

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

$$\begin{array}{lllll} :: \rightarrow \cdots & :: \rightarrow & :: \rightarrow & :: \rightarrow & nil \\ \downarrow & \downarrow & \downarrow & \downarrow & \\ 1 & 3 & 5 & 9 & \end{array}$$

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 $\downarrow$ arrow leads to the head and the leftmost $\rightarrow$ 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!

Getting at the Head and Tail

Slide 403

```
fun null   []    = true
  | null (x::l) = false;
> val null = fn : 'a list -> bool

fun hd (x::l) = x;
> Warning: pattern matching is not exhaustive
> val hd = fn : 'a list -> 'a

tl [7,6,5];
> val it = [6, 5] : int list
```

There are three basic functions for inspecting lists. Note their polymorphic types!

`null : 'a list -> bool`	is a list empty?
`hd   : 'a list -> 'a`	head of a non-empty list
`tl   : 'a list -> 'a list`	tail of a non-empty list

The empty list has neither head nor tail. Applying either operation to `nil` is an error—strictly speaking, an *exception*. The function `null` can be used to check for the empty list before applying `hd` or `tl`.

To look deep inside a list one can apply combinations of these functions, but this style is hard to read. Fortunately, it is seldom necessary because of *pattern-matching*.

The declaration of `null` above has two clauses: 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. ML prints a warning to tell us that calling the function could raise exception `Match`, which indicates failure of pattern-matching.

The declaration of `tl` is omitted because it is similar to `hd`. Instead, there is an example of applying `tl`.

Slide 404

Computing the Length of a List

```
fun nlength []      = 0
  | nlength (x::xs) = 1 + nlength xs;
> val nlength = fn: 'a list -> int
```

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

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:

```
fun nlength []      = 0;
> Warning: pattern matching is not exhaustive
> val nlength = fn: 'a list -> int
fun nlength (x::xs) = 1 + nlength xs;
> Warning: pattern matching is not exhaustive
> val nlength = fn: 'a list -> int
```

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 `fun` declaration replaces any previous one rather than extending it to cover new cases.

Now, let us return to the declaration shown on the slide. The length function is *polymorphic*: it applies to *all* lists regardless of element type! Most programming languages lack such flexibility.

Unfortunately, this length computation is naïve and wasteful. Like `nsum` in Lect. 2, 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.

Slide 405

Efficiently Computing the Length of a List

```
fun addlen (n, [ ])    = n
  | addlen (n, x::xs) = addlen (n+1, xs);
> val addlen = fn: int * 'a list -> int
```

$$
\begin{aligned}
addlen(0, [a, b, c]) &\Rightarrow addlen(1, [b, c]) \\
&\Rightarrow addlen(2, [c]) \\
&\Rightarrow addlen(3, []) \\
&\Rightarrow 3
\end{aligned}
$$

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

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`.

```
fun length xs = addlen(0,xs);
> val length = fn : 'a list -> int
```

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.

Slide 406

Append: List Concatenation

```
fun append([], ys)     = ys
  | append(x::xs, ys) = x :: append(xs,ys);
> val append = fn: 'a list * 'a list -> 'a list
```

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

Here is how append might be declared, ignoring the details of how `@` is made an infix operator.

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. Two lists can be joined if their element types agree.

Slide 407

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

```
fun nrev  []     = []
  | nrev(x::xs) = (nrev xs) @ [x];
> val nrev = fn: 'a list -> 'a list
```

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

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.

Reversing a List in $O(n)$

Slide 408

```
fun revApp ([], ys)     = ys
  | revApp (x::xs, ys) = revApp (xs, x::ys);
> val revApp = fn: 'a list * 'a list -> 'a list
```

$$\begin{aligned} revApp([a,b,c],[]) &\Rightarrow revApp([b,c],[a]) \\ &\Rightarrow revApp([c],[b,a]) \\ &\Rightarrow revApp([],[c,b,a]) \\ &\Rightarrow [c,b,a] \end{aligned}$$

Calling `revApp (xs,ys)` reverses the elements of `xs` and prepends them to `ys`. Now we may declare

```
fun rev xs = revApp(xs,[]);
> val rev = fn : 'a list -> 'a list
```

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.

Slide 409

Lists, Strings and Characters

character constants	`#"A"` `#"\""` ...
string constants	`""` `"B"` `"Oh, no!"` ...
`explode`(s)	*list of the characters in string* s
`implode`(l)	*string made of the characters in list* l
`size`(s)	*number of chars in string* s
s_1 ^ s_2	*concatenation* of strings s_1 and s_2

Strings are provided in most programming languages to allow text processing. At a bare minimum, numbers must be converted to or from a textual representation. Programs issue textual messages to users and analyze their responses. Strings are essential in practice, but they bring up few issues relevant to this course.

The functions `explode` and `implode` convert between strings and lists of characters. In a few programming languages, strings simply are lists of characters, but this is poor design. Strings are an abstract concept in themselves. Treating them as lists leads to clumsy and inefficient code.

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

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

Learning guide. Related material is in *ML for the Working Programmer*, pages 69–80.

Exercise 4.1 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.

Slide 501

List Utilities: take and drop

Removing the first i elements

```
fun take ([], _)     = []
  | take (x::xs, i) = if i>0
                      then x :: take(xs, i-1)
                      else [];

fun drop ([], _)     = []
  | drop (x::xs, i) = if i>0 then drop(xs,i-1)
                             else x::xs;
```

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.

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

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

The special pattern variable `_` appears in both functions. This *wildcard pattern* matches anything. We could have written `i` in both positions, but the wildcard reminds us that the relevant clause ignores this argument.

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

Slide 502

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.

Slide 503

Types with Equality

The membership test has a strange *polymorphic* type.

```
fun member(x, [])    =  false
  | member(x, y::l) = (x=y) orelse member(x,l);
> val member = fn : ''a * ''a list -> bool
```

Here, *''a* stands for any *equality type*.

Equality testing is *OK* for integers but NOT for functions.

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`. Its polymorphism is restricted to the so-called *equality types*. These include integers, strings, booleans, and tuples or lists of other equality types.

Equality testing is not available for every type, however. *Functions* are values in ML, and there is no way of comparing two functions that is both practical and meaningful. *Abstract types* can be declared in ML, hiding their internal representation, including its equality test. Equality is not even allowed for type `real`, though some ML systems ignore this. We shall discuss function values and abstract types later.

If a function's type contains *equality type variables*, such as `''a`, `''b`, then it uses polymorphic equality testing.

Slide 504

Equality Polymorphism

```
fun inter([],     ys) = []
  | inter(x::xs, ys) =
        if member(x,ys) then x::inter(xs, ys)
                        else    inter(xs, ys);
> val inter = fn: ''a list * ''a list -> ''a list
```

ML notices that `inter` uses equality *indirectly*, by `member`.

ML will OBJECT if we apply these functions to non-equality types.

Function **inter** computes the 'intersection' of two lists, returning the list of elements common to both. It calls **member**. The equality type variables propagate: the intersection function also has them even though its use of equality is indirect. Trying to apply **member** or **inter** to a list of functions causes ML to complain of a *type error*. It does so at *compile time*: it detects the errors by types alone, without executing the offending code.

Equality polymorphism is a contentious feature. Some researchers complain that it makes ML too complicated and leads programmers to use linear search excessively. The functional programming language Haskell generalizes the concept, allowing programmers to introduce new classes of types supporting any desired collection of operations.

Building a List of Pairs

Slide 505

```
fun zip (x::xs,y::ys) = (x,y) :: zip(xs,ys)
  | zip _             = [];
```

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

The *wildcard* pattern (_) catches *empty lists.*

THE PATTERNS ARE TRIED IN ORDER.

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 $[z_1, \ldots, z_n]$.

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:

```
  zip : ('a list * 'b list) -> ('a * 'b) list
unzip : ('a * 'b) list -> ('a list * 'b list)
```

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. ML 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.

Building a Pair of Results

Slide 506

```
fun unzip []              = ([],[])
  | unzip ((x,y)::pairs) =
      let val (xs,ys) = unzip pairs
      in  (x::xs, y::ys)
      end;


fun revUnzip ([], xs, ys)           = (xs,ys)
  | revUnzip ((x,y)::pairs, xs, ys) =
      revUnzip(pairs, x::xs, y::ys);
```

Given a list of pairs, `unzip` has to build *two* lists of results, which is awkward using recursion. The version shown about uses the *local declaration* `let` D `in` E `end`, where D consists of declarations and E is the expression that can use them.

Note especially the declaration

```
val (xs,ys) = unzip pairs
```

which binds `xs` and `ys` to the results of the recursive call. In general, the declaration `val` P `=` E matches the pattern P against the value of expression E. It binds all the variables in P to the corresponding values.

Here is 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 `zip` and is possibly clearer, but not every local declaration can be eliminated as easily.

```
fun conspair ((x,y), (xs,ys)) = (x::xs, y::ys);

fun unzip []          = ([],[])
  | unzip(xy::pairs) = conspair(xy, unzip pairs);
```

Making the function iterative yields `revUnzip` above, 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.

Slide 507

An Application: Making Change

```
fun change (till, 0)      = []
  | change (c::till, amt) =
      if amt<c then change(till, amt)
      else c :: change(c::till, amt-c)
> Warning: pattern matching is not exhaustive
> val change = fn : int list * int -> int list
```

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

The till 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.

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 function can terminate either with success or failure. It fails by raising exception `Match`. The exception occurs if no pattern matches, namely if `till` becomes empty while `amount` is still nonzero.

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.

Slide 508

All Ways of Making Change

```
fun change (till, 0)     = [[]]
  | change ([], amt)     = []
  | change (c::till, amt) =
      if amt<c then change(till, amt)
      else
      let fun allc []        = []
            | allc(cs::css) = (c::cs)::allc css
      in allc (change(c::till, amt-c)) @
         change(till, amt)
      end;
```

Let us generalize the problem to find all possible ways of making change, returning them as a list of solutions. Look at the type: the result is now a list of lists.

```
> change : int list * int -> int list list
```

The code will 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.

Slide 509

ALL Ways of Making Change — Faster!

```
fun change(till, 0, chg, chgs)       = chg::chgs
  | change([], amt, chg, chgs)       = chgs
  | change(c::till, amt, chg, chgs) =
      if amt<0 then chgs
      else change(c::till, amt-c, c::chg,
                  change(till, amt, chg, chgs))
```

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.

We shall revisit the 'making change' task later to illustrate exception-handling.

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.

Learning guide. Related material is in *ML for the Working Programmer*, pages 82-107, though you may want to skip some of the harder examples.

Exercise 5.1 How does this version of `zip` differ from the one above?

```
fun zip (x::xs,y::ys) = (x,y) :: zip(xs,ys)
  | zip ([], [])      = [];
```

Exercise 5.2 What assumptions do the 'making change' functions make about the variables `till`, `c` and `amt`? Illustrate what could happen if some of these assumptions were violated.

Exercise 5.3 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?

Slide 601

Sorting: Arranging Items into Order

a few applications:

- fast *search*
- fast *merging*
- finding *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 [9]! Sedgewick [14] 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*: after sorting, they 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.)

How Fast Can We Sort?

Slide 602

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 at this complexity is the best possible [2, pages 86–7]. 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 [12]:

```
local val a = 16807.0  and  m = 2147483647.0
in  fun nextrandom seed =
          let val t = a*seed
          in  t - m * real(floor(t/m))  end
    and truncto k r = 1 + floor((r / m) * (real k))
end;
```

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

```
fun randlist (n,seed,seeds) =
        if n=0  then  (seed,seeds)
        else  randlist(n-1, nextrandom seed, seed::seeds);
val (seed,rs) = randlist(10000, 1.0, []);
```

Never mind how this works, but note that generating statistically good random numbers is hard. Much effort has gone into those few lines of code.

Slide 603

Insertion Sort

Insert does $n/2$ comparisons on average

```
fun ins (x:real, [])     = [x]
  | ins (x:real, y::ys) =
      if x<=y then x::y::ys
              else    y::ins(x,ys);
```

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

```
fun insort []      = []
  | insort (x::xs) = ins(x, insort xs);
```

174 seconds to sort 10,000 random numbers

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.

The type constraint `:real` resolves the overloading of the `<=` operator; recall Lect. 2. All our sorting functions will need a type constraint somewhere. The notion of sorting depends upon the form of comparison being done, which in turn determines the type of the sorting function.

Slide 604

Quicksort: The Idea

- choose a *pivot* element, a
- *Divide*: 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 C. A. R. Hoare, who has just moved from Oxford to 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.

Slide 605

Quicksort: The Code

```
fun quick []      = []
  | quick [x]     = [x]
  | quick (a::bs) =
      let fun part (l,r,[]) : real list =
                (quick l) @ (a :: quick r)
            | part (l, r, x::xs) =
                if x<=a then part(x::l, r, xs)
                        else part(l, x::r, xs)
      in  part([],[],bs)  end;
```

0.74 seconds to sort 10,000 random numbers

Our ML quicksort copies the items. It is still pretty fast, and it is much easier to understand. It takes roughly 0.74 seconds to sort `rs`, 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* [2, page 94]. 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 $O(n^2)$. Randomizing the input makes the worst case highly unlikely.

Slide 606

Append-Free Quicksort

```
fun quik([], sorted)    = sorted
  | quik([x], sorted)   = x::sorted
  | quik(a::bs, sorted) =
      let fun part (l, r, []) : real list =
                quik(l, a :: quik(r,sorted))
            | part (l, r, x::xs) =
                if x<=a then part(x::l, r, xs)
                        else part(l, x::r, xs)
      in  part([],[],bs)  end;
```

0.53 seconds to sort 10,000 random numbers

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 [14]) 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.

Merging Two Lists

Merge joins two sorted lists

Slide 607

```
fun merge([],ys)        = ys : real list
  | merge(xs,[])        = xs
  | merge(x::xs, y::ys) =
      if x<=y then x::merge(xs,  y::ys)
              else y::merge(x::xs,  ys);
```

Generalises *Insert* to two lists

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 (Lect. 4).

Slide 608

Top-down Merge sort

```
fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
      let val k = length xs div 2
      in  merge(tmergesort (take(xs, k)),
                tmergesort (drop(xs, k)))
      end;
```

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

1.4 seconds to sort 10,000 random numbers

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.

Slide 609

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. We have not considered the problem of sorting huge amounts of data using external storage media such as magnetic tape.

Learning guide. Related material is in *ML for the Working Programmer*, pages 108–113.

Slide 701

An Enumeration Type

```
datatype vehicle = Bike
                 | Motorbike
                 | Car
                 | Lorry;
```

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

The datatype declaration adds a new type to our ML session. Type `vehicle` is as good as any built-in type and even admits pattern-matching. 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 `datatype`, 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`.

Slide 702

Declaring a Function on Vehicles

```
fun wheels Bike      = 2
  | wheels Motorbike = 2
  | wheels Car       = 4
  | wheels Lorry     = 18;
> val wheels = fn : vehicle -> int
```

- Datatype constructors can be used in patterns.
- Pattern-matching is fast, even complicated nested patterns.

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

Note: ML does not always catch misspelt constructors. If one appears as the last pattern, it might be taken as a variable name. My book gives an example [13, page 131].

Slide 703

A Datatype with Constructor Functions

```
datatype vehicle = Bike
                 | Motorbike of int
                 | Car       of bool
                 | Lorry     of int;
```

- Constructor functions (like `Lorry`) make *distinct values*.
- Different kinds of `vehicle` can belong to one list:
  `[Bike, Car true, Motorbike 450];`

ML generalizes 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 are functions for creating vehicles.

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 ML, 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).

```
datatype vehicle = Bike
                 | Motorbike of int     (*engine size in CCs*)
                 | Car       of bool    (*true if a Reliant Robin*)
                 | Lorry     of int;    (*number of wheels*)
```

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.

Slide 704

A Finer Wheel Computation

```
fun wheels Bike          = 2
  | wheels (Motorbike _) = 2
  | wheels (Car robin)   =
      if robin then 3 else 4
  | wheels (Lorry w)     = w;
> val wheels = fn : vehicle -> int
```

This function consists of four clauses:

- A Bike has two wheels.
- A Motorbike has two wheels.
- A Reliant Robin has three wheels; all other cars have four.
- A Lorry has the number of wheels stored with its constructor.

There is no overlap between the Motorbike and Lorry cases. Although `Motorbike` and `Lorry` both hold an integer, ML takes the constructor into account. A Motorbike is 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. Most ML systems perform pattern-matching efficiently.

Slide 705

Error Handling: Exceptions

During a computation, what happens if something goes WRONG?

- (Arithmetic overflow or division by zero are hard to predict.)

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 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.

Slide 706

Exceptions in ML

`exception Failure;`	**Declaring**
`exception NoChange of int;`	
`raise Failure`	**Raising**
`raise (NoChange` n`)`	
E `handle Failure =>` E'	**Handling**
E `handle` P_1 `=>` E_1 `|...|` P_n `=>` E_n	**many handlers**

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 ML 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.

The effect of `raise` E is to jump to the most recently-encountered handler that matches E. The matching handler can only be found *dynamically* (during execution); contrast with how ML associates occurrences of identifiers with their matching declarations, which does not require running the program.

One criticism of ML'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`.

```
datatype 'a option = NONE | SOME of 'a;
```

`NONE` signifies 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`.

Slide 707

Making Change with Exceptions

```
exception Change;
fun change (till, 0)      = []
  | change ([],  amt)     = raise Change
  | change (c::till, amt) =
      if amt<0 then raise Change
      else (c :: change(c::till, amt-c))
           handle Change => change(till, amt);
> val change = fn : int list * int -> int list
```

In Lect. 5 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.

Making Change: A Trace

Slide 708

```
change([5,2],6)
5::change([5,2],1) handle C=>change([2],6)
5::(5::change([5,2],˜4) handle C=>change([2],1))
    handle C=>change([2],6)
5::change([2],1) handle C=>change([2],6)
5::(2::change([2],˜1) handle C=>change([],1))
    handle C=>change([2],6)
5::(change([],1)) handle C=>change([2],6)
change([2],6)
```

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)
5::change([5,2],1) handle C => change([2],6)
5::(5::change([5,2],˜4) handle C => change([2],1))
    handle C => change([2],6)
5::change([2],1) handle C => change([2],6)
5::(2::change([2],˜1) handle C => change([],1))
    handle C => change([2],6)
5::(change([],1)) handle C => change([2],6)
change([2],6)
2::change([2],4) handle C => change([],6)
2::(2::change([2],2) handle C => change([],4)) handle ...
2::(2::(2::change([2],0) handle C => change([],2)) handle C => ...)
2::(2::[2] handle C => change([],4)) handle C => change([],6)
2::[2,2] handle C => change([],6)
[2,2,2]
```

Binary Trees, a Recursive Datatype

Slide 709

```
datatype 'a tree = Lf
                 | Br of 'a * 'a tree * 'a 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.

Lists themselves could be declared using datatype:

```
datatype 'a list = nil
                 | cons of 'a * 'a list
```

We could even declare `::` as an *infix constructor*. The only thing we could not define is the `[...]` notation, which is part of the ML grammar.

Slide 710

Basic Properties of Binary Trees

```
fun count Lf              = 0        # of branch nodes
  | count(Br(v,t1,t2)) = 1 + count t1 + count t2

fun depth Lf              = 0   length of longest path
  | depth(Br(v,t1,t2)) = 1 +
                          max(depth t1, depth t2)
```

$$count(t) \leq 2^{depth(t)} - 1$$

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:

```
fun leaves Lf = 1
  | leaves (Br(v,t1,t2)) = leaves t1 + leaves t2;
```

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!

Traversing Trees (3 Methods)

Slide 711

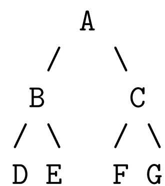
```
fun preorder Lf           = []
  | preorder(Br(v,t1,t2)) =
      [v] @ preorder t1 @ preorder t2;

fun inorder Lf           = []
  | inorder(Br(v,t1,t2)) =
      inorder t1 @ [v] @ inorder t2;

fun postorder Lf           = []
  | postorder(Br(v,t1,t2)) =
      postorder t1 @ postorder t2 @ [v];
```

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 [10]. 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

Slide 712

Efficiently Traversing Trees

```
fun preord (Lf, vs)           = vs
  | preord (Br(v,t1,t2), vs) =
      v :: preord (t1, preord (t2, vs));

fun inord (Lf, vs)           = vs
  | inord (Br(v,t1,t2), vs) =
      inord (t1, v::inord (t2, vs));

fun postord (Lf, vs)           = vs
  | postord (Br(v,t1,t2), vs) =
      postord (t1, postord (t2, v::vs));
```

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 Lect. 6.

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

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

Learning guide. Related material is in *ML for the Working Programmer*, pages 123–147.

Exercise 7.1 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.

Exercise 7.2 Show that the functions `preord`, `inord` and `postord` all take linear time in the size of the tree.

Slide 801

Dictionaries

- **lookup**: find an item in the dictionary
- **update**: 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**

Our *abstract type* supports these operations

. . . but *hides* the implementation!

A dictionary is a data structure that associates values to certain identifiers, called *keys.* When choosing the internal representation for a data structure, it is essential to specify the full set of operations that must be supported. Seldom is one representation best for all possible applications of a data structure; each will support some operations well and others badly.

We consider simple dictionaries that support only *update* (associating a value with an identifier) and *lookup* (searching for the value associated with an identifier). Other operations that could be considered are *delete* (removing an association) and *merge* (combining two dictionaries). Since we are programming in a functional style, update will not modify the data structure. Instead, it will return a modified data structure. This can be done efficiently if we are careful to avoid excessive copying.

Lookup and delete fail unless they find the desired key. We use ML's exceptions to signal failure.

Modern programming languages provide a means of declaring abstract types that export well-defined operations while hiding low-level implementation details such as the data structure used to represent dictionaries. ML provides *modules* for this purpose, but this course does not cover them. (The Java course covers modularity.) Therefore, we shall simply declare the dictionary operations individually at top level. We shall encounter many versions of `lookup`, for example, that ought to be packaged in separate modules to prevent clashes.

Slide 802

Association Lists: Lists of Pairs

```
exception Missing;

fun lookup ([], a)           = raise Missing
  | lookup ((x,y)::pairs, a) =
      if a=x then  y  else  lookup(pairs, a);
> val lookup = fn : (''a * 'b) list * ''a -> 'b

fun update(l, b, y) = (b,y)::l
```

PROBLEMS: Linear search is slow. The lists get too long!

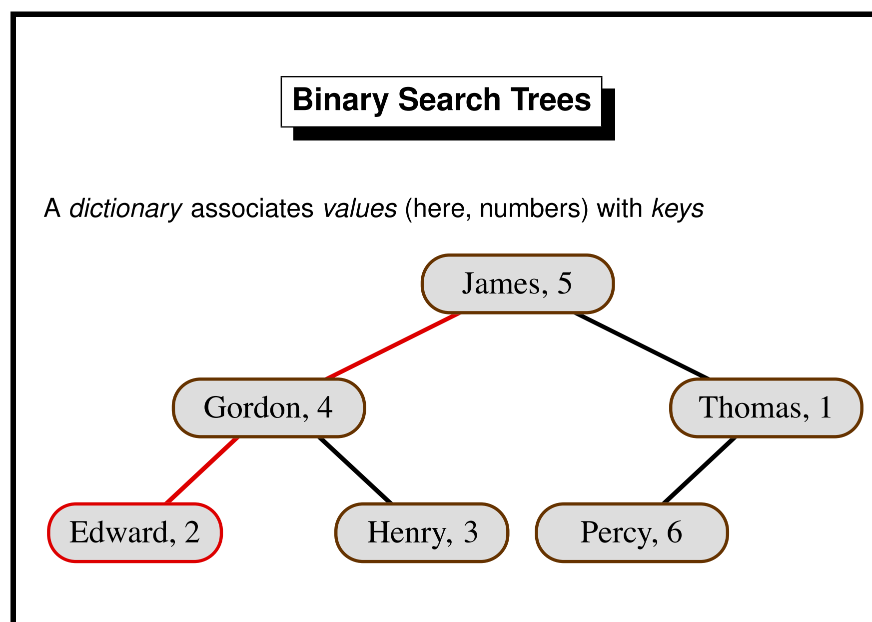
A list of pairs is the most obvious representation for a dictionary.

Lookup is by linear search, which we know to be prohibitively slow: $O(n)$. Association lists are only usable if there are few keys of interest, always near the front. However, note `lookup`'s type: association lists work for any equality type. This generality is their main advantage.

To enter a new (key, value) association, simply put a new pair into the list. This takes constant time, which is the best we could hope for. But the space requirement is huge. It is linear in the number of updates, not in the number of distinct keys, because obsolete entries are never deleted. To delete old entries would require first finding them, increasing the update time from $O(1)$ to $O(n)$.

Function `lookup` is traditionally called `assoc` after a similar function in the language Lisp, which played a historic role in the definition of the Lisp evaluator.

Slide 803



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.

Slide 804

Lookup: Seeks Left or Right

```
exception Missing of string;

fun lookup (Br ((a,x),t1,t2), b) =
      if      b < a then  lookup(t1, b)
      else if a < b then  lookup(t2, b)
      else x
  | lookup (Lf, b) = raise Missing b;
> val lookup = fn : (string * 'a) tree * string
>                   -> 'a
```

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 exception `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 Lect. 7, using the constructor `NONE` for failure.

Slide 805

Update

```
fun update (Lf, b:string, y) = Br((b,y), Lf, Lf)
  | update (Br((a,x),t1,t2), b, y) =
      if b<a
      then  Br ((a,x),  update(t1,b,y),  t2)
      else
      if a<b
      then  Br ((a,x),  t1,  update(t2,b,y))
      else (*a=b*) Br ((a,y),t1,t2);
```

Also $O(\log n)$: it copies the path only, **not whole subtrees!**

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*. (Lect. 15 will discuss using references to create linked structures.) 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 ML are enclosed in the brackets `(*` and `*)`.

Slide 806

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*: updating is an action.

A *functional Array* is a finite map from integers to data.

- Updating is *copying* in `update(A,k,x)`
- The new array equals `A` except at position `k`.

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*.

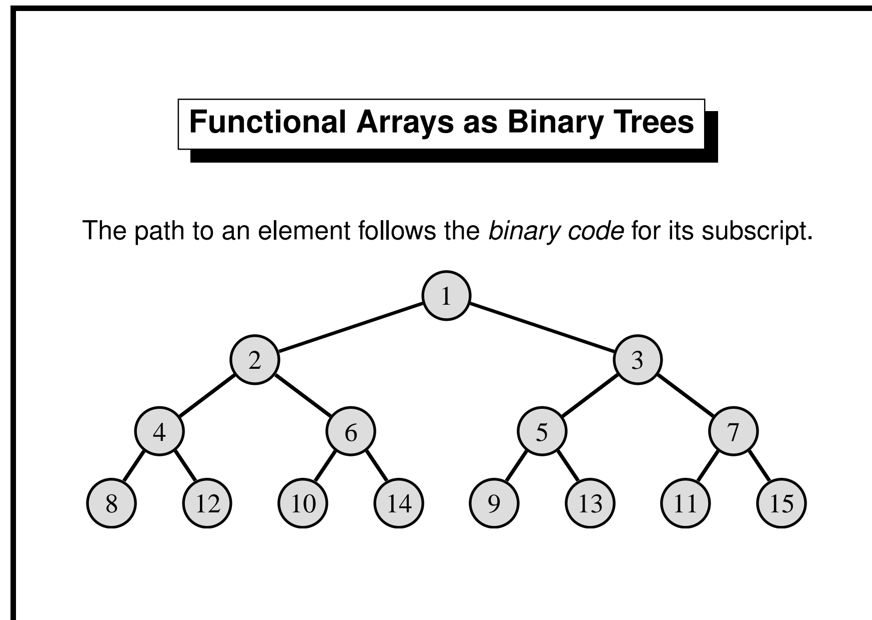
Arrays are the dominant data structure in conventional programming languages. The ingenious use of arrays is the key to many of the great classical algorithms, such as Hoare's original quicksort (the partition step) and Warshall's transitive-closure algorithm.

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 organize data. Here is a summary of our 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, linear in the worst case.
- *Array subscripting*: Least general, requiring keys to be integers, but even worst-case time is logarithmic.

Slide 807



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 indices is one. The upper bound starts at zero (which signifies the empty array) and can grow without limit. This data structure can be used to implement arrays that grow and shrink by adding and deleting elements at either end.

Slide 808

The Lookup Function

```
exception Subscript;

fun sub (Lf, _) = raise Subscript          (*Not found!*)
  | sub (Br(v,t1,t2), k) =
        if k=1 then v
        else if k mod 2 = 0
             then sub (t1, k div 2)
             else sub (t2, k div 2);
```

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 understand 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.

Slide 809

The Update Function

```
fun update (Lf, k, w)          =
      if k = 1 then Br (w, Lf, Lf)
      else raise Subscript                  (*Gap in tree!*)
  | update (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 div 2, w),  t2)
      else Br (v,  t1,  update(t2, k div 2, w))
```

The update function, `update`, 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.

Learning guide. Related material is in *ML for the Working Programmer*, pages 148–159.

Slide 901

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*. (The ML datatype `tree` contains only finite trees, but ML can represent infinite trees by means discussed in Lect. 13.) Depth-first search is almost useless with infinite trees, for if the left subtree is infinite then it will never reach the right subtree.

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.

Slide 902

Breadth-First Tree Traversal — Using Append

```
fun nbreadth []                  = []
  | nbreadth (Lf :: ts)          = nbreadth ts
  | nbreadth (Br(v,t,u) :: ts) =
                v :: nbreadth(ts @ [t,u])
```

Keeps an *enormous queue* of nodes of search

Wasteful use of *append*

25 SECS to search depth 12 binary tree (4095 labels)

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.

Slide 903

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 (Lect. 8) 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.

Slide 904

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 *rear list*

Remove items from *front list*; 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.

Efficient Functional Queues: Code

Slide 905

```
datatype 'a queue = Q of 'a list * 'a list

fun norm(Q([],tls)) = Q(rev tls, [])
  | norm q          = q

fun qnull(Q([],[])) = true  |  qnull _ = false

fun enq(Q(hds,tls), x)  = norm(Q(hds, x::tls))

fun deq(Q(x::hds, tls)) = norm(Q(hds, tls))
```

The datatype of queues prevents confusion with other pairs of lists.

The empty queue, omitted to save space on the slide, has both parts empty.

```
val qempty = Q([],[]);
```

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 normalize their result.

Because queues are in normal form, their head is certain to be in their front part, so `qhd` (also omitted from the slide) looks there.

```
fun qhd(Q(x::_,_)) = x
```

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.

Slide 906

Aside: The case Expression

```
fun wheels v =
    case v of Bike        => 2
         | Motorbike _ => 2
         | Car robin   =>
           if robin then 3 else 4
         | Lorry w     => w;
```

The `case` expression has the form

`case` E `of` P_1 `=>` E_1 `|` $\cdots$ `|` P_n `=>` E_n

It tries the patterns one after the other. When one matches, it evaluates the corresponding expression. It behaves precisely like the body of a function declaration. We could have defined function `wheels` (from Lect. 7) as shown above.

A program phrase of the form P_1 `=>` E_1 `|` $\cdots$ `|` P_n `=>` E_n is called a *Match*. A match may also appear after an exception handler (Lect. 7) and with `fn`-notation to expression functions directly (Lect. 10).

Slide 907

Breadth-First Tree Traversal — Using Queues

```
fun breadth q =
  if qnull q then []
  else
  case qhd q of
      Lf => breadth (deq q)
    | Br(v,t,u) =>
         v :: breadth(enq(enq(deq q, t), u))
```

0.14 secs to search depth 12 binary tree (4095 labels)

200 times faster!

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.

Slide 908

Iterative deepening: Another Exhaustive Search

Breadth-first search examines $O(b^d)$ nodes:

$$1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

$b =$ branching factor

$d =$ depth

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 big problems: it uses too much space. 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 [11] 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.

Slide 909

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.

Slide 910

A Survey of Search Methods

1. **Depth-first**: use a *stack* (efficient but incomplete)
2. **Breadth-first**: use a *queue* (uses too much space!)
3. **Iterative deepening**: use (1) to get benefits of (2)
   (trades time for space)
4. **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.

Learning guide. Related material is in *ML for the Working Programmer*, pages 258–263. For priority queues, see 159–164.

Slide 1001

Functions as Values

In ML, functions can be

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

Functions represent *algorithms* and *infinite data structures*.

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 realized. 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 transforms other functions. Many functionals are familiar from mathematics, such as integral and differential operators of the calculus. To a mathematician, a function is typically an infinite, uncomputable object. We use ML functions to represent algorithms. Sometimes they represent infinite collections of data given by computation rules.

Functions cannot be compared for equality. The best we could do, with reasonable efficiency, would be to test identity of machine addresses. Two separate occurrences of the same function declaration would be regarded as unequal because they would be compiled to different machine addresses. Such a low-level feature has no place in a language like ML.

Slide 1002

Functions Without Names

<u>fn</u> x => E is the function f such that $f(x) = E$

The function (<u>fn</u> n => n*2) is a *doubling function*.

```
(fn n => n*2);
> val it = fn : int -> int

(fn n => n*2) 17;
> val it = 34 : int
```

If functions are to be regarded as computational values, then we need a notation for them. The `fn`-notation expresses a function value without giving the function a name. (Some people pronounce `fn` as 'lambda' because it originated in the λ-calculus.) It cannot express recursion. Its main purpose is to package up small expressions that are to be applied repeatedly using some other function.

The expression (<u>fn</u> n => n*2) has the same value as the identifier `double`, declared as follows:

```
fun double n = n*2
```

The `fn`-notation allows pattern-matching, like <u>case</u> expressions and exception handlers, to express functions with multiple clauses:

<u>fn</u> P_1 => E_1 | ... | P_n => E_n

This rarely-used expression abbreviates the local declaration

<u>let</u> <u>fun</u> $f(P_1)$ = E_1 | ... | $f(P_n)$ = E_n
<u>in</u> f <u>end</u>

For example, the following declarations are equivalent:

```
val not   = (fn false => true | true => false)

fun not false = true
  | not true = false
```

Slide 1003

Curried Functions

A *curried function* returns another function as its *result*.

```
val prefix = (fn a => (fn b => a^b));
> val prefix = fn: string -> (string -> string)
```

`prefix` yields functions of type `string -> string`.

Each of these refers to some value of the argument `a`.

The `fn`-notation lets us package `n*2` as the function `(fn n => n*2)`, but what if there are several variables, as in `(n*2+k)`? If the variable `k` is defined in the current context, then

```
fn n => n*2+k
```

is still meaningful. To make a function of two arguments, we may use pattern-matching on pairs, writing

```
fn (n,k) => n*2+k
```

A more interesting alternative is to *nest* the `fn`-notation:

```
fn k => (fn n => n*2+k)
```

Applying this function to the argument 1 yields another function,

```
fn n => n*2+1
```

which, when applied to 3, yields the result 7.

The example on the slide is similar but refers to the expression `a^b`, where `^` is the infix operator for string concatenation.

Using a Curried Function

Slide 1004

Let's give one of these functions the name `promote`.

```
val promote = prefix "Professor ";
> val promote = fn: string -> string
promote "Mop";
> "Professor Mop" : string
```

Or we can apply `prefix` to two arguments at once:

```
(prefix "Doctor ") "Who";
> val "Doctor Who" : string
```

Function `promote` binds the first argument of `prefix` to the string `"Professor "`; the resulting function prefixes that title to any string to which it is applied.

Note: The parentheses may be omitted in `(fn a => (fn b => E))`. They may also be omitted in `(prefix "Doctor ") "Who"`.

Shorthand for Curried Functions

Slide 1005

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

This syntax is nicer than nested `fn` binders:

```
fun prefix a b = a^b;
> val prefix = ...                          as before

val dub = prefix "Sir ";
> val dub = fn: string -> string
```

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

The n-argument curried function `f` is conveniently declared using the syntax

```
fun f x1 ...xn = ...
```

and applied using the syntax `f E1 ...En`.

We now have two ways—pairs and currying—of expressing functions of multiple arguments. 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 definite integral $\int_x^y f(z)\,dz$, where fixing $x = x_0$ yields a function in y alone.

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:

```
hd [dub, promote] "Hamilton";
> val "Sir Hamilton" : string
```

Here `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, as found in languages like Java and C++.

Slide 1006

Partial Application: A Curried Insertion Sort

```
fun insort lessequal =
    let fun ins (x, []) = [x]
          | ins (x, y::ys) =
              if lessequal(x,y) then x::y::ys
              else y :: ins (x,ys)
        fun sort [] = []
          | sort (x::xs) = ins (x, sort xs)
    in  sort  end;

> val insort = fn : ('a * 'a -> bool)
>                  -> ('a list -> 'a list)
```

The sorting functions of Lect. 6 are coded to sort real numbers. They can be generalized to an arbitrary ordered type by passing the ordering predicate ($\leq$) 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.

Examples of Generic Sorting

Slide 1007

```
insort (op<=) [5,3,9,8];
> val it = [3, 5, 8, 9] : int list

insort (op<=) ["bitten","on","a","bee"];
> val it = ["a", "bee", "bitten", "on"]
>          : string list

insort (op>=) [5,3,9,8];
> val it = [9, 8, 5, 3] : int list
```

Note: `op<=` stands for the `<=` operator regarded as a value. Although infixes are functions, normally they can only appear in expressions such as `n<=9`. The `op` syntax lets us write `op<=(n,9)`, but on the slide we use `op<=` to pass the comparison operator to function `insort`.

To exploit sorting to its full extent, we need the greatest flexibility in expressing orderings. There are many types of basic data, such as integers, reals and strings. On the overhead, we sort integers and strings. The operator `<=` is overloaded, working for types `int`, `real` and `string`. The list supplied as `insort`'s second argument resolves the overloading ambiguity.

Passing the relation $\geq$ for `lessequal` gives a decreasing sort. This is no coding trick; it is justified in mathematics. If $\leq$ is a partial ordering then so is $\geq$.

There are many ways of combining orderings. Most important is the *lexicographic ordering*, in which two keys are used for comparisons. It is specified by $(x', y') < (x, y) \iff x' < x \vee (x' = x \wedge y' < y)$. Often part of the data plays no role in the ordering; consider the text of the entries in an encyclopedia. Mathematically, we have an ordering on pairs such that $(x', y') < (x, y) \iff x' < x$.

These ways of combining orderings can be expressed in ML as functions that take orderings as arguments and return other orderings as results.

Slide 1008

A Summation Functional

Sums the values of $f(i)$ for $1 \leq i \leq m$.

```
fun sum f 0 = 0.0
  | sum f m = f(m) + sum f (m-1);
> val sum = fn: (int -> real) -> (int -> real)

sum (fn k => real (k*k)) 5;
> val it = 55.0 : real
```

sum f m $= \sum_{i=1}^{m} f(i)$

Above we see that $1 + 4 + 9 + 16 + 25 = 55$.

Numerical programming languages, such as Fortran, allow functions to be passed as arguments in this manner. Classical applications include numerical integration and root-finding.

Thanks to currying, ML surpasses Fortran. Not only can `f` be passed as an argument to `sum`, but the result of doing so can itself be returned as another function. Given an integer argument, that function returns the result of summing values of `f` up to the specified bound.

Nesting the Summation Functional

Slide 1009

```
sum (fn i=>
        sum (fn j=>h(i,j))
            n
    )
    m
```

$$= \sum_{i=1}^{m} \sum_{j=1}^{n} h(i,j)$$

`sum (sum f) m` $= \sum_{i=1}^{m} \sum_{j=1}^{i} f(j)$

These examples demonstrate how `fn`-notation expresses dependence on bound variables, just as in ordinary mathematics. The functional `sum` can be repeated like the traditional Σ sign.

Let us examine the first example in detail:

- `h(i,j)` depends upon the variables `i` and `j`
- `fn j=>h(i,j)` depends upon `i` alone, yielding a function over `j`
- `sum (fn j=>h(i,j)) n` depends upon `i` and `n`, summing the function over `j` mentioned above
- `fn i => sum` $\cdots$ `n` depends upon `n` alone, yielding a function over `i`

The expression as a whole depends upon the three variables `f`, `m` and `n`.

Functionals, currying and `fn`-notation yield a language for expressions that is grounded in mathematics, concise and powerful.

Slide 1010

Historical Remarks

Frege (1893): if functions are values, we need **unary** functions only

Schöfinkel (1924): with the right **combinators**, we don't need variables!

Church (1936): the λ-calculus & unsolvable problems

Landin (1964-6): ISWIM: a language based on the λ-calculus

Turner (1979): combinators as an implementation technique

The idea that functions could be regarded as values in themselves gained acceptance in the 19th century. Frege's mammoth (but ultimately doomed) logical system was based upon this notion. Frege discovered what we now call Currying: that having functions as values meant that functions of several arguments could be formalized using single-argument functions only.

Another logician, Schönfinkel, rediscovered this fact and developed combinators as a means of eliminating variables from expressions. Given $\mathbf{K}$ and $\mathbf{S}$ such that $\mathbf{K}xy = x$ and $\mathbf{S}xyz = xz(yz)$, any functional expression could be written without using bound variables. Currying is named after Haskell B. Curry, who made deep investigations into the theory of combinators.

Alonzo Church's λ-calculus gave a simple syntax, λ-notation, for expressing functions. It is the direct precursor of ML's `fn`-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. However, Landin sketched out the main features of functional languages. Turner made the remarkable discovery that combinators (hitherto thought to be of theoretical value only) could an effective means of implementing functional languages that employed lazy evaluation.

Learning guide. Related material is in *ML for the Working Programmer*, pages 171–179. Chapter 9 contains an introduction to the λ-calculus, whch will be covered in the second-year course *Foundations of Functional Programming*.

Exercise 10.1 Write an ML function to combine two orderings lexicographically. Explain how it allows function `insort` to sort a list of pairs, using both components in the comparisons.

Exercise 10.2 Code an iterative version of `sum`, a curried function of three arguments. Does it matter whether the accumulator is the first, second or third argument?

Exercise 10.3 Explain the second example of `sum` on the overhead. What is `(sum f)`?

Slide 1101

map: the 'Apply to All' Functional

```
fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs
> val map = fn: ('a -> 'b) -> 'a list -> 'b list

map (fn s => s ^ "ppy") ["Hi", "Ho"];
> val it = ["Hippy", "Hoppy"] : string list

map (map double) [[1], [2,3]];
> val it = [[2], [4, 6]] : int list list
```

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 in this lecture. We again see the advantages of `fn`-notation, currying and `map`. If we did not have them, the first example on the slide would require a preliminary function declaration:

```
fun sillylist [] = []
  | sillylist (s::ss) = (s ^ "ppy") :: sillylist ss;
```

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:

```
fun double n = n*2;
```

Note that `map` is a built-in ML function. Standard ML's library includes, among much else, many list functions.

Slide 1102

Example: Matrix Transpose

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

```
fun hd (x::_) = x;
fun tl (_::xs) = xs;

fun transp ([]::_) = []
  | transp rows    = (map hd rows) ::
                     (transp (map tl rows))
```

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 ML corresponds to the following transformation on lists of lists:

```
[[a,b,c], [d,e,f]]  ↦  [[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:

```
map hd rows ↦ [a,d]
map tl rows ↦ [[b,c], [e,f]]
```

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.

Slide 1103

Review of Matrix Multiplication

$$\begin{pmatrix} A_1 & \cdots & A_k \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ \vdots \\ B_k \end{pmatrix} = \begin{pmatrix} A_1B_1 + \cdots + A_kB_k \end{pmatrix}$$

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

$$(a_1, \ldots, a_k) \cdot (b_1, \ldots, b_k) = a_1b_1 + \cdots + a_kb_k.$$

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.

Slide 1104

Matrix Multiplication in ML

Dot product of two vectors—a **curried function**

```
fun dotprod []      []      = 0.0
  | dotprod(x::xs)(y::ys) = x*y + dotprod xs ys
```

Matrix product

```
fun matprod(Arows,Brows) =
  let val cols = transp Brows
  in  map (fn row => map (dotprod row) cols)
          Arows
  end
```

The `transp Brows` converts B into a list of columns. It yields 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 `fn`-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` is raised.

Slide 1105

The 'Fold' Functionals

```
fun foldl f (e, [])     = e
  | foldl f (e, x::xs) = foldl f (f(e,x), xs)

fun foldr f ([],    e) = e
  | foldr f (x::xs, e) = f(x, foldr f (xs,e))
```

They do *recursion down a list*:

$$\texttt{foldl}\oplus: \quad (e,\ [x_1,\ldots,x_n]) \longmapsto (\cdots(e\oplus x_1)\oplus\cdots)\oplus x_n$$

$$\texttt{foldr}\oplus: \quad ([x_1,\ldots,x_n],\ e) \longmapsto x_1\oplus(\cdots\oplus(x_n\oplus e)\cdots)$$

These functionals start with an initial value e. They combine it with the list elements one at a time, using the function $\oplus$. While `foldl` takes the list elements from left to right, `foldr` takes them from right to left. Here are their types:

```
> val foldl = fn: ('a * 'b -> 'a) -> 'a * 'b list -> 'a
> val foldr = fn: ('a * 'b -> 'b) -> 'a list * 'b -> 'b
```

Obvious applications or `foldl` or `foldr` are to add or multiply a list of numbers. Many recursive functions on lists can be expressed concisely. Some of them follow common idioms and are easily understood. But you can easily write incomprehensible code, too.

The relationship between `foldr` and the list datatype is particularly close. Here is the list [1,2,3,4] in its internal format:

$$\begin{array}{lllll} :: \rightarrow & :: \rightarrow & :: \rightarrow & :: \rightarrow & nil \\ \downarrow & \downarrow & \downarrow & \downarrow & \\ 1 & 2 & 3 & 4 & \end{array}$$

Compare with the expression computed by `foldr`$(\oplus, e)$. the final *nil* is replaced by e; the conses are replaced by $\oplus$.

$$\begin{array}{lllll} \oplus \rightarrow & \oplus \rightarrow & \oplus \rightarrow & \oplus \rightarrow & e \\ \downarrow & \downarrow & \downarrow & \downarrow & \\ 1 & 2 & 3 & 4 & \end{array}$$

Slide 1106

Defining List Functions Using `foldl/r`

`foldl op+ (0,xs)`	*sum*
`foldr op:: (xs,ys)`	*append*
`foldl (foldl op+) (0,ls)`	*sum of sums!*
`foldl (fn(e,x) => e+1) (0,xs)`	*length*
`foldl (fn(e,x)=>x::e) ([],xs)`	*reverse*

The sum of a list's elements is formed by starting with zero and adding each list element in turn. Using `foldr` would be less efficient, requiring linear instead of constant space. Note that `op+` turns the infix addition operator into a function that can be passed to other functions such as `foldl`. Append is expressed similarly, using `op::` to stand for the cons function.

The sum-of-sums computation is *space-efficient*: it does not form an intermediate list of sums. Moreover, `foldl` is *iterative*. Carefully observe how the inner `foldl` expresses a function to add a number of a list; the outer `foldl` applies this function to each list in turn, accumulating a sum starting from zero.

The nesting in the sum-of-sums calculation is typical of well-designed fold functionals. Similar functionals can be declared for other data structures, such as trees. Nesting these functions provides a convenient means of operating on nested data structures, such as trees of lists.

The length computation might be regarded as frivolous. A trivial function is supplied using `fn`-notation; it ignores the list elements except to count them. However, this length function takes constant space, which is better than naïve versions such as `nlength` (Lect. 4). Using `foldl` guarantees an iterative solution with an accumulator.

Slide 1107

List Functionals for Predicates

```
fun exists p []       = false
  | exists p (x::xs) = (p x) orelse exists p xs;
> exists: ('a -> bool) -> ('a list -> bool)

fun filter p []       = []
  | filter p (x::xs) =
       if p x then x :: filter p xs
              else      filter p xs;
> filter: ('a -> bool) -> ('a list -> 'a list)
```

(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 `orelse`; this aspect of `exists` cannot be obtained using the `fold` functionals.

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

```
fun all p []      = true
  | all p (x::xs) = (p x)  andalso all p xs;
> all: ('a -> bool) -> ('a list -> bool)
```

The `filter` functional is related to `map`. 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.

Applications of the Predicate Functionals

Slide 1108

```
fun member(y,xs) =
    exists (fn x => x=y) xs;

fun inter(xs,ys) =
    filter (fn x => member(x,ys)) xs;
```

Testing whether two lists have no common elements

```
fun disjoint(xs,ys) =
    all (fn x => all (fn y => x<>y) ys) xs;
> val disjoint = fn: ''a list * ''a list -> bool
```

Again, by way of example, we consider applications of the predicate functionals. Lecture 5 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.

Tree Functionals

Slide 1109

```
fun maptree f Lf = Lf
  | maptree f (Br(v,t1,t2)) =
        Br(f v, maptree f t1, maptree f t2);
> val maptree =  fn
>      : ('a -> 'b) -> 'a tree -> 'b tree

fun fold f e Lf = e
  | fold f e (Br(v,t1,t2)) =
        f (v, fold f e t1, fold f e t2);
> val fold = fn
>   : ('a * 'b * 'b -> 'b) -> 'b -> 'a tree -> 'b
```

The ideas presented in this lecture generalize in the obvious way to trees and other datatypes, not necessarily recursive ones.

The functional `maptree` applies a function to every label of a tree, returning another tree of the same shape. Analogues of `exists` and `all` are trivial to declare. On the other hand, `filter` is hard because removing the filtered labels changes the tree's shape; if a label fails to satisfy the predicate, there is no obvious way to include the result of filtering both subtrees.

The easiest way of declaring a `fold` functional is as shown above. The arguments `f` and `e` replace the constructors `Br` and `Lf`, respectively. This functional can be used to add a tree's labels, but it requires a three-argument addition function. To avoid this inconvenience, fold functionals for trees can implicitly treat the tree as a list. For example, here is a fold function related to `foldr`, which processes the labels in inorder:

```
fun infold f (Lf, e) = e
  | infold f (Br(v,t1,t2), e) = infold f (t1, f (v, infold f (t2, e)));
```

Its code is derived from that of the function `inord` of Lect. 7 by generalizing cons to the function `f`.

Our primitives themselves can be seen as a programming language. This truth is particularly obvious in the case of functionals, but it holds of programming in general. 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.

Learning guide. Related material is in *ML for the Working Programmer*, pages 182–190.

Exercise 11.1 Without using `map`, currying, etc., write a function that is equivalent to `map (map double)`. The obvious solution requires declaring two recursive functions. Try to get away with only one by exploiting nested pattern-matching.

Exercise 11.2 Express the functional `map` using `foldr`.

Exercise 11.3 Declare an analogue of `map` for type `option`:

```
datatype 'a option = NONE | SOME of 'a;
```

Exercise 11.4 Recall the making change function of Lect. 5:

```
fun change ...
  | change (c::till, amt) =
      if ...
      else
      let fun allc []        = []
            | allc(cs::css) = (c::cs)::allc css
      in allc (change(c::till, amt-c)) @
         change(till, amt)
      end;
```

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`.

Slide 1201

Computer Algebra

symbolic arithmetic on polynomials, trig functions, ...

closed-form or *power-series* solutions, not NUMERICAL ones

rational arithmetic instead of FLOATING-POINT

For *scientific* and *engineering* calculations

Univariate polynomials $a_n x^n + \cdots + a_0 x^0$

Example of **data representation** and **algorithms in practice**

This lecture illustrates the treatment of a hard problem: polynomial arithmetic. Many operations could be performed on polynomials, so we shall have to simplify the problem drastically. We shall only consider functions to add and multiply polynomials in one variable. These functions are neither efficient nor accurate, but at least they make a start. *Beware*: efficient, general algorithms for polynomials are complicated enough to boggle the mind.

Although computers were originally invented for performing numerical arithmetic, scientists and engineers often prefer closed-form solutions to problems. A formula is more compact than a table of numbers, and its properties—the number of crossings through zero, for example—can be determined exactly.

Polynomials are a particularly simple kind of formula. A polynomial is a linear combination of products of certain variables. For example, a polynomial in the variables x, y and z has the form $\sum_{ijk} a_{ijk}\, x^i y^j z^k$, where only finitely many of the coefficients a_{ijk} are non-zero. Polynomials in one variable, say x, are called *univariate*. Even restricting ourselves to univariate polynomials does not make our task easy.

This example demonstrates how to represent a non-trivial form of data and how to exploit basic algorithmic ideas to gain efficiency.

Slide 1202

Data Representation Example: *Finite Sets*

represent by *repetition-free lists*

representations not *unique*: $\{3,4\}$ $\swarrow$ $\searrow$ $[3,4]$ $[4,3]$

INVALID representations? $[3,3]$ represents no set

ML operations must *preserve* the representation

Representation must promote *efficiency*: try *ordered lists*?

ML does not provide finite sets as a data structure. We could represent them by lists without repetitions. Finite sets are a simple example of *data representation*. A collection of *abstract* objects (finite sets) is represented using a set of *concrete* objects (repetition-free lists). Every abstract object is represented by at least one concrete object, maybe more than one, for $\{3,4\}$ can be represented by $[3,4]$ or $[4,3]$. Some concrete objects, such as $[3,3]$, represent no abstract object at all.

Operations on the abstract data are defined in terms of the representations. For example, the ML function `inter` (Lect. 5) implements the abstract intersection operation $\cap$ provided `inter`(l,l') represents $A\cap A'$ for all lists l and l' that represent the sets A and A'. It is easy to check that `inter` preserves the representation: its result is repetition-free provided its arguments are.

Making the lists repetition-free makes the best possible use of space. Time complexity could be improved. Forming the intersection of an m-element set and an n-element set requires finding all the elements they have in common. It can only be done by trying all possibilities, taking $O(mn)$ time. Sets of numbers, strings or other items possessing a total ordering should be represented by ordered lists. The intersection computation then resembles merging and can be performed in $O(m+n)$ time.

Some deeper issues can only be mentioned here. For example, floating-point arithmetic implements real arithmetic only approximately.

Slide 1203

A Data Structure for Polynomials

polynomial $a_n x^n + \cdots + a_0 x^0$ as list $[(n, a_n), \ldots, (0, a_0)]$

REAL coefficients (should be *rational*)

Sparse representation (no zero coefficients)

Decreasing exponents

$$x^{500} - 2 \quad \textit{as} \quad [(500, 1), (0, -2)]$$

The univariate polynomial $a_n x^n + \cdots + a_0 x^0$ might be represented by the list of coefficients $[a_n, \ldots, a_0]$. This *dense* representation is inefficient if many coefficients are zero, as in $x^{500} - 2$. Instead we use a list of (exponent, coefficient) pairs with only nonzero coefficients: a *sparse* representation.

Coefficients should be *rational numbers*: pairs of integers with no common factor. Exact rational arithmetic is easily done, but it requires arbitrary-precision integer arithmetic, which is too complicated for our purposes. We shall represent coefficients by the ML type `real`, which is far from ideal. The code serves the purpose of illustrating some algorithms for polynomial arithmetic.

Polynomials will have the ML type `(int*real)list`, representing the sum of terms, each term given by an integer exponent and real coefficient. To promote efficiency, we not only omit zero coefficients but store the pairs in decreasing order of exponents. The ordering allows algorithms resembling mergesort and allows at most one term to have a given exponent.

The *degree* of a non-zero univariate polynomial is its largest exponent. If $a_n \neq 0$ then $a_n x^n + \cdots + a_0 x^0$ has degree n. Our representation makes it trivial to compute a polynomial's degree.

For example, `[(500,1.0), (0,~2.0)]` represents $x^{500} - 2$. Not every list of type `(int*real)list` is a polynomial. Our operations may assume their arguments to be valid polynomials and are required to deliver valid polynomials.

Slide 1204

Specifying the Polynomial Operations

- `poly` is the *type* of univariate polynomials
- `makepoly` *makes* a polynomial from a list
- `destpoly` *returns* a polynomial as a list
- `polysum` *adds* two polynomials
- `polyprod` *multiplies* two polynomials
- `polyquorem` computes *quotient* and *remainder*

An implementation of univariate polynomials might support the operations above, which could be summarized as follows:

```
type poly
val makepoly   : (int*real)list -> poly
val destpoly   : poly -> (int*real)list
val polysum    : poly -> poly -> poly
val polyprod   : poly -> poly -> poly
val polyquorem : poly -> poly -> poly * poly
```

This tidy specification can be captured as an ML *signature*. A bundle of declarations meeting the signature can be packaged as an ML *structure*. These concepts promote modularity, letting us keep the higher abstraction levels tidy. In particular, the structure might have the name `Poly` and its components could have the short names `sum`, `prod`, etc.; from outside the structure, they would be called `Poly.sum`, `Poly.prod`, etc. This course does not discuss ML modules, but a modular treatment of polynomials can be found in my book [13]. Modules are essential for building large systems.

Function `makepoly` could convert a list to a valid polynomial, while `destpoly` could return the underlying list. For many abstract types, the underlying representation ought to be hidden. For dictionaries (Lect. 8), we certainly do not want an operation to return a dictionary as a binary search tree. Our list-of-pairs representation, however, is suitable for communicating polynomials to the outside world. It might be retained for that purpose even if some other representation were chosen to facilitate fast arithmetic.

Polynomial addition

Slide 1205

```
fun polysum [] us = us : (int*real)list
  | polysum ts [] = ts
  | polysum ((m,a)::ts) ((n,b)::us) =
      if m>n then
         (m,a) :: polysum ts ((n,b)::us)
      else if n>m then
         (n,b) :: polysum us ((m,a)::ts)
      else (*m=n*) if a+b=0.0 then
         polysum ts us
      else (m, a+b) :: polysum ts us;
```

Our representation allows addition, multiplication and division to be performed using the classical algorithms taught in schools. Their efficiency can sometimes be improved upon. For no particular reason, the arithmetic functions are all curried.

Addition involves adding corresponding coefficients from the two polynomials. Preserving the polynomial representation requires preserving the ordering and omitting zero coefficients.[1]

The addition algorithm resembles merging. If both polynomials are non-empty lists, compare their leading terms. Take the term with the larger exponent first. If the exponents are equal, then create a single term, adding their coefficients; if the sum is zero, then discard the new term.

[1] Some ML compilers insist upon `Real.==(a+b,0.0)` instead of `a+b=0.0` above.

Slide 1206

Polynomial multiplication (1st try)

```
fun termprod (m,a) (n,b)                    term × term
    = (m+n, a*b) : (int*real);

fun polyprod [] us = []                     poly × poly
  | polyprod ((m,a)::ts) us =
      polysum (map (termprod(m,a)) us)
              (polyprod ts us);
```

BAD MERGING; 16 seconds to square $(x+1)^{400}$

Multiplication of polynomials is also straightforward provided we do not care about efficiency; the schoolbook algorithm suffices. To cross-multiply the terms, function `polyprod` forms products term by term and adds the intermediate polynomials.

We see another application of the functional `map`: the product of the term `(m,a)` with the polynomial `ts` is simply

```
map (termprod(m,a)) ts
```

Polynomial multiplication (2nd try)

Slide 1207

```
fun polyprod [] us       = []
  | polyprod [(m,a)] us = map (termprod(m,a)) us
  | polyprod ts us      =
      let val k = length ts div 2
      in  polysum (polyprod (take(ts,k)) us)
                  (polyprod (drop(ts,k)) us)
      end;
```

4 seconds to square $(x+1)^{400}$

The function `polyprod` is too slow to handle large polynomials. In tests, it required about 16 seconds and numerous garbage collections to compute the square of $(x+1)^{400}$. (Such large computations are typical of symbolic algebra.) The inefficiency is due to the merging (in `polysum`) of lists that differ greatly in length. For instance, if `ts` and `us` consist of 100 terms each, then `(termpolyprod (e,c) us)` has only 100 terms, while `(polyprod ts us)` could have as many as 10,000. Their sum will have at most 10,100 terms; a growth of only 1%. Merging copies both lists; if one list is much shorter than the other, then it effectively degenerates to insertion.

A faster algorithm is inspired by mergesort (Lect. 6). Divide one of the polynomials into equal parts, using `take` and `drop`. Compute two products of roughly equal size and merge those. If one polynomial consists of a single term, multiply it by the other polynomial using `map` as above. This algorithm performs many fewer merges, and each merge roughly doubles the size of the result.

Other algorithms can multiply polynomials faster still.

Polynomial division

Slide 1208

```
fun polyquorem ts ((n,b)::us) =
 let fun quo []          qs = (rev qs, [])
       | quo ((m,a)::ts) qs =
           if m<n then (rev qs, (m,a)::ts)
           else
           quo (polysum ts
                (map (termprod(m-n, ~a/b)) us))
               ((m-n, a/b) :: qs)
 in  quo ts []  end;
```

Let us turn to functions for computing polynomial quotients and remainders. The function `polyquorem` implements the schoolbook algorithm for polynomial division, which is actually simpler than long division. It returns the pair (quotient, remainder), where the remainder is either zero or of lesser degree than the divisor.

The functions `polyquo` and `polyrem` return the desired component of the result, using the ML selectors `#1` and `#2`:

```
fun polyquo ts us = #1(polyquorem ts us)
and polyrem ts us = #2(polyquorem ts us);
```

Aside: if k is any positive integer constant, then `#`k is the ML function to return the kth component of a tuple. Tuples are a special case of ML records, and the `#` notation works for arbitrary record fields.

For example, let us divide $x^2 + 1$ by $x + 1$:

```
polyquorem [(2,1.0),(0,1.0)] [(1,1.0),(0,1.0)];
> val it = ([(1, 1.0), (0, ~1.0)], [(0, 2.0)])
```

This pair tells us that the quotient is $x - 1$ and the remainder is 2. We can easily verify that $(x + 1)(x - 1) + 2 = x^2 - 1 + 2 = x^2 + 1$.

The Greatest Common Divisor

Slide 1209

```
fun polygcd [] us = us
  | polygcd ts us = polygcd (polyrem us ts) ts;
```

needed to simplify *rational functions* such as

$$\frac{x^2-1}{x^2-2x+1} \quad \left(= \frac{x+1}{x-1}\right)$$

strange answers

TOO SLOW

Rational functions are polynomial fractions like $(x+1)/(x-1)$. Efficiency demands that a fraction's numerator and denominator should have no common factor. We should divide the both polynomials by their greatest common divisor (GCD).

We can compute GCDs using Euclid's Algorithm, as shown above. Unfortunately, its behaviour for polynomials is rather perverse. It gives the GCD of x^2+2x+1 and x^2-1 as $-2x-2$, and that of x^2+2x+1 and x^5+1 as $5x+5$; both GCDs should be $x+1$. This particular difficulty can be solved by dividing through by the leading coefficient, but Euclid's Algorithm turns out to be too slow. An innocuous-looking pair of arguments leads to computations on gigantic integers, even when the final GCD is just one! (That is the usual outcome: most pairs of polynomials have no common factor.)

The problem of computing the GCD of polynomials is central to the field of computer algebra. Extremely complex algorithms are employed. A successful implementation makes use of deep mathematics as well as skilled programming. Many projects in advanced technology require this same combination of abilities.

Learning guide. Related material is in *ML for the Working Programmer*, pages 114–121.

Exercise 12.1 Code the set operations of membership test, subset test, union and intersection using the ordered-list representation.

Exercise 12.2 Give a convincing argument that `polysum` and `polyprod` preserve the three restrictions on polynomials.

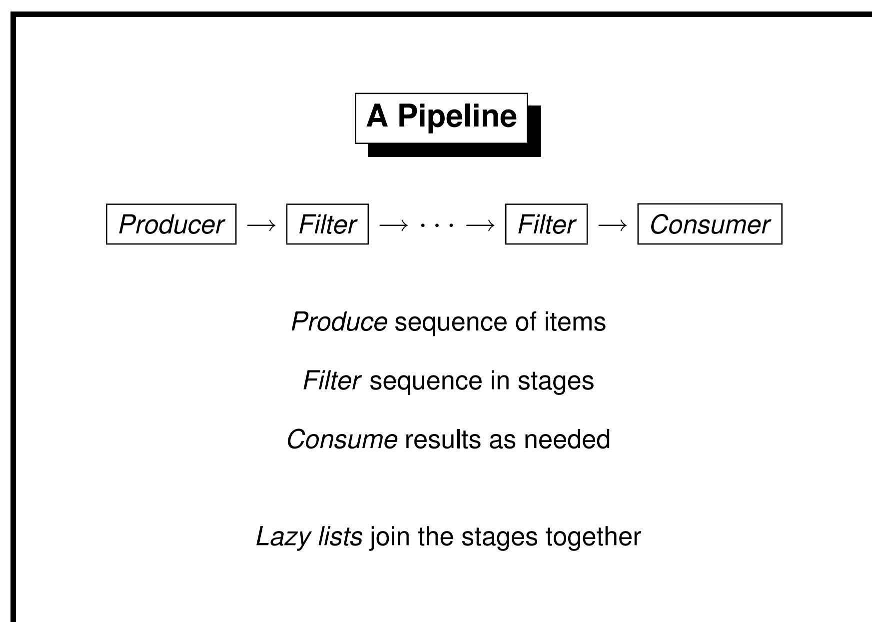
Exercise 12.3 How would you prove that `polysum` correctly computes the sum of two polynomials? Hint: write a mathematical (not ML) function to express the polynomial represented by a list. Which properties of polynomial addition does `polysum` assume?

Exercise 12.4 Show that the complexity of `polysum` is $O(m+n)$ when applied to arguments consisting of m and n terms, respectively.

Exercise 12.5 Give a more rigorous analysis of the asymptotic complexity of the two versions of polynomial multiplication. (This could be difficult.)

Exercise 12.6 If coefficients may themselves be univariate polynomials (in some other variable), then we regain the ability to represent polynomials in any number of variables. For example, $y^2 + xy$ is a univariate polynomial in y whose coefficients are 1 and the polynomial x. Define this representation in ML and discuss ideas for implementing addition and multiplication.

Slide 1301



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 ML 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 ML, we can model pipelines using *lazy lists.*

Slide 1302

Lazy Lists — or *Streams*

Lists of possibly INFINITE length

- elements *computed upon demand*
- *avoids waste* if there are many solutions
- *infinite objects* are a useful abstraction

In ML: implement laziness by *delaying evaluation* of the tail

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 ML, 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.

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

Slide 1303

Lazy Lists in ML

The *empty tuple* `()` and its *type* `unit`

Delayed version of E is `fn()=>`E

sequences

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

fun head (Cons(x,_))  = x;

fun tail (Cons(_,xf)) = xf();
```

`Cons(`x`,`xf`)` has *head* x and *tail function* xf

The primitive ML 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 has several uses:

- It may appear in a data structure. For example, a `unit`-valued dictionary represents a set of keys.
- It may be the argument of a function, where its effect is to *delay evaluation.*
- It may be the argument or result of a *procedure.* (See Lect. 14.)

The empty tuple, like all tuples, is a constructor and is allowed in patterns:

```
fun f () = ...
```

In particular, `fn() =>` 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.

Slide 1304

The Infinite Sequence $k, k+1, k+2, \ldots$

```
fun from k = Cons(k, fn()=> from(k+1));
> val from = fn : int -> int seq

from 1;
> val it = Cons(1, fn) : int seq

tail it;
> val it = Cons(2, fn) : int seq

tail it;
> val it = Cons(3, fn) : int seq
```

Function `from` constructs the infinite sequence of integers starting from k. Execution terminates because of the `fn` enclosing the recursive call. ML displays the tail of a sequence as `fn`, 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.

Slide 1305

Consuming a Sequence

```
fun get(0,xq)           = []
  | get(n,Nil)          = []
  | get(n,Cons(x,xf)) = x :: get(n-1,xf());
> val get = fn : int * 'a seq -> 'a list
```

Get 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 third line of `get`, the expression `xf()` calls the tail function, demanding evaluation of the next element. This operation is called *forcing* the list.

Slide 1306

Sample Evaluation

```
get(2, from 6)
 ⇒ get(2, Cons(6, fn()=>from(6+1)))
 ⇒ 6 :: get(1, from(6+1))
 ⇒ 6 :: get(1, Cons(7, fn()=>from(7+1)))
 ⇒ 6 :: 7 :: get(0, Cons(8, fn()=>from(8+1)))
 ⇒ 6 :: 7 :: []
 ⇒ [6,7]
```

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 `datatype` 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 ML requires references [13, page 327].

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.

Slide 1307

Joining Two Sequences

```
fun appendq (Nil,    yq)      = yq
  | appendq (Cons(x,xf), yq) =
        Cons(x, fn()=> appendq(xf(), yq));
```

A *fair* alternative...

```
fun interleave (Nil,        yq) = yq
  | interleave (Cons(x,xf), yq) =
        Cons(x, fn()=> interleave(yq, xf()));
```

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` (Lect. 4): 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 `fn()=>...`. 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.

Slide 1308

Functionals for Lazy Lists

filtering

```
fun filterq p Nil = Nil
  | filterq p (Cons(x,xf)) =
      if p x
      then Cons(x, fn()=>filterq p (xf()))
      else filterq p (xf());
```

The infinite sequence x, $f(x)$, $f(f(x))$,…

```
fun iterates f x =
      Cons(x, fn()=> iterates f (f x));
```

The functional `filterq` demands elements of `xq` until it finds one satisfying `p`. (Recall `filter`, Lect. 11.) It contains a *force* not protected by a *delay*. If `xq` is infinite and contains no satisfactory element, then `filterq` runs forever.

The functional `iterates` generalizes `from`. It creates the next element not by adding one but by calling the function `f`.

Numerical Computations on Infinite Sequences

```
fun next a x = (a/x + x) / 2.0;
```

Close enough?

Slide 1309

```
fun within (eps:real) (Cons(x,xf)) =
    let val Cons(y,yf) = xf()
    in  if abs(x-y) <= eps then y
        else within eps (Cons(y,yf))
    end;
```

Square Roots!

```
fun root a = within 1E~6 (iterates (next a) 1.0)
```

Calling `iterates (next a) x0)` generates the *infinite series* of approximations to the square root of a for the Newton-Raphson method. As discussed in Lect. 2, the infinite series x_0, $(a + x_0)/2$, ... converges to $\sqrt{a}$.

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 apply the Newton-Raphson method with a tolerance of 10^{-6} and an (awful) initial approximation of 1.0.

This treatment of numerical computation has received some attention in the research literature; a recurring example is *Richardson extrapolation* [7, 8].

Learning guide. Related material is in *ML for the Working Programmer*, pages 191–212.

Exercise 13.1 Code an analogue of `map` for sequences.

Exercise 13.2 Consider the list function `concat`, which concatenates a list of lists to form a single list. Can it be generalized to concatenate a sequence of sequences? What can go wrong?

```
fun concat []      = []
  | concat (l::ls) = l @ concat ls;
```

Exercise 13.3 Code a function to make change using lazy lists. (This is difficult.)

Slide 1401

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*: blocks of memory cells
- *pointer structures* and *linked lists* (next lecture)

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 `case` 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 of computing; they represent one of the first examples of abstraction in programming languages.

ML makes no distinction between commands and expressions. ML provides built-in 'functions' to perform assignment and communication, and these can be used in the traditional (procedural) style. ML programmers normally follow a functional style for most internal computations and use imperative features only for communication with the outside world.

Slide 1402

ML Primitives for References

τ `ref`	*type* of references to type τ
`ref` E	*create* a reference *initial contents* = the value of E
`!`P	return the *current contents* of reference P
P `:=` E	*update* the contents of P to the value of E

The slide presents the ML 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 *pointers* or *locations*). Calling `ref` allocates a new location in the machine store. Initially, this location holds the value given by expression E. Although `ref` is an ML function, it is not a function in the mathematical sense. For example, `ref(0)=ref(0)` evaluates to false.

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 ML 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:

```
ref     'a -> 'a ref
!       'a ref -> 'a
op :=   'a ref * 'a -> unit
```

Slide 1403

Trying Out References

```
val p = ref 5;                              create a reference
> val p = ref 5 : int ref

p := !p + 1;                                now p holds 6

val ps = [ref 77, p];
> val ps = [ref 77, ref 6] : int ref list

hd ps := 3;                                 updating an integer ref

ps;                                         contents of the refs?
> val it = [ref 3, ref 6] : int ref list
```

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 `val` 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.

ML displays a reference value as `ref` 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 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.

Slide 1404

Commands: Expressions with Effects

- C_1;...;C_n causes a series of expressions to be evaluated and returns the value of C_n.
- () is the value returned by a typical command.
- <u>`if`</u> B <u>`do`</u> C_1 <u>`else`</u> C_2 behaves like the traditional control structure if C_1 and C_2 have effects.
- Other ML constructs behave naturally with commands, including 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`;` ... `;`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 `case` much as in conventional languages. ML functions play the role of procedures.

Other languages that combine the functional and imperative programming paradigms include Lisp (and its dialect Scheme), Objective Caml, and even a systems programming language, BLISS (now long extinct).

Slide 1405

Iteration: the `while` Command

while B do C

```
fun length xs =
  let val lp = ref xs          list of uncounted elements
      and np = ref 0                    accumulated count
  in
      while not (null (!lp)) do
          (lp := tl (!lp);  np := 1 + !np);
      !np                          the count is returned!
  end;
```

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.

ML's only 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`). 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 above consists of two assignment commands, executed one after the other. The `while` command is followed by the expression `!np` to return computed length as the function's result. This semicolon need not be enclosed in parentheses because it is bracketed by `in` and `end`.

Private, Persistent References

Slide 1406

```
fun makeAccount (initBalance: int) =
  let val balance = ref initBalance
      fun withdraw amt =
            if amt > !balance
            then !balance
            else (balance := !balance - amt;
                  !balance)
  in withdraw
  end;
> val makeAccount = fn : int -> (int -> int)
```

As you may have noticed, ML's programming style looks clumsy compared with that of languages like C. ML omits the defaults and abbreviations they provide to shorten programs. However, ML's explicitness makes it ideal for teaching the fine points of references and arrays. ML'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 (it could raise an exception).

Look at the (E_1; 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 unauthorized updates.

This example is based on one by Dr A C Norman.

Slide 1407

Two Bank Accounts

```
val student = makeAccount 500;
> val student = fn : int -> int

val director = makeAccount 400000;
> val director = fn : int -> int

student 5;       (*coach fare*)
> val it = 495 : int

director 50000;  (*Jaguar*)
> val it = 350000 : int
```

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 generalize *makeAccount* to return several functions that jointly manage information held in shared references. The functions might be packaged using ML records, which are discussed elsewhere [13, pages 32–36]. Most procedural languages do not properly support the concept of private references, although *object-oriented* languages take them as a basic theme.

Slide 1408

Variables: ML Versus Conventional Languages

- We must write `!p` to get at the *contents* of `p`
- We can write just `p` for the *address* of `p`
- We can store a private reference cells (like `balance`) in functions—simulating **object-oriented programming**
- ML's assignment syntax is V `:=` E instead of $V = E$
- ML has few control structures: only `while` and `case`

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 uninitialized variables can cause errors that are almost impossible to diagnose.

Dereferencing operators (like ML's `!`) are especially unpopular, because they clutter the program text. Many programming languages make dereferencing implicit (that is, automatic).

Slide 1409

ML Primitives for Arrays

τ `Array.array`	*type* of arrays of type τ
`Array.tabulate(`n`,`f`)`	*create* a n-element array $A[i]$ initially holds $f(i)$
`Array.sub(`A`,`i`)`	return the *contents* of $A[i]$
`Array.update(`A`,`i`,`E`)`	*update* $A[i]$ to the value of E

And countless others!

ML arrays are like references that hold n 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.array` is the type of arrays (of any size) with elements from τ.

Calling `Array.tabulate(`n`,`f`)` creates an array of the size specified by expression n, allocating the necessary storage. Initially, element $A[i]$ holds the value of $f(i)$ for $i = 0, \ldots, n-1$.

Calling `Array.sub(`A`,`i`)` returns the contents of $A[i]$.

Calling `Array.update(`A`,`i`,`E`)` modifies the array by storing the value of E as the new contents of $A[i]$; it returns () as its value.

Why is there no function returning a *reference* to $A[i]$? Such a function could replace both `Array.sub` and `Array.update`, but allowing references to individual array elements would complicate storage management.

An array's size is specified in advance to facilitate storage management. Typically another variable records how many elements are actually in use. The unused elements constitute wasted storage; if they are never initialized to legal values (they seldom are), then they can cause no end of trouble. When the array bound is reached, either the program must abort or the array must expand, typically by copying into a new one that is twice as big.

Slide 1410

Array Example: Block Move

```
fun insert (A,kp,x) =
  let val ip = ref (!kp)
  in
      while !ip>0 do
        (Array.update(A, !ip,          (* A[i] := A[i-1] *)
                       Array.sub(A, !ip-1));
         ip := !ip-1);
      Array.update(A, 0, x);
      kp := !kp+1
  end;
```

The main lesson to draw from this example is that arrays are harder to use than lists. Insertion sort and quick sort fit on a slide when expressed using lists (Lect. 6). The code above, roughly the equivalent of `x::xs`, was originally part of an insertion function for array-based insertion sort. ML's array syntax does not help. In a conventional language, the key assignment might be written

```
A[ip] := A[ip-1]
```

To be fair, ML's datatypes and lists require a sophisticated storage management system and their overheads are heavy. Often, for every byte devoted to actual data, another byte must be devoted to link fields, as discussed in Lect. 15.

Function `insert` takes an array `A` whose elements indexed by zero to`!kp-1` are in use. The function moves each element to the next higher subscript position, stores `x` in position zero and increases the bound in `kp`.

We have an example of what in other languages are called reference parameters. Argument `A` has type `'a Array.array`, while `kp` has type `int ref`. The function acts through these parameters only.

In the C language, there are no arrays as normally understood, merely a convenient syntax for making address calculations. As a result, C is one of the most error-prone languages in existence. The vulnerability of C software was dramatically demonstrated in November 1988, when the Internet Worm brought the network down.

Slide 1411

Arrays: ML Versus Conventional Languages

advantages

- Array subscripts are checked! (Unlike C)
- Read-only arrays are available.

DISADVANTAGES

- The syntax is clumsy, especially for updating
- The syntax for arrays of arrays is even worse!

USE ARRAYS WITH CARE, BECAUSE THEY COMPLICATE PROGRAMS.

ML provides immutable arrays, called *vectors*, which lack an update operation. The operation `Vector.tabulate` can be used to trade storage for runtime. Creating a table of function values is worthwhile if the function is computationally expensive.

Here is a table of the main array functions, with their types.

```
Array.array     int * 'a -> 'a Array.array
Array.tabulate  int * (int -> 'a) -> 'a Array.array
Array.sub       'a Array.array * int -> 'a
Array.update    'a Array.array * int * 'a -> unit
```

References give us new ways of expressing programs, and arrays give us efficient access to the hardware addressing mechanism. But neither fundamentally increases the set of algorithms that we can express— unlike communication primitives—and they can make programs harder to understand. No longer can we describe program execution in terms of reduction, as we did in Lect. 2. They can also make storage management more expensive. They should therefore be used with care.

ML's arrays are much safer than C's, however. In C, an array is nothing more than an address indicating the start of a storage area. Nothing indicates the size of the area or where it ends. This flaw in C is largely to blame for the ubiquitous security loopholes caused by *buffer overruns*. In a buffer overrun attack, the hacker sends more data than the receiving program expects, overrunning the area of storage set aside to hold it. He eventually overwrites the program itself, replacing it with the hacker's chosen code.

Learning guide. Related material is in *ML for the Working Programmer*, pages 313–326. A brief discussion of ML's comprehensive input/output facilities, which are not covered in this course, is on pages 340–356.

Exercise 14.1 Comment, with examples, on the differences between an `int ref list` and an `int list ref`.

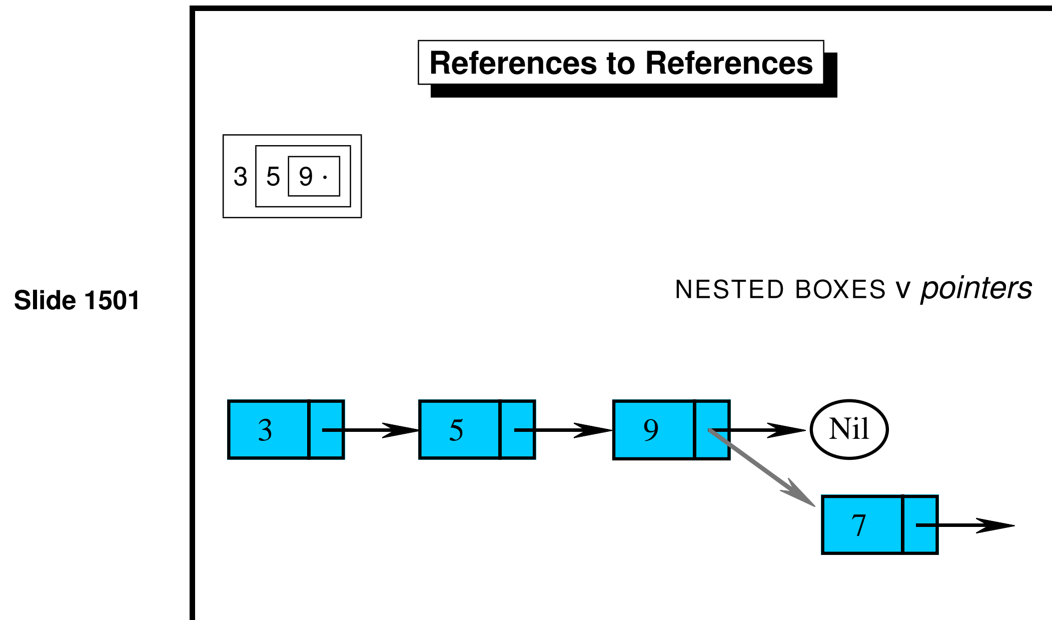
Exercise 14.2 Write a version of function `power` (Lect. 2) using `while` instead of recursion.

Exercise 14.3 What is the effect of `while` (C_1; B) `do` C_2?

Exercise 14.4 Arrays of multiple dimensions are represented in ML 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.

Exercise 14.5 Function `insert` copies elements from $A[i-1]$ to $A[i]$, for $i = k, \ldots, 1$. What happens if instead it copies elements from $A[i]$ to $A[i+1]$, for $i = 0, \ldots, k-1$?

Slide 1501



References can be imagined to be boxes whose contents can be changed. But the box metaphor becomes unworkable when the contents of the box can itself be a box: deep nesting is too difficult to handle. A more flexible metaphor is the *pointer*. A reference points to some object; this pointer can be moved to any other object of the right type. The slide depicts a representation of the list [3,5,9], where the final pointer to `Nil` is about to be redirected to a cell containing the element 7. ML forbids such redirection for its built-in lists, but we can declare linked lists whose link fields are mutable.

Slide 1502

Linked, or Mutable, Lists

```
datatype 'a mlist = Nil
                  | Cons of 'a * 'a mlist ref;
```

Tail can be REDIRECTED

Creating a linked list:

```
fun mlistOf []     = Nil
  | mlistOf (x::l) = Cons (x, ref (mlistOf l));
> val mlistOf = fn : 'a list -> 'a mlist
```

A mutable list is either empty (`Nil`) or consists of an element paired with a pointer to another mutable list. Removing the **`ref`** from the declaration above would make the datatype exactly equivalent to built-in ML lists. The reference in the tail allows links to be changed after their creation.

To get references to the elements themselves, we can use types of the form **`'a ref mlist`**. (We have seen type **`int ref list`** in Lect. 14.) So there is no need for another **`ref`** in the datatype declaration.

Function `mlistOf` converts ordinary lists to mutable lists. Its call to **`ref`** creates a new reference cell for each element of the new list.

Most programming languages provide reference types designed for building linked data structures. Sometimes the null reference, which points to nothing, is a predefined constant called `NIL`. The run-time system allocates space for reference cells in a dedicated part of storage, called the *heap*, while other (mutable) variables are allocated on the stack. In contrast, ML treats all references uniformly.

ML lists are represented internally by a linked data structure that is equivalent to `mlist`. The representation allows the links in an ML list to be changed. That such changes are forbidden is a design decision of ML to encourage functional programming. The list-processing language Lisp allows links to be changed.

Slide 1503

Extending a List to the Rear

```
fun extend (mlp, x) =
  let val last = ref Nil
  in  mlp := Cons (x, last);
      last                          new final reference
  end;

> val extend = fn
>    : 'a mlist ref * 'a -> 'a mlist ref
```

Extending ordinary ML lists to the rear is hugely expensive: we must evaluate an expression of the form `xs@[x]`, which is $O(n)$ in the size of `xs`. With mutable lists, we can keep a pointer to the final reference. To extend the list, update this pointer to a new list cell. Note the new final reference for use the next time the list is extended.

Function `extend` takes the reference `mlp` and an element `x`. It assigns to `mlp` and returns the new reference as its value. Its effect is to update `mlp` to a list cell containing `x`.

Slide 1504

Example of Extending a List

```
val mlp = ref (Nil: string mlist);
> val mlp = ref Nil : string mlist ref

extend (mlp, "a");
> val it = ref Nil : string mlist ref

extend (it, "b");
> val it = ref Nil : string mlist ref

mlp;
> ref(Cons("a", ref(Cons("b", ref Nil))))
```

We start things off by creating a new pointer to `Nil`, binding it to `mlp`. Two calls to `extend` add the elements `"a"` and `"b"`. Note that the first `extend` call is given `mlp`, while the second call is given the result of the first, namely `it`.

Finally, we examine `mlp`. It no longer points to `Nil` but to the mutable list `["a","b"]`.

Slide 1505

Destructive Concatenation

```
fun joining (mlp, ml2) =
        case !mlp of
            Nil          => mlp := ml2
          | Cons(_,mlp1) => joining (mlp1, ml2);

fun join (ml1, ml2) =
    let val mlp = ref ml1          temporary reference
    in  joining (mlp, ml2);
        !mlp
    end;
```

Function `join` performs destructive concatenation. It updates the final pointer of one mutable list to point to some other list rather than to `Nil`. Contrast with ordinary list append, which *copies* its first argument. Append takes $O(n)$ time and space in the size of the first list, while destructive concatenation needs only constant space.

Function `joining` does the real work. Its first argument is a pointer that should be followed until, when `Nil` is reached, it can be made to point to list `ml2`. The function looks at the contents of reference `mlp`. If it is `Nil`, then the time has come to update `mlp` to point to `ml2`. But if it is a `Cons` then the search continues using reference in the tail.

Function `join` starts the search off with a temporary reference to its first argument. This trick saves us from having to test whether or not `ml1` is `Nil`; the test in `joining` either updates the reference or skips down to the 'proper' reference in the tail. Tricks of this sort are quite useful when programming with linked structures.

The functions' types tell us that `joining` takes two mutable lists and (at most) performs some action, since it can only return (), while `join` takes two lists and returns another one.

```
joining : 'a mlist ref * 'a mlist -> unit
join    : 'a mlist * 'a mlist -> 'a mlist
```

Slide 1506

Side-Effects

```
val ml1 = mlistOf ["a"];
> val ml1 = Cons("a", ref Nil) : string mlist

val ml2 = mlistOf ["b","c"];
> val ml2 = Cons("b", ref(Cons("c", ref Nil)))

join(ml1,ml2);

ml1;                                    IT'S CHANGED!?
> Cons("a",
>       ref(Cons("b", ref(Cons("c", ref Nil)))))
```

In this example, we bind the mutable lists `["a"]` and `["b","c"]` to the variables `ml1` and `ml2`. ML's method of displaying reference values lets us easily read off the list elements in the data structures.

Next, we concatenate the lists using `join`. (There is no room to display the returned value, but it is identical to the one at the bottom of the slide, which is the mutable list `["a","b","c"]`.)

Finally, we inspect the value of `ml1`. It looks different; has it changed? No; it is the same reference as ever. The contents of a cell reachable from it has changed. Our *interpretation* of its value of a list has changed from `["a"]` to `["a","b","c"]`.

This behaviour cannot occur with ML's built-in lists because their internal link fields are not mutable. The ability to update the list held in `ml1` might be wanted, but it might also come as an unpleasant surprise, especially if we confuse `join` with append. A further surprise is that

```
join(ml2,ml3)
```

also affects the list in `ml1`: it updates the last pointer of `ml2` and that is now the last pointer of `ml1` too.

Slide 1507

A Cyclic List

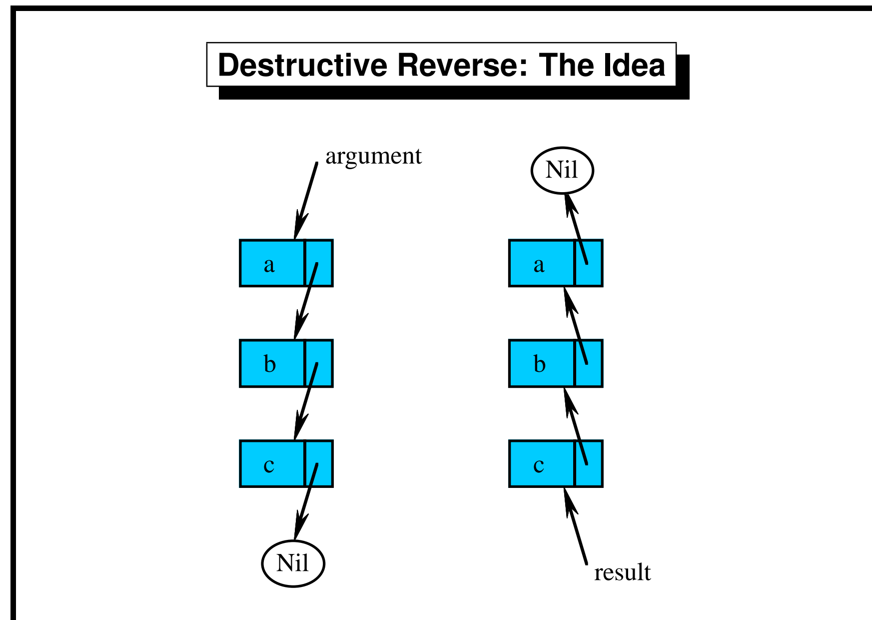
```
val ml = mlistOf [0,1];
> val ml = Cons(0, ref(Cons(1, ref Nil)))

join(ml,ml);
> Cons(0,
>       ref(Cons(1,
>                ref(Cons(0,
>                         ref(Cons(1,...)))))))
```

What has happened? Calling `join(ml,ml)` causes the list `ml` to be chased down to its final link, which is made to point to ... `ml`!

If an object contains, perhaps via several links, a pointer leading back to itself, we have a *cycle*. A cyclic chain of pointers can be disastrous if it is created unexpectedly. Cyclic data structures are difficult to navigate without looping and are especially difficult to copy. Naturally, they don't suit the box metaphor for references! Cyclic data structures do have their uses. A circular list can be used to rotate among a finite number of choices fairly. A dependency graph describes how various items depend upon other items; such dependencies can be cyclic.

Slide 1508



List reversal can be tricky to work out from first principles, but the code should be easy to understand.

Reverse for ordinary lists copies the list cells while reversing the order of the elements. *Destructive* reverse re-uses the existing list cells while re-orienting the links. It works by walking down the mutable list, noting the last two mutable lists encountered, and redirecting the second cell's link field to point to the first. Initially, the first mutable list is `Nil`, since the last link of the reversed must point to `Nil`.

Note that we must look at the reversed list from the opposite end! The reversal function takes as its argument a pointer to the first element of the list. It must return a pointer to the first element of the reversed list, which is the last element of the original list.

Slide 1509

A Destructive Reverse Function

```
fun reversing (prev, ml) =
    case ml of
        Nil          => prev          start of reversed list
      | Cons(_,mlp2) =>
          let val ml2 = !mlp2                    next cell
          in  mlp2 := prev;                      re-orient
              reversing (ml, ml2)
          end;
> reversing: 'a mlist * 'a mlist -> 'a mlist

fun drev ml = reversing (Nil, ml);
```

The function `reversing` redirects pointers as described above. The function needs only constant space because it is tail recursive and does not call `ref` (which would allocate storage). The pointer redirections can be done in constant space because each one is local, independent of other pointers. It does not matter how long the list is.

Space efficiency is a major advantage of destructive list operations. It must be set against the greater risk of programmer error. Code such as the above may look simple, but pointer redirections are considerably harder to write than functional list operations. The reduction model does not apply. We cannot derive function definitions from equations but must think explicitly in terms of the effects of updating pointers.

Slide 1510

Example of Destructive Reverse

```
val ml = mlistOf [3, 5, 9];
> val ml =
> Cons(3, ref(Cons(5, ref(Cons(9, ref Nil)))))

drev ml;
> Cons(9, ref(Cons(5, ref(Cons(3, ref Nil)))))

ml;                                        IT'S CHANGED!?
> val it = Cons(3, ref Nil) : int mlist
```

In the example above, the mutable list [3,5,9] is reversed to yield [9,5,3]. The effect of `drev` upon its argument `ml` may come as a surprise! Because `ml` is now the last cell in the list, it appears as the one-element list [3].

The ideas presented in this lecture can be generalized in the obvious way to trees. Another generalization is to provide additional link fields. In a *doubly-linked* list, each node points to its predecessor as well as to its successor. In such a list one can move forwards or backwards from a given spot. Inserting or deleting elements requires redirecting the pointer fields in two adjacent nodes. If the doubly-linked list is also cyclic then it is sometimes called a *ring buffer* [13, page 331].

Tree nodes normally carry links to their children. Occasionally, they instead have a link to their parent, or sometimes links in both directions.

Learning guide. Related material is in *ML for the Working Programmer*, pages 326–339.

Exercise 15.1 Write a function to copy a mutable list. When might you use it?

Exercise 15.2 What is the value of `ml1` (regarded as a list) after the following declarations and commands are entered at top level? Explain this outcome.

```
val ml1 = mlistOf[1,2,3]
and ml2 = mlistOf[4,5,6,7];
join(ml1, ml2);
drev ml2;
```

Exercise 15.3 Code destructive reverse using `while` instead of recursion.

Exercise 15.4 Write a function to copy a cyclic list, yielding another cyclic list holding the same elements.

References

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, 1971.

[4] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Reprinted as Chapter 4 of Bell and Newell [3], first published in 1946.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[6] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.

[7] Matthew Halfant and Gerald Jay Sussman. Abstraction in numerical methods. In *LISP and Functional Programming*, pages 1–7. ACM Press, 1988.

[8] John Hughes. Why functional programming matters. *Computer Journal*, 32:98–107, 1989.

[9] Donald E. Knuth. *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison-Wesley, 1973.

[10] Donald E. Knuth. *The Art of Computer Programming*, volume 1: *Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1973.

[11] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[12] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988. Follow-up discussion in Comm. ACM 36 (7), July 1993, pp. 105-110.

[13] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[14] Robert Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[15] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1993.

[16] Å. Wikström. *Functional Programming using ML*. Prentice-Hall, 1987.