

# ***Foundations of Computer Science***

## Computer Science Tripos Part IA

*Lawrence C Paulson*  
Computer Laboratory  
University of Cambridge

lp15@cam.ac.uk

# Contents

<b>1</b>	<b>Introduction to Programming</b>	<b>1</b>
<b>2</b>	<b>Recursion and Efficiency</b>	<b>15</b>
<b>3</b>	<b>Lists</b>	<b>27</b>
<b>4</b>	<b>More on Lists</b>	<b>38</b>
<b>5</b>	<b>Sorting</b>	<b>49</b>
<b>6</b>	<b>Datatypes and Trees</b>	<b>60</b>
<b>7</b>	<b>Dictionaries and Functional Arrays</b>	<b>72</b>
<b>8</b>	<b>Functions as Values</b>	<b>83</b>
<b>9</b>	<b>Sequences, or Lazy Lists</b>	<b>96</b>
<b>10</b>	<b>Queues and Search Strategies</b>	<b>106</b>
<b>11</b>	<b>Polynomial Arithmetic</b>	<b>118</b>
<b>12</b>	<b>Elements of Procedural Programming</b>	<b>128</b>

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

The programming in this course is based on the language 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 cover many fundamental data structures and learn 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* (2nd ed), which also contains many exercises. You can find relevant exam questions in the Part IA papers from 1998 onwards. From 2013, mutable lists are no longer part of the syllabus.

The course materials webpage<sup>1</sup> contains information on various ML systems that you can install on your computer. There are copies of lecture notes, sample programs, programming exercises and other useful documents. You will also find a cryptic but complete description of Standard ML's syntax.

Thanks to David Allsopp, Stuart Becker, Gavin Bierman, Chloë Brown, Silas Brown, Qi Chen, David Cottingham, William Denman, Robert Harle, Daniel Hulme, Frank King, Jack Lawrence-Jones, Joseph Lord, Dimitrios Los, Farhan Mannan, James Margetson, David Morgan, Frank Stajano, Thomas Tuerk, Xincheng Wang, Philip Withnall and Assel Zhiyenbayeva for pointing out errors in these notes. Corrections and suggestions are welcome and will be acknowledged 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.

- Paulson, Lawrence C. (1996). *ML for the Working Programmer*. Cambridge University Press (2nd ed.).
- Mads Tofte (2009). *Tips for Computer Scientists on Standard ML*. This “cheat sheet” overview of ML is available at <http://www.itu.dk/people/tofte/publ/tips.pdf>.
- Hansen, Michael and Rischel, Hans (1999) *Introduction to Programming Using SML*. Addison-Wesley.
- Ullman, J. D. (1998) *Elements of ML97 Programming*. Prentice Hall.
- M. Felleisen and D. P. Friedman (1998). *The Little MLer*. MIT Press.

---

<sup>1</sup><http://www.cl.cam.ac.uk/teaching/current/FoundsCS/>

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: dates over a certain interval

*Concrete* level: typically 6 characters: YYMMDD  
(where each character is represented by 8 bits)

Date crises caused by INADEQUATE internal formats:

- *Digital's PDP-10*: using 12-bit dates (good for at most 11 years)
- *2000 crisis*: 48 bits could be good for lifetime of universe!

Digital Equipment Corporation's date crisis occurred in 1975. The PDP-10 was a 36-bit mainframe computer. It represented dates using a 12-bit format designed for the tiny PDP-8. With 12 bits, one can distinguish  $2^{12} = 4096$  days or 11 years.

The most common industry format for dates uses six characters: two for the year, two for the month and two for the day. The most common "solution" to the year 2000 crisis is to add two further characters, thereby altering file sizes. Others have noticed that the existing six characters consist of 48 bits, already sufficient to represent all dates over the projected lifetime of the universe:

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

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  $1.066 \times 10^3$ .

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

The concept of DATA TYPE involves

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

Computer arithmetic can yield *incorrect answers*!!

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

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

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

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

Slide 104

### Goals of Programming

- to describe a computation so that it can be done *mechanically*:
  - Expressions* compute *values*.
  - Commands* cause *effects*.
- to do so **efficiently** and CORRECTLY, giving the right answers quickly
- to allow easy *modification* as needs change
  - Through an orderly *structure* based on abstraction principles
  - Such as modules or (Java) classes

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

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

*Object-oriented programming* is the most complicated approach to modularity. *Classes* define concepts, and they can be built upon other classes. Operations can be defined that work in appropriately 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 when you study Java.

Slide 105

### 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.

Standard ML is the outcome of years of research into programming languages. It is unique, defined using a mathematical formalism (an *operational semantics*) that is both precise and comprehensible. Several 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, consider installing Standard ML on it. Our preferred implementation is Poly/ML,<sup>1</sup> because it is efficient and well-supported. Alternatives such as SML of New Jersey and Moscow ML are worth considering. On the other hand, other ML dialects such as OCaml and F# expect a different syntax and will not run any of our examples as presented.

---

<sup>1</sup><http://www.polym1.org/>



Slide 106

**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. This sort of encapsulated computation is called a *function*. 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.

### Raising a Number to a Power

Slide 107

```
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. You may also have seen recursion in the product rule for differentiation:  $(u \cdot v)' = u \cdot v' + u' \cdot v$ . In finding the derivative of  $u \cdot v$ , we recursively find the derivatives of  $u$  and  $v$ , combining them to obtain the desired result. The recursion is meaningful because it terminates: we reduce the problem to two smaller problems, and this cannot go on forever. The 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:

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

*Remark:* function `npower` contains both an integer constant (`0`) and a real constant (`1.0`). The decimal point makes all the difference. The ML system will notice.

Slide 108

### An Aside: Overloading

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
```

Now for a tiresome but necessary aside. In most languages, the types of arguments and results must always be specified. ML is unusual that it normally infers the types 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. All programming languages have trouble points such as these.

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. (We discuss equality further in a later lecture.)

Note that `x <> y` is ML for `x ≠ y`.

Slide 109

### Conditional Expressions and Type `bool`

`if b then x else y`

`not(b)`    **negation of  $b$**

`p andalso q`     $\equiv$     `if p then q else false`

`p orelse q`     $\equiv$     `if p then true 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

$(\text{if true then } x \text{ else } y) = x$

$(\text{if false then } x \text{ else } y) = y$

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: here, 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.)

### Efficiently Raising a Number to a Power

```

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);

```

Slide 110

*Mathematical Justification:*

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

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

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

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

We use the function `even`, declared previously, to test whether the exponent is even. Integer division (`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 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 or else raise rec
sharing sig signature struct structure
then type val where while with withtype
```

The negation of  $x$  is written  $\sim x$  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.

Slide 112

**Syntax You Must Know****Declarations**val  $Id = E$ fun  $Id\ Args = E$ **Expressions**if  $E$  then  $E$  else  $E$  $E$  andalso  $E$  $E$  orelse  $E$  $E : T$ 

This page summarises the syntax presented so far. Here,  $Id$  denotes an identifier,  $E$  denotes an expression and  $T$  denotes a type.

As we have seen, simple values and functions can be declared. The argument of a function can be a single identifier or a group of arguments. This will be generalised again later.

The simplest expressions are identifiers (declared as above) and constants. They can be combined using infix operators, as outlined on the previous slide. We have seen the if-then-else expression form, which makes a choice depending on a truth value. We have seen the built-in operators andalso and orelse, which combine truth values.

Finally, we have seen that a type constraint such as `: real` can be inserted in a program. An expression that has been constrained in this way is still an expression.

A comprehensive specification of ML syntax is available on the course website. See Course materials, and near the bottom of this page, Full syntax of Standard ML.

**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.)

**Exercise 1.3** Why would no competent programmer write an expression of the form if . . . then true else false? What about expressions of the form if . . . then false else true?

**Exercise 1.4** Functions `npower` and `power` both have type constraints, but only one of them actually needs it. Try to work out which function does not need its type constraint merely by looking at its declaration.

**Exercise 1.5** Because computer arithmetic is based on binary numbers, simple decimals such as 0.1 often cannot be represented exactly. Write a function that performs the computation

$$\underbrace{x + x + \cdots + x}_n$$

where  $x$  has type `real`. (It is essential to use repeated addition rather than multiplication!) Different ML systems deliver different results. Arithmetic under Poly/ML is much more accurate than under SML/NJ. The value computed with  $n = 1000000$  and  $x = 0.1$  may print as `100000.0`, which looks exact. If that happens, then evaluate the expression `it - 100000.0`.

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

**Exercise 1.6** Another example of the inaccuracy of floating-point arithmetic takes the golden ratio  $\phi \approx 1.618 \dots$  as its starting point:

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

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



Slide 201

**Expression Evaluation**

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

Sample evaluation for power:

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

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

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

We write  $E \Rightarrow E'$  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.

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

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

**Example: Summing the First  $n$  Integers**

Slide 202

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

$$\begin{aligned} \text{nsum } 3 &\Rightarrow 3 + \text{nsum } 2 \\ &\Rightarrow 3 + (2 + \text{nsum } 1) \\ &\Rightarrow 3 + (2 + (1 + \text{nsum } 0)) \\ &\Rightarrow 3 + (2 + (1 + 0)) \Rightarrow \dots \Rightarrow 6 \end{aligned}$$

The function call  $\text{nsum } n$  computes the sum  $1 + \dots + n$  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  $\text{nsum } 0$  is reached. Meanwhile, the computer must store the numbers in an internal data structure, typically the *stack*. For large  $n$ , say  $\text{nsum } 10000$ , 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 203

**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
```

$$\begin{aligned}\text{summing}(3, 0) &\Rightarrow \text{summing}(2, 3) \\ &\Rightarrow \text{summing}(1, 5) \\ &\Rightarrow \text{summing}(0, 6) \Rightarrow 6\end{aligned}$$

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 204

**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.

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

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

### Stupidly Summing the First $n$ Integers

Slide 205

```
fun stupSum n =  
  if n=0 then 0  
  else n + (stupSum(n-1) + stupSum(n-1)) div 2;
```

The function calls itself  $2^n$  times!

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

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

The function `stupSum` calls itself twice in each recursive step. This function is contrived, but many mathematical formulas refer to a particular quantity more than once. In ML, we can create a local binding to a computed value using the *local declaration* syntax `let D in E end`, where  $D$  is a series of declarations. In the following expression,  $y$  is computed once and used twice:

```
let val y = power(x,20)  
in f(y) + g(x,y) end
```

Fast hardware does not make good algorithms unnecessary. On the contrary, faster hardware magnifies the superiority of better algorithms. Typically, we want to handle the largest inputs possible. If we double our processing power, what do we gain? How much can we increase  $n$ , the input to our function? With `stupSum`, we can only go from  $n$  to  $n + 1$ . We are limited to this modest increase because the function's running time is proportional to  $2^n$ . With the function `npower`, defined in the previous lecture, 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  $n^2$ .

Slide 206

### Some Illustrative Figures

<i>complexity</i>	<i>1 second</i>	<i>1 minute</i>	<i>1 hour</i>	<b>gain</b>
$n$	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 of runtime given an input of size  $n$*

This table (excerpted from a 40-year-old book! [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 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.

### Comparing Algorithms: $O$ Notation

Slide 207

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

$|f(n)|$  is bounded for some *constant*  $c$  and all *sufficiently large*  $n$ .

*Intuitively*, Look at the *most significant* term.

Ignore *constant factors*: they seldom dominate and are often transitory

**Example:** consider  $n^2$  instead of  $3n^2 + 34n + 433$ .

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

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

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

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  $f(n) = f'(n)$ . Rather,  $f(n) = O(g(n))$  states that  $g$  is an *upper bound* for  $f$ . A number of related concepts can be defined, such as ‘little o’ notation:  $f(n) = o(g(n))$  states that  $g$  is a *lower bound* for  $f$ .

Slide 208

### 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 that  $n^2 + 50n + 36/n^2$  converges to 1 for increasing  $n$ .

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

$O(n^c)$  is contained in  $O(n^d)$

$O(c^n)$  is contained in  $O(d^n)$

$O(\log n)$  is contained 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 209

**Common Complexity Classes**

$O(1)$	<i>constant</i>
$O(\log n)$	<i>logarithmic</i>
$O(n)$	<i>linear</i>
$O(n \log n)$	<i>quasi-linear</i>
$O(n^2)$	<i>quadratic</i>
$O(n^3)$	<i>cubic</i>
$O(a^n)$	<i>exponential (for fixed <math>a</math>)</i>

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 210

### Sample Costs in $O$ Notation

<i>function</i>	<i>time</i>	<i>space</i>
npower, nsum	$O(n)$	$O(n)$
summing	$O(n)$	$O(1)$
$n(n+1)/2$	$O(1)$	$O(1)$
power	$O(\log n)$	$O(\log n)$
stupSum	$O(2^n)$	$O(n)$

Recall (Lect. 2) that npower computes  $x^n$  by repeated multiplication while nsum naïvely computes the sum  $1 + \dots + n$ . 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 211

### Some Simple Recurrence Relations

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

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

Some *recurrences*:

$$T(n + 1) = T(n) + 1 \qquad O(n)$$

$$T(n + 1) = T(n) + n \qquad O(n^2)$$

$$T(n) = T(n/2) + 1 \qquad O(\log n)$$

$$T(n) = 2T(n/2) + n \qquad O(n \log n)$$

To analyse 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  $T(n) = (n - 1) + \dots + 1 = n(n - 1)/2 = O(n^2)$ . 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  $T(n) = O(\log n)$ .

**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** Add a column to the table shown in Slide 206 with the heading *60 hours*.

**Exercise 2.3** Let  $g_1, \dots, g_k$  be functions such that  $g_i(n) \geq 0$  for  $i = 1, \dots, k$  and all sufficiently large  $n$ . Show that if  $f(n) = O(a_1 g_1(n) + \dots + a_k g_k(n))$  then  $f(n) = O(g_1(n) + \dots + g_k(n))$ .

**Exercise 2.4** Find an upper bound for the recurrence given by  $T(1) = 1$  and  $T(n) = 2T(n/2) + 1$ . You should be able to find a tighter bound than  $O(n \log n)$ .

Slide 301

**Lists**

```
[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  $x_1, \dots, x_n$  all have the same type (say  $\tau$ ) then the list  $[x_1, \dots, x_n]$  has type  $(\tau)\text{list}$ .

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  $i$ th 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 302

### The List Primitives

*The two kinds of list*

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

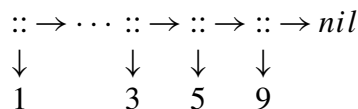
`x :: l` is the list with head `x` and tail `l`

*List notation*

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

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]:



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!

Slide 303

### Getting at the Head and Tail

```

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

```

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

The declaration of `null` 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 similar to `hd`.

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

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

Symbols `'a`, `'b`, ... are called *type variables* and stand for any types. Code written using these functions is checked for type correctness at compile time. And this guarantees strong properties at run time, for example that the elements of any list all have the same type.

These functions are polymorphic because they do nothing to list elements but copy them. Note that ML does all type checking at compile time. This is very different from “dynamically typed” languages such as Python or Scheme, where values are tagged with their types at runtime and checked only then.

### Computing the Length of a List

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

Slide 304

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

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 305

### 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
```

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

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.

### Append: List Concatenation

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

Slide 306

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

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. It tells us that two lists can be joined if their element types agree.

Slide 307

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

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

```
nrev[a, b, c] ⇒ nrev[b, c] @ [a]
               ⇒ (nrev[c] @ [b]) @ [a]
               ⇒ ((nrev[] @ [c]) @ [b]) @ [a]
               ⇒ (([] @ [c]) @ [b]) @ [a] ⇒ ... ⇒ [c, b, a]
```

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

$$0 + 1 + 2 + \dots + n = n(n + 1)/2.$$

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.

Slide 308

### Reversing a List in $O(n)$

```

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

```

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

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  $i = 0, \dots, n$  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 309

### Lists, Strings and Characters

character constants	#"A"    #"\""	...
string constants	" "    "B"    "Oh, no!"	...
<code>explode(s)</code>	<i>list of the characters in string s</i>	
<code>implode(l)</code>	<i>string made of the characters in list l</i>	
<code>size(s)</code>	<i>number of chars in string s</i>	
$s_1 \wedge s_2$	<i>concatenation of strings <math>s_1</math> and <math>s_2</math></i>	

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

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

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).

**Syntax You Must Know****Declarations**

Slide 310

 $\underline{\text{fun}} \text{ } Id \text{ } Pat_1 = E_1 \mid \dots \mid Id \text{ } Pat_n = E_n$ **Expressions**

[ ]

[  $E_1, \dots, E_n$  ]

In this lecture, the syntax of a function declaration has been generalised. A function declaration can consist of any number of separate clauses, each with its own pattern describing the argument. These patterns are matched from left to right when the function is actually called.

A special syntax for list expressions has also been introduced.

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

**Exercise 3.1** Code a recursive function to compute the sum of a list's elements. Then code an iterative version and comment on the improvement in efficiency.

**Exercise 3.2** Code a function to return the last element of a non-empty list. How efficiently can this be done?

**Exercise 3.3** Code a function to return the list consisting of the even-numbered elements of the list given as its argument. For example, given  $[a, b, c, d]$  this it should return  $[b, d]$ .

**Exercise 3.4** Consider the polymorphic types in these two function declarations:

```
fun id x = x;  
> val id = fn: 'a -> 'a  
fun loop x = loop x;  
> val loop = fn: 'a -> 'b
```

Explain why these types make logical sense, preventing run time type errors, even for expressions like `id [id [id 0]]` or `loop true / loop 3`.

### List Utilities: take and drop

#### Removing the first $i$ elements

Slide 401

```

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.

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

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 402

**Linear Search**

find  $x$  in list  $[x_1, \dots, x_n]$  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.

### Types with Equality

The membership test has a strange *polymorphic* type.

Slide 403

```
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*, making you write instead *Real.==(x,y)*, though some ML systems ignore this rule.

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.

### Equality Polymorphism

Slide 404

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

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

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

Slide 405

$$\left. \begin{array}{l} [x_1, \dots, x_n] \\ [y_1, \dots, y_n] \end{array} \right\} \mapsto [(x_1, y_1), \dots, (x_n, y_n)]$$

The *wildcard* pattern (`_`) matches *anything*.

THE PATTERNS ARE TESTED IN ORDER.

A list of pairs of the form  $[(x_1, y_1), \dots, (x_n, y_n)]$  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  $[z_1, \dots, z_n]$ .

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.

Slide 406

### Building a Pair of Results

```

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 above uses the *local declaration* `let D in E end`, where *D* consists of declarations and *E* is the expression that can use them. The `let`-construct counts as an expression and can be used (perhaps wrapped within parentheses) wherever an expression is expected.

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

### An Application: Making Change

Slide 407

```
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`, signifying that no pattern matches, namely if `till` becomes empty while `amt` is still nonzero. (Exceptions will be discussed later.)

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

Slide 408

### 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;

```

Now generalize the problem to return the list of **all possible ways** of making change. 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.

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

Slide 409

### 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.

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.



**Syntax You Must Know**

Slide 410

**Declarations**val *Pat* = *E***Expressions**let *D* in *E* end

Value declarations have been generalised in this lecture: a pattern can be given on the left-hand side rather than just an identifier. Then the expression *E*, once evaluated, is matched against this pattern (in case of failure, an exception will occur). This version takes apart in a structured value.

The local declaration let *D* in *E* end embeds the declaration *D* within the expression *E*. It is useful within a function, to perform intermediate computations using the arguments. *D* can be a single value or function declaration, or a long series of declarations.<sup>1</sup>

---

<sup>1</sup>A series of declarations, each optionally terminated using a semicolon, is actually regarded as a single declaration in ML.

**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 4.1** Code a function to implement set union, by analogy with `inter` above. It should avoid introducing repetitions, for example the union of the lists `[4,7,1]` and `[6,4,7]` should be `[1,6,4,7]` (though the order does not matter).

**Exercise 4.2** Code a function that takes a list of integers and returns two lists, the first consisting of all nonnegative numbers found in the input and the second consisting of all the negative numbers.

**Exercise 4.3** How does this version of `zip` differ from the one above?

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

**Exercise 4.4** What assumptions do the ‘making change’ functions make about the variables `till` and `amt`? Describe what could happen if these assumptions were violated.

**Exercise 4.5** Show that the number of ways of making change for  $n$  (ignoring order) is  $O(n)$  if there are two legal coin values. What if there are three, four, ... coin values?

**Exercise 4.6** We know nothing about the functions `f` and `g` other than their polymorphic types:

```
> val f = fn: 'a * 'b -> 'b * 'a
> val g = fn: 'a -> 'a list
```

Suppose that `f(1, true)` and `g 0` are evaluated and return their results. State, with reasons, what you think the resulting values will be.

### Sorting: Arranging Items into Order

a few applications:

- fast *search*
- fast *merging*
- finding *duplicates*
- *inverting tables*
- *graphics* algorithms

Slide 501

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 [5]! Sedgewick [10] 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.)

Slide 502

### How Fast Can We Sort?

typically count *comparisons*  $C(n)$

there are  $n!$  permutations of  $n$  elements

each comparison eliminates *half* of the permutations

$$2^{C(n)} \geq n!,$$

$$\text{therefore } C(n) \geq \log(n!) \approx n \log n - 1.44n$$

The usual measure of efficiency for sorting algorithms is the number of comparison operations required. Mergesort requires only  $O(n \log n)$  comparisons to sort an input of  $n$  items. It is straightforward to prove that this complexity is the best possible [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 [8]. Never mind how this works: generating statistically good random numbers is hard. Much effort has gone into those few lines of code.

```
local val a = 16807.0 and m = 2147483647.0
in fun nextrand seed =
    let val t = a*seed
    in t - m * real(floor(t/m)) end
and trunc 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, nextrand seed, seed::seeds);
val (seed,rs) = randlist(10000, 1.0, []);
```

Slide 503

### 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 504

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

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

### Quicksort: The Code

Slide 505

```

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.

### Append-Free Quicksort

Slide 506

```

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 [10]) 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

```
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

Slide 507

*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. 3).

Slide 508

**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 509

**Summary of Sorting Algorithms**

Optimal is  $O(n \log n)$  comparisons

*Insertion sort*: simple to code; too slow (*quadratic*) [174 secs]

*Quicksort*: fast on average; *quadratic* in worst case [0.53 secs]

*Mergesort*: optimal in theory; often slower than quicksort [1.4 secs]

MATCH THE ALGORITHM TO THE APPLICATION

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

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

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

Slide 510

**Syntax You DON'T Need to Know****Declarations**local  $D_1$  in  $D_2$  end

The local declaration local  $D_1$  in  $D_2$  end embeds the declaration  $D_1$  within another declaration,  $D_2$ . The effect is the same as writing the two declarations,  $D_1$  and  $D_2$ , one after another, except that  $D_1$  has no effect outside the local . . . end brackets. In the example above, the declarations of  $a$  and  $m$  were kept local to the declaration of the function `nextrandom`.

While occasionally useful, this declaration form is never actually necessary.

**Learning guide.** Related material is in *ML for the Working Programmer*, pages 108–113. The ultimate source is *Sorting and Searching*, volume 3 of *The Art of Computer Programming* by Donald E. Knuth. Look here for the beautiful Fibonacci sort, designed in the days when truly large amounts of data could only be stored on magnetic tape.

**Exercise 5.1** Another sorting algorithm (selection sort) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements. State, with justification, the time complexity of this approach.

**Exercise 5.2** Implement selection sort (see previous exercise) using ML.

**Exercise 5.3** Another sorting algorithm (bubble sort) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible. State, with justification, the time complexity of this approach.

**Exercise 5.4** Implement bubble sort (see previous exercise) using ML.

Slide 601

### 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`.

### Declaring a Function on Vehicles

Slide 602

```
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 [9, page 131].

Slide 603

### 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 604

**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.

### 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.

Slide 605

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

Slide 606

### Exceptions in ML

```
exception Failure;
```

**Declaring**

```
exception NoChange of int;
```

```
raise Failure
```

**Raising**

```
raise (NoChange n)
```

```
E handle Failure => E1
```

**Handling**

```
E handle Pat1 => E1 | ... | Patn => En
```

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`.

### Making Change with Exceptions

Slide 607

```
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. 4 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 608

```
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)
```

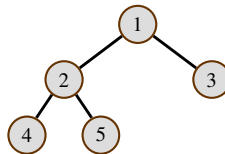
Here is the full execution. Observe how the exception handlers nest and how they drop away once the given expression has returned a value.

```
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]
```

Slide 609

### Binary Trees, a Recursive Datatype

```
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.

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

```
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.

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

```
datatype shape = Null
                | Join of shape * shape
```

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

Slide 610

### 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 +
                        Int.max(depth t1, depth t2)
```

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

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  $\text{leaves}(t) = \text{count}(t) + 1$ . 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!

Slide 611

### Syntax You Must Know

#### Declarations

datatype *TypeSpec* = *Con*<sub>1</sub> | ... | *Con*<sub>*n*</sub>  
exception *Con*

#### Matches

*Pat*<sub>1</sub> => *E*<sub>1</sub> | ... | *Pat*<sub>*n*</sub> => *E*<sub>*n*</sub>

#### Expressions

raise *E*  
*E* handle *M*

A datatype declaration introduces a type, which may be a name like `vehicle` or a type operator like `'a option`. In the latter case, more than one type operand is possible, as in `('a, 'b) vtree`. Here *Con* is a constructor specification, either just *Id* or *Id of T*. It is also used when declaring an exception.

A *Match* associates patterns with expressions.<sup>1</sup> A Match forms part of an exception handler, but we shall see other uses below. It behaves like a function declared with multiple cases: the patterns are matched one by one against a given value. The first matching pattern *Pat*<sub>*i*</sub> selects the corresponding expression *E*<sub>*i*</sub> for evaluation.

The general way to raise an exception is raise *E* because ML exceptions are simply values. In practice, *E* is usually just the name of an exception.

An expression of the form *E handle M* includes an exception handler. If *E* raises an exception, then this exception (which is an ML value, remember) is given to *M*, which will select another expression to evaluate.

<sup>1</sup>Match is also the name of a built-in exception. Please do not confuse the two.



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

**Exercise 6.1** Give the declaration of an ML datatype for The days of the week. Comment on the practicality of such a datatype in a calendar application.

**Exercise 6.2** Write an ML function taking a binary tree labelled with integers and returning their sum.

**Exercise 6.3** Examine the following function declaration. What does `ftree (1,n)` accomplish?

```
fun ftree (k,n) =  
  if n=0 then Lf else Br(k, ftree(2*k, n-1), ftree(2*k+1, n-1));
```

**Exercise 6.4** Give the declaration of an ML datatype for arithmetic expressions that have the following possibilities: real numbers, variables (represented by strings), or expressions of the form  $-E$ ,  $E + E$ ,  $E \times E$ .

**Exercise 6.5** Continuing the previous exercise, write a function that evaluates an expression. If the expression contains any variables, your function should raise an exception indicating the variable name.

Slide 701

### Dictionary

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

Ideally, an *abstract type* should provide these operations

... but *hide* the internal data structures!

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

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

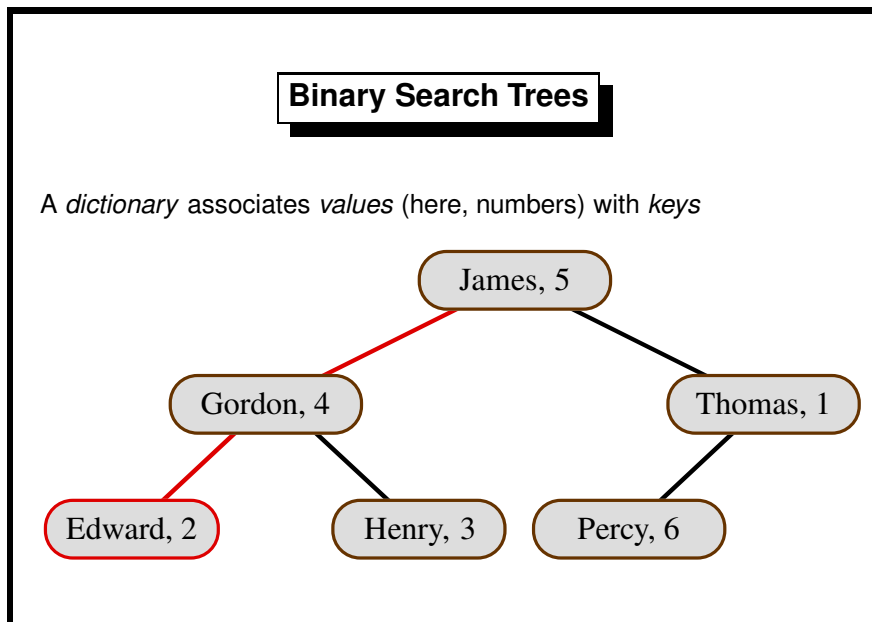
An *association list*—a list of pairs—is the simplest dictionary representation. Lookup is by linear search, and therefore slow:  $O(n)$ . Association lists are only usable if there are few keys in use. However, they are general in that the keys do not need a concept of ordering, only equality.

```
fun lookup ([], a)           = raise Missing
  | lookup ((x,y)::pairs, a) =
    if a=x then y else lookup(pairs, a);
fun update(l, b, y) = (b,y)::l
```

To enter a new (key, value) pair, simply “cons” it to the list. This takes constant time, which is the best we could hope for. But the space requirement is huge: linear in the number of updates, not in the number of distinct keys. Obsolete entries are

never deleted: that would require first finding them, increasing the update time from  $O(1)$  to  $O(n)$ .

Slide 702



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.

**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*!

Slide 703

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. 6, using the constructor NONE for failure.

Slide 704

### 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!**

If you are familiar with the usual update operation for this sort of tree, you may wonder whether it can be implemented in ML, where there is no direct way to replace part of a data structure by something else.<sup>1</sup>

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

The comparison between  $b$  and  $a$  allows three cases:

- *smaller*: update the left subtree; share the right
- *greater*: update the right subtree; share the left
- *equal*: update the label and share both subtrees

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

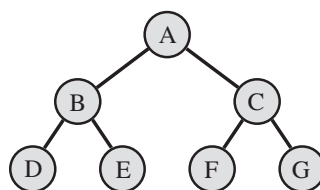
<sup>1</sup>Standard ML includes *reference types* that can be used to define *linked* data structures, but these techniques are no longer covered in this course.

**Aside: Traversing Trees (3 Methods)**

```
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];
```

Slide 705

*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 [6]. We can code these ‘visiting orders’ as functions that convert trees into lists of labels. Algorithms based on these notions typically perform some action at each node; the functions above simply copy the nodes into lists. Consider the tree



- *preorder* visits the label first (‘Polish notation’), yielding ABDECFG
- *inorder* visits the label midway, yielding DBEAFCG
- *postorder* visits the label last (‘Reverse Polish’), yielding DEBFGCA. You will be familiar with this concept if you own an RPN calculator.

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

### 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));
```

Slide 706

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. 5.

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

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

These three types of tree traversal are related in that all are depth-first. They each traverse the left subtree in full before traversing the right subtree. Breadth-first search (Lect. 10) is another possibility. That involves going through the levels of a tree one at a time.

Slide 707

### Arrays

A conventional array is an indexed storage area.

- It is updated *in place* by the command  $A[k] := x$
- The concept is inherently *imperative*.

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

- Updating implies *copying* to return  $\text{update}(A, k, x)$
- The new array equals  $A$  except that  $A[k] = x$ .

*Can we do updates efficiently?*

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

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

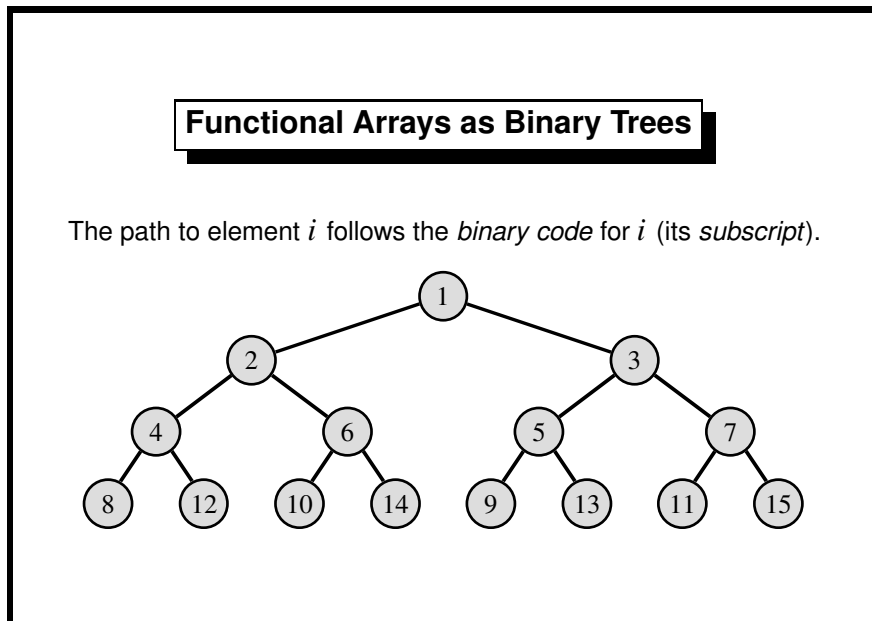
The drawback is that subscripting is a chief cause of programmer error. That is why arrays play little role in this introductory course.

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

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



Slide 708



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. Inspection of the diagram above should make it clear that these trees are always balanced: the left subtree can have at most one node more than the right subtree, recursively all the way down. (This assumes that the array is defined for subscripts  $1 \dots n$  with no gaps; an array defined only for odd numbers, for example, would obviously be unbalanced.)

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

### The Lookup Function

```
exception Subscript;
```

Slide 709

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

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

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

### The Update Function

Slide 710

```

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.

**Exercise 7.1** Draw the binary search tree that arises from successively inserting the following pairs into the empty tree: (Alice, 6), (Tobias, 2), (Gerald, 8), (Lucy, 9). Then repeat this task using the order (Gerald, 8), (Alice, 6), (Lucy, 9), (Tobias, 2). Why are results different?

**Exercise 7.2** Code an insertion function for binary search trees. It should resemble the existing update function except that it should raise the exception `Collision` if the item to be inserted is already present.

**Exercise 7.3** Continuing the previous exercise, it would be natural for exceptional `Collision` to return the value previously stored in the dictionary. Why is that goal difficult to achieve?

**Exercise 7.4** Describe an algorithm for deleting an entry from a binary search tree. Comment on the suitability of your approach.

**Exercise 7.5** Code the delete function outlined in the previous exercise.

**Exercise 7.6** Show that the functions `preorder`, `inorder` and `postorder` all require  $O(n^2)$  time in the worst case, where  $n$  is the size of the tree.

**Exercise 7.7** Show that the functions `preord`, `inord` and `postord` all take linear time in the size of the tree.

**Exercise 7.8** Write a function to remove the first element from a functional array. All the other elements are to have their subscripts reduced by one. The cost of this operation should be linear in the size of the array.

Slide 801

### Functions as Values

In ML, functions can be

- passed as *arguments* to other functions,
- returned as *results*,
- put into lists, trees, etc.,

```
[fn n => n*2, fn n => n*3, fn k => k+1];  
> val it = [fn, fn, fn]: (int -> int) list
```

- but **not** tested for equality.

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  $\alpha$  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 operates on other functions. Many functionals are familiar from mathematics: for example, the differential operator maps functions to their derivatives, which are also functions. To a mathematician, a function is typically an infinite, uncomputable object. We use ML functions to represent algorithms. Sometimes they represent infinite collections of data given by computation rules.

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

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

### Functions Without Names

$\underline{fn} \ x \Rightarrow E$  is the function  $f$  such that  $f(x) = E$

The function  $(\underline{fn} \ n \Rightarrow n*2)$  is a *doubling function*.

Slide 802

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

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

The main purpose of `fn`-notation is to package up small expressions that are to be applied repeatedly using some other function. The expression  $(\underline{fn} \ n \Rightarrow n*2)$  has the same value as the identifier `double`, declared as follows:

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

The `fn`-notation allows pattern-matching, just as in exception handlers, to express functions with multiple clauses:

```
fn P1 => E1 | . . . | Pn => En
```

This rarely-used expression abbreviates the local declaration

```
let fun f(P1) = E1 | . . . | f(Pn) = En
in f end
```

For example, the following declarations are equivalent:

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

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

### 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`.

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

Slide 803

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

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

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

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

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

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

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

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

*Note:* The parentheses may be omitted in  $(\text{fn } a \Rightarrow (\text{fn } b \Rightarrow E))$ . They may also be omitted in  $(\text{prefix } \text{"Doctor "}) \text{"Who"}$ .

Slide 804

### Shorthand for Curried Functions

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

This curried function 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).

In ML, an  $n$ -argument curried function  $f$  can be declared using the syntax

$$\text{fun } f \ x_1 \ . \ . \ . \ x_n = E$$

and applied using the syntax  $f \ E_1 \ . \ . \ . \ E_n$ . If  $f$  is not recursive, then it is equivalent to the function expressed via nesting as follows:

$$\text{fn } x_1 \Rightarrow \dots (\text{fn } x_n \Rightarrow E)$$

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

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

```
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*, like in Java.



Slide 805

### Partial Application: A Curried Insertion Sort

```

fun insert 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 insert = fn : ('a * 'a -> bool)
>                               -> ('a list -> 'a list)

```

The sorting functions of Lect. 5 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, `insert` is a curried function. Given its first argument, a predicate for comparing some particular type of items, it returns the function `sort` for sorting lists of that type of items.

Some examples of its use:

```

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

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

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

```

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

Slide 806

**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 below. 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 807

**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  $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$  is  $\begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$ , which in ML corresponds to the following transformation on lists of lists:

$$[[a,b,c], [d,e,f]] \mapsto [[a,d], [b,e], [c,f]]$$

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 808

### Review of Matrix Multiplication

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

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

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

A simple case of matrix multiplication is when  $A$  consists of a single row and  $B$  consists of a single column. Provided  $A$  and  $B$  contain the same number  $k$  of elements, multiplying them yields a  $1 \times 1$  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 809

### 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.

### List Functionals for Predicates

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

Slide 810

```
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`.

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 811

```
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
```

Lecture 4 presented the function `member`, which tests whether a specified value can be found as a list element, and `inter`, which returns the “intersection” of two lists: the list of elements they have in common.

But remember: the purpose of list functionals is not to replace the declarations of popular functions, which probably are available already. It is to eliminate the need for separate declarations of ad-hoc functions. When they are nested, like the calls to `all` in `disjoint` above, the inner functions are almost certainly one-offs, not worth declaring separately.

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

*Historical Note:* Alonzo Church’s  $\lambda$ -calculus gave a simple syntax,  $\lambda$ -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  $\lambda$ -calculus had a tremendous influence on the design of functional programming languages. McCarthy’s Lisp was something of a false start; it interpreted variable binding incorrectly, an error that stood for some 20 years. But in 1966, Peter Landin (of Queen Mary College, University of London) sketched out the main features of functional languages.

**Syntax You Must Know**

Slide 812

**Declarations** $\underline{\text{fun}} \text{ } Id \text{ } Pat_1 \dots Pat_n = E$ **Expressions** $\underline{\text{fn}} \text{ } M$ 

Function declarations are now generalised so that curried functions can be expressed easily.

A  $\lambda$ -expression usually has the form  $\underline{\text{fn}} \text{ } x \Rightarrow E$ . But in general, a match  $M$  may follow the  $\underline{\text{fn}}$  keyword, which allows non-recursive pattern matching in the function. This generality is seldom used, and you only really need to remember the simple version.

Recall that matches were introduced at the end of Lect. 6.



**Learning guide.** Related material is in *ML for the Working Programmer*, pages 171–190. Chapter 9 contains an introduction to the  $\lambda$ -calculus.

**Exercise 8.1** What does the following function do, and what are its uses?

```
fun sw f x y = f y x;
```

**Exercise 8.2** There are many ways of combining orderings. The *lexicographic ordering* uses two keys for comparisons. It is specified by

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

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

**Exercise 8.3** Without using `map`, write a function `map2` such that `map2 f` is equivalent to `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with only one by exploiting nested pattern-matching.

**Exercise 8.4** The type `option`, declared below, can be viewed as a type of lists having at most one element. (It is typically used as an alternative to exceptions.) Declare an analogue of the function `map` for type `option`.

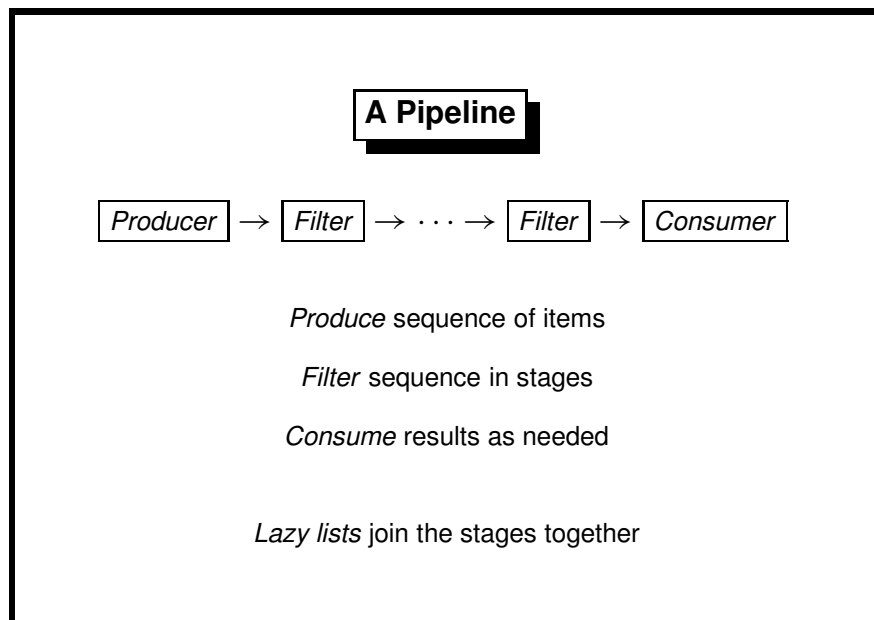
```
datatype 'a option = NONE | SOME of 'a;
```

**Exercise 8.5** Recall the making change function of Lect. 4:

```
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 901



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 902

### 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*.

### Lazy Lists in ML

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

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

Slide 903

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

**sequences**

```
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. 12.)

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  $E$ .

**The Infinite Sequence  $k, k + 1, k + 2, \dots$** **Slide 904**

```
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.

### Consuming a Sequence

Slide 905

```
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 906

**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 [9, 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`.

### Joining Two Sequences

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

Slide 907

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. 3): 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.



**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  $x, f(x), f(f(x)), \dots$* 

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

Slide 908

The functional `filterq` demands elements of `xq` until it finds one satisfying `p`. (Recall `filter`, the analogous operation for ordinary lists.) It contains a *force* not protected by a *delay*. If `xq` is infinite and contains no satisfactory element, then `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?*

```
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)
```

Slide 909

The *Newton-Raphson method* is widely used for computing square roots. The infinite series  $x_0, (a/x_0 + x_0)/2, \dots$  converges rapidly to  $\sqrt{a}$ . The initial approximation,  $x_0$ , is typically retrieved from a table, and is accurate enough that only a few iterations of the method are necessary. Calling `iterates (next a) x0` generates the *infinite series* of approximations to the square root of  $a$  using the Newton-Raphson method. To compute  $\sqrt{2}$ , the resulting series begins 1, 1.5, 1.41667, 1.4142157, 1.414213562 ..., and this last figure is already accurate to 10 significant digits!

Function `within` searches down the lazy list for two points whose difference is less than `eps`. It tests their absolute difference. Relative difference and other 'close enough' tests can be coded. Such components can be used to implement other numerical functions directly as functions over sequences. The point is to build programs from small, interchangeable parts.

Function `root` uses `within`, `iterates` and `next` to apply Newton-Raphson with a tolerance of  $10^{-6}$  and a (poor) initial approximation of 1.0.

This treatment of numerical computation has received some attention in the research literature; a recurring example is *Richardson extrapolation* [3, 4].

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

**Exercise 9.1** Code an analogue of map for sequences.

**Exercise 9.2** 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 9.3** Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint:* to benefit from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq.`)

**Exercise 9.4** A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an ML datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. (Taken from the exam question 2008 Paper 1 Question 5.)

Slide 1001

**Breadth-First v Depth-First Tree Traversal**binary trees as *decision trees*Look for *solution nodes*

- *Depth-first*: search one subtree in full before moving on
  - *Breadth-first*: search all nodes at level  $k$  before moving to  $k + 1$
- Finds *all* solutions — nearest first!

Preorder, inorder and postorder tree traversals all have something in common: they are *depth-first*. At each node, the left subtree is entirely traversed before the right subtree. Depth-first traversals are easy to code and can be efficient, but they are ill-suited for some problems.

Suppose the tree represents the possible moves in a puzzle, and the purpose of the traversal is to search for a node containing a solution. Then a depth-first traversal may find one solution node deep in the left subtree, when another solution is at the very top of the right subtree. Often we want the shortest path to a solution.

Suppose the tree is *infinite*, or simply extremely large. Depth-first search is almost useless with such trees, for if the left subtree is infinite then the search will never reach the right subtree. ML can represent infinite trees by the means discussed in Lect. 9. Another tree representation (suitable for solving solitaire, for example) is by a function `next : pos -> pos list`, which maps a board position to a list of the positions possible after the next move. For simplicity, the examples below use the ML datatype `tree`, which has only finite trees.

A *breadth-first* traversal explores the nodes horizontally rather than vertically. When visiting a node, it does not traverse the subtrees until it has visited all other nodes at the current depth. This is easily implemented by keeping a list of trees to visit. Initially, this list consists of one element: the entire tree. Each iteration removes a tree from the head of the list and adds its subtrees after the end of the list.

**Breadth-First Tree Traversal — Using Append**

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

**Slide 1002**

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 1003

### 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. 7) 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 1004

### Efficient Functional Queues: Idea

Represent the queue  $x_1 x_2 \dots x_m y_n \dots y_1$

by any **pair of lists**

$$([x_1, x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

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  $\text{enq}(q, x)$  performs  $q \leftarrow x :: q$ , then this operation will be  $O(n)$ . We could represent queues by reversed lists, implementing  $\text{enq}(q, x)$  by  $x :: q$ , but then the  $\text{deq}$  and  $\text{qhd}$  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

$$([x_1, x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

represents the queue  $x_1 x_2 \dots x_m y_n \dots y_1$ .

The front part of the queue is stored in order, and the rear part is stored in reverse order. The  $\text{enq}$  operation adds elements to the rear part using  $\text{cons}$ , since this list is reversed; thus,  $\text{enq}$  takes constant time. The  $\text{deq}$  and  $\text{qhd}$  operations look at the front part, which normally takes constant time, since this list is stored in order. But sometimes  $\text{deq}$  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 1005

```
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.



**Aside: The case Expression****Slide 1006**

```
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

$$\text{case } E \text{ of } Pat_1 \Rightarrow E_1 \mid \dots \mid Pat_n \Rightarrow E_n$$

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. 6) as shown above.

Recall that a program phrase of the form  $Pat_1 \Rightarrow E_1 \mid \dots \mid Pat_n \Rightarrow E_n$  is called a Match (Lect. 6). A match may also appear after an exception handler (Lect. 6) and with `fn`-notation to express functions (Lect. 8).

**Breadth-First Tree Traversal — Using Queues****Slide 1007**

```
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.

### Iterative deepening: Another Exhaustive Search

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

$$1 + b + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} \quad \begin{array}{l} b = \text{branching factor} \\ d = \text{depth} \end{array}$$

*Recompute* nodes at depth  $d$  instead of storing them

Time factor is  $b/(b - 1)$  if  $b > 1$ ; complexity is still  $O(b^d)$

Space required at depth  $d$  drops from  $b^d$  to  $d$

Slide 1008

Breadth-first search is not practical for infinite trees: it uses too much space. Large parts of the tree have to be stored. Consider the slightly more general problem of searching trees whose branching factor is  $b$  (for binary trees,  $b = 2$ ). Then breadth-first search to depth  $d$  examines  $(b^{d+1} - 1)/(b - 1)$  nodes, which is  $O(b^d)$ , ignoring the constant factor of  $b/(b - 1)$ . Since all nodes that are examined are also stored, the space and time requirements are both  $O(b^d)$ .

*Depth-first iterative deepening* combines the space efficiency of depth-first with the ‘nearest-first’ property of breadth-first search. It performs repeated depth-first searches with increasing depth bounds, each time discarding the result of the previous search. Thus it searches to depth 1, then to depth 2, and so on until it finds a solution. We can afford to discard previous results because the number of nodes is growing exponentially. There are  $b^{d+1}$  nodes at level  $d + 1$ ; if  $b \geq 2$ , this number actually exceeds the total number of nodes of all previous levels put together, namely  $(b^{d+1} - 1)/(b - 1)$ .

Korf [7] shows that the time needed for iterative deepening to reach depth  $d$  is only  $b/(b - 1)$  times that for breadth-first search, if  $b > 1$ . This is a constant factor; both algorithms have the same time complexity,  $O(b^d)$ . In typical applications where  $b \geq 2$  the extra factor of  $b/(b - 1)$  is quite tolerable. The reduction in the space requirement is exponential, from  $O(b^d)$  for breadth-first to  $O(d)$  for iterative deepening. Of course, this assumes that the tree itself is not stored in memory.

### Another Abstract Data Type: Stacks

Slide 1009

- `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 1010

### 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.

**Syntax You Must Know**

Slide 1011

**Expressions**case  $E$  of  $M$ 

Although introduced relatively late in this course, case expressions are convenient and useful. Note that the body is simply a match.

These expressions are a generalisation of a programming construct invented by C. A. R. Hoare in 1964. (The C programming language has a corrupt variant of this construct, called *switch*, which allows execution to fall from one place to the next. This behaviour is almost never wanted and is the cause of many bugs.)

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

**Exercise 10.1** Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented above.

**Exercise 10.2** The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

**Exercise 10.3** Write a version of the function shown on slide 1007 using a nested let construction rather than case.

**Exercise 10.4** Iterative deepening is inappropriate if  $b \approx 1$ , where  $b$  is the branching factor. What search strategy is appropriate in this case?

**Exercise 10.5** Consider the following ML function.

```
fun next n = [2*n, 2*n+1];
```

If we regard it as representing a tree, where the subtrees are computed from the current label, what tree does `next 1` represent?

Slide 1101

### 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  $a_n x^n + \dots + a_0 x^0$

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.

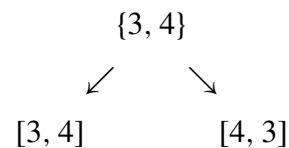


Slide 1102

### Data Representation Example: *Finite Sets*

represent by *repetition-free lists*

representations not *unique*:



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. 4) 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.

### A Data Structure for Polynomials

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

REAL coefficients (should be *rational*)

*Sparse* representation (no zero coefficients)

*Decreasing* exponents

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

Slide 1103

The univariate polynomial  $a_n x^n + \dots + a_0 x^0$  might be represented by the list of coefficients  $[a_n, \dots, a_0]$ . 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  $a_n x^n + \dots + a_0 x^0$  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 1104

### 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 [9]. 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. 7), 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 1105

```
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.<sup>1</sup>

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.

<sup>1</sup>Some ML compilers insist upon `Real.==(a+b,0.0)` instead of `a+b=0.0` above.

### 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  $(x + 1)^{400}$

Slide 1106

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
```

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

**Polynomial multiplication (2nd try)****Slide 1107**

```
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  $(x + 1)^{400}$*

A faster algorithm is inspired by mergesort (Lect. 5). 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 1108

```

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  $k$ th 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

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

Slide 1109

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

**Exercise 11.2** Give a convincing argument that polysum and polyprod preserve the key restrictions on polynomials: all coefficients are nonzero and the exponents are decreasing.

**Exercise 11.3** Show that the complexity of polysum is  $O(m + n)$  when applied to arguments consisting of  $m$  and  $n$  terms, respectively.

**Exercise 11.4** Give a more rigorous analysis of the asymptotic complexity of the two versions of polynomial multiplication. (Difficult!)

**Exercise 11.5** Experiment with a real computer algebra system. Student discounts are frequently available for the leading systems, Maple and Mathematica. Several open source computer algebra systems exist too, such as sage (<http://www.sagemath.org/>) and REDUCE (<http://www.reduce-algebra.com/>).

Slide 1201

### 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
- *linked structures*, especially *linked lists*

Procedural programming is programming in the traditional sense of the word. A program *state* is repeatedly transformed by the execution of *commands* or *statements*. A state change might be local to the machine and consist of updating a variable or array. A state change might consist of sending data to the outside world. Even reading data counts as a state change, since this act normally removes the data from the environment.

Procedural programming languages provide primitive commands and control structures for combining them. The primitive commands include *assignment*, for updating variables, and various *input/output* commands for communication. Control structures include *if* and *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; 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 mainly for communication with the outside world.

Slide 1202

### ML Primitives for References

$\tau \text{ ref}$	<i>type of references to type <math>\tau</math></i>
$\text{ref } E$	<i>create a reference</i> <i>initial contents = the value of <math>E</math></i>
$! P$	<i>return the current contents of reference <math>P</math></i>
$P := E$	<i>update the contents of <math>P</math> to the value of <math>E</math></i>

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 *locations*). Calling `ref` allocates a new location in memory. 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  $\tau$  is some ML type, then  $\tau \text{ ref}$  is the type of references to cells that can hold values of  $\tau$ . Please do not confuse the type `ref` with the function `ref`. This table of the primitive functions and their types might be useful:

<code>ref</code>	<code>'a -&gt; 'a ref</code>
<code>!</code>	<code>'a ref -&gt; 'a</code>
<code>op :=</code>	<code>'a ref * 'a -&gt; unit</code>

Slide 1203

**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 on the *left*-hand side of the `:=` denote locations and those on the *right*-hand side denote the contents. A special 'address of' operator may be available to override the convention and make a reference on the right-hand side to denote a location. Logically this is a mess, but it makes programs shorter.

The list `ps` is declared to hold a new reference (initially containing 77) as well as `p`. Then the new reference is updated to hold 3. The assignment to `hd ps` does NOT update `ps`, only the contents of a reference in that list.

Slide 1204

**Commands: Expressions with Effects**

- Basic commands update references, write to files, etc.
- $C_1; \dots; C_n$  causes a series of expressions to be evaluated and returns the value of  $C_n$ .
- A typical command returns the empty tuple:  $()$
- if  $B$  then  $C_1$  else  $C_2$  behaves like the traditional control structure if  $C_1$  and  $C_2$  have effects.
- Other ML constructs behave naturally with commands, including case expressions and recursive functions.

We use the term *command* informally to refer to an expression that has an effect on the state. All expressions denote some value, but they can return  $()$ , which conveys no actual information.

We need a way to execute one command after another. The construct  $C_1; \dots; C_n$  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).

### Iteration: the **while** Command

while *B* do *C*

Slide 1205

```
fun length xs =
  let val lp = ref xs           list of uncounted elements
      val 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.

Slide 1206

**Private, Persistent References**

```
exception TooMuch of int;  
fun makeAccount (initBalance: int) =  
  let val balance = ref initBalance  
    fun withdraw amt =  
      if amt > !balance  
      then raise TooMuch (amt - !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, raising 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.

**Two Bank Accounts****Slide 1207**

```
val student = makeAccount 500;  
> val student = fn : int -> int  
  
val director = makeAccount 4000000;  
> val director = fn : int -> int  
  
student 5;          (*coach fare*)  
> val it = 495 : int  
  
director 150000;    (*Ferarri*)  
> val it = 3850000 : 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 [9, 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 1208

### ML Primitives for Arrays

$\tau$ Array.array	type of arrays of type $\tau$
Array.tabulate( $n, f$ )	create a $n$ -element array $A[i]$ initially holds $f(i)$
Array.sub( $A, i$ )	return the <i>contents</i> of $A[i]$
Array.update( $A, i, E$ )	update $A[i]$ to the value of $E$
<b>And countless others!</b>	

```

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

```

ML arrays are like references that hold several elements instead of one. The elements of an  $n$ -element array are designated by the integers from 0 to  $n - 1$ . The  $i$ th array element is usually written  $A[i]$ . If  $\tau$  is a type then  $\tau$  Array.array is the type of arrays (of any size) with elements from  $\tau$ .

Calling Array.tabulate( $n, f$ ) creates an array of the size specified by expression  $n$ . Initially, element  $A[i]$  holds the value of  $f(i)$  for  $i = 0, \dots, n - 1$ . Like ref, it allocates mutable storage to hold the specified values.

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

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

ML provides immutable arrays, which are called *vectors*. The operation Vector.tabulate can be used to build a read-only table of function values: worthwhile if the function is computationally expensive.

ML's arrays are much safer than C's. In C, an array is nothing more than an address indicating the start of a storage area. Nothing indicates the size of the area. Therefore C programs are vulnerable to *buffer overrun attacks*: a 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 virus code.

### Array Examples

Slide 1209

```
val ar = Array.tabulate(20, fn i => i*i);  
> val ar = [|0,1,4,9,16,25,...|] : int array  
  
Array.sub(ar,2);  
> val it = 4 : int  
  
Array.sub(ar,20);  
> uncaught exception Subscript  
  
Array.update(ar,2,~33);  
ar;  
> val it = [|0,1,~33,9,16,25,...|] : int array
```

In this session, the identifier `ar` is bound to an array of 20 elements, which are initially set to the squares of their subscripts. (Different ML systems have different ways of displaying an array's value.) The array's third element (which actually has subscript 2) is inspected and found to be four. The second call to `Array.sub` supplies a subscript that is out of range, so ML rejects it.

By calling `Array.update`, we then modify the element with subscript 2. Note however that we cannot modify the array's length. If we outgrow the array, we have to create a new one, copy the data into it, and then forget the old array. Typically the new array would be double the size of the old one, so that the cost of copying is insignificant.

ML provides numerous operators for modifying, computing over and searching in arrays. Many are analogous to functions on lists. For example, `Array.exists` takes a boolean-valued function and returns true if an array element satisfies it.

Slide 1210

**References: ML Versus Conventional Languages**

- We must write `!p` to get the *contents* of `p`
- We write just `p` for the *address* of `p`
- We can store private reference cells (like `balance`) in functions—simulating **object-oriented programming**
- ML's assignment syntax is `V := E` instead of `V = E`
- ML has few control structures: only `while` and `case`
- ML has no nice syntax for arrays (especially for updating an array)

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.

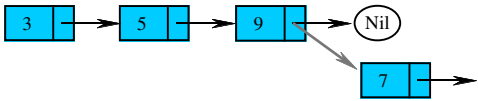
Dereferencing operators (like ML's `!`) are especially unpopular, because they clutter the program text. Virtually all programming languages make dereferencing implicit (that is, automatic).

It is generally accepted these days that a two-dimensional array `A` is nothing but an array of arrays. An assignment to such an array is typically written something like `A[i, j] := x`; in C, the syntax is `A[i][j] = x`. Higher dimensions are treated analogously. The corresponding ML code is a horrible combination of `Array.update` and `Array.sub`.

Slide 1211

### What More is There to ML?

You can easily create linked (mutable) lists.



```

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

```

- *Libraries* interfacing to various operating systems and services
- *Modules*: structures, signatures and functors
- Threads and “futures” packages for *multi-core programming*

It is worth mentioning that ML’s references fully suffice for coding the sort of linked data structures taught in algorithms courses, and is illustrated in the figure above. The programming style is a little different from the usual, but the principles are the same. There are many examples in *ML for the Working Programmer*, pages 326–340. ML also provides comprehensive input/output primitives for various types of file and operating system.

ML’s system of modules include *structures*, which can be seen as encapsulated groups of declarations, and *signatures*, which are specifications of structures listing the name and type of each component. Finally, there are *functors*, which are analogous to functions that combine a number of argument structures, and which can be used to plug program components together. These primitives are useful for managing large programming projects.

This lecture course has used Standard ML as a basis for teaching the basic principles of programming. However, ML was designed primarily for writing research software, especially for automated theorem proving. Substantial software can be coded in ML. The Isabelle proof checker includes over 150,000 lines of Standard ML, structured as purely functional modules connected by Poly/ML’s threads package (a nonstandard language extension) to provide effective parallelism. For example, on an eight-core machine, a speed-up of six is not unusual. And Standard ML’s sister language, OCaml, is also heavily used both in research and even in the finance sector. While C and Java dominate the scene, there is some room for languages that not only achieve reasonable performance but avoid the worst programming hazards and have a hope of producing the right answers.

**Syntax You Must Know**

Slide 1212

**Expressions** $E_1; \dots; E_n$ while  $E_1$  do  $E_2$ 

Although this chapter introduces the new paradigm of procedural programming, it introduces little new syntax. We informally talk about commands, but as far as ML is concerned, there are only expressions. References and the operations on them are simply types and functions. The only new syntax is that shown above.

- Semicolons can be used to combine expressions sequentially. In practice, all of the expressions apart from the last had better have side-effects, since their values are discarded.
- The iteration of a command can be expressed using while. Arguably, ML would benefit from the addition of further iteration structures, such as some sort of indexed loop (a for statement).

**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 12.1** Comment, with examples, on the differences between an `int ref list` and an `int list ref`.

**Exercise 12.2** Write a version of function `power` (Lect. 2) using `while` instead of recursion.

**Exercise 12.3** What is the effect of `while (C1; B) do C2`?

**Exercise 12.4** Write a function to exchange the values of two references, `xr` and `yr`.

**Exercise 12.5** 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. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

## 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] Matthew Halfant and Gerald Jay Sussman. Abstraction in numerical methods. In *LISP and Functional Programming*, pages 1–7. ACM Press, 1988.
- [4] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
- [5] Donald E. Knuth. *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison-Wesley, 1973.
- [6] Donald E. Knuth. *The Art of Computer Programming*, volume 1: *Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1973.
- [7] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
- [8] 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.
- [9] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
- [10] Robert Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.