Book Summary

The comprehensive manual "Computer Science Essentials - Navigating the Digital World" demystifies the complicated and constantly changing field of computer science for readers from all walks of life. This book acts as a crucial road map for comprehending the digital world in an age where technology affects every area of our life. The book begins by covering the basics before introducing readers to important ideas in computer science like algorithms, programming languages, and data structures. It offers useful insights for more seasoned readers while making these topics understandable to beginners through simple explanations and examples from real-world applications.

Readers dig into complex subjects like artificial intelligence, cybersecurity, and the moral implications of technology as they move through the book. The work places an immense value on practical applications to make sure readers can use their newly acquired information in a variety of professional and domestic situations.The text stresses the value of critical thinking, problem-solving, and digital literacy along the trip. It gives readers the skills they need to effectively navigate the digital world, whether they are students, teachers, professionals, or just curious people.

The book "Computer Science Essentials" is a requirement-read for the twenty-first century for anyone wishing to comprehend and succeed in the constantly shifting digital world.

UNESCO LAUREATE Prof. Sir Bashiru Aremu
Professor & Vice Chancellor
@ Crown University Int'l Chartered Inc. (CUICI) USA

Prof. Dr. K. Mahammad Rafi
Founder President @ eSkillGrow Virtual University LLC, USA
Board Member & Global Director
@ Crown University Int'l Chartered Inc. (CUICI) USA

Dr. Mir Iqbal Faheem
Professor & Director of Technical Campus,
Deccan Group of Institutions, Hyderabad, India

Dr. Shaik Abdul Jaffar , Professor,
Department of Electronics and Instrumentation Engineering,
Deccan College of Engineering and Technology,
Hyderabad, India

9 788196 623654

Registered in "Global Register of Publishers "

Computer Science Essentials - Navigating The Digital World

Doctorate Int'l Publications

Computer Science Essentials

Navigating The Digital World

Prof.Sir.Bashiru Aremu
Prof.Dr.K .Mahammad Rafi
Dr.Mir.Iqbal Faheem
Dr. Shaik Abdul Jaffar

Doctorate Int'l Publications
Dept. of Research & Publications
eSkillGrow Virtual University LLC USA
CoE @India, Japan, Germany, Poland, Malaysia

Prof.Sir.Bashiru Aremu
Prof.Dr.K .Mahammad Rafi
Dr.Mir.Iqbal Faheem
Dr. Shaik Abdul Jaffar

Imprint

Cover Image: www.canva.com

publisher:
Doctorate publications
is an International Publishing house
under Department of Research & Publications
@ eSkilllGrow Virtual University LLCs (regd as per USA govt int'l laws)

USA :

International Regd agent office at Delaware and California, USA
16192.coastal highway city of Lewes.

Administrative Office of Registered agents inc. 90 state street, ste 700 office 40, albany 12207 , county: albany, New York city, USA

e-101 kitchawan rd, Yorktown heights, ny 10598, USA

GERMANY:
34/09-a, geschwister-scholl-straße 7, d-39307 genthin, germany

JAPAN:
a-19-21, ihonbashihakozakichō, chūō-ku, tōkyō-to-103-0015, japan.

POLAND:
b-2/45, ul. a. kręglewskiego 11, 61-248-2 poznań, poland

INDIA:
4&5, arpita enclave, karmanghat, Hyderabad, telangana

Computer Science Essentials

Navigating The Digital World

Doctorate Publications

Index

Chapter 1 Introduction

There are two goals for this course. Teaching programming comes first and it should be evident.

The second step is to introduce some basic ideas in computer science, particularly those related to algorithm design. Although the majority of students will already have some programming expertise, there are still individuals whose programming skills can be improved with more understanding of fundamental concepts. If you have considerable experience and find certain parts of the course to be slow, kindly keep this in mind.

This course's programming is based on the ML programming language and mostly uses the functional programming paradigm. Comparing functional programmes to those written in traditional languages like C, they are typically shorter and simpler to grasp.

We'll be able to cover the majority of data structure types used in programming in a few of weeks. The training also covers the fundamentals of efficiency estimation.

Courses taught in the computer lab are now required to include a learning guide that includes recommendations for additional reading, discussion topics, exercises, and sample exam questions. Such materials are related to each lecture in this course.

The additional reading is primarily taken from the second edition of my book, ML for the Working Programmer, which also includes a tonne of exercises. The only exam questions that are pertinent come from the Part 1A examinations from June 1998.

Many thanks to Frank Stajano, James Margetson, Frank King, Joseph Lord, Stuart Becker, Silas Brown, and Frank King for pointing out mistakes in these notes. Please let me know if you find any more mistakes or passages that are particularly difficult to understand. I'll credit your idea if I utilise it in the subsequent printing.

Suggested Reading List

Naturally, these notes are most similar in style to my own book. Another general introduction to ML is found in Ullman's book. A pretty odd tutorial on recursion and types is The Little MLer. Harrison is less directly relevant, but still worth taking into account. O-notation is described in Introduction to Algorithms.

Understanding Computer Abstraction Levels

Mastering Computer Abstraction

Children can use computers, but NO ONE can truly comprehend them. Using several layers of abstraction, master complexity

At most, concentrate on 2 or 3 levels!

Recurring issues:

- What services should be offered at each level?
- How should they be implemented utilising lower-level services?
- The interface: The communication channel between the two layers

A fundamental idea in computer science is that complex systems can only be comprehended in terms of levels, with each level further separated into several functions or services. The marketed services should be available through the interface to the higher level. It should also restrict access to the channels used to carry out those services, which is as crucial. This abstraction barrier enables changes at one level to be made without having an impact on higher levels. For instance, it's crucial that existing programmes continue to function on a faster version of a CPU when it's designed by a manufacturer. The programme shouldn't be able to tell the difference between the old and new processors.

Example I: Dates

Abstract level: names for dates over a certain range

Concrete level: typically 6 characters: YYMMDD

Date crises caused by INADEQUATE internal formats:

- Digital's PDP10: using 12bit dates (good for at most 11 years)
- 2000 crisis: 48 bits could be good for lifetime of universe!

Lessons:

- There are numerous ways to portray information.
- Make a mistake and you'll pay

The date crisis at Digital Equipment Corporation happened in 1975. A 36-bit mainframe computer, the PDP-10. It displayed dates in a 12-bit format created for the little PDP-8. One may distinguish 212 = 4096 days or 11 years with 12 bits.

Six characters—two for the year, two for the month, and two for the day—make up the standard industrial format for dates. The most typical "solution" to the year 2000 problem is to increase file sizes by two more characters. Others have noted that the six characters that are already in use contain 48 bits each, which is already sufficient to represent every date for the anticipated duration of the universe:

$2^{48} = 2.8 \times 10^{14}$ days = $7{:}7 \times 10^{11}$ years!

In contrast to how computers are typically represented, mathematicians think in terms of unbounded ranges. A powerful programming language like ML makes changing the program's representation simple. However, conversion issues will still remain if les in the outdated representation are prevalent everywhere. In the computer industry as a whole, compatibility with outdated systems is a challenge.

Example II: Floating Point Numbers

Computers can store real numbers like 1.066 x 103 and integers like 1066.

Two integers can be used to represent a floating-point number.

There may be various precisions for either type of number.

The idea of a DATA TYPE:

- how a value is represented
- the suite of available operations

Any pocket calculator provides floating point numbers. A float's mantissa (fractional component) and exponent are both integers on the inside. Two real numbers may be provided as complex numbers.

There are already three tiers of numbers!

Most computers also let us choose from a variety of precisions. Integers typically range from 231 1 (i.e. 2,147,483,647) to -231 in 32-bit precision, while reals are accurate to roughly six decimal places and can reach sizes of around 1035.

64-bit precision is frequently recommended for reals. How can we manage so many different types of numbers? If we perform floating-point arithmetic on an integer, the outcome is unpredictable and could even differ from one chip version to another.

Early programming languages like Fortran forbade programmers from combining both real and imaginary numbers in a calculation by requiring variables to be defined as either integer or real. Programmes may now handle a wide variety of data types, including text and symbols. The concept of data type is used in modern languages to guarantee that only actions that are pertinent to the datum are performed on it.

All information is stored inside a computer as bits. It is hard to tell which type a specific bit pattern belongs to unless some bits have been set aside specifically for that purpose (as in languages like Lisp and Prologue). In the majority of languages, types are not stored while programmes are being executed; instead, types are used by the compiler to create accurate machine code.

Some Abstraction Levels in a Computer

user

highlevel

language

operating system

device drivers, : : :

machine language

registers & processors

gates

silicon

These are only a few of the levels that a computer could be able to identify.

The majority of large-scale systems have layers inside them. For instance, a management information system might be made up of various database systems that have been more-or-less neatly slapped together.

There are many layers to the communication protocols used on the Internet.

Each layer is responsible for a different duty, such as applying encryption to make unsecured links secure and making unreliable links reliable by retrying transmissions that are not acknowledged. It may sound difficult, but many personal PCs already have the required software.

This course primarily focuses on writing programmes in the high-level language ML.

What is Programming?

- to describe a computation so that it can be done mechanically

  expressions compute values

  commands cause effects
- to do so **efficiently**, in both coding & execution
- to do so CORRECTLY, solving the right problem
- to allow easy modification as needs change

comparison of small- and large-scale programming

Small-scale programming entails creating code to carry out straightforward, well-defined tasks. Expressions for describing mathematical formulas and other things are provided by programmes. (Fortran, the formula translator, made this initial contribution. In a programme, commands specify how control should move from one area to another.

As we write code in the conventional manner, layer by layer, we finally find ourselves programming in-the-large: combining big modules to accomplish a potentially vague purpose. If the components weren't ever meant to function together, it gets difficult.

Programmers require a range of abilities:

- to convey requirements so they can address the appropriate issue
- to apply mathematics to arrive at accurate and straightforward solutions;
- to rationally organise solutions so they can be understood and modified;
- to assess costs and determine in advance whether a given strategy is practical.

All of these topics will be covered in the course, but the programmes will be too straightforward to greatly increase the chance that the requirements will be met incorrectly.

The Role of Floating Point in Computing

Results are ALWAYS wrong—do we know how wrong?

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

Lessons:

- Innovations are frequently mocked as luxury items for slothful individuals.
- their HIDDEN COSTS can be worse than the obvious ones
- luxuries often become necessities

The foundation of numerical calculation is floating-point, which is essential for research and engineering. Read this next: [3, page 97]

Therefore, it would appear to us that it is not at all evident if the slight advantages of a binary point system outweigh the loss of memory space and the heightened complexity of the control and arithmetic circuits.

One of the greatest figures in the early history of computing was Von Neumann.

How could he be so mistaken? The same thing keeps happening:

- • Time-sharing was designed for users who were too slow to stand in line with decks of punched cards and support many interactive sessions, like on Thor.
- • Automatic storage management, sometimes known as rubbish collection, was designed for persons who were too indolent to complete tasks on their own.
- • People who were too indolent to use line-oriented editors should use screen editors.

To be fair, some concepts only gained traction after hardware development brought down their price.

For instance, designing aircraft uses floating-point maths, but would you want to fly in one? Under °oating-point conditions, even proper code can produce wildly wrong results. A hidden cost is that the danger of inaccuracy surpasses the circuit complexity increase.

Fortunately, there are ways to assess how correct our responses are. A skilled coder will employ them.

Why Program in ML?

It is conversational

It has a loose concept of data type.

The underlying hardware is concealed: no accidents

Mathematics makes it simple to understand programmes.

It sets things apart from UPDATE THE STORE.

It oversees our storage for us. Years of study into programming languages have led to ML. It stands out from other languages in that it was defined using an operational semantics, a mathematical formalism that is exact and understandable. Due to the formal specification, there are a number of commercially backed compilers that are readily available, and there are very few incompatibilities between them.

Due to its connection to mathematics, ML programmes may be created and comprehended without having to consider every last aspect of how the computer would execute them. A programme can terminate, but it cannot crash because it is still under the ML system's control. Even yet, it still manages to be respectably efficient and offers lower-level primitives to those that require them. Most other programming languages enable direct access to the underlying system and even attempt to carry out illegitimate activities, which leads to crashes.

Writing and executing programmes is the only method to learn programming.

Install ML on your computer if you have one. I suggest Moscow ML1, which is quick and compact and works on PCs, Macs, and Unix. With the exception of some module-related characteristics that are not covered in this course, it comes with substantial libraries and supports the entire language. Additionally accessible under PWF is Moscow ML.

An option is Cambridge ML. Because of Arthur Norman, it has a Windows-based user interface, but the compiler uses the outdated, sluggish Edinburgh ML. Many of the examples in my book [12] won't function because it supports an outdated version of ML.

A Practical Example: The Area of a Circle: A = ¼r2

```
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
```

This straightforward ML session's first line declares a value. As a result, the moniker pi now represents the actual value 3.14159. These names are known as identifiers. The stated identifier's name (pi) and type (real) are echoed by ML.

The second line computes the area of the circle with radius 1.5 using the formula A = πr^2. We use pi as an abbreviation for 3.14159. Because it is written in between its two operands, the * symbol, also known as an in x operator, is used to represent multiplication.

The computed value (about 7.07) and its type (again, real) are provided by ML in response. The identifier it, which ML provides to allow us to refer to the value of the most recent expression supplied at top level, has officially been declared by us.

So that we don't have to remember the formula, we should offer the service "compute the area of a circle" to enable abstract work. Therefore, the function area is declared on the third line. The function has type real->real; take note that it returns another real number computed using the area formula for every real number r.

The function area is called in the fourth line with 2.0 as the argument. The area of a circle of radius 2 is approximately 12.6. It should be noted that both during declaration and usage, the brackets surrounding a function's parameter are optional.

Pi is used in the function to represent 3.14159. Pi cannot be "assigned to" or changed in any other way, contrary to what you may have seen in other programming languages. Even if we subsequently issue a fresh val declaration for pi, its meaning within area will remain unchanged.

Integers; Multiple Arguments & Results

```
fun toSeconds (mins, secs) = secs + 60*mins;
> val toSeconds = fn : int * int       int ⟶
fun fromSeconds s = (s div 60, s mod 60);
> val fromSeconds = fn : int >
int * int
toSeconds (5,7);
> val it = 307 : int
fromSeconds it;
> val it = (5, 7) : int * int
```

How many seconds are there in m minutes and s seconds given that there are 60 seconds in a minute? Calculation is handled by the function toSeconds. It requires two parameters that are put between brackets.

Right now, we're utilising integers. The real number sixty is represented as 60.0; the integer sixty is represented as 60. The multiplication operator * is overloaded because it is used for both real and int types. Additionally overloaded is the + addition operator. Similar to other programming languages, we can write secs+60 * mins instead of secs+(60*mins) since multiplication (and division) take precedence over addition (and subtraction).

The infix operators div and mod, which express integer division and remainder, are illustrated by the inverse of toSeconds. A pair of results, once more in brackets, are returned by the function fromSeconds.

Pay close attention to the two functions' types:

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

fromSeconds : int -> int * int

They tell us that an ML function may accept any number of parameters and return any number of outputs, maybe of different kinds, and that fromSeconds maps an integer to a pair of integers and toSeconds maps an integer to an integer pair.

Summary of ML's numeric types

int: the integers

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

real: the floatingpoint numbers

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

The symbols val and fun that are italicised are keywords and cannot be used as identifiers. The entire list of ML's keywords is shown here.

abstype as well as and as case datatype end if else Either raise the rec sharing sig or handle the eqtype exception function fun functor if in include infix infixr let local nonfix of op open. struct structure for the signature, followed by type val where while with withtype

Please take note that the negation of x is represented as x rather than -x. In most languages, minus and subtraction are represented by the same symbol, but in ML, all operators—whether found in x or not—are regarded as functions. While minus only requires one integer, subtraction requires two; therefore, these two operations must have different names. In a similar vein, we cannot use +x.

Computer numbers have a finite range, and if it is exceeded, an overflow error results. Integers of any size can be represented by several ML systems.

If reals and integers need to be mixed in a calculation, ML has functions to do so:

real : int -> real convert an integer to the corresponding real

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

Since ML's libraries are segmented into modules, we utilise compound identifiers to refer to library functions, such Math.sqrt. Library units in Moscow ML are loaded using commands like load"Math";. There are many library functions, in addition to the typical numerical ones, including text-processing and operating system routines.

Please refer to a textbook for more information on ML's syntax. You can find mine [12] and Wikstrom's [15] in many college libraries. A glance at Ullman [14] in the library of the Computer Lab is also recommended.

Learning guide. Pages 1–47, notably 17–32, of ML for the Working Programmer contain pertinent information.

Exercise 1.1 Storing years as two digits and interpreting them so that 50 represents 1950 and 49 implies 2049 is one way to fix the year 2000 problem. Comment on the advantages and disadvantages of this strategy.

Exercise 1.2 The code ML functions to (a) compare two years (b) add/subtract some given number of years from another year using the date representation from the previous problem. (You might need to wait until the following class to learn about ML's comparison operators.)

Chapter 2 Recursive Functions

Raising a Number to a Power

```
fun     npower (x,n) : real =
        if   n=0
        then  1.0
        else x * npower(x, n1);
> val   npower = fn : real * int > real
```

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

$x^0 = 1$

$x^{n+1} = x \times x^n$:

The nonnegative integer x is raised to the power n by the function npower. Recursively, the function calls itself. Since the laws outlined above describe exponentiation, this idea should be recognisable to mathematicians. The ML programmer makes extensive use of recursion.

For $n \geq 0$, the equation $x^{n+1} = x \times x^n$ yields an obvious computation:

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

Evidently, the equation is valid even for negative n. The corresponding calculation, however, continues indefinitely:

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

Here's a tedious but necessary digression. The types of arguments and results must always be defined in the majority of languages. When it comes to type inference, ML is uncommon because it typically determines the types on its own. The function npower has a type restriction to state that its outcome is real, but sometimes ML needs a hint. When overloading would otherwise leave a function's type unclear, such restrictions are necessary. In earlier versions, ML either emits an error message or automatically selects the type int.

Despite the greatest efforts of language designers, these problems exist in all programming languages. In this case, type inference and overloading, they are compromises brought about by attempting to achieve the best of both worlds.

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**

The arithmetic operators are frequently overloaded in programming languages. We don't want different operators for various integer types! Some languages only have one type of number that automatically converts between several formats; this is slow and may result in rounding errors that are not expected.

Almost everywhere allows type constraints. On any instance of x in the function, we can place one. The outcome of the function can be limited:

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 acceptable given that the types of x and y match and equality testing is possible for those types.

Note that x <> y is ML for x 6= y.

Conditional Expressions and Type bool

if b then x else y

not(b) **negation of** b

p andalso q ≡ if p then q else false

p orelse q ≡ if p then true else q

A Booleanvalued function!

fun even n = (n mod 2 = 0);

> val even = fn : int > bool

The computer's capacity to detect circumstances and take appropriate action is one of its distinguishing characteristics. In the beginning, a programme might rely on the sign of a number to jump to a specific address. John McCarthy later explained the meaning of the conditional expression to satisfy.

(if true then x else y) = x

(if false then x else y) = y

ML initially assesses B before evaluating the statement if B then E1 else E2.

ML assesses E1 if the result is true and E2 otherwise. The two expressions E1 and E2 are only evaluated in one case! Recursive functions like the one npower above would continue indefinitely if both were evaluated.

An expression of type bool with the values true and false controls the if-expression. Tests have an independent position in modern programming languages rather than being integrated into "conditional branch" constructions.

The relational operators and = can be used to express tests or Boolean expressions. The Boolean operators for negation (not), conjunction (andalso), and disjunction (orelse) can be used to combine them. To verify whether an integer is even, for example, new properties might be introduced as functions.

Note. The second operand of the andalso and orelse operators is only evaluated when necessary. They are not considered to constitute functions: All of the arguments are evaluated using ML functions. Any two-argument function in machine learning (ML) can be converted to an infix operator.

Raising a Number to a Power, Revisited

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

Mathematical Justification:

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

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

$2_{12} = 4^6 = 16^3 = 16$ X $256^1 = 16$ £ 256 = 4096:

We only require two lg n multiplications instead of n, where lg n is the logarithm of n to base 2.

To determine whether the exponent is even, we use the function even, which was previously declared. Divide 2n + 1 by 2, and the result is n since integer division (div) truncates the result to an integer.

Only when a recurrence is guaranteed to end can it be used as a calculation rule.

If n is greater than zero, n is less than both 2n and 2n + 1. The exponent will be decreased to 1 after a sufficient number of recursive calls. If n is less than 0, the equations still hold, but the resulting calculation takes forever.

Our logic expects accuracy in arithmetic; fortunately, using floating-point, the calculation behaves itself.

Expression Evaluation

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

Sample evaluation for power:

power(2; 12) ⇒ power(4; 6)

⇒ power(16; 3)

⇒ 16 X power(256; 1)

⇒ 16 X 256) 4096:

The expression Ei is first reduced to Ei+1 starting with E0 and continues in this manner until a value v is reached. A value resembles a number that cannot be diminished further. To express that E is lowered to E0, we write E⇒ E1. They are mathematically equal: E = E1, but the calculation is always made from E to E0 rather than the other way around. Only expressions and the values they return are subject to evaluation. This interpretation of computation may seem overly constrained. It is undoubtedly distant from computer hardware, but that might work to its advantage. Expression evaluation is perfect for the conventional idea of calculating solutions to issues.

Computers communicate with the outer world as well. They first require a method for receiving problems and offering solutions. Industrial operations are monitored and managed by numerous computer systems. Although now commonplace, this function of computers was not first anticipated. A concept of observable and modifiable states is necessary for modelling it. Finally, we arrive at typical programmes (known to those of you who know C, for example), which are composed of commands. We can update the state by assigning values to variables or conducting input/output.

We continue to programme at the expression level for the time being, which is known as functional programming.

Example: Summing the First n Integers

```
fun nsum  n =
if  n=0  then     0
          else n + nsum (n1);
> val nsum = fn: int ->int
nsum 3 ⇒ 3 + nsum 2
⇒ 3+ (2 + nsum 1)
⇒ 3+ (2 + (1 + nsum 0))
⇒ 3+ (2 + (1 + 0)) ⇒ … ⇒ 6
```

The initial n in the function name "nsum n" refers to how naively it computes the sum of 1 + ¢ ¢ ¢ + n. Parentheses nested within parentheses suggest an actual issue rather than just being a quirk of our notation. A collection of numbers is gathered by the function, but no additions can be made until nsum 0 is reached. The numbers must be kept in the computer's internal data structure, usually the stack, in the meantime. Stack overflow could cause the computation to fail for high n, like nsum 10000.

We are all aware that additions might be made as we go. How do we get the computer to perform that?

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
summing (3; 0) ⇒ summing (2; 3)
                ⇒ summing (1; 5)
                ⇒ summing (0; 6) ) 6
```

A running total is a second input that the function summing accepts. If n is 0, the running total is returned; otherwise, the sum is increased and the process is repeated. The additions are made right away; the recursive calls don't nest.

Iterative or tail-recursive functions are recursive functions whose computation does not nest. (Such calculations approximate those that can be carried out in traditional languages using while-loops.)

By adding an input similar to total, often known as an accumulator, many functions can be made iterative.

Sometimes the efficiency advantage is worthwhile, other times it is not. Due to nesting occurring anytime the exponent is odd, the function power is not iterative. It becomes iterative when a third input is added, however this complicates the algorithm and very slightly improves performance; for 32-bit integers, the maximum amount of nesting is 30 for the exponent 231 -1.

A coding style that uses far more arguments than necessary results from an obsession with tail recursion. Write simple code first, avoiding only blatant inefficiencies. There are tools available for identifying the root reason of the program's slowness. Always keep in mind the phrase "Keep It Simple, Stupid."

I believe you have all realised by now that the arithmetic progression formula may be used to perform the summing much more effectively.

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

Computing Square Roots: NewtonRaphson

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

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

Let's now examine a different kind of algorithm. Finding equations' roots can be done quite effectively using the Newton-Raphson method. Many common functions are computed using it in numerical libraries, and reciprocals are computed using it in hardware.

Using a formula derived from the equation to be solved, compute new values starting with an approximation, x0, and then moving on to x1, x2,... The new approximations will quickly converge to the original estimation if it is sufficiently close to the root.

The square root of an is calculated using the formula above. The computation is shown in the ML session. We get to the square root by x3 with complete machine precision after making the initial assumption that x0 = 1:5. A little more time spent on the session reveals that the convergence has taken place, with x4 = x3:

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

A Square Root Function

```
fun      findRoot (a, x, epsilon) =
let val nextx = (a/x + x) / 2.0
in
        if abs(xnextx) < epsilon*x then nextx
else findRoot (a, nextx, epsilon)
end;
fun sqrt a = findRoot (a, 1.0, 1.0E~10);
> sqrt = fn : real -> real
sqrt 64.0;
> val it = 8.0 : real
```

Using the initial guess x as a starting point, the function findRoot employs Newton-Raphson to calculate the square root of a with a relative accuracy of £. It comes to an end when further approximations fall inside the tolerance 2x, when $|x_i - x_{i+1}| < \epsilon x$.

Fundamentally, this recursive function differs from earlier ones like power and summation. For them, we can predict exactly how many steps they will require and the outcome. It is challenging to calculate the number of steps needed for convergence for findRoot. It may alternate between two approximations that differ somewhat.

See how the following approximation, nextx, is declared. Despite only being computed once, this value is used three times. Let D in E end typically declares the items in D but limits their visibility to the expression E.

(Remember that identifiers declared using the val command cannot be allocated to.)

The sqrt function guesses 1.0 at first. The starting approximation for Newton-Raphson is obtained from a table in a practical application. Only 256 entries would be in the table if it were indexed by, say, eight bits extracted from a. Convergence is guaranteed within a given number of steps, usually two or three, by a solid initial prediction. With no convergence test, the loop converts to straight-line code.

Learning guide. Pages 48–58 of ML for the Working Programmer include pertinent information. The more keen student could be interested in the type checking material (pages 63–67).

Exercise 2.1 Create a power function that is iterative.

Exercise 2.2 Try computing 1=a using the formula xi+1 = xi(2 xia). Without a reasonable initial approximation, it might not even converge.

Exercise 2.3 Has two functions, npower and power, but only one of them actually requires the other. By examining a function's declaration, try to determine which ones do not require a type constraint.

Chapter 3 'O' Notation: Estimating Costs in the Limit

A Silly Square Root Function

```
fun nthApprox (a,x,n) =
        if n=0
then x
else (a / nthApprox(a,x,n1) +
nthApprox(a,x,n1))
/ 2.0;
```

Calls itself 2^n times!

Higher expenses result from larger inputs, but what is the growth rate?

The objective of nthApprox is to compute xn using the Newton-Raphson method from the first approximation x0. formula $x_{i+1} = (a/x_i + x_i)/2$. It is evident that it is inefficient to repeat the recursive call and hence the computation.

Let val... in E end can be used to stop the repetition from happening. Even better would be to use an existing abstraction and call the function nextApprox.

Fast hardware does not eliminate the need for good algorithms. Faster hardware, on the other hand, magnifies the supremacy of stronger algorithms. Usually, we want to be able to manage the greatest inputs. How much more input can our function handle if we buy a machine that is twice as powerful as our current one? We can only go from n to n + 1 using nthApprox.

The function's running time is proportional to 2n, therefore our growth is constrained to this small increment. The function npower, which is described in Lesson 2, allows us to go from n to 2n, or solve issues that are twice as huge. Going from n to n2, with power, we can still do much better.

The term "asymptotic complexity" describes how expenses rise as inputs are increased. Costs often relate to either time or space. Because it requires time to use space, space complexity can never be greater than time complexity. Space complexity frequently pales in comparison to time complexity.

This lecture discusses how to calculate various programme costs and how to estimate them.

It uses the outstanding texts Concrete Mathematics [5] and Introduction to Algorithms [4] as a concise introduction to a challenging topic.

Some Illustrative Figures

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

complexity = milliseconds needed for an input of size n

This table (extracted from Aho et al. [1, page 3]) shows how different time complexity affect the results. How many milliseconds are needed to process an input of size n is shown in the left-hand column. The remaining entries display the biggest n that can be processed in the allotted amount of time (one second, minute, or hour).

The table shows how the size of an input can be processed in relation to time. The amount of the input increases as we increase the computer time per input from one second to one minute, and ultimately to one hour.

Complexities n and n lg n, the top two rows, rise quickly. The bottom two begin somewhat close to one another, but n3 quickly pulls well away from 2n. Even if given enormous resources, an algorithm with exponential complexity cannot handle vast inputs. The complexity might instead take the form nc, where c is a constant. (We ascribe a polynomial level of complexity.) The cost is then increased by a fixed factor when the argument is doubled. Although that is significantly better, the approach might not be thought of as practical if c > 3

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

Comparing Algorithms

Look at the most significant term

Ignore constant factors

- they are seldom significant
- they depend on extraneous details

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

A complex formula is typically used to determine a program's cost. Often, we should only take into account the most important term. Even if 99n is larger for n 99, the n2 term will eventually predominate if the cost is n2 + 99n + 900 for an input of size n. Although the constant term 900 appears large, as n rises, it quickly loses significance.

Cost constants are frequently disregarded. For starters, they rarely matter; in the long run, 100n2 will be preferable to n3. Constant factors only matter if the leading phrases are otherwise the same.

A second issue is that constant factors are rarely accurate. They are dependent on specifics like the software, operating system, and hardware being used. We can compare algorithms in a way that holds true under a variety of conditions by omitting constant components.

In actuality, constant elements occasionally matter. An algorithm's expenses will have a high constant factor if it is overly complex. The theoretically fastest technique for multiplication only surpasses the conventional approach for extremely large values of n.

O **Notation (And Friends)**

f (n) = O(g(n)) provided /f(n)j ≤ c/g(n)/

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

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

f (n) = Ω(g(n)) means g is an **lower** bound on f

f (n) = θ(g(n)) means g gives **exact** bounds on f

Efficiency, or more specifically, asymptotic complexity, is frequently described using the 'Big O' notation. Given that its argument is prone to infinity, it is about the limit of a function. It is an abstraction that satisfies the ad hoc standards we just went over. According to the concept, sufficiently large means that for all n greater than n0, /f(n)/ c/g(n). A finite number of exceptions to the bound are to be disregarded by n0, such as situations in which 99n exceeds n2.

Additionally, the notation disregards constant variables like c. With each f, we might employ a distinct c and n0.

It is misleading to write f(n) = O(g(n)) because this is not an equation.

Please be reasonable. We cannot conclude that f(n) = f0(n) from f(n) = O(n) and f0(n) = O(n).

In terms of g, the expression f(n) = O(g(n)) offers an upper constraint on f. The dual notation f (n) = -(g(n)) () /f(n)j c/g(n))/ can be used to specify a lower bound for all sufficiently big n. The expression f(n) = £(g(n) is used to conjoin the expressions f(n) = O(n) and f(n) = -(g(n)).

People frequently confuse O(g(n)) with £(g(n)) and utilise it as if it provided a tight constraint. Due to the upper bound provided by O(g(n)), if f(n) = O(n), then f(n) = O(n2). Tricky exam questions take use of this reality.

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** $O(\sqrt{n})$

We can readily think about the costs of algorithms using the O notation.

- Constant factors, such as the second in O(2g(n)), disappear: we can use O(g(n)) with a definition that has twice the value of c.
- The base of logarithms doesn't matter since constant factors disappear.
- Substantial terms are dropped. Consider the value of n0 required in the formula f(n) = O(n2 + 50n+ 36) to understand that O(n2 + 50n+ 36) is equivalent to O(n2).

It is simple to verify that using n0 + 25 for n0 and maintaining the same amount of c results in f(n) = O(n2) using the rule (n + k)2 = n2 + 2nk + k2.

If c and d are constants (i.e., unaffected by n), then 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 $O(n^c)$

O(cn) gives a tighter bound than O(dn), therefore to claim that O(cn) is contained in O(dn) means the former. For instance, the reverse is not true if f(n) = O(2n), in which case f(n) = O(3n) trivially.

Common Complexity Classes

O(1) constant

O(log n) logarithmic

O(n) linear

O(n log n) quasilinear

$O(n^2)$ quadratic

$O(n^3)$ cubic

$O(a^n)$ exponential (for fixed a)

Because logarithms grow extremely slowly, O(log n) complexity is ideal. The base of the logarithm is unimportant since O notation disregards constant factors!

We may include O(n log n), which is sometimes referred to as quasi-linear and scales up well for high n, under linear.

Matrix addition is an illustration of quadratic complexity: it takes n2 additions to create the sum of two n£n matrices. The size of matrices that we can multiply in a reasonable amount of time is constrained by the cubic complexity of matrix multiplication. Although there is an O(n2.81) method that is potentially superior, it is too complex to be very useful.

We are constrained to very small values of n by an exponential growth rate of 2n.

The cost already exceeds one million with n = 20. The worst-case scenario might not occur in everyday situations, though. In the worst situation, ML type-checking is exponential, but not for most programmes.

Sample Costs in O Notation

function time space

npower, nsum O(n) O(n)

summing O(n) O(1)

n(n + 1)=2 O(1) O(1)

power O(log n) O(log n)

nthApprox O(2^n) O(n)

Remember from Lesson 2 that although nsum naively computes the sum of 1 +... + n, npower computes xn by repeated multiplication. Each executes O(n) arithmetic operations, as is clear. They utilise O(n) amount of space and are not tail recursive. Because of its repetitive behaviour, the function summing can operate in constant space. It is a variant of nsum with an accumulating parameter. We can avoid having to define the units used to measure space by using O notation.

The units picked can affect the outcome even when constant factors are ignored.

It is possible to think of multiplication as a single unit of cost. The cost of multiplying two n-digit numbers for big n, however, is a crucial issue, particularly given that public-key cryptography employs numbers with hundreds of digits.

There aren't many real-world objects that can be completed or stored indefinitely.

The number of bits needed to store n is O(log n). If a program's cost is O(1), then we have likely assumed that some of the operations it performs are also O(1). This is because we normally don't anticipate ever needing to go beyond the capabilities of the hardware's built-in arithmetic.

It is convenient to write O(log n) for power because the precise number of operations relies on n in a convoluted manner depending on how many odd numbers emerge. The space cost of an accumulating argument could be reduced to O(1).

Solving Simple Recurrence Relations

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

Typical base case: T(1) = 1

Some recurrences:

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

Examine the ML declaration of a function to analyse it. Equations that repeat for the cost function Usually, T(n) can be read o®. We can assign a cost of one unit to the basic scenario because we don't consider constant factors. We can alternatively assign a unit cost to the constant labour performed in the recursive phase; since we only require an upper bound, this unit corresponds to the higher of the two actual costs. If using different constants makes the algebra simpler, we could do so.

Remember our function nsum, for instance:

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

It calls itself recursively with argument n and performs a fixed amount of work (an addition and a subtraction) when given n + 1. We discover that T(0) = 1 and T(n + 1) = T(n) + 1 are recurrence equations. We may simply confirm the closed form by substitution: T(n) = n + 1 is unambiguously the closed form. Costs are linear.

This function performs O(n) work by using nsum with n + 1 as the input. We can assert that this call requires exactly n units by once again neglecting constant factors.

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

The recurrence equations T(0) = 1 and T(n + 1) = T(n) + n are what we arrive at. T(n) = (n 1) +...+1 = n(n 1)=2 = O(n2) is an obvious conclusion. There is a quadratic cost.

The recurrence equation for the function power, which divides its input n into two, is T(n) = T(n=2) + 1. T(n) = O(log n) since T(2n) = n+1.

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

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

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

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

We now examine the nthApprox function that was presented at the beginning of the course.

The term 2T(n) of the recurrence contains representations of the two recursive calls. Regarding the constant e®ort, we can select units such that both constants are one, despite the fact that the recursive situation requires more work than the basic case. (Recall that we are looking for an upper bound rather than the precise cost.)

Let's resolve the recurrence equations for T(n). Knowing the closed form—which in this case is obviously something like 2n—helps.

We obtain 1, 3, 7, 15,.. when evaluating T(n) for n = 0, 1, 2, 3,.. We can simply demonstrate T(n) = 2n+1 - 1 using induction on n. We must examine the default scenario:

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

To replace T(n) with 2n+1 - 1 in the inductive stage, we can assume our equation for T(n + 1). Resting is simple.

We have shown that T(n) = O(2n+1 1), but 2n is also an upper bound, and we are free to choose two as the constant factor. T(n) hence equals O(2n).

The above demonstration is not very formal. The traditional approach to demonstrating that f(n) = O(g(n)) is to use the definition of O notation. However, using the definition of T(n), an inductive demonstration of T(n) c2n encounters problems because this bound is too loose. The proof is allowed to proceed by tightening the bound to T(n) c2n1.

Exercise 3.2 Try the above-mentioned proof. What is it saying regarding c?

An O(n log n) **Recurrence**

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

Proof that $T(n) \leq cn \lg n$ for some constant c and $n \geq 2$:

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

When a function divides its input into two equal parts, performs O(n) work, then executes itself recursively on each, the recurrence equation is created.

Such balancing is advantageous. Instead, dividing the input into unequal chunks of 1 and n 1 size results in the quadratic-complexity recurrence T(n + 1) = T(n) + n.

The outcome of changing the original equations to use the closed form T(n) = cn lg n is displayed on the slide. This is yet another induction-based proof. The final step is valid if c 1.

But something isn't right. Because cn lg n = 0, which is not an upper bound for T(1), if n = 1 then the basic case fails. Although it is simpler to keep in mind that O notation allows us to disregard a finite number of problematic circumstances, we could search for an exact closed form for T(n). By using n = 2 and n = 3 as base cases, n = 1 is completely disregarded. For c 2, the constraints T(2) 2c lg 2 and T(3) 3c lg 3 can be met. T(n) thus equals O(n log n).

In these recurrences, incidentally, n=2 stands for integer division. To be accurate, we should write bn=2c to denote truncation to the next lower integer. B(2n+1)=2c = n gives the odd number divided by two, which is half. For instance, if n is an integer, b2:9c = 2 and bnc = n.

Learning guide. You could have a look at Chapter 2 of Introduction to Algorithms [4] for a more in-depth discussion of complexity.

Exercise 3.3 Determine the recurrence provided by T(1) = 1 and T(n) = 2T(n=2) + 1's upper bound. There should be a tighter bound available than O(n log n).

Exercise 3.4 Prove that the recurrence

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

is O(n log n). Truncation to the next larger integer is indicated by the notation dxe; for instance, d3:1e = 4.

IV Foundations of Computer Science 34

Chapter 4

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
```

Repetition is important in lists since they are an ordered collection of elements. Therefore, [3,5,9] is different from both [5,3,9] and [3,3,5,9].

A list's elements must all be of the same type. A list of numbers and a list of (integer, string) pairs are shown above. Lists of lists are also possible, for example [[3], [], [5, 6]], which has the type int list list.

In the general situation, the list [x1;... ; xn] has type ()list if x1,..., and xn are all of the same type (let's say).

The simplest data structure for handling collections of elements is a list. In conventional languages, arrays are used. The elements of an array are accessible by subscripting; for instance, the expression A[i] returns the array's ith entry. However, arrays should always be avoided in favour of higher-level data structures because subscripting problems are a well-known source of developer annoyance.

Two lists are joined together with the add infix operator (@). Additionally included is rev, which flips a list. These are shown in the earlier session.

The List Primitives

The two kinds of list

nil or [] is the empty list

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

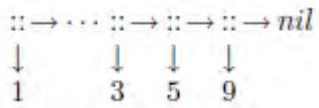
List notation

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

Cons (for "construct") is an operator that adds a new element to the top of an already-existing list. It is important to note that:: is an O(1) operation even though we shouldn't get too caught up in the specifics of implementation.

No matter how long the final list is, it always consumes the same amount of time and space. Internally, lists are represented by a linked structure; when an element is added, it is simply hooked to the front of the existing structure. Furthermore, that structure still represents the same list as it did previously; to see the updated list, one must look at the newly generated new:: node (or cons cell).

The first item on the list is compressed here [3, 5, 9], as follows:



Taking a list's first element (its head) or its tail (its remaining elements) both need a fixed amount of time. Every action just follows a link. The leftmost ↓ arrow in the accompanying figure points to the tail, while the first → arrow points to the head. The head of the original list is the second element once we get the tail, etc.

The list of all elements besides the head is the tail, not the tail itself.

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
```

There are three fundamental ways to inspect lists. Take note of their polymorphic species!

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

There is no head or tail on the empty list. Any operation that is applied to nil results in an error or, strictly speaking, an exception. Before applying hd or tl, the function null can be used to check for an empty list.

Combinations of these functions can be used to explore deeply inside a list, however this format is difficult to read. Fortunately, pattern-matching makes it rarely necessary.

There are two clauses in the aforementioned null declaration: one for empty lists, in which case it returns true, and another for non-empty lists, in which case it returns false.

There is only one sentence, for non-empty lists, in the declaration of HD above.

The function returns x, which is the head, and they have the form x::l. In order to notify us that calling the function can result in the Match exception, which denotes a failure of pattern-matching, ML prints a warning.

Given that tl and hd are similar, the statement of tl is omitted. Instead, a tl application example is provided.

Computing the Length of a List

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

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

Recursion is mostly used in list processing. This is only a simple illustration; patterns can be more intricate.

Note how the clauses of the function are separated by a vertical bar (|).

One function declaration serves as the basis for two scenarios. Consider the following flawed code to comprehend its function:

```
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
```

Not one announcement, but two. We first specify that nlength is a function that only works with empty lists. The function is then re-declared to only handle non-empty lists and never produce a result. We observe that instead of being extended to include new situations, a second fun declaration simply replaces any prior one.

Let's go back to the declaration that was presented on the slide. The length function is polymorphic, meaning that it works with any list and any element type!

Such flexibility is rare in programming languages.

Sadly, this length calculation is inefficient and naive. It is not tail-recursive, unlike nsum in Lecture 2. Where n is the length of the input, it requires O(n) space. As per usual, the answer is to make more and more arguments.

Efficiently Computing the Length of a List

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

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

We can make patterns as complex as we want. The two patterns in this instance are (n,[]) and (n,x::xs).

Addlen is yet another polymorphic function. It refers to the integer accumulator of its type.

We may now declare a length function that is efficient. It serves as nothing more than a wrapper for addlen, providing zero as n's initial value.

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

This version is iterative; the recursive calls do not nest. O(1) space is required.

Since it requires at least n steps to determine the length of an n-element list, it is clear that its time requirement is O(n).

Append: List Concatenation

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

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

Ignoring the specifics of how @ is converted into an infix operator, here is how add might be declared.

Additionally not iterative is this function. It first examines its first argument before setting up a series of 'cons' operations (::) and then performing them.

Given that n is the length of its first argument, it uses O(n) space and time.

Its second argument does not affect its costs.

It may become iterative if an argument keeps building, but it would be quite complicated.

Concatenation necessitates copying every element from the first list, so the iterative version would still take O(n) space and time.

As a result, we are unable to expect asymptotic increases; at best, we can reduce the constant factor involved in O(n), but adding complexity is likely to result in a rise in that factor. By no means add an accumulator out of habit.

Note the polymorphic type of append. If two lists' element types match, they can be merged.

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

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

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

Because add copies its first parameter, it is used improperly and the reverse function is incredibly inefficient. When append is used on a list with length n > 0, it copies the reversed tail if nrev is given a list with length n > 0. Cons are called once more when building the list [x], totaling n calls. It takes n - 1 additional conses to reverse the tail, and so on. There are a total of censes.

0 + 1 + 2 + … + n = n(n + 1)=2:

Therefore, O(n2) is the time complexity. Because the duplicates don't all exist at once, space complexity is merely O(n).

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
```

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

The elements of xs are reversed and prepended to ys when revApp (xs,ys) is called. We can now declare

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

Given an n-element list, it is clear that this reverse function executes simply n conses. We could count the number of conse precisely for both reverse functions, not only up to a fixed factor. O notation is still helpful for describing the entire running time because a cons's time depends on the system.

The function is iterative thanks to the accumulator y. But the loss of append results in a gain in complexity. Reduction in strength is a term used to describe the replacement of one expensive operation (append) with a number of inexpensive actions (cons). It first appeared when many computers lacked a hardware multiply instruction, making it more economical to repeatedly add up the sequence of products i X r for i = 0,..., n. We shall see several instances of eliminating the add as a method of reducing strength.

Constraining an accumulator results in the opposite outcome. The iterative function may be substantially slower than the recursive one if that necessitates the usage of an additional list reversal.

Lists, Strings and Characters

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

Most programming languages offer strings to support text processing.

Numbers must at the very least be transformed into or out of a textual representation.

Users receive text messages from programmes, which then evaluate their responses.

Although strings are necessary in practise, they rarely raise concerns that are pertinent to this course.

When converting between strings and lists of characters, the functions implode and explode. Strings are sometimes just lists of characters in some programming languages, however this is bad design. In and of itself, strings are an ethereal idea.

The result of treating them as lists is awkward, ineffective code.

In a similar vein, characters are not strings of length 1, but rather a basic idea.

In ML, character constants take the form #"c", where c can be any character.

The comma character, for instance, is #",".

Along with the aforementioned operators, strings can also use the relations = > >=, which provide alphabetical order (more accurately, lexicographic order with respect to ASCII character codes).

Learning guide. Pages 69–80 of ML for the Working Programmer include pertinent information.

Exercise 4.1 To calculate the total of a list's elements, write a recursive function. Then create an iterative version and discuss how the efficiency has increased.

Chapter 5 More on Lists

List Utilities: take and drop

Removing the first i elements

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

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

This course analyses further list utilities, demonstrates other recursion patterns, and ends with a brief programme for generating change.

The take and drop functions split a list into sections, returning or removing the first i elements.

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

In next lectures, take and drop will be applied. For recursive processing, they often partition a collection of elements into equal halves.

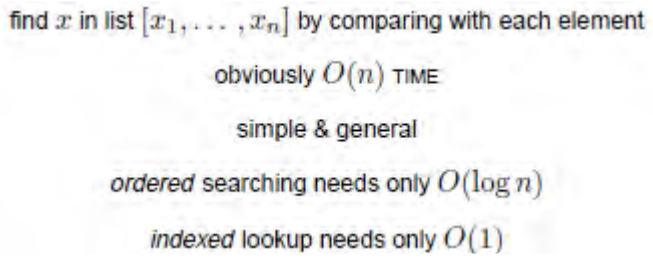
In both functions, the unique pattern variable _ may be found. This pattern's wildcard matches are universal. Although we had the option of writing i in either position, the wildcard serves as a reminder that the relevant clause rejects this claim.

Although making function take iterative wouldn't increase its effectiveness, it already isn't. Up to i list elements must be copied for the work to be completed in O(i) space and time.

Simply put, the function drop skips through i list items. O(i) time is needed, but just constant space. It is considerably quicker than take and iterative. Both operations have an O(i) time complexity, however skipping elements instead of copying them is quicker because drop's constant factor is smaller.

The same type of list is returned by both functions, which take a list and an integer. Therefore, they have the type 'a list * int -> 'a list.

Linear Search



The simplest method of finding a desired item in a collection is a linear search: just go through each item one at a time. If x is in the list, it will typically be located in n=2 steps, making even the worst case clearly O(n).

In order to find objects in large collections of data exponentially faster than O(n) in O(log n) time, items are typically organised or indexed. With the normal caveat that machine constraints are not exceeded, even O(1) is feasible (using a hash table).

Effective indexing techniques are crucial; for example, think of web search engines. However, due to its simplicity and generality, linear search is frequently used to search small collections and serves as the foundation for more sophisticated algorithms.

Types with Equality

Membership test has strange polymorphic type …

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

OK for integers; *NOT OK* for functions

```
fun inter([],    ys) = []
  | inter(x::xs, ys) =
       if member(x,ys) then x::inter(xs, ys)
                       else    inter(xs, ys);
```

To determine whether x appears in l, the function member does a linear search.
The inter function determines the 'intersection' of two lists and returns the list of elements that are shared by both. Member, so called.
ML's interpretation of equality makes it simple to code these functions.
All of the list functions that we have come across so far are polymorphic, which means that they may be used with any type of list. However, not all types are eligible for equality testing. In ML, functions are values, and there is no useful or meaningful way to compare two functions. In machine learning, abstract types can be specified, masking their intrinsic representation and equality test. Later, we'll talk about abstract types and function values.

The majority of commonly encountered varieties have comparable concrete components. Integers, strings, reals, booleans, and tuples/lists of related kinds are examples of equality types.

Polymorphic equality testing is used by a function, as indicated by type variables "a, "b, etc. in the type of the function. Variables of the equality type spread. Despite its indirect application of equality, the intersection function also has these properties:

```
> val inter = fn: ''a list * ''a list -> ''a list
```

ML complains of a type problem when applying member or inter on a list of functions. It performs this at compile time and does not run the offending code; instead, it only looks for flaws in the types.

The trait of equality polymorphism is controversial. Some languages make the concept general. Some academics claim that this complicates machine learning (ML) too much and forces programmers to utilise linear search a lot.

Building a List of Pairs

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

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

Wildcard pattern catches *empty lists*

PATTERNS ARE TRIED IN ORDER

Each xi is connected to a yi via a list of pairs of the form [(x1; y1);... ; (xn; yn)].

A phone book may theoretically be viewed as one of these lists, where xi ranges over names and yi over the appropriate phone number. In such a list, linear search can very slowly identify the yi associated with a given xi or vice versa.

In some instances, the items of another list [z1;...;zn] may have been used to construct the (xi; yi) pairings by applying a function to them.

Lists of pairs are created and broken down by the methods zip and unzip. Zip pairs up corresponding list members, whereas unzip reverses this action. What they do is reflected in their types:

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

Zip discards extra items at the end of the longer list if the lists are not the same length. Only two non-empty lists are compatible with its initial pattern. A wildcard pattern, the second pattern could match anything. The first pattern is tried first since ML tests the clauses in the specified order. Only arguments where at least one of the lists is empty are sent to the second.

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);
```

Unzip must create two lists of results from a list of pairs, which is problematic when using recursion. The version displayed here makes use of the local declaration allow D in E end, where D is a collection of declarations and E is an expression that can make use of them.

Note especially the declaration

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

This links the outcomes of the recursive call to xs and ys. In general, the formula val P = E compares the value of the expression E to the pattern P. All of the variables in P are connected to their respective values.

Here is a version of unzip that substitutes a function (conspair) for the local declaration in the recursive call to split the pair of lists apart. Although it may be clearer and defines the same computation as the previous version of zip, not all local declarations can be removed with the same ease.

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

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

The result of making the function iterative is the straightforward revUnzip mentioned above.

Multiple results can be built simultaneously in various argument positions through iteration.

Reversing the input to revUnzip will rectify the output lists' reverse order construction. Even though iteration has advantages, the overall costs will presumably be higher than those of unzip.

An Application: Making Change

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

- Base case = 0
- Largest coin first
- Greedy algorithm; CAN FAIL!

There are an infinite number of coins in the till. To prevent handing only pennies as change, start with the largest coins. As an example, the list of lawful coin values is stated in descending order as follows: 50, 20, 10, 5, 2, and 1. (Remember that the element that is easiest to access is at the top of a list.) Simple observations serve as the basis for the code for change.

- There are no coins at all in the change for zero. (Note how the first phrase has a pattern of 0.)
- Use the largest coin you can find to get a non-zero amount. Use it if it's tiny enough, and adjust the amount as necessary.
- Discard any coins that are excessively large.

Although no one ever considers changing for zero, this is the most straightforward way to cause the algorithm to crash. If they do nothing in their base case, the majority of repeated processes become the simplest. A beginner programmer's base case is frequently one of one rather than zero.

Either success or failure can cause the function to end. It fails by raising the Match exception. If no pattern is found, the exception is when the till empties while the amount is still nonzero.

Unfortunately, failure is still possible even when change is possible. The opportunistic "largest coin first" strategy is to blame. Assuming we have two coins with values of 5 and 2, the only method to get change for 6 is 6 = 2 + 2 + 2, ignoring the value of 5. Usually, greedy algorithms are effective, but not in this case.

ALL Ways of Making Change

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

Let's generalise the issue to identify every potential course of action and present them as a list of potential answers. Examine the type: the outcome is currently a list of lists.

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

Exceptions will never be raised by the code. If the till is empty and the amount is nonzero, it returns [], expressing failure by returning an empty list of solutions.

There is only one way to make change if the quantity is 0; the outcome should be [[]].

The base scenario for success is this.

There are two options for solving nontrivial problems: either use a coin (if possible) and reduce the amount accordingly, or ignore the present coin value.

To use c, the current coin, the function allc is locally declared. To adjust for amt-c, it adds an extra c to each of the answers the recursive call returned.

A list of coins is designated as cs, whereas a collection of such lists is designated as css. The final's' in the word suggests a plural.

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))
```

Yet another accumulating parameter!

Stepwise refinement

The previous slide's modify function's many:: and add operations are removed with two extra arguments. The first, chg, compiles the coins already selected; one assessment of c::Many of allc's assessments are replaced by chg.

The second, chgs, compiles the list of existing solutions; it does not require append. The speed of this version is significantly faster than the previous one.

Change still happens very slowly for the obvious reason that there are a growing number of solutions in the area being modified. There are 4366 different ways to express 99 using 50, 20, 10, 5, 2, and 1.

Later, we'll go back to the "making change" operation to demonstrate how to handle exceptions.

Our three change functions serve as an example of a fundamental programming method called progressive refining. Write a very basic programme first, then add requirements one by one. Refinements to efficiency should be added last. Writing the simpler programme taught one something about the task, even if it cannot be used in the upcoming version and must be deleted.

Learning guide. Pages 82–107 of ML for the Working Programmer provide related information, however you might wish to skip some of the more challenging cases.

Exercise 5.1 What makes this zip version different from the previous one?

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

Exercise 5.2 What presumptions are made by the'making change' functions regarding the variables until, c, and amt? Explain what may occur if some of these presumptions were not true.

Exercise 5.3 If there are two legal coin values, demonstrate that the number of ways to make change for n (ignoring order) is O(n). What if there were 3, 4, or more coin values?

Chapter 6 Sorting

Sorting: Arranging Items into Order

a few applications:

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

The most extensively researched aspect of algorithm design is probably sorting. Sorting and searching are covered in-depth in volume one of Knuth's The Art of Computer Programming series [8]! Sorting is also covered by Sedgewick [13]. There are innumerable uses for sorting.

A collection can be rapidly searched by sorting the items. Remember that there are O(n) steps needed to search through n objects in linear search. Binary search is possible on sorted collections and takes just O(log n) time. The goal of a binary search is to compare the item being sought with the centre item (in position n=2) and then, depending on the comparison's outcome, to dismiss either the left half or the right half. Instead of lists, binary search requires arrays or trees; we'll talk about binary search trees later.

To create a larger sorted file, two sorted files can be combined fast. Finding duplicates is one of the uses; they are adjacent after sorting.

Names are arranged alphabetically in a phone book. Instead, the same data can be organised by phone number (helpful for law enforcement) or street address (valuable for junk mail companies). Information can be applied in various ways by being sorted in different ways.

Insertion sort, quicksort, mergesort, and heapsort are examples of common sorting algorithms. The first three of these will be discussed. Every algorithm has its benefits. Runtimes for DECstation computers are provided as a specific benchmark for comparison. (These were built using an early RISC design called the MIPS chip.)

How Fast Can We Sort?

typically count comparisons C(n)

there are n! permutations of n elements

each comparison distinguishes two permutations

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

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

The quantity of comparison operations needed is the standard metric for sorting algorithm efficiency. Mergesort sorts an input of n items using only O(n log n) comparisons. It is simple to demonstrate that this complexity is the best one there is. [1, pages 86{7]. Each comparison separates two of the n! permutations that are possible with the given set of n elements. By resolving C(n), the lower bound on the number of comparisons, $2^{C(n)} \geq n!$; therefore $C(n) \geq \log(n!) \approx n \log n - 1.44n$.

We utilise the following source of fictitious random integers [11] to compare the sorting algorithms:

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

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

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

Whatever, just keep in mind that it's challenging to get statistically reliable random numbers. Those few lines of code represent a lot of work.

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(n2) comparisons on average

fun insort [] = []

| insort (x::xs) = ins(x, insort xs);

174 seconds to sort 10,000 random numbers

One by one, elements from the input are copied to the output. The output is always in order since each new item is entered into the appropriate location.

It would serve no reason to write iterative versions of these routines. Insertion sort is slow not because it is recursive but because it does O(n2) comparisons and involves a lot of list copying. With a quadratic runtime of 174 seconds for our sample compared to the next-worst number of 1.4 seconds, it is practically unusable.

Because it is simple to code and exemplifies the ideas, insertion sort merits consideration. Mergesort and heapsort, two effective sorting algorithms, are advancements of insertion sort.

The overloading of the = operator is resolved by the type constraint:real; remember Lect. 2. We'll need a type constraint someplace for each of our sorting functions.

The concept of sorting is dependent on the type of comparison being made, which in turn establishes the sorting function's nature.

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

C. A. R. Hoare, who recently relocated from Oxford to Microsoft Research, Cambridge, devised quicksort. Divide and conquer is a fundamental algorithmic design approach that Quicksort employs. Quicksort selects the pivot value, designated as some value a from the input. It divides the remaining items between those with an a and those without an a. Recursively sorting each component, it places the smaller component before the larger. The most brilliant aspect of Hoare's technique was the ability to perform the partition on the fly by swapping array components. Before recursion was well recognised, quicksort was created, and people found it to be quite challenging to understand.

As per customary, a functional programming-based list version will be taken into consideration.

Quicksort: The Code

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

0.74 seconds to sort 10,000 random numbers

In our ML quicksort, the items are copied. Even though it is slower, it is much simpler to grasp. We sort our list of random numbers, rs, in about 0.74 seconds.

There are three clauses in a function declaration. The first one deals with empty lists, the second one with singleton lists (those with the form [x]), and the third one with lists with two or more members. To increase speed, lists of up to five items are sometimes regarded as special circumstances.

The input is divided using an as the pivot by the locally specified function portion. Items for the left (a) and right (> a) halves of the input are accumulated by the arguments l and r, respectively.

It is simple to demonstrate that, on average, quicksort performs n log n comparisons [1, page 94]. The pivot often has an average value with random data that divides the input into two roughly equal sections. We have the O(n log n) recurrence T(1) = 1 and T(n) = 2T(n=2) + n.

It is approximately 235 times faster than insertion sort in our scenario.

Quicksort's execution time is quadratic in the worst-case scenario! When the input is almost sorted or reverse sorted, for instance. The work is not evenly distributed; almost everything ends up in one partition. There is an O(n2) recurrence between T(1) = 1 and T(n + 1) = T(n) + n. By generating random input, the worst case is extremely unlikely.

Append Free Quicksort

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

0.53 seconds to sort 10,000 random numbers

The quicksort algorithm's combine stage gathers the results from the sorted list. We once more eliminated add using the accepted method. The elements of xs are reversed and prepended to the list sorted when quik(xs,sorted) is called.

If you thoroughly examine the portion, you will see that quik(r,sorted) is executed first. A is after that added to this sorted list. Finally, the elements of l are sorted by calling quik once again.

The acceleration is substantial. A Pascal imperative quicksort (adapted from Sedgewick [13]) is just marginally quicker than function quik. The near-agreement is unexpected given that lists have higher computing costs than arrays. In practical implementations, comparisons are the primary expense and overheads are even less significant.

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

A larger sorted list is created by merging two smaller sorted lists. It does up to m+n comparisons, where m and n are the input list lengths. If n = 1 then merging degenerates to insertion, doing significant work for little gain. If m and n are nearly equal, then we have a quick approach to build sorted lists.

Many sorting algorithms are built on merging; we examine a divide-and-conquer technique. Because arrays are challenging to programme for, mergesort is rarely used in traditional programming; nevertheless, it works well with lists. If the input is non-trivial, it splits it into two nearly equal pieces before sorting them again and merging them. Function merge uses a deep recursion rather than iteration. For the same reasons that apply to add, an iterative version is of little use (Lec. 4).

Topdown 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;

Worst case scenario: O(n log n) comparisons

Sorting 10,000 random numbers takes 1.4 seconds. Instead of selecting a pivot (like in quicksort), the mergesort split stage divides the input by simply counting out half of the elements. Recursive calls are used in the conquer stage once more, and merging is used in the combine stage.

The list rs is sorted by the function tmergesort in about 1.4 seconds.

With the same recurrence equation as in the average case of quicksort, mergesort performs O(n log n) comparisons in the worst situation. We always have T(n) = 2T(n=2) + n since take and drop divide the input into two equal portions (they differ by no more than one element).

In the case, Quicksort is approximately three times as quick. But the worst-case scenario is quadratic! Although sluggish, merge sort is safe. Which algorithm is the best then?

Top-down mergesort has been demonstrated. There are also bottom-up algorithms.

They begin with a set of lists with a single element and combine adjacent lists until there is only one left. A re-nement begins with a list of increasing or decreasing runs of input items and takes advantage of any original order within the input.

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

The worst scenario of Quicksort cannot be disregarded. A complexity of O(n2) is disastrous for large n. Mergesort is slower than quicksort for random data, but it has an O(n log n) worst case running time, which is ideal.

It is important to mention non-comparison sorting. Using their radix form, we can sort a huge number of tiny numbers in O(n) time. Since no comparisons are made, this result does not refute the comparison-counting hypothesis. Only if the largest integer is predetermined in advance can linear time be achieved; as n approaches infinity, an increasing percentage of the items are duplicates. Simply said, it is a specific case. There are numerous different sorting algorithms. The issue of sorting enormous volumes of data using external storage medium like magnetic tape has not been discussed.

Learning guide. Pages 108–113 of ML for the Working Programmer include pertinent information.

Chapter 7 Datatypes and Trees

An Enumeration Type

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

Our ML session gains a new type as a result of the datatype declaration. Type vehicle admits pattern matching and performs as well as any built-in type.

Constructors are the name given to the four new vehicle type designations.

The different types of cars might be represented by the numbers 0 to 3 and matched using pattern matching as in the wheels example above. The code would be challenging to read and even more challenging to maintain, though. Think about including the tricycle as a new vehicle. All the numbers would need to be adjusted if we were to include it before Bike. Such modifications are simple to make with datatype, and the compiler can (at least occasionally) alert us when it comes across a function definition without a Tricycle case.

It's likewise bad practise to represent automobiles using strings like "Bike," "Car," etc. It takes a while to compare string values, and the compiler can't forewarn us that misspellings like "MOtorbike" will cause our code to fail.

The declaration of types like vehicles is supported by the majority of computer languages.

They are known as enumeration types because they are made up of a list of identifiers. The days of the week or colours are two more frequent examples. Since the compiler

selects the integers for us, we won't accidentally confuse Bike with Red or Sunday thanks to type-checking.

Note that not all misspelt constructors are caught by ML [12, page 131].

A NonRecursive Datatype

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

- Distinct constructors make distinct values
- Can put different kinds of vehicle into one list:

[Bike, Car true, Motorbike 450];

Enumeration types are generalised in ML to enable data to be connected to each constructor. The other three constructors are functions for generating vehicles, while the constructor Bike is a vehicle unto itself.

Complex declarations should have comments because it may be difficult for us to recall the functions of the different int and bool components.

Comments are encapsulated in brackets in ML (* and *). Programmers should use comments in their code to describe design choices and important aspects of algorithms (sometimes by referencing 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*)
```

A bicycle, a Reliant Robin, and a sizable motorcycle are represented in the list on the slide. It resembles a mixed-type list since it contains both numbers and booleans. Datatypes mitigate the effects of the requirement that all list components have the same type, which is essentially a list of vehicles.

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

The cases involving the motorbike and the truck do not overlap. Although both Motorbike and Lorry are in possession of an integer, ML considers the constructor. Any Lorry is different from a Motorcycle.

One notion that has many different variations and distinct properties is vehicles. Similar to how datatypes are used to represent such things in most programming languages. (These records, whose tag elds serve as the constructors, are also referred to as union types or variation records.)

Nested Pattern Matching

```
fun greener (_, Bike) = false
        | greener (Bike, _) = true
        | greener (_, Motorbike _) = false
        | greener (Motorbike _, _) = true
        | greener (_, Car _) = false
        | greener (Car _, _) = true
        | greener (_, Lorry _) = false;
        note wildcards in patterns
        order of evaluation is crucial!
```

Here is a function to compare vehicles for environmental friendliness as another pattern-matching example. A bike is more environmentally friendly than a motorbike, which is greener than a car, which is greener than a truck or trailer. The code displayed above is intricate and significantly depends on ML's evaluation order. Nothing is more environmentally friendly than a bike, a bike is more environmentally friendly than everything else, a motorbike is more environmentally friendly than anything else, etc. The code is challenging to read because the trues and falses alternate.

List all of the valid situations for clarity, and let one wildcard catch all of the invalid cases. But what would happen if we added a couple more constructors to this style?

```
fun greener (Bike, Motorbike _) = true
        | greener (Bike, Car _) = true
        | greener (Bike, Lorry _) = true
        | greener (Motorbike _, Car _) = true
        | greener (Motorbike _, Lorry _) = true
        | greener (Car _, Lorry _) = true
        | greener _ = false;
```

True equations do not contain clauses that depend on the method of evaluation. The best method for comparing multiple constructors is to convert them both to integers and then compare those (using).

The lesson to be learned from this example is that patterns can incorporate tuples, lists, numbers, texts, and datatype constructors. The size of patterns and the quantity of clauses in a function declaration are both unlimited.

The majority of ML systems are effective at matching patterns.

Error Handling: Exceptions

exception Failure;	**Declaring**
exception NoChange of int;	
raise Failure	**Raising**
raise (NoChange n)	
E handle P1 => E1 | … | Pn => En	**Handling**

INELEGANT? ESSENTIAL!

Because it is not always feasible to predict in advance whether or not a search will result in a dead end or whether a computation will result in mistakes like overflow or division by zero, exceptions are required.

Raise E has the effect of continuously aborting function calls until it finds a handler that matches E. In contrast to how ML correlates occurrences of identifiers with their matching declarations, which does not necessitate running the programme, the matching handler can only be determined dynamically (during execution).

A function's type does not always imply whether it can raise an exception, which is one critique of ML's exceptions. Alternatively, it can return a value for the datatype.

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

NONE denotes an error, whereas SOME returns the appropriate solution. Although this strategy appears straightforward, it has the disadvantage of requiring numerous checks for NONE throughout the code.

ML types of exceptions are exn. It is a unique datatype with added exception declarations in its constructors. Constructor names are exceptions.

Making Change with Exceptions

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

We discussed the issue of change in Lesson 5. Since the greedy algorithm there always took the biggest coin, it was unable to express 6 using only 5 and 2. We only require one answer, so returning the list of all alternatives avoids that issue fairly expensively.

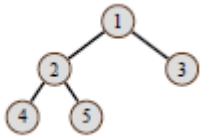
We can programme a backtracking algorithm that can reverse previous choices if it encounters a dead end by using exceptions. If we run out of coins (with a non-zero value) or if the amount decreases, the exception Change is raised. The largest coin is always tried, but we surround the recursive function in an exception handler, which reverses the decision if it is incorrect.

Pay close attention to how recursion and exceptions interact. The exception handler always reverses the most recent action, perhaps leaving other actions to be reversed at a later time. If making a change is truly difficult, exception Change will eventually be raised without a handler to catch it and will be notified at the highest level.

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 tree is a type of data structure that has numerous branches. Trees can stand in for mathematical expressions, logical equations, computer programmes, English sentence structure, etc.

Nearly as fundamental as lists are binary trees. They can offer effective information retrieval and storage. Each node in a binary tree is either empty (Lf) or a branch with a label and two subtrees (Br).

Datatype could be used to declare lists themselves:

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

Even as an infix constructor, we could use::. The [:::] notation, which is a component of the ML language, is the only thing we were unable to define.

Basic Properties of Binary Trees

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

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

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

Pattern-matching is used to represent functions on trees in a recursive manner.

The two aforementioned functions are comparable to length on lists. Here is a third way to gauge how big a tree is:

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

This function is unnecessary because, for any tree t, leaves(t) = count(t)+1, which is a fundamental truth about trees that can be demonstrated using induction. The inequality depicted on the slide also has a simple inductive proof.

A 20-depth tree has a storage capacity of 220-1, or around one million components. These elements have short access pathways, especially when compared to lists with millions of items!

Traversing Trees (3 Methods)

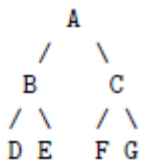
```
fun preorder Lf           = []
  | preorder(Br(v,t1,t2)) =
      [v] @ preorder t1 @ preorder t2;

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];
```

A tree is traversed by looking at each node in a certain order. Preorder, inorder, and postorder are the three types of tree traversal identified by D. E. Knuth [9]. These 'visiting orders' can be codified as functions that turn trees into lists of labels. The functions above only copy the nodes into lists; nevertheless, algorithms based on these principles often carry out some operation at each node.

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

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));
```

Since the appends in the recursive calls are inefficient, the functions depicted on the preceding slide are, regrettably, quadratic in the worst scenario. We (as per usual) add an accumulating argument to address that issue. Compare how appends were removed from quicksort in Lect with how each function generates its result list.

Each of these functions has an equation that links it to the corresponding function on the preceding slide. For instance, inord(t; vs) = inorder(t)@vs

Learning guide. Pages 123–147 of ML for the Working Programmer include pertinent information.

Exercise 7.1 Demonstrate that, in the worst scenario, the time requirements for the functions preorder, inorder, and postorder are all O(n2), where n is the size of the tree.

Exercise 7.2 Demonstrate that the time required by the functions preord, inord, and postord is linear in the size of the tree.

Chapter 8 Dictionaries and Functional Arrays

Dictionaries

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

Abstract type: conceal implementation while supporting operations!

A dictionary is a type of data structure that links particular keys to values. It is crucial to identify the entire set of operations that need to be supported when deciding on the internal representation for a data structure. Rarely is one representation best for all potential uses of a data structure; each will support different operations with varying degrees of success.

We take into consideration simple dictionaries that just allow for update (linking a value to an identifier) and lookup (finding the value linked to an identifier). The actions of delete (removing an association) and merge (combining two dictionaries) may also be taken into consideration. We are using a functional style of programming, thus update won't change the data structure. Instead, a changed data structure will be returned. We can accomplish this effectively if we take care to limit our copying.

Unless the desired key is found, lookup and delete are unsuccessful. ML's exceptions are used by us to indicate failure.

Abstract types can be declared in modern programming languages so that well-defined methods can be exported while low-level implementation details, such as the data structure used to represent dictionaries, are concealed. For this reason, ML offers modules, although this course does not cover them. (Modularity is covered in the Java course.) As a result, we will just define each dictionary operation at the top level. For instance, there will be numerous versions of lookup that should be packed in different modules to avoid conflicts.

Association Lists: Lists of Pairs

exception Missing;

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

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

LINEAR SEARCH IS SLOW A LISTS CAN GET VERY LONG!

The simplest straightforward way to represent a dictionary is as a list of pairs. Lookup is done by a linear search, which is O(n) inherently slow.

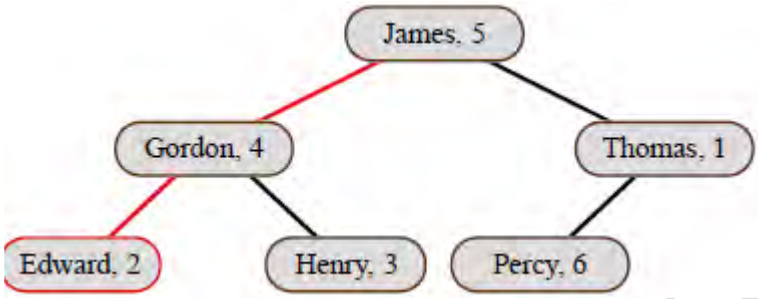
Association lists are only useful if there are few, prominent interest-keys. However, remember lookup's association lists type is applicable to all equality types. Their key benefit is their generality.

Put a new pair into the list to add a new (key, value) association. The best we might aspire for is that this takes constant time. But a lot of area is needed. Because outdated entries are never destroyed, it is linear in the number of updates rather than the number of unique keys. The update time would increase from O(1) to O(n) if old entries needed to be found before they could be deleted.

A analogous function in the language Lisp, which was historically important in the creation of the Lisp evaluator, inspired the traditional name for function lookup, assoc.

Binary Search Trees

A Dictionary: Associates values with keys



An essential use of binary trees is in binary search trees. They function for keys like strings that have a complete ordering. Each branch of the tree has a pair of keys and values; the left subtree of each branch has smaller keys, while the right subtree holds larger keys. Update and lookup for a tree of size n both take O(log n) if the tree remains adequately balanced. Given random data, the tree is expected to remain balanced at these moments in the typical scenario.

All keys in the left subtree are smaller (or equal) at a particular node, whereas all trees in the right subtree are larger.

In the worst situation, an unbalanced tree has a linear access time. Examples include creating trees by repeatedly inserting elements in ascending or descending order; quicksort is closely related to this method. The sorting method known as treesort is created by first creating a binary search tree and then converting it to inorder.

In the worst scenario, self-balancing trees like Red-Black trees can achieve O(log n). The implementation of these is challenging.

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
```

O(log n) access time —if balanced!

If the required key is less than the present key or higher than it, lookup in the binary search tree moves to the left subtree, and vice versa. If it comes across an empty tree, it raises the exception Missing.

We need to specify the functions for a certain type, in this case string, because an ordering is required. The exception Missing now makes reference to that type: if a lookup is unsuccessful, it returns the missing key. By utilising the constructor NONE for failure and the type choice of Lesson 7, the exception might be avoided.

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);
```

copies the path to the new node!

An excellent example of functional programming is the update procedure. Similar to a lookup, it searches but uses recursive calls to build a new tree around the update's outcome. While the other subtree remains unaltered, one has been updated. The internal representation of trees makes ensuring that sections of the tree that have not changed are shared rather than copied. (Lect. 15 will cover how to build connected structures using references.) As a result, updating just copies the new node's path from the root. For a well balanced tree, its time and space requirements are both O(log n).

Three situations are possible based on the comparison of b and a:

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

The comment (*a=b*) in the function definition should be noted. In ML, comments are encased in brackets (* and *).

Arrays

Conventional Array = indexed storage area

- updated in place: A[k] := x
- inherently imperative: requires actions

Functional Array = finite mapping on integers

- updated by copying: update(A, k, x)
- new mapping equals A except at k

Can we do this efficiently?

Only by counting from the front can one get to the elements of a list.

A path from the root leads to various parts of a tree. Such structural details are concealed by an array, whose components are all uniformly identified by numbers. Random access is instantaneous access to any area of a data structure.

The main data structure used in traditional programming languages is an array. Many of the famous classical algorithms, including Warshall's transitive-closure algorithm and Hoare's original quicksort (the partition phase), rely on the clever use of arrays.

The disadvantage is that subscripting frequently results in programmer error.

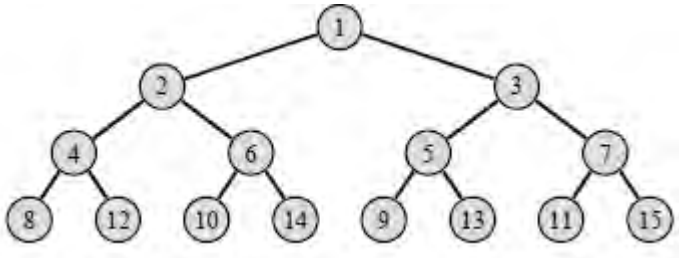
Due to this, arrays are not heavily utilised in this beginner course.

Below is a description of functional arrays that shows another approach to organise data using trees. A list of our dictionary data structures, listed in diminishing generality and increasing efficiency, is provided below:

- Linear search: the most general, just requiring equality on keys, but linear time inefficient.
- Binary search: Requires a key ordering. In the best situation, access times are linear; in the average case, they are logarithmic.
- Array subscripting is the least general and requires integer keys, although even in the worst-case scenario, time is logarithmic.

Functional Arrays as Binary Trees

Path follows binary code for subscript



This straightforward illustration, credited to W. Braun, ensures the equilibrium of the tree. Access complexity is always O(log n), which is the ideal value. Access to conventional arrays is substantially quicker in terms of real running time because it just requires a few hardware instructions. It is common to assume that array access is O(1), which (as always) assumes that hardware limitations are never reached.

Indexes in an array have a lower bound of one. The upper bound has an unlimited growth potential and starts at 0, which represents an empty array. With the addition and deletion of components at each end, this data structure can be utilised to construct arrays that expand and contract.

Lookup

exception Subscript;

```
fun sub (Lf, _) = raise Subscript
  | 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 subscript is divided by two by the lookup function sub until 1 is reached.

The function follows the left subtree if the remainder is 0, otherwise the right. When it encounters a leaf, it raises an exception to indicate a problem. Subscript.

The binary code of the subscript serves as a useful metaphor for understanding array access. In binary, the subscript has a leading one since it must be a positive integer. Reverse the remaining bits bits after discarding this one. The route from the root to the subscript can be found by considering zero as left and one as right.

The significance of binary is frequently explained in popular literature as being driven by hardware: since a circuit is either on or off. The contrary is almost entirely true. Digital electronics designers take great pains to suppress the continuous behaviour that would otherwise develop. The use of binary in algorithms—where an if-then-else decision results in binary branching—is what gives binary its actual significance.

Binary branching is used in data structures like trees and algorithms like mergesort to lower cost from O(n) to O(log n). The smallest integer divisor that may accomplish this reduction is two. (Larger divisors are only infrequently useful because they lower the constant factor, like in the case of B-trees.) Another example of how algorithms based on binary choices are simple is the ease with which binary arithmetic is simpler than decimal arithmetic.

Update

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 subscript is continually divided by two by the update function. It knows the element position when it reaches a value of one.

Then another branch with the new label replaces the branch node.

As long as no new nodes need to be formed in between, a branch can replace a leaf and increase the array's size. For arrays without gaps in their subscripting, this is sufficient.

The data structure can be changed to support sparse arrays, in which the majority of subscript places are undefined. Exception Subscript means that the slot cannot be created and does not exist.

NONE and SOME cannot simply replace this use of exceptions.

Keep in mind that k = 1 is used in two different experiments. If a leaf has been reached, it returns a branch and increases the array by one. The result is an updated array element if we are still at a branch node.

An array can be shrunk by one using a similar technique.

Learning guide. Pages 148–159 of ML for the Working Programmer include pertinent information.

Chapter 9 Queues and Search Strategies

BreadthFirst v DepthFirst Tree Traversal

binary trees as decision trees

Look for solution nodes

- Depthfirst : search one subtree in full before moving on
- Breadthfirst: search all nodes at level k before moving to k + 1

Finds all solutions — nearest first!

All three types of tree traversals—preorder, inorder, and postorder—are depth first. The left subtree is fully traversed before the right subtree at each node. Although depth-first traversals are simple to code and occasionally efficient, they are not appropriate for all problems.

Assume that the tree represents the possible solutions to a puzzle, and that the goal of the traverse is to find a node that contains the answer. When another solution is at the very top of the right subtree, a depth-first traverse may find one solution node far down in the left subtree. We frequently seek the quickest route to a resolution.

Assume the tree is endless. (The ML datatype tree only contains finite trees, but ML is also capable of representing infinite trees using the techniques covered in Lesson 13). With infinite trees, depth-first search is practically worthless because it will never reach the right subtree if the left subtree is endless.

Instead of exploring the nodes vertically, a breadth-first traversal does so.

When visiting a node, it first visits all other nodes at the current depth before traversing the subtrees. Keeping a list of trees to visit makes it simple to put this into practise. This list initially includes the complete tree. A tree is taken out of the list's head and its subtrees are added to the list's tail at each step.

BreadthFirst Tree Traversal — Using Append

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

Keeps an enormous queue of nodes of search

Wasteful use of append

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

This simplistic version of breadth-first search is particularly inefficient. Following the subtrees (all at depth d + 1) of the trees that have already been visited, the list of trees when the search is at depth d of the tree contains all the remaining trees at depth d. The list may already have 1024 elements at depth 10. It takes up a lot of room and makes matters worse by egregiously abusing the word "append." Just two components are included into the lengthy list ts after evaluating ts@[t,u].

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

If queues are used in place of lists for breadth-first search, the process is substantially faster.

A queue depicts a sequence and enables elements to be added to the tail from the head. Due to the First-In-First-Out (FIFO) discipline, the item that has been in the queue the longest will be the next to be eliminated. Although lists can construct queues, appending elements to the tail is a subpar solution.

If we add a method to delete the first array element, our functional arrays from Lesson 8 will work. (See page 156 of ML for the Working Programmer.) For a queue of length n, each operation would need O(log n) time.

We'll outline an effective, strictly functional, and list-based representation of queues. When amortised, or averaged over a queue's lifespan, operations require O(1) time.

A queue is typically represented by an array in programming. The front and back of the queue, which may wrap around the end of the array, are indicated by two indices. The coding is a little challenging. Even worse, an upper bound must be placed on the queue's length.

IX Computer Science Foundations 85

Efficient Functional Queues: Idea

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

by any **pair of lists**

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

Add new items to rear list

Remove items from front list ; if empty move rear to front

Amortized time per operation is O(1)

Both ends of a queue need to have efficient access: the front for removal and the back for input. Access should ideally be constant time, O(1).

It might seem that lists are unable to offer this access. This operation will take O(n) if enq(q,x) executes q@[x]. We might implement enq(q,x) by x::q and represent queues as reversed lists, but the deq and qhd operations would be O(n). A series of n queue operations might then take O(n2) time, making linear time intolerable.

The solution is to represent a queue by a pair of lists, where

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

represents the queue $x_1 x_2 \ldots x_n y_n \ldots y_1$.

The queue's front portion is stored in chronological order, and its back portion is stored in the opposite manner. Due to the fact that this list is inverted, the enq operation adds elements to the rear section using cons; as a result, enq requires constant time. Since this list is stored in order, the deq and qhd operations just examine the front portion, which typically takes constant time. Deq can occasionally remove the final component from the front section, in which case the rear component gets switched around to become the new front component.

The cost per operation throughout the course of any full execution, on average, is referred to as amortised time. The research below shows that the average cost per operation remains constant, even for the worst-case scenario of execution.

Efficient Functional Queues: Code

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))
```

Queues' datatype avoids misunderstanding with other lists of pairs.

To conserve space on the slide, the empty queue's two halves were left out.

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

By converting a queue to its normal form, the norm function makes sure that the front portion is never empty unless the queue as a whole is empty. Deq and Enq functions call norm to normalise their output.

as queues are in regular form, qhd (also deleted from the PowerPoint) looks at their front part as that is where their head will undoubtedly be.

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

Let's examine the cost of an execution that includes n enq operations and n deq operations (in any possible order), beginning with an empty queue. Each enq operation will add a component to the back section by performing one cons.

Each component of the rear section is moved to the front part since the ultimate queue must be empty. The associated reversals execute one disadvantage for each element. As a result, 2n cons operations, or an average of 2 per operation, make up the overall cost of the sequence of queue operations. O(1) is the amortised time.

There's a problem. Reversing a lengthy list could require up to n 1 cons; they do not all have to be distributed equally. The method is inappropriate for real-time programming, where deadlines must be met, due to unpredictable delays.

Aside: The case Expression

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

The case expression has the form

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

One after the other, it attempts the patterns. It evaluates the matching expression when one matches. It acts exactly like the function declaration's body. As demonstrated above, we could have defined function wheels (from Lec. 7).

A program phrase of the form $P_1 =¿ E_1 — \cdots — P_n =¿ E_n$ is called a Match. In addition, a match may occur with the fn-notation to expression functions immediately (Lecture 10) and after the exception handler (Lecture 7).

BreadthFir Tree Traversal — Using Queues

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

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

200 times faster!

This function uses a different data structure to implement the same method as nbreadth. It uses type queue to represent queues rather than type list.

I used both functions on the complete binary tree of depth 12, which has 4095 labels, to compare how effective they were. While breadth only took 0.15 seconds, the method nbreadth took 30 seconds, 200 times faster.

The speedup would be greater for larger trees. Making the appropriate data structure choice is quite profitable.

Iterative deepening: Another Exhaustive Search

Breadthfirst search examines O(bd) nodes:

$$1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} \qquad b = \text{branching factor} \qquad d = \text{depth}$$

Recompute nodes at depth d instead of storing them

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

Space required at depth d drops from bd to d

For complex issues, breadth-first search is impractical since it takes up too much room. In the case of binary trees, b = 2, making the slightly more general problem of locating trees with b branches. Then, ignoring the constant factor of b=(b 1), a breadth-first search to depth d investigates (bd+1 1)=(b 1) nodes, which is O(bd).) The space and time requirements are both O(bd) because all investigated nodes are also saved.

The 'nearest-first' characteristic of breadth-first search is combined with the space efficiency of depth-first iterative deepening. With rising depth bounds, it repeatedly does depth first searches, tossing the results of the previous search after each execution. In order to locate a solution, it searches to depth 1, then depth 2, and so on. As a result of the exponential growth in the number of nodes, we can afford to throw off earlier findings. Level d + 1 has bd+1 nodes; if b 2, this number actually exceeds the sum of all nodes at all preceding levels, i.e. (bd+1 1)=(b 1).

Korf [10] demonstrates that if b > 1, the time required for breadth-first search to reach depth d is only b=(b 1) times that of iterative deepening. Both algorithms have the same time complexity, O(bd), making this a constant factor.

The additional factor of b=(b 1) is relatively reasonable in normal instances where b 2. With iterative deepening, the space need drops exponentially from O(bd) for breadth-first to O(d).

Another Abstract Data Type: Stacks

- empty is the empty stack
- null tests whether a stack is empty
- top returns the element at the top of a stack
- pop discards the element at the top of a stack
- push adds an element at the top of a stack

A stack is a series where items can only be added to or taken out of the head. The item that has been in the queue for the shortest amount of time is the one that will be removed next from a stack according to the Last-In-First-Out (LIFO) discipline. Given that the head is impacted by both cons and hd, lists can easily create stacks. Nevertheless, stacks are sometimes seen as an imperative data structure rather than a list because push or pop only modifies an existing stack rather than creating a new one.

A stack is frequently implemented in traditional programming languages by storing the elements in an array and counting them using a variable called the stack pointer. Most language processors use an internal stack to keep track of recursive function calls.

A Survey of Search Methods

1. **Depthfirst**: use a stack (efficient but incomplete)
2. **Breadthfirst**: use a queue (uses too much space!)
3. **Iterative deepening**: use (1) to get benefits of (2) (trades time for space)
4. **Bestfirst**: use a priority queue (heuristic search)

The data structure determines the search!

The data structure used to store pending subtrees can be used to categorise search operations. In functions like inorder, a stack is implicitly used by depth-first search but can be made explicit. Such nodes are kept in a queue via breadth-first search.

Keeping the nodes in an ordered sequence known as a priority queue is a significant variation. The nodes in the priority queue are subjected to some type of ranking algorithm, with higher-ranked nodes being prioritised over lower-ranked ones.

The ranking function frequently calculates the separation between a node and a solution. If the estimate is accurate, the answer is found quickly. Best-first search is the name of this approach.

Although it is slow, the priority queue can be maintained as a sorted list.

On the whole, binary search trees would be superior, and more complex data structures would further improve things.

Learning guide. Pages 258–263 of ML for the Working Programmer include pertinent information. 159–164 discuss priority queues.

Chapter 10 Functions as Values

Functions as Values

Functions can be

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

functions represent algorithms and infinite data structures

What abstractions a programming language allows can be used to gauge its progress. Examples include parametric types like ® list and conditional expressions, which are descended from conditional leaps based on the sign of some numeric variable. Although the notion of using functions as values in a calculation first came up, it took some time before the concept was fully realised. Few programming languages go to the trouble of allowing functions to be returned as results, although many computer languages enable functions to be supplied as arguments to other functions.

A functional or higher-order function in mathematics is a function that modifies other functions. Numerous mathematical concepts, such as the calculus's integral and differential operators, are functionals. A function is often an infinite, uncomputable object to a mathematician. In order to represent algorithms, we employ ML functions. They can occasionally be limitless collections of data determined by computing rules. The equality of functions cannot be compared. Testing the identity of machine addresses would be the most we could hope for in terms of efficiency. Due to the fact that they would be compiled to different machine addresses, two distinct instances of the same function declaration would be viewed as unequal.

In a language like ML, such a low-level feature has no place.

Functions Without Names

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

The function (fn n => n*2) is a *doubling function*

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

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

We need a notation for functions if they are to be regarded as computational values. Without naming the function, the fn-notation expresses a function value. (Because it originates in the -calculus, some people pronounce fn as 'lambda'.) It is unable to convey recurrence. Its primary role is to compile brief expressions that must be used repeatedly with another function.

The result of the expression (fn n => n*2) is the same as the double-declared identifier:

fun double n = n * 2

Similar to case expressions and exception handlers, the fn-notation enables pattern-matching to express functions with many clauses:

fn P_1 => E_1 | ... | P_n => E_n

This uncommon phrase shortens the local proclamation.

let fun $f(P_1)$ = E_1 | ... | $f(P_n)$ = E_n
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

returning another function as its result

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

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

promote "Mop";
> "Professor Mop" : string
```

When there are multiple variables, like in (n*2+k), how do we package n*2 as the function (fn n => n*2)? In the event that the variable k is defined in the current situation, then

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

is still relevant. We can utilise patternmatching on pairs, defining a function with two arguments.

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

Nesting the fn-notation is a more intriguing option:

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

This function produces another function when applied to the argument 1,

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

It produces the number 7 when applied to the number 3.

Similar examples may be found on the slide, although they use the expression ab instead, where the symbol for string concatenation is. The function promote prefixes any string to which it is applied with the title "Professor," binding the first parameter of prefix to the string "Professor" in the process.

Note: That in (fn a => (fn b => E) the brackets are optional.

Syntax for Curried Functions

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

prefix "Doctor " "Who";
> val "Doctor Who" : string

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

Allows partial application

Conveniently, the n-argument curried function f is declared using the syntax

fun f x_1 ... x_n = ...

and applied using the syntax f E_1… E_n.

We now have two methods for describing functions with many arguments: pairs and currying. Currying allows for partial application, which is advantageous when changing the first parameter results in a function that is valuable on its own. The definite integral is a mathematical concept where setting x = x0 results in a function that only affects y. Although the function hd (which returns the head of a list) isn't curried, it can be used in some expressions that require the curried application syntax:

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

The function (dub) that results from applying hd to a list of functions is then applied to the string "Hamilton." In object-oriented programming, which is used in languages like Java and C++, the concept of running code stored in data structures is fully developed.

A Curried Insertion Sort

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

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

Lect. 6's sorting algorithms are programmed to sort real values. By passing the ordering predicate () as an input, they can be generalised to any ordered type.

Locally declared functions ins and sort make use of lessequal.

It might not be immediately apparent, but insort is a curried function. It returns the function sort for sorting lists of that kind of items given as its first argument a predicate for comparing a certain type of item.

Examples of Generic Sorting

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

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

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

Note: Op= refers to the equals sign when used as a value. Despite being functions, infixes often only exist in phrases like n=9.

We require the highest degree of expression flexibility to fully benefit from sorting. Basic data come in a variety of forms, including texts, reals, and integers. We sort strings and integers on the overhead. With support for the types int, real, and string, the operator = is overloaded. The overloading ambiguity is resolved by the list provided as insort's second parameter.

A diminishing sort results from passing the relation for lessequal. This is not a coding trick; it has a mathematical foundation. Both if and only if are incomplete orderings. Orderings can be combined in numerous ways. The lexicographic ordering, which uses two keys for comparisons, is of utmost significance. It is specified by $(x', y') < (x, y) \iff x' < x \vee (x' = x \wedge y' < y)$. Consider the wording of the entries in an encyclopaedia as an example of how portion of the data frequently has no bearing on the ordering. We have an ordering on pairs in mathematics such that $(x', y') < (x, y) \iff x' < x$.

In machine learning, these methods of mixing orderings can be represented by functions that accept parameters of orderings and return outcomes of other orderings.

A Summation Functional

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

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

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

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

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

Functional arguments can be supplied in this way in numerical programming languages like Fortran. Numerical integration and root-finding are examples of classic applications. Currying makes ML superior to Fortran. Furthermore, the outcome of passing f as an argument to sum has the ability to return itself as another function. That function returns the total of all values off up to the specified bound when given an integer argument.

Applying the Summation Functional

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

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

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

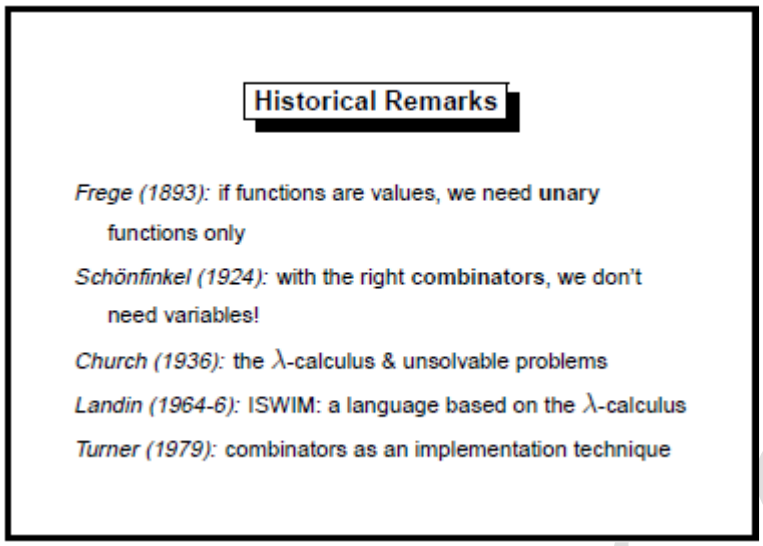
These examples show how fn-notation conveys dependency on bound variables in a manner that is consistent with conventional mathematics. Similar to the conventional sign, the functional sum can be iterated.

Let's look more closely at the first illustration:

- The factors i and j influence the variable h(i,j).
- fn j=>h(i,j) produces a function over j by being dependent only on i.
- fn i=>sum... n depends only on n and produces a function over i;
- sum (fn j=>h(i,j)) n depends on i and n, summing the function over j described above

The three variables f, m, and n all play a role in the statement as a whole.
A language for expressions that is based on mathematics, succinct, and potent is produced via functionals, currying, and fn-notation.

Historical Remarks:



In the 19th century, the notion that functions may be seen as values in and of themselves became widely accepted. This idea served as the foundation for Frege's enormous (but ultimately futile) logical system. Frege found what is now known as Currying: by employing single-argument functions solely, he was able to formalise functions with many parameters.

This truth was rediscovered by another logician, Schonfinkel, who also created combinators to remove variables from expressions. Any functional expression that has K and S such that Kxy = x and Sxyz = xz(yz) can be stated without the need of bound variables. Currying is named for Haskell B. Curry, who conducted extensive research on the combinator theory.

The -calculus by Alonzo Church provided a straightforward syntax and notation for expressing functions. It was created before ML's fn-notation. Church's thesis asserts that this accurately specifies the set of functions that can be calculated effectively. It was shortly demonstrated that his system had computational power comparable to Turing computers.

The creation of functional programming languages was greatly influenced by the -calculus. McCarthy's Lisp had kind of a false start because it misinterpreted variable binding, which persisted for around 20 years.

Landin did, however, outline the primary characteristics of functional languages. Turner made the incredible discovery that combinators, which were previously solely regarded to have theoretical utility, could be used to create functional languages with slow evaluation.

Learning guide. Pages 171–179 of ML for the Working Programmer include pertinent information. The second-year Foundations of Functional Programming course will cover the -calculus, which is introduced in Chapter 9.

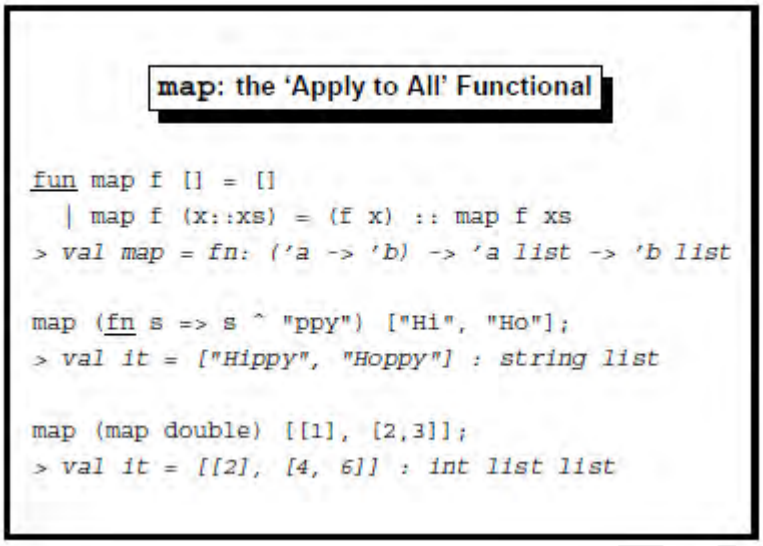
Exercise 10.1 Create an ML function that combines two lexicographic orderings. Describe how this enables the function insort to sort a list of pairs while utilising both elements in the comparisons.

Exercise 10.2 Implement an iterative sum, a three-argument curry function. What difference does it make whether the accumulator is the first, second, or third argument?

Exercise 10.3 - Describe the second sum on the overhead. (sum f) - what is it?

Chapter 11 List Functionals

Functional Map:



A function is applied to each element of a list by the functional map, which then returns a list of the function's output. Apply to all is a fundamental operation, and in this lesson, we'll see various uses for it. We can once more observe the benefits of currying and fn-notation. The first example on the slide would need a preliminary function definition if not for them (and map):

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

A lengthy list of declarations can be condensed into a single expression by using functionals, as in our second example. This coding method can be as crystal-clear as it is confusing at times. We can finally capture common programme structures by treating functions as values.

The second example uses double, which is a straightforward integer doubling function:

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

Note that the ML function map is a built-in one. Among many other things, the library of Standard ML has numerous list functions.

Matrix Transpose:

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))
```

You can think of a matrix as a list of rows, each of which contains a list of matrix elements.

Comparing this format to the standard one (using arrays), it is not particularly efficient. However, lists of lists frequently appear, and by using examples from well-known matrix operations, we can show how to handle them. The extent of ML for the Working Programmer is Gaussian elimination, which is shockingly simple.

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

```
[[a,b,c], [d,e,f]]  ↦  [[a,d], [b,e], [c,f]]
```

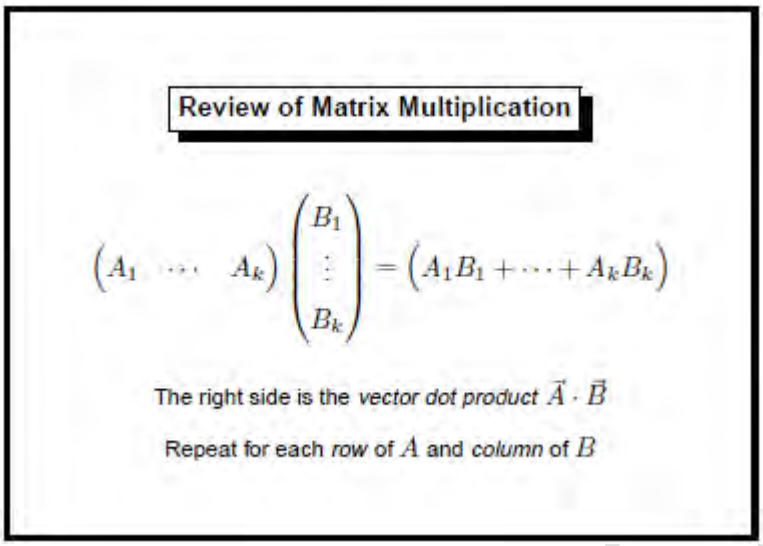
The operation of function transp is straightforward. Map hd extracts the matrix's first column if rows is the matrix to be transposed, and Map tl extracts the matrix's second column:

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

The latter matrix is transposed via a recursive function and is then given the column [a,d] as its first row.

If not for map, the two functions would need to be stated independently.

Matrix Multiplication:



The right side is the vector dot product ~A ¢ ~B

Repeat for each row of A and column of B

The dot product of two vectors is

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

A single row and a single column in A and a single column in B constitute a straightforward case of matrix multiplication. The dot product depicted above is the sole member of the 1 £ 1 matrix that results from multiplying A and B, provided that they both have the same number of k of elements.

If A and B are m X k and k X n matrices, respectively, then AXB is a m X n matrix. The (i; j) element of A £ B is the dot product of row i of A and column j of B for each i and j.

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

The (1,1) element above is computed by

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

Three nested loops are typically used when programming matrix multiplication in a traditional programming language. Subscripting is prone to errors and frequently runs slowly because of unnecessary internal calculations.

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 transfer B is transformed into a list of columns by brows. It produces a list, the items of which are the columns of B. A row of A is multiplied by the columns of B to produce each row of A X B.

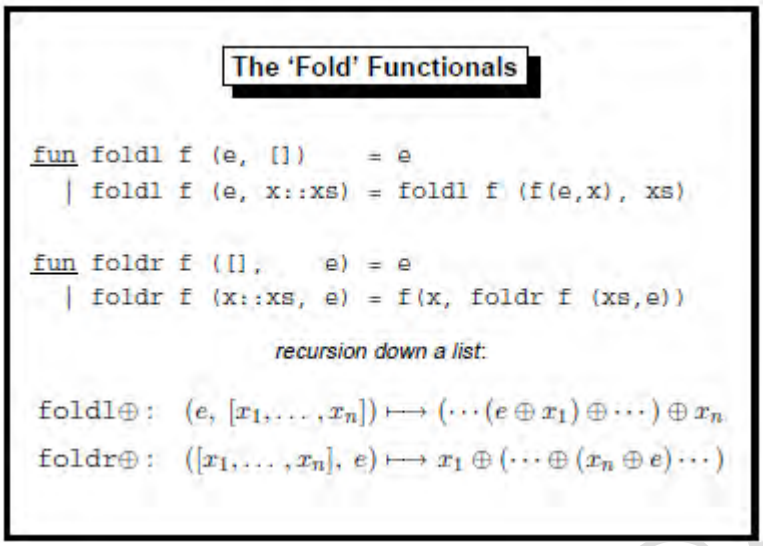
Dotprod can be used on an A row because it is curried. All of B's columns are then subjected to the resulting function. Another instance of partial application with currying is available.

Dotprod is applied to each row of A in the outer map. The inner map applies dotprod row to each column of B using the fn-notation. Contrast this version with the one on page 89 of ML for the Working Programmer, which omits the use of map and calls for two extra function declarations.

The two vectors in the dot product function must be the same length.

If not, a Match exception is raised.

The 'Fold' Functionals:



These functions have the value e as their starting value. They employ the function to combine it with the list items one at a time. Foldr takes the list components from right to left, whereas foldl takes them in the opposite direction. Here are some of them:

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

Foldl and Foldr are commonly used to multiply or add lists of numbers. There are numerous compact ways to express recursive functions on lists. Some of them are simple to understand and use familiar idioms. But it's also simple to create confusing code.

The list datatype and foldr have a particularly strong relationship.

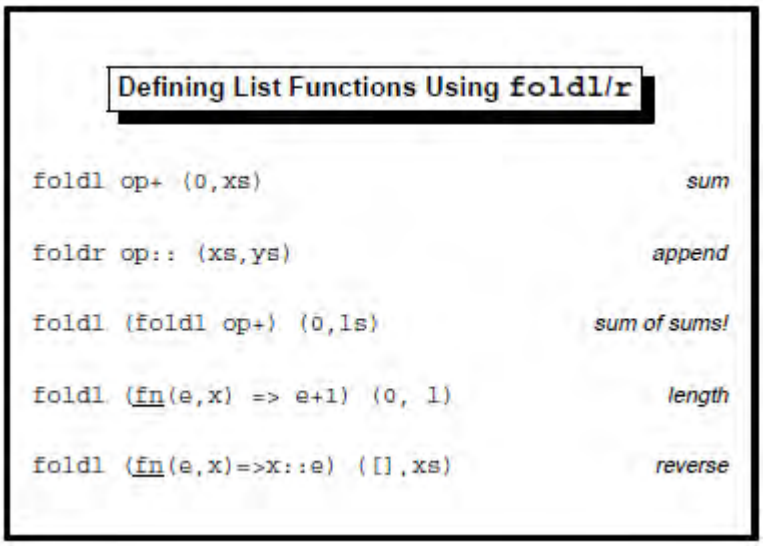
The list [1,2,3,4] is presented below in its internal structure.

$$\begin{array}{cccccc} :: & \to & :: & \to & :: & \to & :: & \to & nil \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ 1 & & 2 & & 3 & & 4 & & \end{array} \qquad ©$$

Compare to the expression generated by the foldr(©;e) function. The cons are changed to ©, and the final nil is changed to e.

$$\begin{array}{cccccc} \oplus & \to & \oplus & \to & \oplus & \to & \oplus & \to & e \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ 1 & & 2 & & 3 & & 4 & & \end{array}$$

Defining List Functions Using foldl/r:



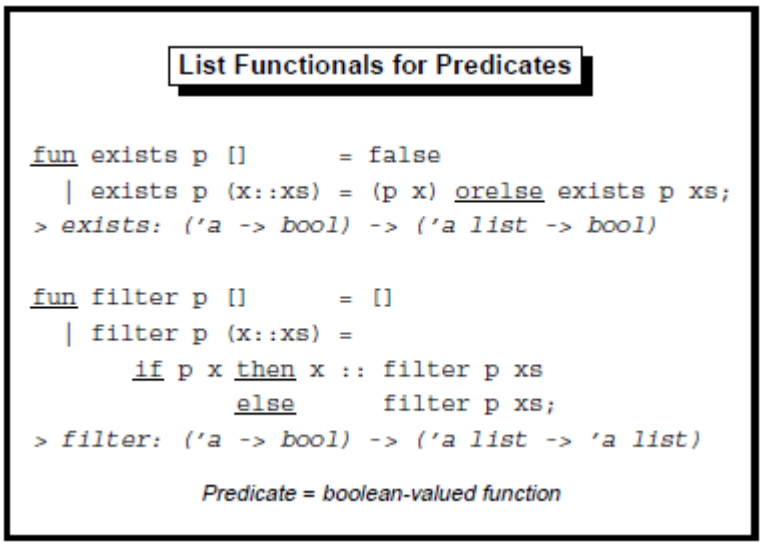
A list's elements are added one at a time, beginning with zero, to generate the list's total. Foldr would require linear space rather than constant space, making it less efficient. Be aware that the infix addition operator becomes a function that can be handed to other functions, like foldl, when you use op+. Similar syntax is used for append, using op:: to stand for the cons function.

The computation of the sum-of-sums is space-efficient since it does not create an intermediate list of sums. Foldl is also iterative. Pay close attention to how the inner foldl describes a function to add the numbers in a list; the outer foldl then uses this function to add the numbers in each list in turn, beginning with zero.

The nesting in the calculation of the sum-of-sums is typical of fold functionals with good design. Other data structures, like trees, can be declared to have equivalent functionals. These functions can be easily used to operate on nested data structures, such as trees or lists, by nesting them.

The length calculation can be viewed as unnecessary. Using fn-notation, a simple function is provided that only counts the list elements. However, unlike na3ve versions like nlength (Lect. 4), this length function occupies constant space. Using foldl ensures an accumulator-based iterative solution.

List Functionals for Predicates:



A predicate is converted into a predicate over lists by the functional exists.

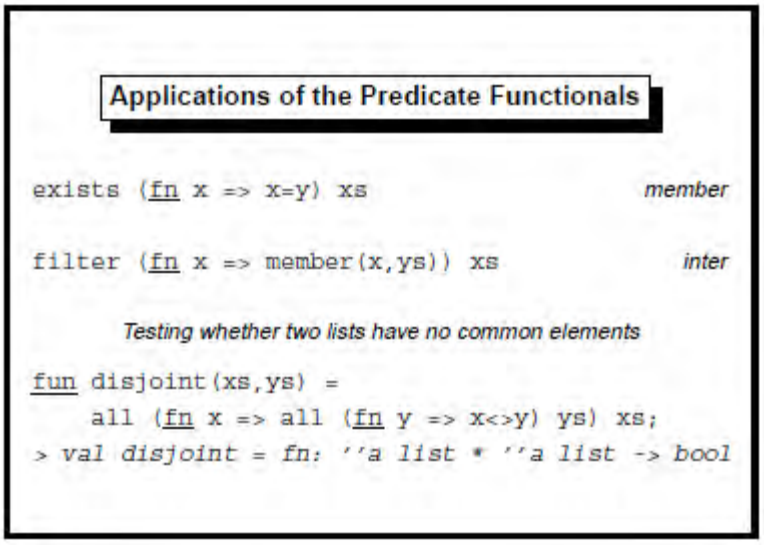
Exists p evaluates if a given list element satisfies p, returning true if it does. Due to the behaviour of orelse, if it finds one, it instantly quits looking for more; this element of exists cannot be attained via fold functionals.

In addition, we have a functional to determine whether every element of the list satisfies the condition. If it comes upon a counterexample, it also gives up looking.

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

The map-related functionality of the filter. It applies a predicate to each element of the list, returning the list of elements that fulfil the predicate rather than the results, which could only be true or false.

Applications of the Predicate Functionals:

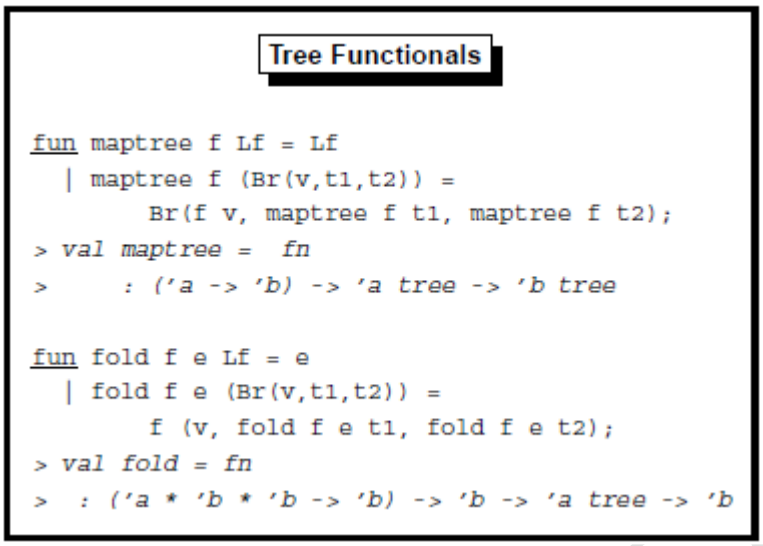


We take predicate functional applications once more as an example.

In lecture 5, the functions member and inter were covered. The function member determines if a given value can be found as a list element while the function inter returns the "intersection" of two lists, or the list of members they share.

But keep in mind that list functionals are not intended to replace the declarations of widely used functions, which are undoubtedly already available. It is done to do away with the requirement for individual declarations of ad-hoc functions. The inner functions are almost definitely one-offs when they are nested, like the calls to all in disjoint above, and are therefore not worth stating separately.

Tree Functionals:



The concepts covered in this lecture naturally generalise to trees and other datatypes, not just recursive ones.

The functional maptree creates another tree with the same shape by applying a function to each label of a tree. There are analogues to and they are all easily declared. However, filter is challenging since the form of the tree is altered when the filtered labels are removed, and there is no obvious way to incorporate the filtered results from both subtrees if a label does not fulfil the predicate.

Declaring a fold functional in the manner displayed above is the simplest method. The constructors Br and Lf are correspondingly replaced by the arguments f and e. The labels of a tree can be added using this functional, however it needs a three-argument addition function. Fold functionals for trees can implicitly treat the tree as a list to get around this problem. Here is a fold function associated with fold as an illustration, which processes the labels in order:

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

By extending cons to the function f, its code is derived from that of the function inord in Lesson 7.

A programming language could be viewed in our primitives themselves. This reality applies to all forms of programming, although it is especially clear in the case of

functionals. The work of programming includes expanding our programming language to include notation for resolving the current issue. The levels of notation that we create ought to match the levels of abstraction that naturally exist in the issue area.

Learning guide. Pages 182–190 of ML for the Working Programmer include pertinent information.

Exercise 11.1: Create a function that is equivalent to map (map double) without requiring map, currying, etc. Declaring two recursive functions is necessary for the simple solution. Use nested pattern-matching to your advantage to get away with just one.

Exercise 11.2: uses foldr to express the functional map.

Exercise 11.3: Declare a map equivalent for the option type: datatype 'a option = NONE | SOME of 'a;

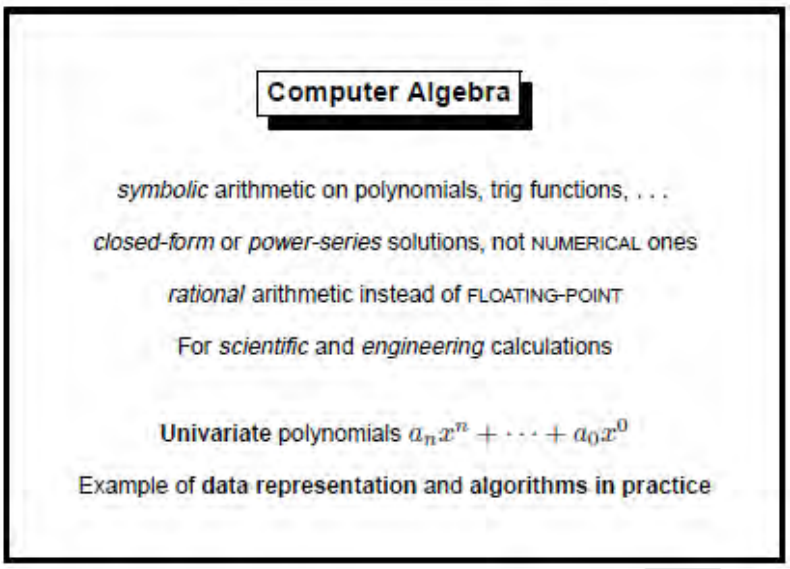
Exercise 11.4: Recall Lect. 5's function of making changes:

```
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.

Chapter 12 Polynomial Arithmetic

Computer Algebra:

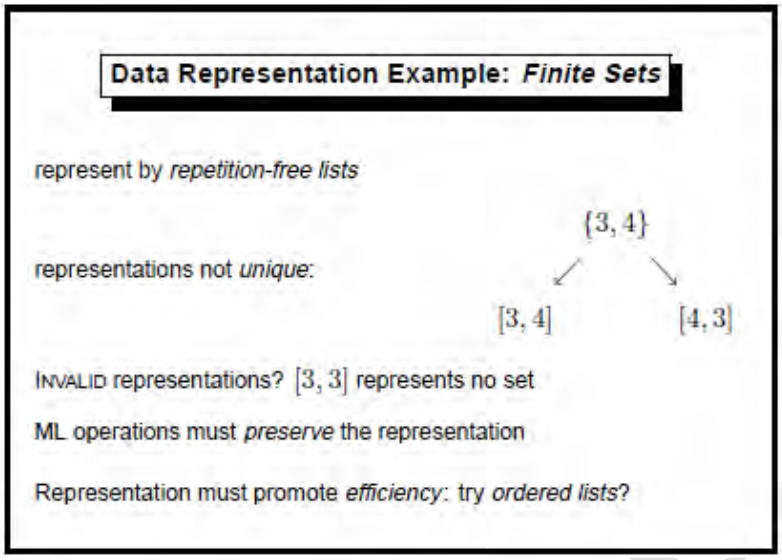


This lecture illustrates the treatment of a hard problem: polynomial arithmetic. Many operations that could be performed on polynomials; the general problem is too ambitious; compromises must be made. We shall have to simplify the problem drastically. We end up with functions to add and multiply polynomials 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. Univariate polynomials are those that have only one variable, such as x. Our task is difficult even after limiting ourselves to univariate polynomials.

This example shows how to efficiently describe a non-trivial type of data and use fundamental algorithmic concepts.

Data Representation:



Finite sets are not offered as a data structure by ML. Lists without repeats could be used to represent them. A straightforward illustration of data representation is finite sets. A set of concrete things (repetition-free lists) is used to represent a collection of abstract objects (finite sets). Every abstract object has at least one physical counterpart, and sometimes more, as in the case of f3; 4g, which can be represented by [3; 4] or [4; 3]. Certain concrete items, such [3; 3], don't even represent any abstract things.

The representations are used to specify operations on the abstract data.

In the case where inter(l; l') represents A A' for all lists l and l' that represent the sets A and A', the ML function inter (Lect. 5) implements the abstract intersection operation. It is simple to verify that inter maintains the representation since, as long as its arguments are sound, its return is devoid of repetition.

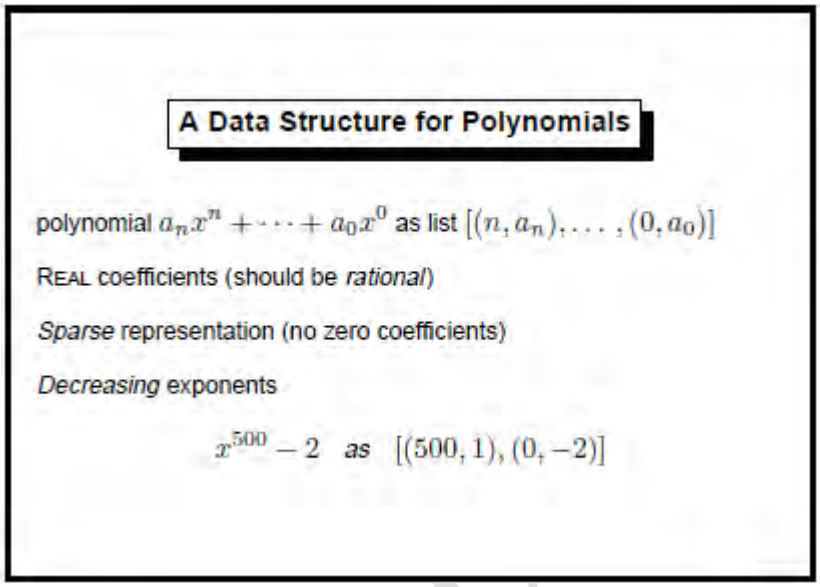
The lists are most efficiently utilised when they are devoid of repetition.

The intricacy of time could be increased. Finding all the elements they share is necessary to form the intersection of an m-element set and an n-element set. Only by exploring every possibility and taking O(mn) time will it be accomplished. Ordered lists should be used to represent collections of integers, strings, or other objects that have a general

ordering. The intersection algorithm can be completed in O(m + n) time and resembles merging.

Only here can some deeper topics be mentioned. For instance, °oatingpoint arithmetic approximates actual arithmetic.

Data Structure for Polynomials:



The univariate polynomial $a_n x^n + \cdots + a_0 x^0$ might be represented by the list of coefficients $[a_n, \ldots, a_0]$. If many of the coefficients are zero, as in x500 - 2, this dense representation is ineffective. Instead, we employ a sparse representation, which is a list of (exponent, coefficient) pairs with only nonzero coefficients.

Rational numbers, or pairs of integers without a common factor, should be used as coefficients. Although arbitraryprecision integer arithmetic is necessary for exact rational arithmetic, our needs don't call for it. We will use the less-than-ideal ML type real to express coefficients.

The code's objective is to demonstrate a few polynomial arithmetic algorithms.

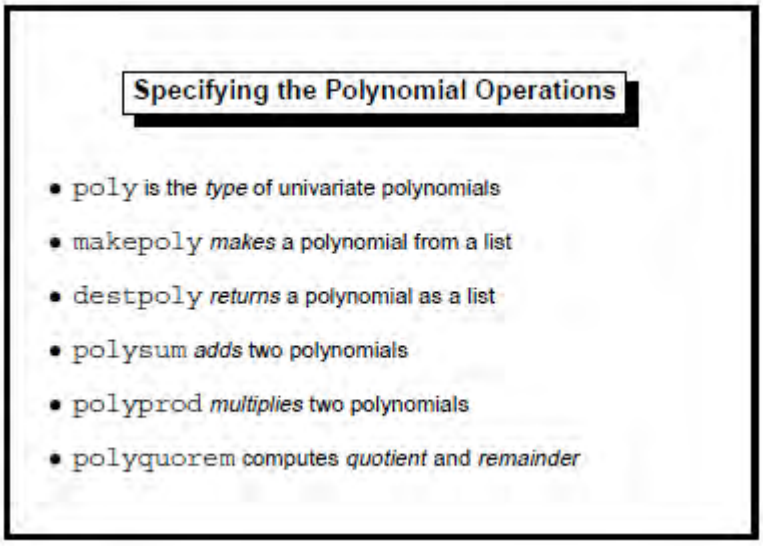
The ML type (int*real)list will be used to represent the sum of terms in polynomials, with each term denoted by an integer exponent and a real coefficient.

We don't only leave out 0 coefficients; we also store the pairs in decreasing exponent order to maximise efficiency. The ordering permits techniques similar to mergesort and permits a particular exponent to be present in no more than one phrase.

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

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

Polynomial Operations:



The aforementioned operations, which may be supported by an implementation of univariate polynomials, can be summed up 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
```

An ML signature can be created from this neat specification. An ML structure can be created out of a collection of declarations that satisfy the signature.

These ideas encourage modularity, enabling us to maintain orderly higher abstraction layers. In particular, the structure might be called Poly, and its parts would go by the short names sum, prod, etc.; they would be referred to as Poly.sum, Poly.prod, etc. from outside the structure. Although ML modules are not covered in this course, my book [12] provides a modular treatment of polynomials. Large systems must be built using modules.

Dest-poly may return the underlying list, while the function makepoly could turn a list into a valid polynomial. The underlying representation for many abstract types should be hidden. We definitely do not want an operation to return a dictionary as a binary search tree for dictionaries (Lect. 8). However, we can convey polynomials to the outside world

using our list-of-pairs form. Even if a different form is chosen to enable quick arithmetic, it might still be kept for that purpose.

Polynomial Addition:

Polynomial addition

```
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;
```

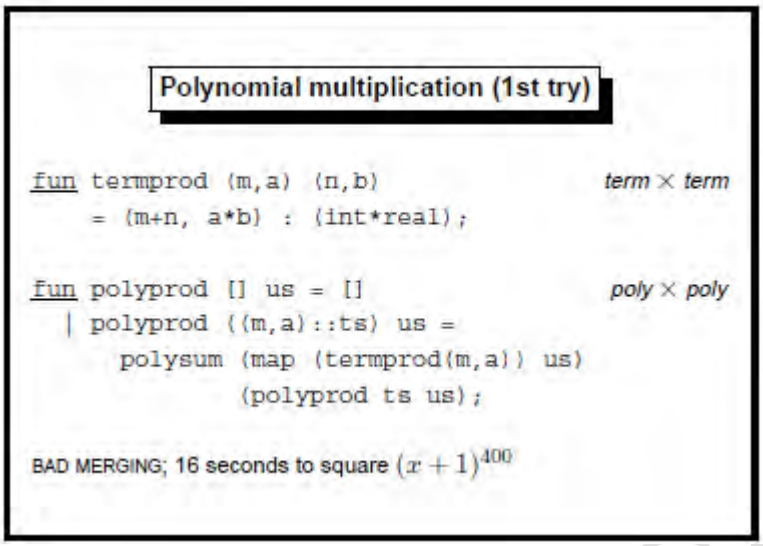
Our representation enables the use of the traditional methods taught in schools for addition, multiplication, and division. Their effectiveness can occasionally be increased. The arithmetic operations are all carried out, without any apparent rationale.

The two polynomials' matching coefficients are added during addition.

It is necessary to maintain the ordering and leave out the zero coefficients in order to preserve the polynomial form.

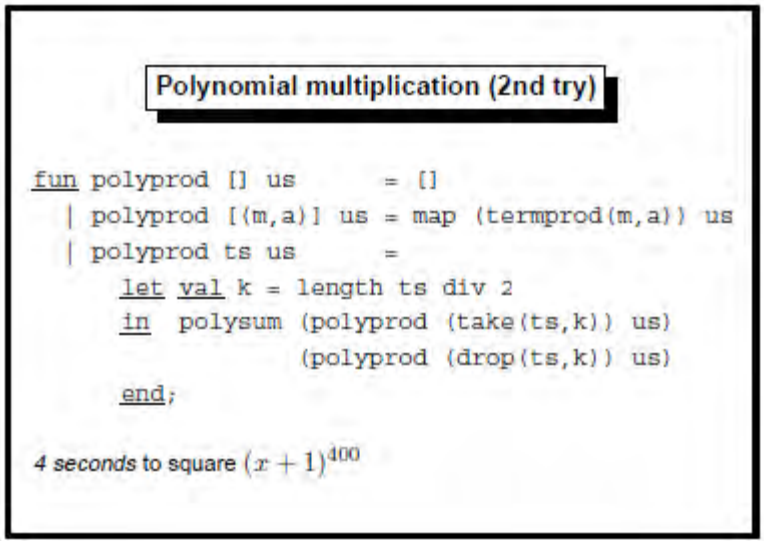
The algorithm for adding is similar to merging. Compare the leading terms of both polynomials if they are both nonempty lists. Start by calculating the term with the largest exponent. If the exponents are identical, combine their coefficients to form a single term; if the sum is zero, toss the new term.

Polynomial Multiplication:



If efficiency is not a concern, polynomial multiplication is also simple and only requires the textbook algorithm. Function polyprod creates products term by term and adds the intermediary polynomials to cross-multiply the terms.

Another use of the functional map may be seen in the expression map (termprod(m,a)) ts, which is the product of the term (m,a) and the polynomial.

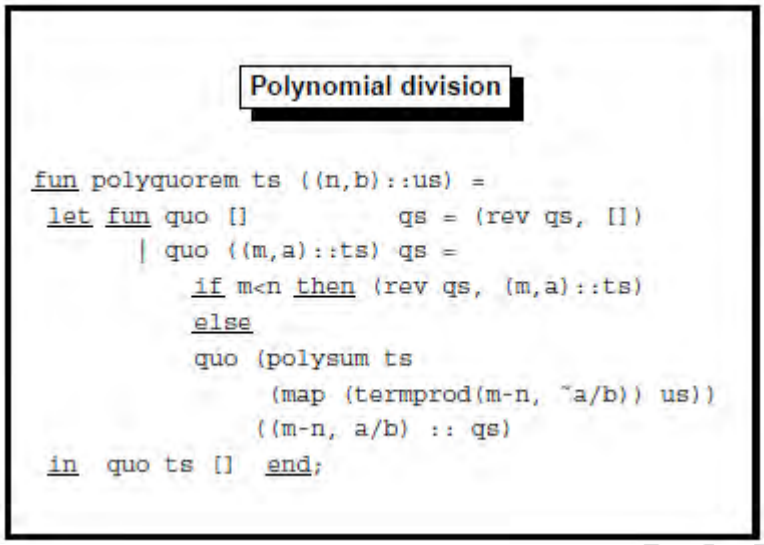


Large polynomials cannot be handled by polyprod because of its slowness. In tests, computing the square of (x+1)400 took roughly 16 seconds and several garbage collections. (Symbolic algebra frequently involves calculations of this size.)

The combining (in polysum) of lists with vastly different lengths is the cause of the inefficiency. For example, if ts and us each have 100 terms, then (termpolyprod (e,c) us) only has 100 terms, whereas (polyprod ts us) may have 10,000 terms. At most 10,100 terms will be added to their total, a rise of just 1%. If one list is significantly shorter than the other, merging essentially degenerates into insertion.

Mergesort (Lecture 6) serves as the inspiration for a quicker algorithm. Using take and drop, divide one of the polynomials into equal pieces. Create two goods that are nearly the same size, then combine them. If a polynomial only has one term, multiply it by the other polynomial using the above-mentioned map. With this approach, the number of merges is significantly lower, and each merge roughly doubles the size of the outcome. There are still quicker polynomial multiplication algorithms.

Polynomial Division:



Now let's discuss how to compute the quotients and remainders of polynomials.

The polynomial division algorithm, which is really simpler than long division, is implemented via the polyquorem function. The pair (quotient, remainder) is returned, with the remainder being either zero or less significant than the divisor.

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

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

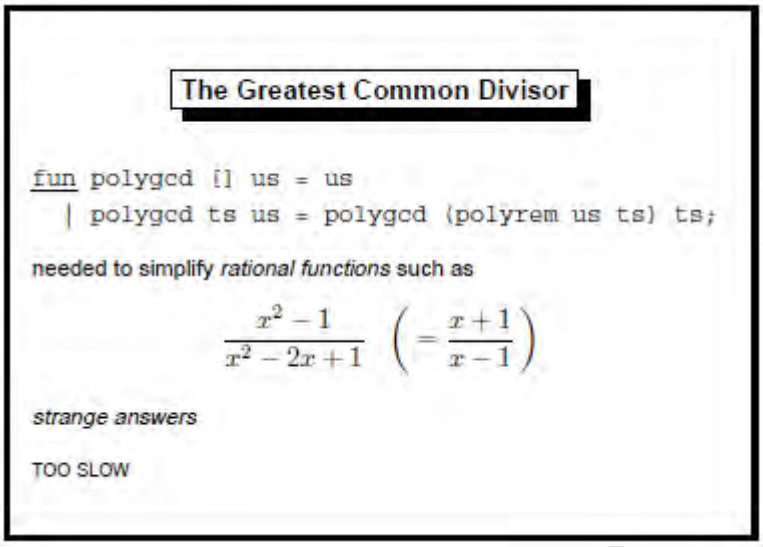
Aside: #k is an ML function that returns the kth component of a tuple if k is any positive integer constant. Tuples are a particular case of ML records, and any record field can use the # notation.

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:



Polynomial fractions like (x + 1)=(x¡1) are examples of rational functions. Efficiency necessitates that there be no common factor between a fraction's numerator and denominator. The two polynomials should be divided by their GCD, or greatest common divisor.

Euclid's Algorithm can be used to compute GCDs, as was demonstrated above. Unfortunately, it behaves pretty bizarrely for polynomials. The GCDs of x2+2x+1 and x2 -1 are given as -2x - 2 and 5x+5, respectively; both GCDs should be x+1. Divide through by the leading coefficient can be used to solve this particular problem, however Euclid's algorithm is too slow. Even if the final GCD is merely one, an innocent-looking pair of parameters causes calculations on enormous integers! The majority of pairings of polynomials don't have a common factor, which is the usual result.

The core challenge in computer algebra is calculating the GCD of polynomials. There are deployed very sophisticated algorithms. Both sophisticated programming and profound mathematics are required for a good implementation. This similar set of skills is necessary for numerous advanced technology initiatives.

Learning guide. Pages 114–121 of ML for the Working Programmer include relevant information.

Exercise 12.1 uses the ordered-list format to code the set operations membership test, subset test, union, and intersection.

Exercise 12.2 Make a strong case for how polysum and polyprod maintain the three limits on polynomials.

Exercise 12.3: How would you demonstrate that polysum computes the sum of two polynomials correctly? Tip: To express the polynomial represented by a list, construct a mathematical (not ML) function. Which characteristics of polynomial addition does polysum assume?

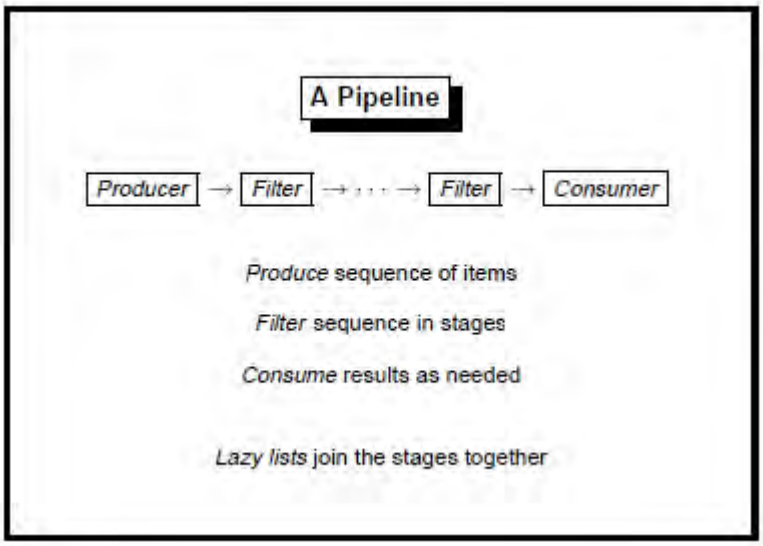
Exercise 12.4 Demonstrate that when polysum is used on arguments made up of m and n terms, respectively, the complexity is O(m + n).

Exercise 12.5 Analyse the asymptotic complexity of the two polynomial multiplication algorithms with greater rigour. (This can be challenging.)

Exercise 12.6 We regain the capacity to represent polynomials in any number of variables if coecients can themselves be univariate polynomials (in some other variable). As an illustration, the univariate polynomial y2 + xy has the coefficients 1 and the polynomial x. Discuss strategies for implementing addition and multiplication while defining this representation in ML.

Chapter 13 Sequences, or Lazy Lists

A Pipeline:



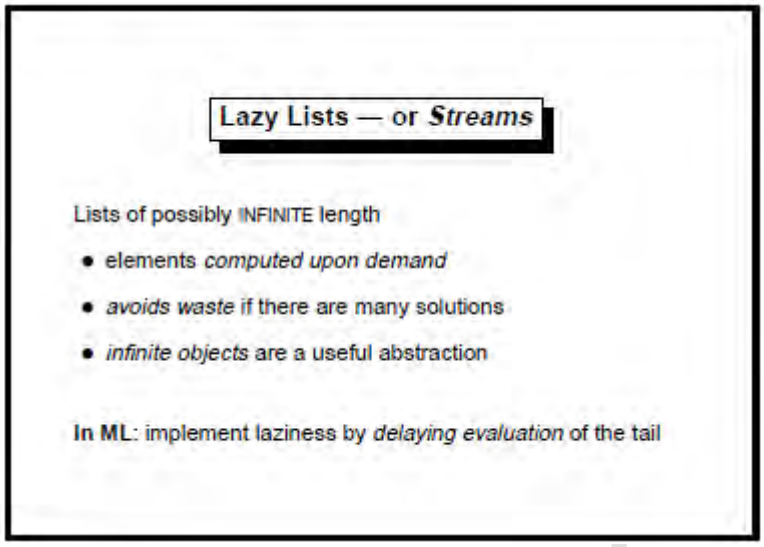
Programmes can be divided into two categories. A sequential programme accepts a problem to solve, works on it for some time, and then ends with the solution. The extensive numerical simulations carried out on supercomputers serve as an example. The majority of our ML algorithms also match this model.

Reactive programmes, whose purpose it is to engage with the environment, are at the other extreme. They operate for however long is required and communicate continuously. The computer software that runs many modern aircraft is a good illustration. Concurrent processes that are engaged at the same time and in communication make up reactive programmes frequently.

Concurrency is too complex to be covered in this course, but we can construct straightforward pipelines like the one above. The Producer outputs a stream of data that it represents from one or more sources. The filter stages transform the input stream into an output stream, possibly requiring many input items to produce a single output item. The consumer takes however many components are required.

Everything is computed except in response to the Consumer's request for a new datum, which controls the pipeline. The Filter stages are carried out in succession as necessary to complete the computation. The programmer configures the data requirements but is unsure of what will occur when. The idea of concurrent computation is a delusion. Similar concepts are provided by the Unix operating system's pipes, which connect processes. In ML, lazy lists can be used to model pipelines.

Lazy Lists:



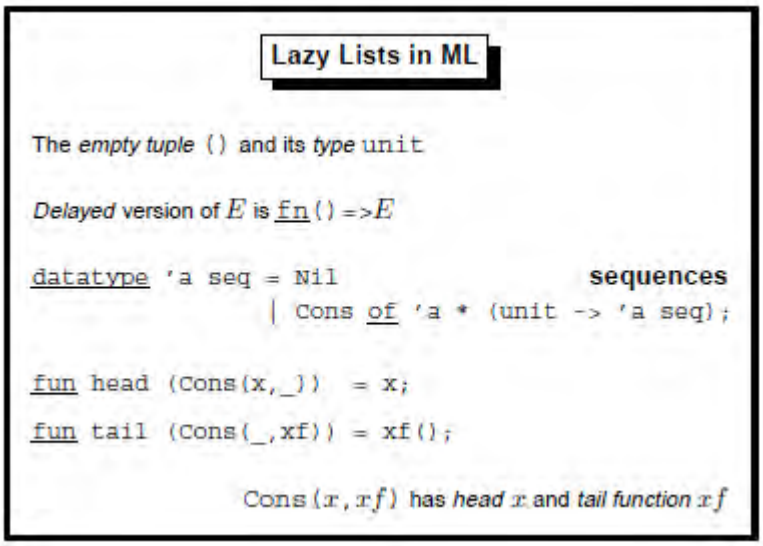
Lists that are too lazy can be useful. Making changes is one of many algorithms that can produce numerous solutions when only a few are necessary. With lazy lists, we can pretend that infinite series do exist when the original issue is that they don't!

We are now working with computations that are infinite, or at least unbounded.

Upon request, a potentially limitless supply of data is processed one component at a time. These programmes have more potential for error and are more difficult to understand than ones that terminate.

There are certain wholly functional languages that employ lazy evaluation everywhere, like Haskell. All lists are lazy, and even the if-then-else construct can be a function. In ML, we can declare a type of lists so that the tail is only evaluated when necessary. Lazy evaluation is stronger, yet delayed evaluation is sufficient for our needs.

For input/output channels, the conventional word stream is reserved in machine learning terminology. Sequences of lazy lists will be our term.



One element, denoted by the symbol (, is present in the primitive ML type unit. This component can be thought of as a 0-tuple, with unit representing the nullary Cartesian product. (Consider the relationship between the number 1 and multiplication.)

In circumstances where no data is required, the empty tuple acts as a placeholder. It can be used for a variety of things:

- It might show up in a data structure. A unit-valued dictionary, for instance, represents a collection of keys.
- It might be a function argument whose effect is to postpone evaluation.
- It might be the justification or outcome of a process. (Lecture 14.)

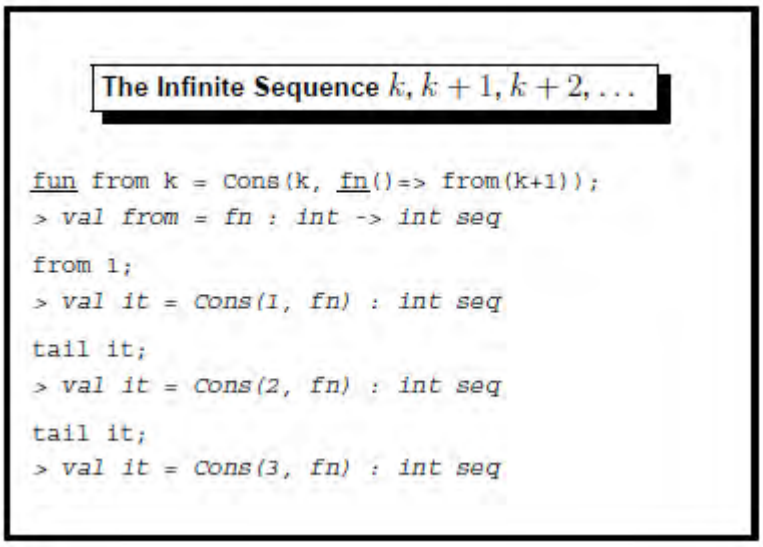
Like all tuples, the empty one has a constructor and is accepted in patterns:

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

The function that accepts a unit-type input and outputs the value of E in particular is called fn() -> E. Despite the fact that there is only one argument, (), evaluation of expression E does not occur until the function is called.

Simply put, the function postpones the evaluation of E.

The Infinite Sequence:

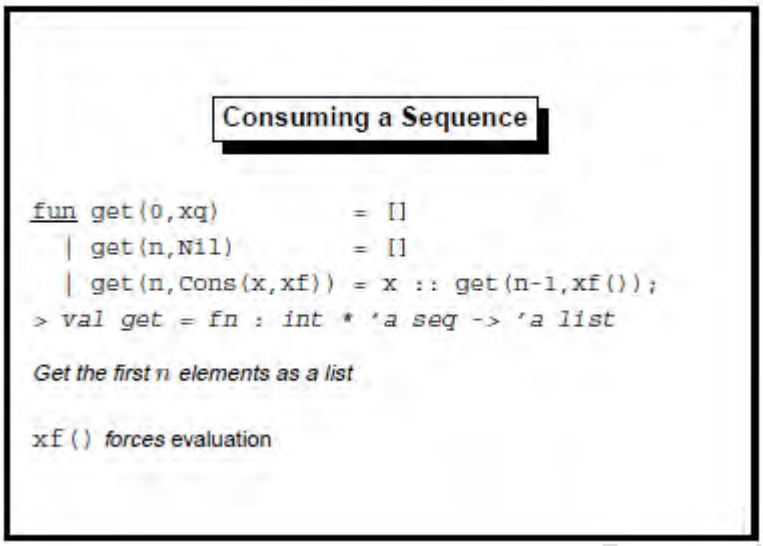


The function from creates an unlimited series of integers beginning with k. The fn encapsulating the recursive call causes execution to end. The tail of a sequence is shown by ML as fn, where fn is a function value.

The following sequence element is created with each call to tail. This may go on forever. Because the expense of constructing the dummy function will outweigh the cost of computing a sequence member, this example has no practical utility.

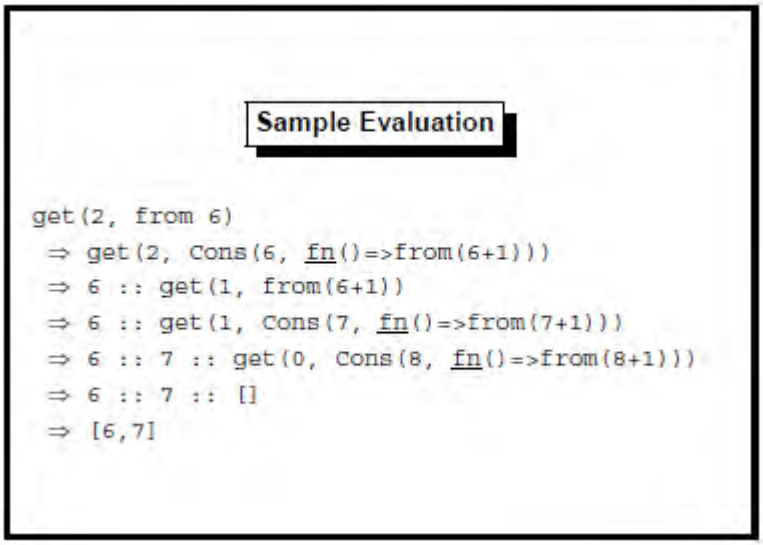
Inefficient lists typically have substantial overheads.

Consuming a Sequence:



A sequence is transformed into a list via the function get. It takes the first n elements and, if n is greater than 0, all of them. This process can end only if the sequence is finite.
The expression xf() calls the tail function on the third line of get, requiring evaluation of the subsequent element. Calling this procedure "forcing the list"

Sample Evaluation:

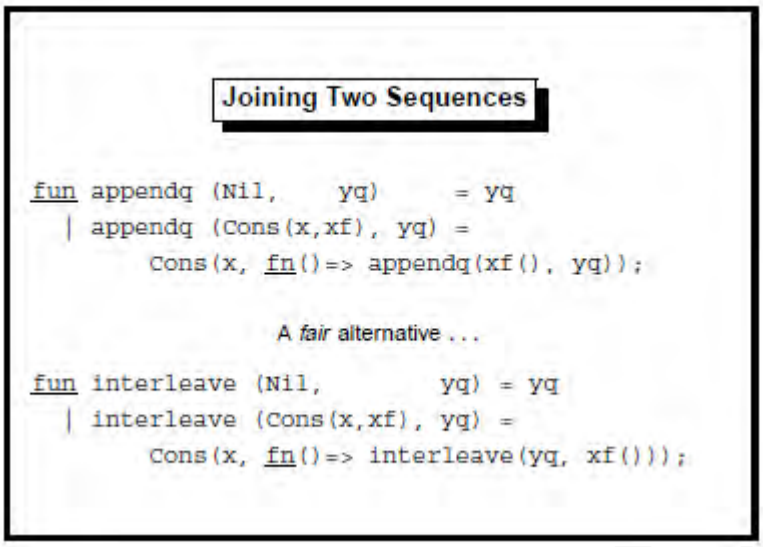


In this case, we are requesting two elements from the infinite series. Actually, six, seven, and eight items are computed. Our implementation is a bit hasty. This issue might be prevented by a more complex datatype declaration. Another issue is that forcing causes a list element to be evaluated more than once when it is repeatedly examined. The outcome of the initial evaluation would be saved for subsequent use in a lazy programming language.

References are necessary to achieve the same result in ML [12, page 327].

We are fortunate that the computation is limited rather than becoming limitless. The expression 6+1 is even present in the original sequence's tail.

Joining Two Sequences:



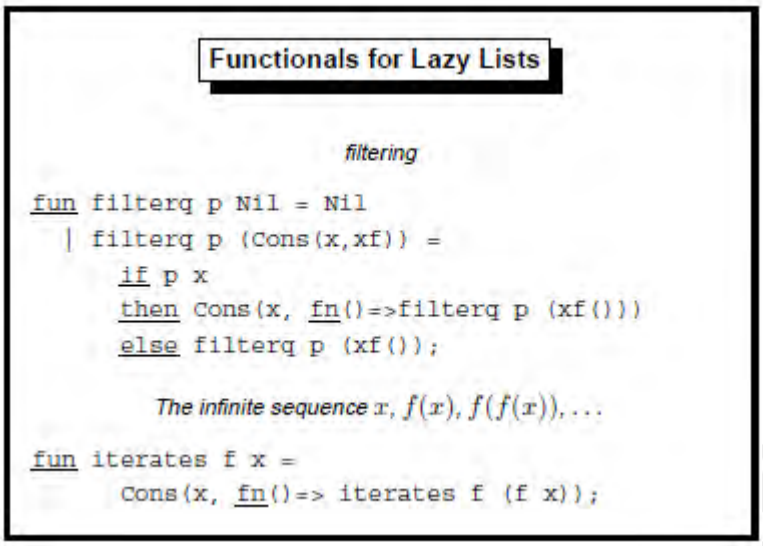
There are sequence analogues for the majority of list functions and functionals, but weird things can still occur. Can you reverse an endless list?

Appendq concatenates two sequences, which is exactly the same as append (Lecture 4) in concept. Appendq never gets to its second argument, which is lost, if the first argument is infinite. Infinite sequence concatenation is not particularly fascinating.

By switching the two arguments in each recursive call, the function interleave gets around this issue. No elements are lost in the combination of the two lazy lists. The best technique to merge two potentially limitless streams of information into one is by interleaving.

Keep in mind that each xf() in both function declarations is encased in a fn()->... Every force is contained by a delay. These actions make the functions inefficient. A force that isn't contained in a delay, like in get above, runs the risk of being evaluated in its whole.

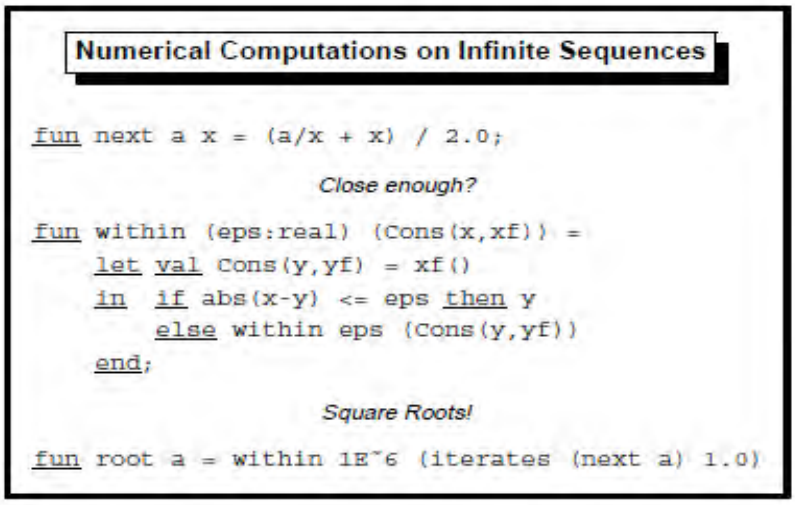
Functionals for Lazy Lists:



Up until it discovers one satisfying p, the functional filterq wants elements of xq. (Lect. 11, Recall Filter) It contains a force that is not delayed. Filterq runs indefinitely if xq is infinite and lacks a satisfying element.

Iterating functionally generalises from. Instead of adding one, it instead calls the function f to generate the subsequent element.

Numerical Computations on Infinite Sequences:



The Newton-Raphson method provides an infinite series of approximations to the square root of a by calling iterates (next a) x0). In Lec. 2, it was discussed that the infinite series x0, (a + x0)/2,... Converges to Function within looks for two points whose difference is less than eps in the lazy list. It measures the extent of their disparity. There are 'near enough' checks like relative difference that can be programmed. Other numerical functions can be implemented directly using these elements as functions over sequences. The idea is to construct programmes from tiny, replaceable components.

Function Root applies the Newton-Raphson method with a tolerance of 106 and a (terrible) beginning approximation of 1.0, then iterates and continues.

In academic literature, this method of numerical computing has drawn some attention; Richardson extrapolation being a frequent example [6, 7].

Learning guide. Pages 191–212 of ML for the Working Programmer include pertinent information.

Exercise 13.1 Create a map equivalent for sequences.

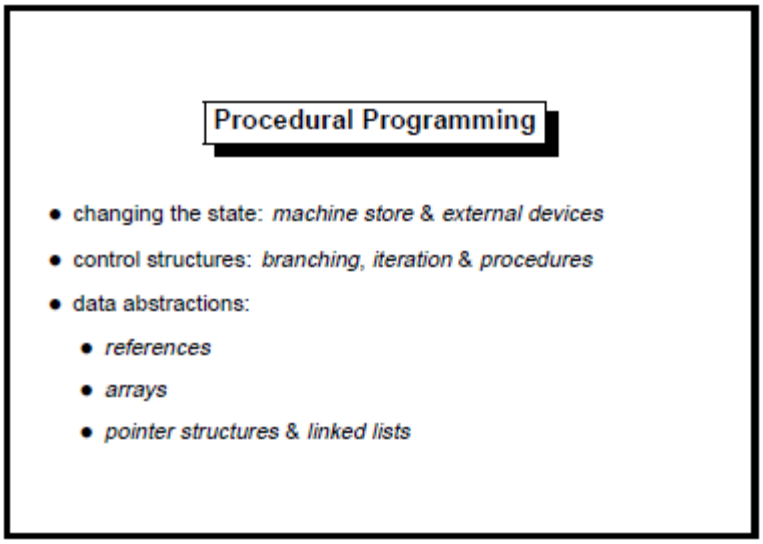
Exercise 13.2 Consider the list function concat, which joins a number of lists together to create a single list. Can a series of lists be concatenated in general? What might go wrong?

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

Exercise 13.3 Create a function that uses lazy lists to perform changes. (This is challenging.)

Chapter 14 Elements of Procedural Programming

Procedural Programming

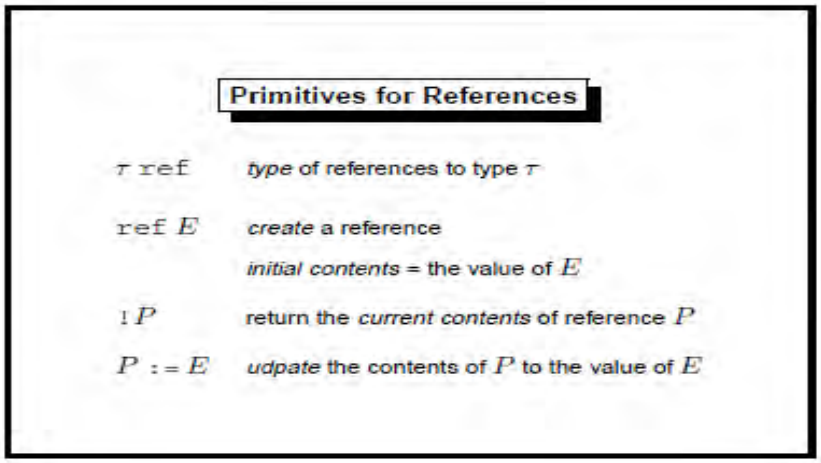


The traditional definition of programming is procedural programming. The repetition of commands or statement execution changes the state of a programme. A local machine state change could involve changing a variable or array. A state transition could involve communicating information to external parties. Since reading data typically removes it from the environment, doing so counts as a state change.

Primitive commands and control structures for mixing them are provided by procedural programming languages. Assignment, which updates variables, and a number of input/output commands for communication are among the elementary commands. For conditional execution, control structures include the if and case statements, as well as while-style repeated statements. Programmers are able to package their own commands into arguments-taking procedures. Such'subroutines' were necessary from the very beginning of computing, and they are one of the earliest examples of abstraction in programming languages.

No distinction is made between commands and expressions in ML. To carry out assignment and communication, ML has built-in functionalities. The same as in other languages, they can be used with if and case. The ML functions act as the procedures. For the majority of internal computations, ML programmers often utilise a functional style, and they only use imperative elements when communicating with the outside world.

Primitives for References:



The ML primitives are shown on the slide, however they are parallels in the majority of languages, often heavily disguised. The ability to create references (or allocate storage), access the contents of reference cells, and update those cells is required.

Create references (also known as pointers or locations) with the ref function.

A new place in the machine storage is assigned when you call the ref. This place initially has the value indicated by expression E. Despite being an ML function, ref is not a function in the traditional meaning of the word. Ref(0)=ref(0), for instance, evaluates to false.

When a reference is sent to the function!, its contents are returned. Dereferencing is the term for this action. It is obvious that! is not a mathematical function because its output depends on the store.

Expression P, which must return a reference p, and E are evaluated via the assignment P:=E. It holds the value of E at address p. Although it updates the store, syntactically, := is a function and P:=E is an expression. It returns the value () of type unit, like many other functions that modify the state.

If 'τ is some ML type, then 'τ ref is the type of references to cells that can hold values of 'τ . Please do not mix up type and function references. You might find this table of primitive functions and their kinds useful:

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

A Simple Session:

A Simple Session

```
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 states that p will initially contain a reference to the integer 5. It admits assignment since its type is int ref, not just int. Val bindings are immutable and never alter as a result of assignment. Unless superseded by a new usage of p, the identifier p will always denote the reference mentioned in its declaration. Only the reference's contents are changeable.

A reference value is shown by ML as ref v, where value v is the contents.

Although understandable, this notation does not allow us to determine whether two references with the same value are truly the same reference. The disadvantages of displaying a reference as a machine address are clear!

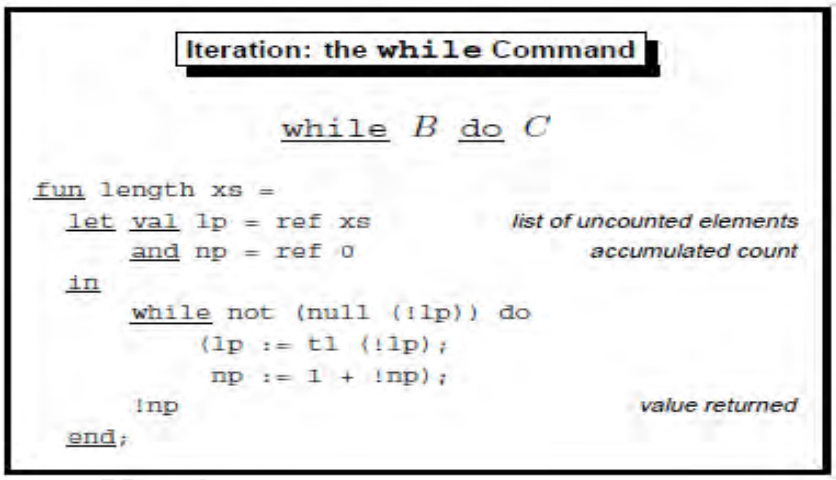
All variables in traditional languages can be altered. Even if the language supports reference types, we declare something like p: int without naming them. If an initial value

is not provided, whatever was previously there at that address may be returned. Uninitialized variables can contain illegal values that result in mistakes that are nearly impossible to identify.

The expression!p in the first assignment returns the reference's current contents, which are 5. The assignment switches what's in p to what's in 6. Due to its inconvenience, most languages lack an explicit dereferencing operator (like!). Instead, by tradition, references that appear on the left side of the := signify places, whereas references that appear on the right side signify contents. There might be a specific "address of" operator that can override the convention and insert a reference to a location on the right side. Although this is illogical, it shortens programmes.

A new reference (with an initial contents of 77) and p are both declared to be held by the list ps. The new reference is then updated to hold 3 after that. Only the contents of a reference in that list are updated by the assignment to HD ps.

Iteration: The While Command



Once we have the ability to alter the state, we must do it frequently. Recursion can be used for this, however it is cumbersome to have to declare a procedure for each loop, and compilers for traditional languages rarely use tail-recursion.

Early programming languages didn't support repetition all that well. The programmer had to use goto instructions to create loops, and an if-controlled second goto was used to end the loop. The most fundamental of the looping constructs offered by modern languages is while B do C.

In the event that the boolean expression B is true, command C is carried out, and the command is repeated. The while command ends if B evaluates to false, sometimes without even executing C once.

While is the only looping construct offered by ML, and the command returns the value (). The construct (E1;... ;En), which evaluates the expressions E1 to En in the specified sequence and returns the value of En, is also significant. The other expressions' values are ignored because their goal is to alter the state.

The function length declares references to retain both the number of elements now being counted (np) and the length of the list being examined (lp). Since the list has at least one element and is not empty, we skip over the element by assigning it to the tail and count it. Two assignment commands are executed one after the other in the while loop's body. The expression!np is used after the while command to return computed length as the function's output. Because it is surrounded by the words in and end, this semicolon does not require parentheses.

Private References:

Private References

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

You may have noticed that ML's programming approach appears awkward when compared to languages like C. The defaults and abbreviations they use to condense programmes are omitted by ML. But ML's explicitness makes it perfect for imparting

knowledge of the subtleties of references and arrays. Compared to references in other languages, ML references are more flexible.

The makeAccount function simulates a bank. When the function is called with a given initial balance, a new reference (balance) is created to keep the account balance and a function (withdraw) with exclusive access to that reference is returned. Calling withdraw subtracts the specified amount from the balance and returns the updated balance. Money can be put in by withholding a negative sum. The if-construct prevents an overdraft by raising an exception and preventing the account from becoming negative.

Take a look at the else part's (E1; E2) construct. In the first expression, the trivial value () and the account balance are both updated. The second expression,!balance, does not return the reference itself because doing so would permit unauthorised modifications but instead provides the balance as it is at the moment.

Based on a previous example by Dr. A. C. Norman.

Two Bank Accounts:

Two Bank Accounts

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

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

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

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

Every time make Account is called, a copy of withdraw that contains a brand-new instance of the reference balance is returned. There is no other way to access the account balance than through the matching withdraw feature, just like with a real bank passbook.

The reference cell becomes unreachable if that function is dropped; ultimately, the computer will recover it, just as banks close down inactive accounts.

Two persons can be seen here managing their accounts. Neither can take money from the other, for better or worse.

Make Account could be generalised to produce a number of operations that collaborate to handle data stored in shared references. The use of ML records, which are addressed previously [12, pages 32–36], may be used to package the functions.

Although object-oriented languages consider private references to be a fundamental feature, most procedural languages do not handle them well.

Primitives for Arrays:

Primitives for Arrays

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

Similar to references, ML arrays carry n entries as opposed to 1. The integers 0 to n- 1 are used to identify the elements of an n-element array. The typical notation for the ith array element is A[i]. Array.array is the type of arrays (of any size) with items from if is a type.

By allocating the required storage, the function Array.tabulate(n,f) generates an array with the size indicated by expression n. When i = 0,..., n-1, element A[i] initially contains the value of f(i).

The results of calling Array.sub(A,i) are the items in A[i].

When Array.sub(A,i,E) is used, the array is modified by adding the value of E to the contents of A[i], and () is returned as the result.

Why doesn't any function return a pointer to A[i]? Both Array.sub and Array.update may be replaced by such a function, but enabling references to specific array members would make storage management more difficult.

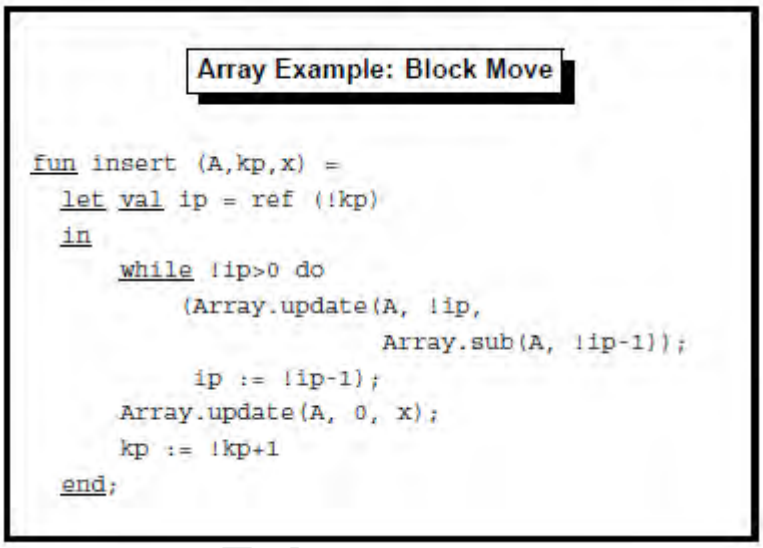
In order to simplify storage management, an array's size is predetermined.

Usually, another variable keeps track of the number of pieces that are actually in use.

The unused elements represent wasted storage; if they are never initialised to valid values (which seldom happens), they can lead to a great deal of issues.

The programme must terminate or the array must grow, usually by copying into a new one that is twice as big, when the array bound is reached.

Array Example: Block Move



The primary takeaway from this example is that using arrays is more difficult than using lists. Slides that express insertion sort and rapid sort using lists (Lect. 6). Originally, the code above—roughly x::xs—was a component of an insertion function for an array-based insertion sort. The array syntax in ML is useless. The primary task may be written in a conventional language.

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

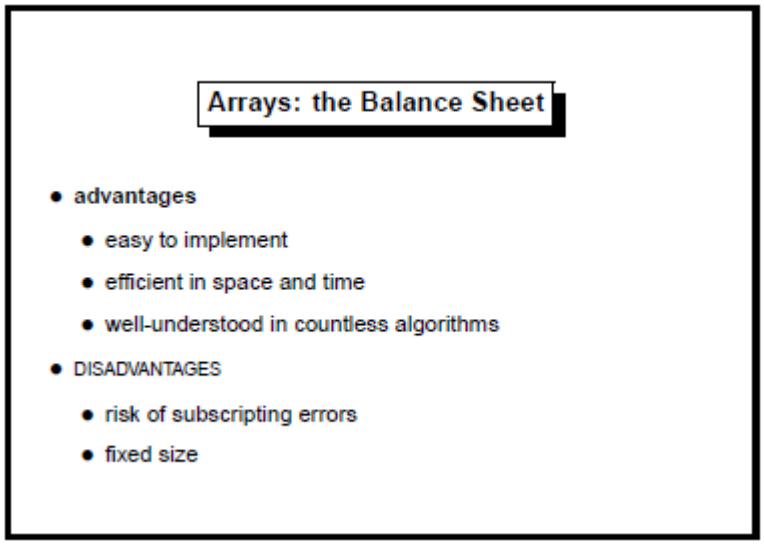
To be honest, ML's datatypes and lists have high overheads and call for a complex storage management solution. In many cases, link fields must take up an additional byte for every byte allocated to actual data, as was covered in Lesson 15.

An array A whose members are in use and are indexed by zero to!kp-1 is passed to the insert function. The function increases the bound in kp, shifts each element to the subsequent higher subscript position, and stores x in position zero.

We have an illustration of what is referred to as a reference parameter in other languages. While kp is of type int ref, argument A is of type 'a Array.array. The function only functions using these parameters.

The only thing that resembles an array in the C language is a simple syntax for calculating addresses. C is consequently one of the most error-prone programming languages ever created. When the Internet Worm brought the network to a halt in November 1988, the vulnerability of C software was vividly illustrated.

Arrays: The Balance Sheet



We can now express programmes in new ways thanks to references, and arrays provide us quick access to the hardware addressing system. However, neither fundamentally broadens the range of algorithms we can represent, and they make software more difficult to comprehend. As we did in Lesson 2, we can no longer speak of programme execution in terms of reduction. They also raise the cost of storage management. Therefore, it is advisable to limit their use.

ML offers immutable arrays without an update operation, known as vectors.

To exchange storage for runtime, utilise the Vector.tabulate operation. If the function requires a significant amount of computing, it is beneficial to create a table of function values.

The number of algorithms we can define grows as a result of input/output operations since they enable interacting with the outside world.

This table lists the primary array functions along with their kinds.

```
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
```

Learning guide. Pages 313–326 of ML for the Working Programmer include pertinent information. Pages 340–356 provide a quick overview of ML's extensive input/output capabilities, which are not addressed in this course.

Exercise 14.1 Comment on the differences between an int ref list and an int list ref using examples.

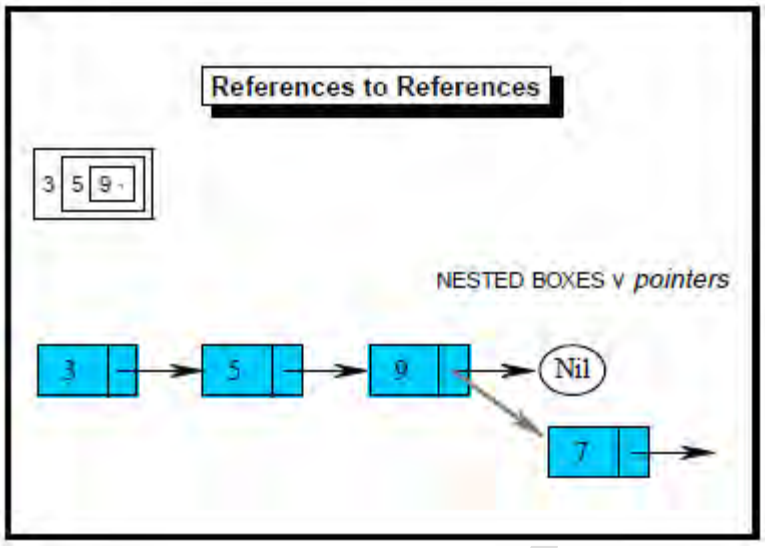
Exercise 14.2 Instead of utilising recursion, create a while-based version of the function power from Lesson 2.

Exercise 14.3 What effect does doing C2 when (C1; B) is doing C1 have?

Exercise 14.4 In ML, arrays of arrays are used to represent arrays with multiple dimensions. Create functions that (a) take an input of n and produce a n X n identity matrix, and (b) transpose a m X n matrix.

Exercise 14.5 For i = k,..., 1, the insert function duplicates elements from A[i1] to A[i]. What would happen if elements were copied from A[i] to A[i+1], where i = 0,...., k - 1?

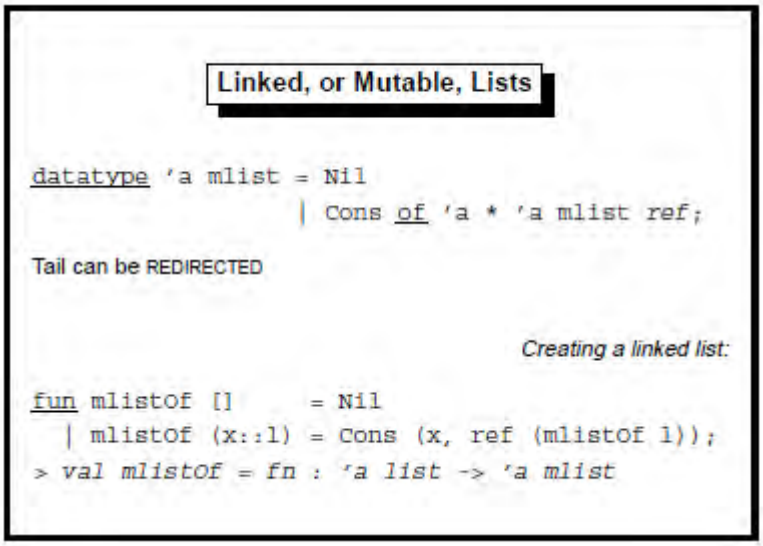
References to References:



References can be compared to boxes with movable contents.

However, the box metaphor breaks down when the contents of the box can also be another box because deep nesting is too challenging to manage. The pointer serves as a more adaptable metaphor. A reference is a pointer that can be transferred to any other object of the appropriate type. The final pointer to Nil in the list [3,5,9] is about to be redirected to a cell containing the value 7, which is against ML's built-in list policy. However, we can declare linked lists with modifiable link fields.

Linked, or Mutable, Lists:



Either empty (Nil) or made up of an element and a pointer to another changeable list, a mutable list can be modified. The declaration above would become exactly similar to built-in ML lists if the ref were removed. Links can be modified after they are created thanks to the reference in the tail.

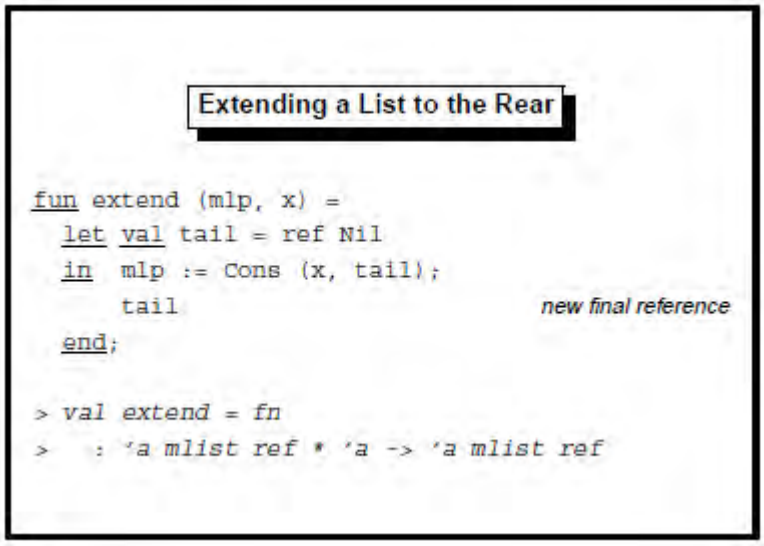
We can use types of the form 'a ref mlist to obtain pointers to the actual components. (In Lec. 14, we saw a type int ref list.) Therefore, the datatype definition does not require the addition of another ref.

The mlistOf function changes regular lists into mutable lists. For each entry of the new list, its call to ref creates a new reference cell.

For creating linked data structures, the majority of computer languages include reference types. Sometimes a predefined constant named NIL serves as the null reference, which always leads to nothing. While other (mutable) variables are allocated on the stack, reference cells are given space in a special area of storage called the heap by the run-time system. In contrast, ML handles every reference the same way.

A linked data structure that is equivalent to mlist is used internally to represent ML lists. The representation enables changing the links in an ML list. It is by design of ML to restrict such changes in order to promote functional programming. Lisp, a list-processing language, supports the modification of links.

Extending a List to the Rear:

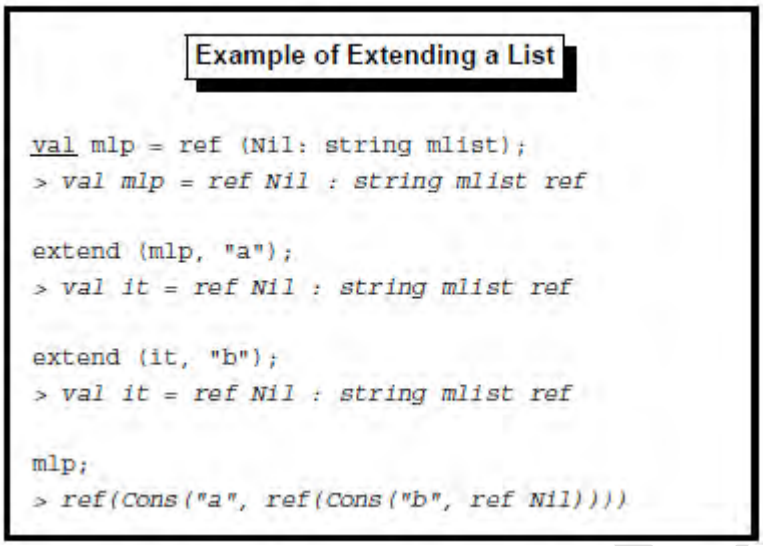


Ordinary ML lists are extremely expensive to extend to the rear because we have to evaluate an expression of the form xs@[x], which is O(n) in the size of xs.

We can maintain a pointer to the final reference using mutable lists. Update this pointer to a new list cell to make the list longer. For usage the following time the list is expanded, take note of the new final reference.

A reference mlp and an element x are passed to the extend function. The new reference is returned as its value after the assignment to mlp. It updates mlp to a list cell that contains x as a result.

Example of Extending a List:

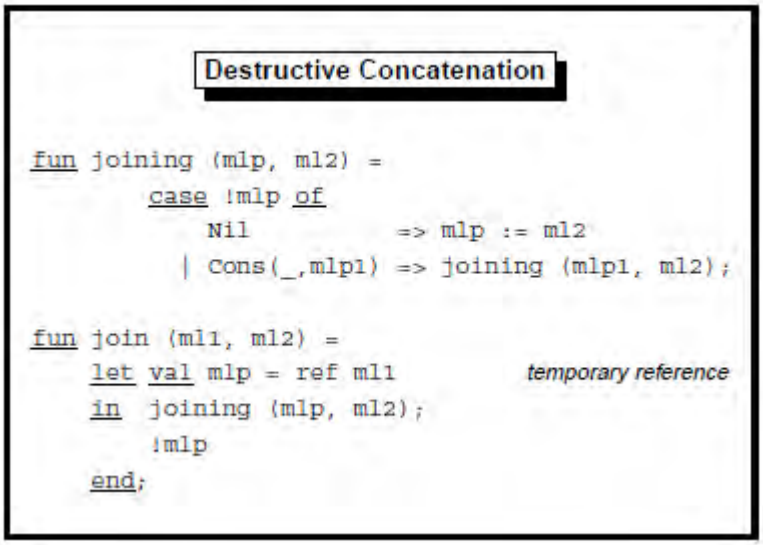


We begin by making a fresh pointer to Nil and binding it to mlp.

The items "a" and "b" are added by two calls to extend. As you can see, the first extend call receives mlp, while the second call receives the first call's outcome, in this case it.

We then look at mlp. It now points to the mutable list ["a","b"] rather than Nil.

Destructive Concatenation:



Concatenation is performed destructively by function join. It changes the end pointer of one changeable list from pointing to Nil to pointing to another list.

Comparatively speaking, a regular list append replicates its first parameter. Contrary to destructive concatenation, which simply requires constant space, append requires O(n) time and space equal to the size of the initial list.

The actual work is done by function joining. Its first argument is a pointer that should be followed until it may be changed to point to list ml2 when Nil is reached. The function scans the information in the reference mlp. The moment has come to update mlp to point to ml2 if it is Nil. However, the search is continued using the reference in the tail if it is a con.

A temporary reference to the function join's first argument is used to initiate the search. The joining test either updates the reference or skips down to the "proper" reference in the tail, saving us from having to test whether or not ml1 is Nil. Such techniques can be quite helpful when developing connected structures.

The types of the functions reveal that join takes two mutable lists and returns another one, whereas joining requires two mutable lists and can only return ().

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

Side Effects:

Side-Effects

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

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

join(ml1,ml2);

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

In this instance, the variables ml1 and ml2 are bound to the changeable lists ["a"] and ["b","c"]. We can conveniently read off the list elements in the data structures thanks to ML's reference value display technique.

The lists are then joined together using join. (There isn't enough area to show the returned value, but it is the same as the mutable list ["a","b","c"] at the bottom of the slide.)

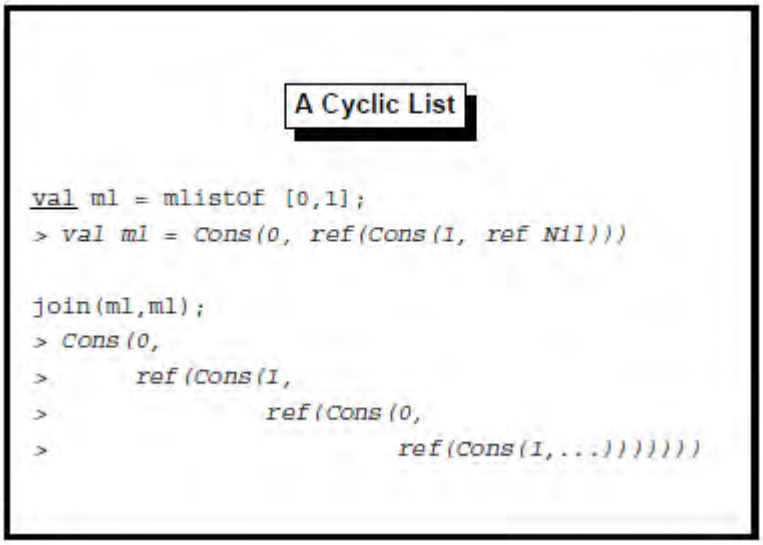
Finally, we look at ml1's value. Has it changed? It seems to have changed.

No, it is still the same reference. Reachable from it, the contents of a cell have altered.

The value of a list has been reinterpreted from ["a"] to ["a","b","c".

The internal link fields of ML's built-in lists are immutable, preventing this behaviour from happening. If we confuse join with add, the ability to change the list contained in ml1 might be desired but it also might come as an unwelcome surprise. The fact that join(ml2,ml3) also affects the list in ml1 is a further surprise. This is because it changes the last pointer of ml2, which is now also the last pointer of ml1.

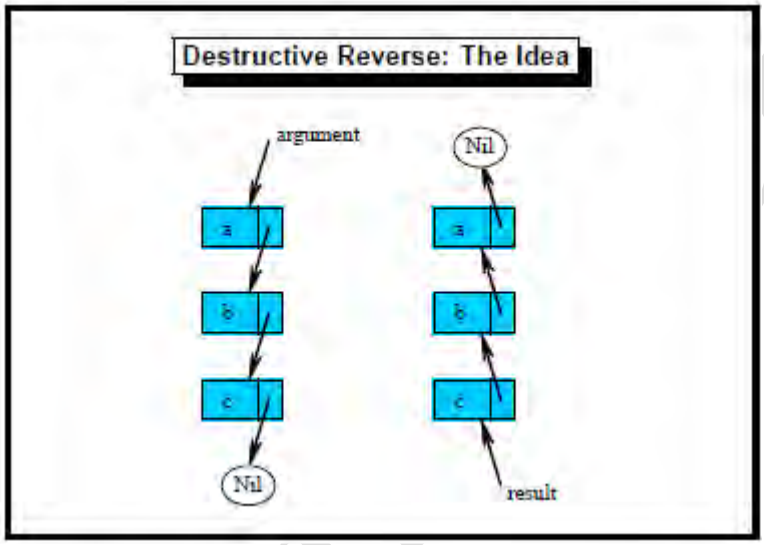
A Cyclic List:



What took place? The list ml is chased down to its final link, which is made to point to... ml, when join(ml,ml) is called.

We have a cycle if an item contains a pointer that points back to itself, possibly over numerous links. If it forms unexpectedly, a cyclic chain of pointers can be disastrous. Without using loops, cyclic data structures are challenging to explore and are particularly challenging to replicate. They obviously don't fit the metaphor of the box for references! There are applications for circular data structures. To fairly rotate over a limited number of options, use a circular list. A dependency graph illustrates the relationships between different elements, some of which may be cyclic.

Destructive Reverse: The Idea

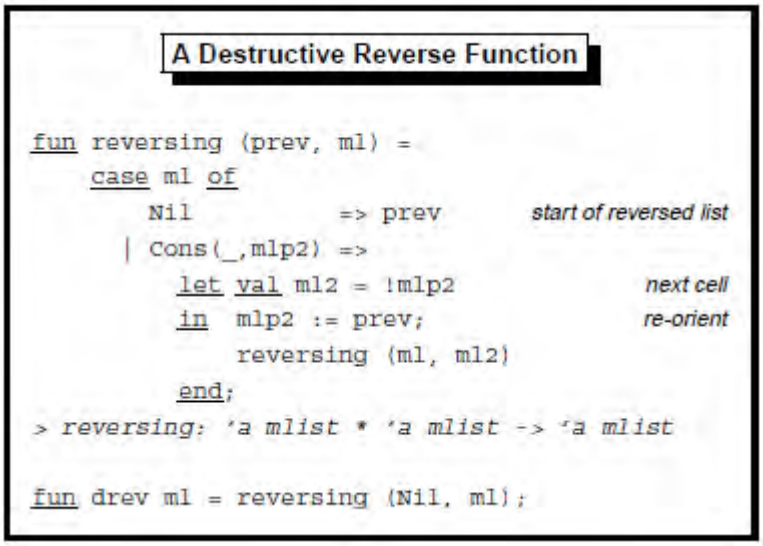


List reversal can be challenging to grasp from a first-principles perspective, but the code should be clear.

Reverse for standard lists duplicates the list cells while flipping the elements' positions. Destructive reverse repositions the links while using the preexisting list cells. The way it operates is to traverse the mutable list, note the previous two mutable lists found, and then change the link field in the second cell to point to the first. Since the last link of the reversed must point to Nil, the first changeable list is initially Nil.

Keep in mind that we must view the inverted list from the other end! A pointer to the list's top entry is the argument for the reversal function. A pointer to the final element of the original list, which is the first element of the reversed list, must be returned.

A Destructive Reverse Function:



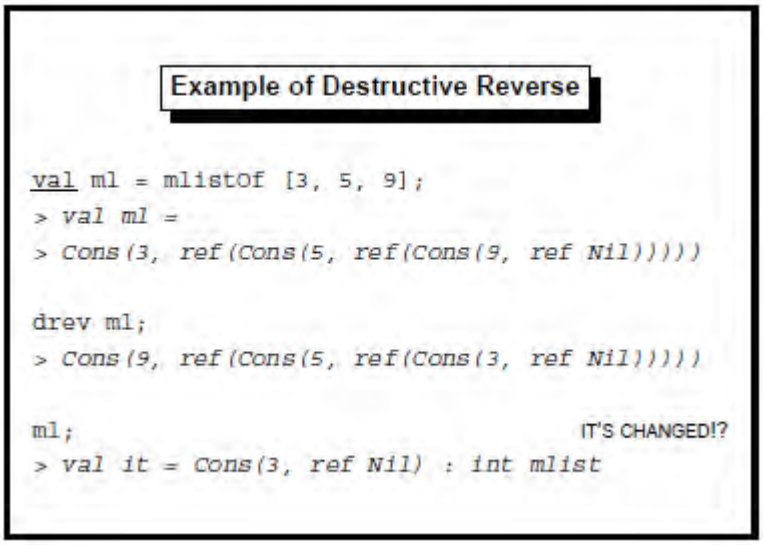
Reversing redirects points in the manner previously mentioned. Because the function is tail recursive and does not call ref (which would allocate storage), it just requires constant space. Since each pointer redirection is local and independent of other pointers, it can be performed in constant space.

No matter how large the list is, it is irrelevant.

Destructive list operations have the significant advantage of being space efficient. It needs to be weighed against the higher likelihood of programmer error. Although the aforementioned code appears straightforward, pointer redirections are much more challenging to write than practical list operations. The reduction model is irrelevant.

We must clearly consider the consequences of changing pointers instead than deriving function definitions from equations.

Example of Destructive Reverse:



The changeable list [3,5,9] in the aforementioned example is reversed to produce [9,5,3]. It could be unexpected how drev affects its argument ml. It now appears as the one-element list [3] because ml is the last cell in the list.

The obvious extension of the concepts discussed in this lecture to trees is possible. To offer more connection fields is another generalisation. Each node in a doubly-linked list points to both its predecessor and its successor. One can travel forward or backward in such a list by starting at a specific location. The pointer fields of two neighbouring nodes must be redirected in order to add or remove elements. A ring buffer may be used to refer to a doubly-linked list if it is also cyclic [12, page 331].

Links to the children of tree nodes are typically present. They occasionally have linkages to both of their parents, or links in both directions.

XV Computer Science Foundations 152

Learning guide. Pages 326–339 of ML for the Working Programmer include pertinent information.

Exercise 15.1 Make a clone of a mutable list function. Would you ever use it?

Exercise 15.2 What does ml1 (a list) contain once the declarations and commands below are entered at the top level? Describe this result.

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

Exercise 15.3 Replace recursion with while to code destructive reverse.

Exercise 15.4 Create a function that duplicates cyclic lists, producing new cyclic lists with identical members.

References

[1] Alfred V. Aho, John E. Hopcroft, and Je®rey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

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

[3] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Reprinted as Chapter 4 of Bell and Newell [2], first published in 1946.

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

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

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

[7] John Hughes. Why functional programming matters. Computer Journal, 32:98{107, 1989.

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

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

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

[11] 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.

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

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