

Foundations of Computer Science

Introduction to Programming

Abstraction and Abstraction Barrier:

Abstraction is the idea that large systems can only be understood in levels, with each level further subdivided into functions, with the higher level supplying the services. The idea is that you don't need to know about the lower levels to use something, only about the next highest level.

Recurring issues:

1. What services at each level
2. How to implement them using lower-level services
3. How to allow levels to communicate

Abstraction Barrier: Allowing one level of a system to be changed without affecting levels above it. When a chip manufacturer changes the processor, existing programs must continue to be able to run on them.

Date

This is an example of problem of ensuring that old formats of items continue to be useable. Programming languages like ML allow you to define the method of storing information, but the system must still allow for old formats to work.

Methods of storing:

1. *Abstract Level* – Dates over a certain interval
2. *Concrete Level* – Typically YYMMDD (with one byte for each digit).
 - a. However, this threw up the issue of the millennium crisis, when they could easily simply add another two digits.
 - b. However, already using 48 bits, which is enough for the entire lifetime of the universe.
3. *Digital's PDP-10*: Uses 12 bit dates
 - a. This only allows for dates for 11 years, therefore it doesn't work

Representational Abstraction and Datatypes

The idea of representational abstraction is that we invoke a type of an object without caring about how it is implemented in hardware.

Datatypes:

1. How a value is represented inside the computer
2. The suite of operations given to the programmers regarding the datatypes.
3. Valid and invalid (or exceptional) results.

Possible Inaccuracies

Floating point inaccuracies are common, with errors spiralling out of control since not all numbers are necessarily representable using the floating-point representation in hardware. (For this, a mantissa and an exponent are stored).

Precisions

Range of precisions are possible, for example 32 bits, 64 bits etc which defines the amount of memory space which would be given to that.

Goal of Programming

1. To describe a computation so that it can be done mechanically.
 - a. Expressions compute values
 - i. For describing mathematical formulae and suchlike.
 - ii. Original contribution of FORTRAN (Formula Translator)
 - b. Commands cause effects
 - i. Commands describe how control should flow from one part of the program to another.
2. To do so efficiently and correctly, giving the right answers quickly
3. To allow easy modification as needs change
 - a. Through orderly structure based on abstraction principles
 - i. Such as modules and classes (OOP)
 - Modules encapsulate a body of code, allowing outside access only through a programmer-defined interface.
 - **Abstract Datatypes:** Simpler version of this concept, which implement a single concept such as dates or floating-point numbers.

Why ML

1. Interactive
2. Flexible notion of data types.
3. **Hides the underlying hardware, ie no crashes.**
4. Programs can easily be understood mathematically.
5. Distinguishing naming something from updating memory (you have no idea what's happening in the memory).
6. Manages storage for us

ML Items

Value Declaration: Makes a name (**identifier**) stand for an item (such as `val pi = 3.14159;`)

Infix Operators: Infix operator is effectively a function which acts on two arguments which is defined as 'infix'. When defined as infix, the function name is placed between the two arguments – such as '`a * b`'

Functions: A method of encapsulated computation taking a number of inputs and returning an output (this output may be a unit – `()`)

Recursion: Recursion is the idea that a function calls itself, normally with different arguments and a base case at which the recursion stops.

Overloading: Overloading is the idea that the type of a function may be ambiguous, since a number of operators (`~`, `+`, `-`, `*`) and relations (`<`, `<=`, `>`, `>=`) are defined for both integers and reals. In this case, the type of the function can be defined using the following (`' : real'` for example) or otherwise the ML compiler defaults to an integer.

Boolean Expressions: Boolean Expressions are any expressions which return one of two values, either a 'true' or a 'false'. They are expressed using relational operators as well as

Boolean operators for negation (not), conjunction (andalso) or disjunction (orelse). **N.B. andalso and orelse work by stopping evaluating as soon as possible, i.e. if there is a 0 in the first argument of an andalso, it will stop (and same for a 1 in orelse).**

Recursion and Efficiency

Iterative Methods

A recursive function whose computation does not nest is called iterative or tail-recursive, by introducing another element, often known as the accumulator. This ensures that the space complexity is much reduced generally and therefore it is often better to use.

```
fun summing(n, total) =  
  if n=0 then total  
  else summing(n-1, n+total);  
  
fun summing (n) =  
  if n=0 then 0  
  else n + summing(n-1);
```

Recursion vs Iteration

NB: Iterative functions are functions which produce computations reflecting those which can be done using while-loops in conventional languages. All other 'iterative' functions are tail-recursive functions

- Tail-recursion is efficient only if the compiler detects it.
- Iterative functions save space and often run faster.
- However, it leads to functions having more arguments than are necessary.

Generally, write straightforward code first (often recursive) simply avoiding gross inefficiency and then consider how long it would take to run.

Time Complexities

Asymptotic Complexity: It refers to how costs – normally time or space – grow with increasing inputs (as the number of inputs go to infinity). **Space complexity can never exceed time complexity as it takes time to do anything with space.**

Notation:

1. **O Notation** defined by

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

where $f(n)$ is bounded by some constant c for all sufficiently large n .

2. Ω is a lower bound for a function asymptotically.
3. Θ is an exact bound – ie $\Omega(f(n)) = O(f(n)) = \Theta(f(n))$

O Notation

$O(2g(n)) = O(g(n))$
 $O(\log_{10} n) = O(\ln n)$ (since it is a scalar factor to convert between the two)
 $O(n^2 + 50n + 36) = O(n^2)$
 $O(n^2)$ is contained within $O(n^3)$
 $O(2^n)$ is contained within $O(2^n)$ – ie if $f(n) = O(2^n) \Rightarrow f(n) = O(3^n)$ by triviality but the reverse doesn't hold.
 $O(\log n)$ is contained within $O(n^{1/2})$

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

Recurrence Relations

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

Lists

A list is an ordered series of elements, with repetitions and ordering being significant. All elements must have the same type.

@ = append two lists

rev = reverses a list

Lists are **represented internally with a linked structure**, where adding a new element (Cons'ing) simply 'hooks' the new element to the front of the existing structure.

Nil or [] is the empty list

$x :: l$ is the list with head of x and the tail (containing the rest of the elements) is l

$::$ (the process of adding to the head of the list) is an **$O(1)$** operation.

Calculating Length

Both recursive and iterative are $O(n)$ space complexity, but iterative is only $O(1)$ space complexity, while recursive is $O(n)$ space complexity.

```

fun nlength(n, []) = n
|   nlength(n, x::xs) = length(n+1, xs);

fun length(xs) = nlength(0, xs);
  
```

List Concatenation (and reversal)

The append function is clearly an $O(n)$ function in terms of both space and time where n is the length of the first argument (the function is independent of the length of the second argument).

Therefore using the append function in another function, such as the badReverse adds a lot of time complexity, therefore making it $O(n^2)$ trivially in this case ($= 0 + 1 + 2 \dots + n = \frac{1}{2} n(n+1) = O(n^2)$). badReverse has a space complexity of $O(n)$ because copies of items don't exist at the same time.

goodReverse on the other hand makes use of the cons system and uses another item as an accumulator to make the process much more efficient ($O(n)$ in terms of both time and space)

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

fun badReverse [] = []
| badReverse (x :: xs) = badReverse(xs) @ [x]

fun goodReverse ([], ys) = ys
| goodReverse (x :: xs, ys) = goodReverse(xs, x::ys)
```

Take and Drop

```
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;
```

Both make use of the wildcard pattern ('_') which says that it could be anything and the pattern would be matched (in that place).

Both 'take' and 'drop' are $O(i)$ in time complexity and take is $O(i)$ in space complexity. Drop is $O(1)$ in space complexity.

Linear Search

Used when the list where the item to be found in is unordered and unindexed. It is $O(n)$. When the list is ordered, it take $O(\log n)$, while and indexed lookup takes $O(1)$.

Polymorphism

'a', 'b', etc (alpha, beta, etc) are used as type variables and can stand for any type. Code written using these functions is checked for type correctness at compile time. This guarantees strong properties at run time, such as all the elements of any list are the same type.

Equality Types

Equality types are types for which equality testing is allowed – these are not things like functions, where equality is impossible to test. It is even not allowed for reals, though some ML systems ignore this rule.

Abstract types can be declared in ML, hiding their internal representation, thereby the contents of the equality test can be changed so that two things can be tested to be equal.

Zip and Unzip

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

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);
```

Making Change

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

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

Strings and Characters

In ML, strings are an abstract concept in and of themselves, and aren't simply a list of characters and so they should not be treated as such.

Characters are similarly not strings of length one but are a primitive concept of their own. They have the form `#'c'` where `c` is any character. Special characters are coded using escape sequences using the backslash character.

Functions

- `explode(s)` – list of characters in string `s`
- `implode(l)` – string of characters in list `s`
- `size(s)` – length of string `s`
- `s1 ^ s2` – concatenation of strings `s1` and `s2`
- `s1 < s2` (similarly for `<=`, `>`, `>=`) – uses a lexicographic order with respect to ASCII character codes (alphabetic order) to determine if Boolean expression is true or false.

Sorting

Why sort?

1. Fast search
 - a. $O(\log n)$ instead of $O(n)$ to find an item using a binary search
2. Fast merges

3. Finding duplicates
4. Graphics algorithms

Random Number Generator

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

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

Optimal Speed of Sorting

$n!$ permutations of n elements and each comparison compares exactly two combinations. Therefore, the number of comparisons $C(n)$ can be said to be the following:

$$2^{C(n)} \geq n!$$
$$\text{Hence, } C(n) \geq \log(n!) \approx n \log(n) - 1.44n$$
$$\text{Therefore, } C(n) \geq n \log(n)$$

Therefore, sorting algorithms such as mergesort, which have time complexity of $O(n \log(n))$ have technically got the optimal time complexity.

Insertion Sort

Insertion Sort takes each item of the array and places it in the correct position in the list one at a time, therefore taking $O(n^2)$ comparisons asymptotically. It is therefore not at all efficient. It also takes up lots of space ($O(n)$).

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

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

Quicksort

Quicksort is much faster (also can work in-place, therefore having space complexity of $O(1)$) and works in the following way:

1. Choose a pivot element
2. Divide list into two sublists: those greater than (or equal) to the pivot and those less than the pivot
3. Recursively sort the sublists

- Combine the sorted lists by appending one to the other.

Obviously, it seems well suited to using appending of lists, however, we can also do it without appends, which would of course make the system more efficient.

In terms of specific time complexity, we can consider both the average case and the worst-case complexity. In the **average case**, the pivot will fall in the middle of the list, therefore making the problem of this sort:

$$T(n) = 2T\left(\frac{1}{2}n\right) + n, \quad T(1) = 1$$

This can be followed down to:

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

However, in the worst case, where the item is either reverse sorted or almost sorted. In this case, nearly all the elements fall on one side of the pivot, therefore:

$$T(n) = T(n-1) + n, \quad T(1) = 1$$

This becomes:

$$T(n) = O(n^2)$$

However, we can attempt to avoid this by simply randomising the order of inputs before attempting to sort them, therefore reducing the likelihood that they are sorted.

```
fun quickSortWithAppend [] = []
| quickSortWithAppend [x] = [x] (* This singleton makes the code 20% faster *)
| quickSortWithAppend (a :: bs) =
  let fun partition (l, r, []) : real list = quick (l) @ (a :: quick (r))
      | partition (l, r, x :: xs) = if (x <= a) then partition (x::l, r, xs)
                                   else partition(l, x::r, xs)
  in partition([], [], bs)
  end;

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

Merging and Merge Sort

Merging two lists takes at most $O(m+n)$ where m and n are the length of the two lists.

Merge Sort takes the idea of splitting the list into two parts, sorting each one (recursively) and then merging the two lists together.

This has time complexity as:

$$T(n) = 2T\left(\frac{1}{2}n\right) + n, \quad T(1) = 1$$

This becomes:

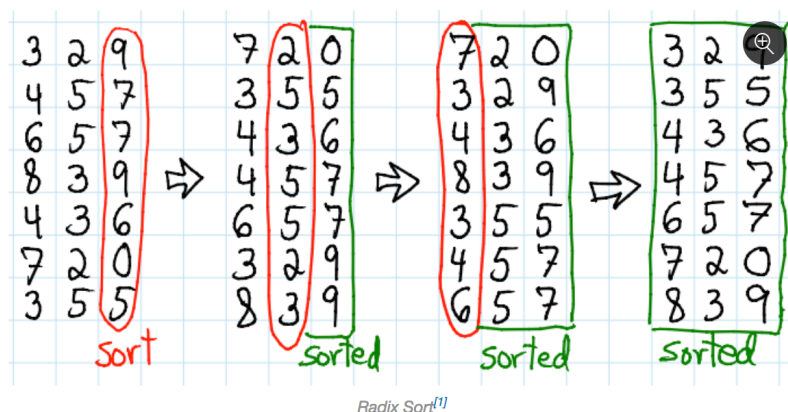
$$T(n) = O(n \log(n))$$

Though this is the same as the average case as quicksort, this is often actually much slower than quicksort, however is safe, always taking the same amount of time whereas merge sort can take quadratic time.

We consider only the top-down merge sort (where we split them into subsequently smaller lists), however, the bottom-up merge sort system also exists where we start with a list of one-element lists and repeatedly merge lists until we only have one list.

Radix Sort

This is a sorting system by which we look at the representation of an integer, therefore not completing any comparisons between the numbers themselves, just parts of them. For example:



You first sort the LSB, followed by the next significant bit and so on until the entire thing is sorted.

The time complexity of this is (where d is the number of digits of the numbers and b is the number of different values that can be represented by one digit):

$$O(d(n + b))$$

For small integers, where d is a small constant, it becomes $O(bn)$ which is equivalent to $O(n)$.

Datatypes and Trees

Declaring Datatypes

You can define an **enumeration type**, where a datatype can be one of a number of identifiers, as below. This is better than using an integer to represent them as it is far more flexible, allowing people to add items easily without changing the numbers that are used. Additionally, better than using a string (the identifier), because then the code would not work in the case of mistypes, but would likely return no error. **You can do pattern matching on enumeration types in functions, such as the function below.**

The four identifiers that a vehicle could be are each a constructor of the object of that type.

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

```
fun wheels Bike = 2
|   wheels Motorbike = 2
|   wheels Car = 4
|   wheels Lorry = 18;
```

As the identifiers are constructors, ML allows data to be associated with each constructor, eg:

```
datatype vehicle2 = Bike
                  | Motorbike of int
                  | Car of bool
                  | Lorry of real;
```

Here vehicle2 is an example of a concept consisting of several varieties with distinct features. In other programming languages, these are often represented by things close to datatypes, often called **union types** or **variant records** (where **tag fields** are effectively the constructor).

Exceptions

Exceptions are necessary because it is not always possible to tell in advance whether or not an error will occur. Rather than crashing, programs should check whether things have gone wrong and attempt an alternative computation.

Exception handling allows us to recover properly:

1. **Raising an exception** abandons the current computation
2. **Handling the exception** attempts an alternative computation
3. **Raising and handling can be far apart in code**
4. **Errors of different sorts can be handled separately**

In ML, before an exception can be used, it must be declared, as follows:

```
exception Failure;
exception NoChange of int;
```

Exception names are constructors of the special datatype **exn**. **This lets exception handlers use pattern matching.**

In order to raise and handle exceptions, the following can be used:

```
raise Failure;
raise (NoChange 10)

handle Failure => E (* E is a function that should be called *)
handle NoChange (n) => return n
```

N.B. Unlike in Java there is no indication that a function can throw an exception. One alternative to exceptions is to return a value of a new datatype

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

Where NONE represents than an error and SOME x is the solution x. While clean, this would be very annoying, needing to check for NONE in a lot of places (not just being able to handle it).

Binary Trees

A tree is a data structure with multiple branching, providing efficient storage and retrieval of information. In a binary tree, a node is either empty (leaf node) or is a branch with a label and two subtrees.

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

Counting number of nodes

```
fun count Lf = 0
|   count Br (_, tl, tr) = 1 + count tl + count tr;
```

Finding depth of tree

```
fun depth Lf = 0
|   depth Br (_, tl, tr) = 1 + Int.max(depth tl, depth tr);
```

Dictionaries and Functional Arrays

Dictionary Idea

A dictionary attaches values to identifiers (known as the key). The key is unique and you should be able to locate the value using the key in a dictionary data structure.

A dictionary is an example of an abstract data type given it provides specified operations while hiding low level details of how exactly the operations work and how the data is stored.

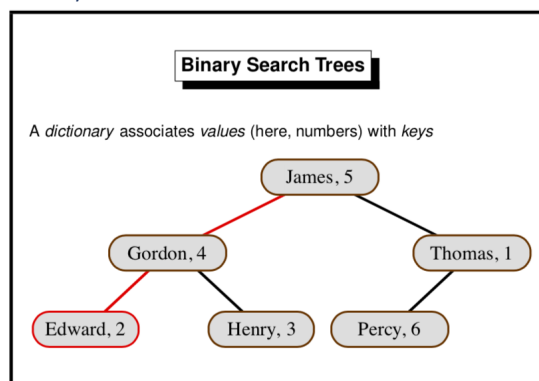
For a dictionary, the following operations must be provided:

1. **Lookup**: find an item in the dictionary
2. **Update (insert)**: replace an item in the dictionary
3. **Delete**: remove an item from the dictionary
4. **Empty**: have a null dictionary
5. **Missing Exception**: when there are errors in lookup and delete

Association List

The simplest way of working would be with an association list – a list of two-tuples. Lookup is slow, taking $O(n)$, however they are often used if there is no way to order, simply having an equality in the type. To add a new item to the list, a new tuple must simply be 'cons'ed to the list – this takes constant time. However, the update and lookup times mean that this is generally not used.

Binary Search Trees



Each branch of the tree carries a (key, value) pair with the left subtree containing items with smaller keys and the right subtree containing items with larger keys. If the tree is reasonably balanced, then update and lookup both take $O(\log n)$ for a tree of size n . Meanwhile, an unbalanced tree has a linear ($O(n)$) access time in the worst case.

In addition to being used for dictionaries, binary search trees are also used for sorting – adding all elements to a tree, before doing an in-order traversal to get the items in order.

There are also self-balancing trees, called Red-Black trees, which are $O(\log n)$ for access in the worst case. However, these are generally rather hard to implement.

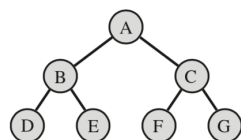
```
exception Missing of string;

fun lookup (Lf, b) = raise Missing b;
| lookup (Br((a,x), tl, tr), b) = if (b < a) then lookup(tl, b)
                                else if (b > a) then lookup(tr, b)
                                else x;

fun update (Lf, b : string, y) = Br ((b, y), Lf, Lf)
| update (Br((a,x), tl, tr)) = if (b < a) then Br((a,x), update(tl, b, y),
tr)
                                else if (b > a) then Br((a,x, tl, update(tr,
b, y)))
                                else Br((a,y), tl, tr);
```

Since (for update) the paths are simply copied, the parts of the tree are shared, not copied.

Traversing Trees



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

Uses

- Preorder is often used to copy the tree.
- Inorder is used for sorting, getting the tree in the correct order.
- Postorder converts from infix to Reverse Polish Notation, used by compilers.

N.B. All these types of tree traversal are depth first traversals. They each traverse the left subtree before traversing the right one.

```
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));

```

Intersection of two Binary Trees

```

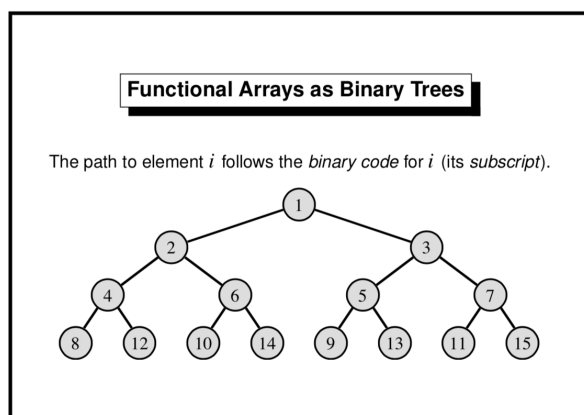
fun intersection (Lf, t2, out) = out
| intersection (Br((a,x), tl, tr), t2, out) =
  let val newTree = intersection(tl, t2, (intersection(tr,t2, out )))
  in
    if (lookup(u,a) = x) then update(newTree, a, x) else newTree
    handle Missing _ => newTree
  end;

```

Functional Arrays

A conventional array is an indexed storage area and is updated in place by the command $A[k] := x$

A functional array is a finite map from integers to data, where updating implies copying with every other item of the array equalling what it was before, except the item to be changed has now been updated.



In a functional array, items are stored in the position of a binary tree according to the diagram above, ie $A[2]$ will be the left label of the first branch.

As can be seen, the lower bound for array indices is 1; the upper bound is 0 (this represents an empty array) and can grow without limit.

Since the tree is clearly always balanced, the access time for updating or adding is $O(\log n)$

```

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);

```

```

fun update(Lf, k, w) = if k = 1 then Br(w, Lf, Lf)
                      else raise Subscript
|   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, w), t2)
                                    else Br(v, t1, update(t2, k, w));

```

Dictionary Methods

Linear Search	More general, only needs equality on keys but inefficient – linear time
Binary Search	Needs an ordering on keys. Logarithmic access time in the average case but linear in worst case.
Array Subscripting	Least general, requiring keys to be integers, but even worst-case time is logarithmic.

Functions

In ML, functions can be passed as arguments to other functions, returned as results, put into lists, trees, etc, but **cannot** be tested for equality.

They don't necessarily need names, the function can be referred to as:

`Fn x => f(x)`

You can also have pattern matching in this definition, for example:

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

Currying

Currying is the technique of expressing a function taking multiple arguments as nested functions, each taking a single argument. You can call the main function and fill in the first argument(s) and receive another function which has the other arguments as the arguments to the function. This is known as **partial application** when fixing the first argument(s) yields a useful function in and of its own. An example:

```

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

insert(op<=) [5,3,9,8]; (* Returns [3,5,8,9] *)
insert(op<=) ["bitten","on","a","bee"]; (* Returns them alphabetically ordered *)
insert (op>=) (* Returns a function to sort descending *)

```

Map

The functional map applies a function on every element of a list, returning a list of the function's result, effectively this (though it is a built in ML function):

```
fun map f [] = []  
|   map f (x :: xs) = (f x) :: map f xs;
```

Example 1: Transpose a Matrix

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

Example 2: Matrix Multiplication (lookup if unsure how it works – page 92)

```
fun dotprod [] = 0.0  
|   dotprod(x :: xs, y :: ys) = x*y + dotprod(xs, ys);  
  
fun matprod(Arows, Brows) =  
  let val cols = transp Brows  
  in map(fn row => map (dotprod row) cols) Arows  
  end;
```

Predicate Functionals

Exists

Transforms a predicate into a predicate over a list. Given a list, exists p checks whether the list has some element which satisfies p (making it return true immediately).

```
fun exists p [] = false  
|   exists p (x::xs) = (p x) orelse (exists p xs);
```

Filter

Applies a predicate to all list elements of a list and returns a list of the elements which satisfy the predicate.

```
fun filter p [] = []  
|   filter p (x::xs) = if (p x) then x :: (filter p xs)  
                       else filter p xs;
```

All

Functional to test whether all elements of a list satisfy a predicate.

```
fun all p [] = true  
|   all p (x :: xs) = (p x) andalso all p xs;
```

Examples:

```
fun member(y, xs) = exists (fn x => x=y) xs;  
fun inter(xs, ys) = filter (fn x => member(x,ys)) xs;  
fun disjoint(xs, ys) = all (fn x => all (fn y => x<>y) ys) xs;
```

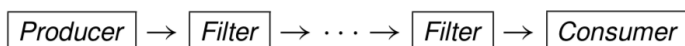
Lazy Lists

Pipeline

There are two types of program:

- **Sequential Program**
 - Accepts problem to solve, processes then terminates with the result.
 - This is most of the code that is written.
- **Reactive Programs**
 - Interacts with the environment, communicating constantly, running for as long as is necessary.
 - For example, the software controlling an airplane.
 - Often consists of concurrent processes running at the same time and communicating with one another.

Lazy Lists allow us to start to consider a reactive program pipeline, where the program receives more data upon command (in this case upon command by the user).



Pipeline consists of:

- **Produce** sequence of items
- **Filter** sequence in stages
- **Consume** results as needed

Lazy Lists join the stages together

Definition

Lazy Lists are lists of possibly infinite length with elements computed upon demand. This avoids waste if there are many solutions as a defined number of solutions can easily be found. In ML, you implement laziness by having a tail which is delayed in its evaluation until something else happens – the tail function is called.

Also, infinite objects are generally a useful abstraction for a system to utilise a problem which could have an unbounded number of results.

Implementation

In ML, Lazy Lists (or sequences) are implemented as the following:

```

datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq);
fun head (Cons(x, _)) = x;
fun tail (Cons(_, xf)) = xf();
  
```

Therefore a sequence item is made of the item itself and a function (taking 'no' argument) and getting the next item. The next item is not calculated until the tail function is called.

Example: Infinite Sequence of integers

```

fun from k = Cons(k, fn()=> from(k+1));
from 1;
  
```

Convert to List

Since it could technically be an infinite list, we must instead of converting the entire sequence, only convert the first n elements of the sequence, hence:

```

fun get (0, xq) = []
  
```



```
| get (n, Nil) = []
| get (n, Cons(x, xf)) = x :: get(n-1, xf());
```

Joining

While we could technically append one sequence onto another as below, it makes no sense at all, just because the first sequence that we are attempting to get through has the potential to be infinitely long. Therefore, we instead choose to interleave normally, which means taking one element from each sequence in turn:

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

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

Functionals

Filter

Filter demands elements from the sequence until it finds one which satisfies the predicate.

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

Iterates

Iterates generalizes 'from', creating the next element by calling function f. It is also the sequential equivalent of map.

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

Numerical Computations

Lazy Lists can easily be used to complete mathematical functions, for example finding the square root of a number using the Newton-Raphson approximation.

Example: Square Root

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

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

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

Queues and Search Strategies

Breadth-First vs Depth-First Tree Traversal

In the case of searching, we can represent all the data in a binary tree (using it as the **decision tree**), while we are looking for solution nodes.

In the case of a depth first search, we search one subtree in full before moving on and searching through the rest of the tree. If simply a solution (of non-caring depth) is required to be found, then a depth-first search is generally effective, since it is so easy to code.

In the case of the breadth first search, we search every node at a certain level (depth) before moving onto the next depth. It will always generate the shortest path to the solution (if this is relevant).

Breadth-First Traversal

Using Append

Breadth-first searches are often not very efficient, but it is especially bad when you use an append, as below. This is because the list 'ts' can be very long, containing all of the items that need to be visited. Therefore, appending only two items to it (which requires copying everything in ts) would be very inefficient.

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

Queues

BFS becomes much faster if we replace lists by queues. Queues are a sequence, allowing elements to be taken from the head and added to the tail. This is FIFO discipline (First-In-First-Out). While they could be implemented using lists (and append) this would be highly inefficient. It would be better off implementing them with a functional array as long as we have a function to delete the first element. Here, each operation would take $O(\log n)$ time. Conventional programming represents a queue using an array with two indices to point to the beginning and end of the queue which may wrap around the end of the array. However, this is difficult to implement and the length must have an upper bound.

However, using two lists, you can get a representation of queues, which when take $O(1)$ time when **amortized (averaged over the lifetime of the queue)**.

You have a pair of lists to represent the queue, adding items to the rear list. You remove items from the front list and if the front is empty, you move items from the rear to the front (reverse the rear list and make it the front list).

A queue must offer the following:

- Qempty – empty queue
- Qnull – tests whether queue is empty
- Qhd – returns the element at the head of the queue
- Deq – discards the element at the head of the queue
- Enq – adds an element to the rear of the queue

The code is therefore as follows:

```
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));

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

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

```

Looking at the time-complexity of queues, for each element which is enqueued to the queue (and then subsequently dequeued), it takes 1 cons to add it to the rear list and 1 cons to move it to front list whenever it is required. Therefore it is $O(2)$ which is equivalent to $O(1)$.

However, the time when the queue must be normalised is determined at run time and so there may be unpredictable delays. Therefore, this approach is unsuitable for real-time programming.

Case Expression

Case can be used for pattern matching, as per this example:

```

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

```

BFS Code

```

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

```

Iterative Deepening

In some places, a BFS is not practical for large trees, taking up far too much space, as large parts of the tree have to be stored. A BFS search examines the number of nodes = the below where b is the branching factor and d is the depth:

$$1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$$

Since all the nodes that are examined are also stored, the space requirements are also equal to $O(b^d)$.

Depth-first iterative deepening combines the space efficiency of depth-first searching with the 'nearest-first' property of breadth-first searching. It performs repeated depth-first searching with increasing depth bounds, each time discarding the result of the previous search. Therefore, it searches to depth 1, then depth 2 etc.

It can be shown that the time needed for iterative deepening to reach depth d is only $\frac{b}{b-1}$ times that for the time required for a breadth first search. This is a constant factor – both have time complexity = $O(b^d)$, but iterative deepening has space complexity of $O(d)$.

Depth-First Search

Stacks

Stack is a sequence such that items can be added or removed from the head only, obeying a LIFO (Last-In-First-Out) discipline. Lists can easily be used to implement stacks. However, stacks are often regarded as an imperative data structure (popping or pushing should affect the original stack not return a new one).

In conventional programming languages, a stack is often implemented by storing items in an array, using a stack pointer to count them.

Stacks must have the following things:

- Empty – ability to have an empty stack
- Null – ability to test for an empty stack
- Top – ability to return the item at the top of stack
- Pop – ability to remove the item at the top of the stack
- Push – ability to add an element to the top of the stack

Search Methods Conclusion

1. DFS – use a stack
 - a. It is efficient but incomplete – does not always return the best solution
2. BFS – use a queue
 - a. Effective but uses too much space
3. Iterative deepening – effectively uses (1) to get the benefits of (2)
 - a. It trades time for space
4. **Best-First – uses a priority queue**
 - a. Nodes are an ordered sequence, placing a ranking function to the nodes. For example, may be to estimate the distance from the node to a solution. If the estimate is good, the solution is located quickly.
 - b. The priority queue can be kept as a sorted list, although this is slow. Binary search tree would be better.

Polynomial Arithmetic

A polynomial is a linear combination of products of certain variables. Polynomials in one variable is called univariate. For the section, we only consider univariate closed-form polynomials. Being able to do maths on polynomials is highly useful, as it would allow us to derive and use formulas for science and engineering.

Data Structure for Polynomials

We might represent a polynomial with a dense representation – using a list of coefficients $[a_n, a_{n-1}, \dots, a_0]$ where the formula is $a_n x^n + a_{n-1} x^{n-1}$. This is very inefficient if many coefficients are zero.

Therefore, better to use a sparse representation, where a list of (exponent, coefficient) pairs with only non-zero coefficients being stored. For the coefficient, while it should be a rational number (stored as an integer * integer (a/b)), it will be stored as a real, because this makes it much easier to complete.

Therefore, polynomial will have type of (int*real) list, representing the sum of each term. Additionally, to promote efficiency, the pairs will be stored in descending order of exponents, with only one term having each exponent.

Polynomial Operations

In specifying a module for polynomial operations, eg Poly, you must define the functions and the ML signature of each function. They may be:

- Poly is the type of univariate polynomials
- Makepoly makes a polynomial from a list
- Destpoly returns a polynomial as a list
- Polysum adds two polynomials
- Polyprod multiplies two polynomials
- Polyquorem computes a quotient and a remainder

ML Signature

```
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
```

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 (m < n) then (n,b) :: polysum ((m,a)::ts) us
  else if a+b = 0.0 then polysum ts us
  else (m, a+b) :: polysum ts us;
```

Multiplication

Do multiplication in a very merge-sort like method, computing products of roughly equal parts and then merges them together.

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

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

Division

```

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

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

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

```

GCD

```

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

```

This is using Euclid's algorithm, however, for polynomials its behaviour is slightly odd. It gives the GCD of $x^2 + 2x + 1$ and $x^2 - 1$ as $-2x - 2$ and $x^2 + 2x + 1$ and $x^5 + 1$ as $5x + 5$. Both should have the answer $x + 1$. However, this difficulty can be fixed by dividing through by the leading coefficient.

However, more importantly, the algorithm is far too slow, and therefore this algorithm shouldn't be used.

Procedural Programming

Definition

- Procedural programming is programming where the programming state is repeatedly transformed by the execution of commands or statements.
 - A state change might be local to the machine and consist of updating a variable or array, or consist of sending data to the outside world. (Even reading data counts as a state change, since this act normally removes the data from the environment).
- It also provides primitive commands and control structures for combining them.
 - Primitive commands include **assignment** for updating variables and **input / output** commands for communication
 - Control commands include branching, iteration and procedures.
- Use abstractions of the computer's memory
 - **References** to memory cells
 - **Arrays** for blocks of memory cells
 - **Linked Structures**; especially linked lists

References

References offer a way of creating a reference, thereby getting at a specific memory cell which prevails.

The function, `ref` creates a reference (a location), allocating a new location in memory. It initially contains the value given by the expression `E`. Though an ML function, it is not a mathematical function – same input will likely give different output, ie `ref(0) = ref(0)` will return false.

The function `!` dereferences a reference, returning its contents.

The assignment function (returns unit) `P:=E` evaluates expression `P`, which must return a reference `p`. It stores at address `p`, the value of `E`.

`τ ref` *type of references to type τ*

`ref E` *create a reference*
 initial contents = the value of E

`! P` *return the current contents of reference P*

`P := E` *update the contents of P to the value of E*

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

N.B.1 assignment will never change val bindings, they are immutable. Only the contents of the reference are mutable.

N.B.2 most languages do not have an explicit dereferencing operator (like `!`) because of its inconvenience. Instead, by convention, occurrences of the reference on the left hand side of the `:=` denote locations and those on the right denote the contents. (Sometimes there is a special ‘address of’ available to override the convention.

Commands

A command refers to an expression that has an effect on the state. All expressions denote some value, but they can return `()`, which conveys no information.

The construct `C1; C2; ...; Cn` evaluates the expressions `C1` to `Cn` in the order given and returns the value of `Cn`. The values of the other expressions are discarded; their only purpose is to change the state.

Iteration

ML’s only looping construct is `while`, which returns the value `()`. The construct works like ‘while `B` do `C`’, where `B` is a Boolean expression and `C` is series of commands with a return value (this can be a unit).

Example: Bank Account

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

> val makeAccount = fn : int -> (int -> int)
```

The function `makeAccount` models a bank. Calling the function with an initial balance creates a new reference (balance) to maintain the account balance and returns a function (`withdraw`) having sole access to that reference.

Each call to `makeAccount` returns a copy of `withdraw` holding a fresh instance of the reference balance. There is no access to the account balance except via the corresponding `withdraw` function. If that function is discarded, the reference cell becomes unreachable – the computer will eventually get rid of it.

We can generalise `makeAccount` to return several functions that jointly manage information held in shared references. (The functions might be packaged using ML records, which are discussed elsewhere. Most procedural languages do not support the concept of private references, although OOP has it as a basic theme).

Arrays

ML arrays are like references that hold several elements instead of one. The primitives are as follows:

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

And countless others!

This has ML signature of:

<code>Array.array</code>	<code>int * 'a -> 'a Array.array</code>
<code>Array.tabulate</code>	<code>int * (int -> 'a) -> 'a Array.array</code>
<code>Array.sub</code>	<code>'a Array.array * int -> 'a</code>
<code>Array.update</code>	<code>'a Array.array * int * 'a -> unit</code>

Other things in ML

Mutable (linked) lists are easy to create:

<code>datatype 'a mlist = Nil Cons of 'a * 'a mlist ref;</code>

ML's system of **Modules** includes **structures**, which can be seen as encapsulated groups of **declarations and signatures**, which are specifications of structures listing the **name and type of each component**. Finally, there are **functors**, which are analogous to functions that combine a number of argument structures, and which can be used to plug program components together.

ML also provides comprehensive input / output primitives for various types of file and operating system.