Part IA Foundations of Computer Science Notes

1 Recursion and efficiency

- In normal recursion the stack is built up, reducing efficiency
- Adding another **accumulator** argument can make a function **tail recursive** or **iterative**.
  - Tail recursion is only efficient if the compiler detects it
  - Mainly saves space, although it can run faster
- Space complexity never exceeds time complexity, it takes time to do anything with space
- Simple Recurrence Relations:

$$\begin{array}{rcl|r} T(n+1) & = & T(n)+1 & O(n) \\ T(n+1) & = & T(n)+n & O(n^2) \\ T(n) & = & T(n/2)+1 & O(log\, n) \\ T(n) & = & 2T(n/2)+n & O(n\, log\, n) \end{array}$$

- Pattern matching can be done in a function using the `case` keyword, eg.

```
case E0 of Pat1 => E1 | ... | Patn => En
```

2 Lists

- Lists are ordered series of elements, in which receptions are significant
- Built in functions are:
  - Append (@)
  - Reverse (rev)
  - Cons (::)
- A function can be polymorphic, this allows flexibility in the types of its arguments
- **Type variables** can stand for any type, this are denoted `'a`, `'b`, etc.
- **Equality type variables** can stand for any type that can be equality tested
  - Are denoted by `''a`, `''b`, etc.
  - Functions and abstract types can bot be equality tested
  - Technically reals can not be equality tested, but a lot of system ignore this
- **Reduction in strength** - When expensive operations are replaced with equivalent but less expensive operations
- The functions `take` and `drop` divide a list into two parts
- The function `member` tests if an element is within a list
- The `zip` function pairs up corresponding elements in a pair of lists, discarding surplus
- The `unzip` function inverts the operation of `zip`

- The `interleave` function combines two lists, eg.

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

3 Strings and Characters

- Characters are represented using a hash symbol, eg. `#"A"`
- A string represented using quotation marks eg. `"Test"`
- The `explode(s)` function converts a string into a list of characters
- The `implode(l)` function converts a list of characters into a string
- The `size(s)` function gives the number of characters in a string
- A carat (^) is used to concatenate two strings

4 Sorting

4.1 Insertion Sort

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

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

- Repeated inserts elements in the correct position into a sorted list
- Requires $O(n^2)$ comparisons

4.2 Qicksort

```
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;
```

- Uses divide and conquer
- Chose a pivot a and partition into two lists, one $\leq a$ and one $> a$
- On average has a complexity of $O(nlog\,n)$, but has a worst can of $O(n^2)$

4.3 Merge Sort

- Two sorted list can be combined to produce an ordered list, with a complexity of $O(n+m)$, where n and m are the lengths of the two lists
- The `merge` function is used

```
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);
```

- Merge sort divides the input into two roughly equal parts, sorts them recursively, and then merges them

```
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;
```

- Merge sort has a worst case of $O(nlog\,n)$
- Merge sort has a large constant factor, making it slower then quicksort
- Could be improved by passing down the length of the new lists each time

5 Datatypes and Trees

5.1 User Datatypes

- User datatypes can be declared in ML, using the `datatype` keyword
- These can be treated as enumeration types or have data associated with them
- If there is data associated with the type then a constructor is declared
- User datatypes can be pattern matched

5.2 Functional Queues

- FIFO - First-In-First-Out
- Has amortized time per operation of $O(1)$
- Represented by a pair of lists
  - Add new items to rear list
  - Remove items from front list
  - If front list empty move rear to front
- Defined as follows

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

- The function `norm` is used to normalise the queue, it is called every time something is added or removed from the queue, eg.

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

5.3 Trees

- A data structure with multiple branching is called a tree
- A binary tree is a tree with each node with empty (`Lf`) or is a branch (`Br`) with two children
- A binary tree is declared as follows

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

- Trees can be traversed in three different ways
  - **Preorder** - Visits the label first, Polish Notation

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

  - **Inorder** - Visits the label midway, the sorted list

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

  - **Postorder** - Visits the label last, Reverse Polish Notation

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

- Best-first searching uses a priority queue, and orders nodes by some ranking function
  - Can use a sorted list, but slow
  - Better to use a binary search tree
  - Fancier data structures improve on both

5.3.1 Breadth-First Tree Traversal

- Can be done using a list and append - but inefficient
- Instead use a queue

```
fun breadth q =
     if qnull q then []
     else
     case ghd q of
        Lf => breadth (deq q)
      | Br (v, l, r) => v::breadth (enq (enq (deq q, l), r));
```

- Unpractical for infinite trees, it uses too much space
- At a depth d in a tree with a branching factor of b, breadth-first search examines $\frac{b^{d+1}-1}{b-1}$ nodes, which is $O(b^d)$

5.3.2 Iterative Deepening

- Combines the space efficiency of depth-first with the nearest-first property of breadth-first search
- It performs depth-first search with increasing depth bounds, ie. examining all at depth 1, then 2
- Discards the result of previous searches

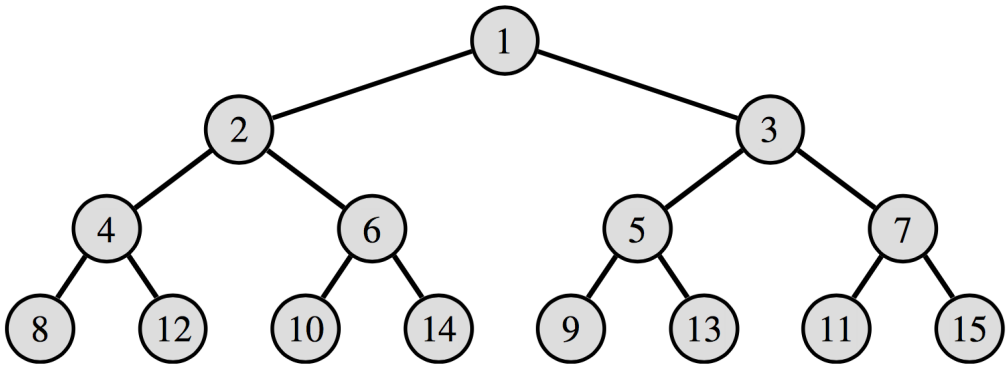
5.4 Dictionaries

- A dictionary is a collection of keys mapped to values
- They have the following functions
  - Lookup - Find an item in the dictionary
  - Update (insert) - Replace (store) an item in the dictionary
  - Delete - Remove an item from the dictionary
  - Empty - The null dictionary
  - Missing - Exception for errors in lookup and delete
- Can be achieved using a list, but inefficient
- Binary search trees are a more efficient way of implementing this
  - For any given node all keys in the left subtree are less than or equal to the node's key and all those in the right subtree are greater than
  - Worst case lookup of $O(n)$, for unbalanced tree
  - The update function only changes the tree along the path from root to insertion point, the rest is unchanged and shared in memory
  - The following update function is used

```
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 Br ((a ,y), t1, t2);
```

5.5 Functional Arrays

- A functional array is a finite map from integers to data
- Each node in a binary tree is labelled with a positive integer, with 1 at the root



- The tree is always balanced because of how elements are inserted into it
- The lookup function divides the subscript by two until 1 is reached. If the remainder is 0 then the function follows the left subtree, otherwise the right
- If the function reaches are leaf, it raises an exception

- The `sub` function is as follows

```
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);
```

- Subscript can be thought of in binary
  - Subscript must be positive and therefore have a leading one, discard this
  - Reverse the remaining bits
  - Interpret zero as left and one as right, to give the path from root to the subscript
- The `update` function updates the element at that subscript, adding it to the end of the array if required

```
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 div 2, w), t2)
             else Br (v, t1, update (t2, k div 2, w));
```

- Using an exception is not easily replaced by NONE and SOME
- The following function is used to cons an item onto the head of an array

```
fun tcons v Lf = Br (v, Lf, Lf)
  | tcons v (Br (w, t1, t2)) = Br (v, tcons w t2, t1);
```

- The following function converts an array to a list

```
fun toList Lf = []
  | toList (Br(v, l, r)) = v::interleave(toList l, toList r);
```

5.6 Finite Sets

- Represent by repetition-free lists
- Can be implemented using ordered lists

6 Exceptions

- Raising an exception abandons the current computation
- Handling an exception attempts an alternative computation
- Exceptions can be pattern matched
- Exceptions are declared with the `exception` keyword, like datatypes they can has more data associated with them, eg.

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

- An exception can be raised with the `raise` keyword, eg.

```
raise Failure;
raise (NoChange n);
```

- An exception can then be handled with the `handle` keyword, this allows for pattern matching, eg.

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

- Exception names are constructors of the datatype `exn`
- An alternative to exceptions is have functions return an option datatype, eg.

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

  - This has the disadvantage of having to test for `NONE` in code

- Exceptions can be used to backtrack in code

7 Functions as Values

7.1 Functions as First Class Citizens

- Functions are first class citizens
  - Functions can be passed as arguments to other functions
  - Returned as results
  - **Can not** be tested for equality
- A functional or higher-order function is a function that operates on other functions
- Nameless functions can be declared using `fn` notation

```
fn x => E

fn n => n*n
```

  - Can pattern match in nameless functions, eg.

```
fn Pat1 => E1 | ... | Patn => En
```

7.2 Curried Functions

- Curried functions return another function as a result
- A function taking multiple arguments can be expressed as nested functions each taking one argument
- A function returning function is just a function of two arguments
- Curried functions allows partial application, providing one argument will return a function requiring one fewer arguments, eg.

```
fun foo a b = a * b;

foo 4 ≡ 4 * b
```

- A function can be got from an operator using the `op` keyword, eg. `op<=`
- A polymorphic sort can be made by taking the less than or equal to operator as an argument, eg.

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

7.2.1 The Map Function

- `map` applies a function to every element of a list, returning a list of the functions results
- Has the following declaration,

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

7.2.2 The Exists Function

- Given a predicate the `exists` function tests if it hold for any element in a list
- Has the following declaration,

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

7.2.3 The Filter Function

- Given a predicate the `filter` function returns a list of all elements in an input list that satisfy the predicate
- Has the following declaration,

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

8 Lazy Lists

- Lists of possibly infinite length
- Element computed upon demand
- Laziness implemented by delaying evaluation of the tail
- Avoids waste if there are many solutions
- Evaluation of the tail is called forcing the sequence
- Defined as follows

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

- The function `fn() => E` will delay evaluation until a unit is passed to it
- The following defines an infinite sequence of integers starting at k

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

- The function `get`, is passed an integer n and a sequence, it returns the first n elements of the sequence as a list
- The lazy list append function will never get to its second argument if the first sequence is infinite
- The function `interleave` will combine two potentially infinite lazy lists, eg.

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

- The list functions `filter` and `map` can be altered to work with sequences, giving `filterq` and `mapq` respectively

9 Procedural Programming

9.1 Commands

- Procedural programs can change the state of the machine
- They can interact with its environment
- They use data abstractions of the computer's memory
  - References to memory cells
  - Arrays, blocks of memory cells
  - Linked structures, especially linked lists
- References point to mutable areas in memory, assignment never changes `val` bindings, which are imutable
- Calling `ref` will allocate a new location in memory, with a type τ `ref`, where τ is a type
- The function `!` will return the contents of a reference, eg. `!P`
- The function `:=` updates the contents of a reference, eg. `P := E`
- To allocate the number 1 and increment it, the following must be done,

```
val i = ref 1;
i := !i + 1;
```

- Commands are expressions that has an effect on the state of the machine, eg. update references, perform I/O etc.
  - A typical command returns an empty tuple, `()`
  - C_1`; ... ;`C_n causes a series of expressions to be evaluated and returns the value of C_n

9.2 Iteration

- The `while` loop tests a Boolean conditions and executes the body while this condition is true
- Has the following syntax

```
while B do C
```

- This is a command and returns an empty tuple `()`
- If chained must be followed by a semicolon `;`

9.3 Arrays

- Arrays are like references that hold several values
- Arrays are zero indexed
- Arrays have a type τ `Array.array`
- An array is created using the `Array.tabulate(n, f)` function, where `n` is the size of the array and `f` is a function such that `A[i] = f(i)`
- The function `Array.sub(A, i)` returns the contents of `A[i]`
- The command `Array.update(A, i, E)` updates `A[i]` to the value `E`