**Decision Making**

**Pattern Matching**

* Describes combinations of values that can be passed into the function
* Use \_ for variables that are not used in the function body
* Pattern order matters

**Guards**

* Permit function behaviour to vary based on arbitrary expressions
* Evaluated in order
* End with an otherwise case

**If … then … else**

* Analogous to ternary conditional
* Return type in then and else must be the same

**Case**

* Pattern matching for arbitrary expressions
* Form: function var = case (var) of

‘case’ -> Result …

**Recursion**

* Need one or more bases cases and one or more recursive cases
* Primitive: recursion with (n-1)
* General: recursion with anything either than (n-1)
* Mutual: Two or more functions that call each other

**Tuples and Lists**

* Tuples combine data into one object, fixed size, can be heterogenous
* Lists combine data into one object, variable in size, must be heterogenous
* List Functions:
  + Head – returns head of a list
  + Tail – returns all elements of a list except for the head
  + Take – returns first n elements from a list
  + Drop – returns list constructed by dropping first n elements
  + Length – returns length of a list
  + : – constructor for lists, inserts elements at the head
  + ++ – concatenate two lists together
  + !! – retrieve an element by index from a list
* List comprehensions
  + [calculation involving v | v <- old list, condition, … , condition ]
* List comprehensions with multiple lists forms a nested loop type of behaviour
* To process parallel lists, use **zip** 
  + zip creates a list of tuples from the elements of both lists
  + reverse process with unzip
* words – Turn a string into a list of the words in it
* unwords – Turn a list of words into one string
* sort – sorts a list

**Higher Order Functions**

* A higher order function is one that takes one or more functions as a parameter and/or returns a function as it’s result
  + Type signature use (a -> a) to denote a function
* Map – A function that computes a new list by applying a function to every element in a list
  + map <function> [a]
* filter – A function that retains or discards items in a list
  + filter <Bool function> [a]
  + Keep items that return true, discard items that return false
* Polymorphic functions
  + A function that can be applied to multiple types of data
  + Have one or more type variables in type definition
* Folding
  + Reduces a list to a single value by repeatedly applying a function to the value computed so far and the next it in the list
  + Inserting an operator/function between each value ina list
* Foldl1 and foldr1 only work on lists that contain at least one element
* Foldl and foldr are more general, and can be applied to an empty list
* Foldl
  + Function call: foldl <helper> <initial value> [a]
  + Helper signature: helper :: type a of list -> [a] -> [a]
* Foldr
  + Function call : foldr <helper> <initial value> [a]
  + Helper signature: helper :: [a] -> type a of list -> [a]
* zipWith – combines elements of two lists, performing an arbitrary operation
* sortBy – sortBy :: (a -> a -> Ordering) -> [a] -> [a]
  + Ordering has values LT, GT, EQ
* Lambda functions
  + \ parameters -> calculation with parameters
  + \x y -> x + y
  + Can be used within map and filter to make one line expressions
* Can Return a lamba expression
  + Flip reverses the order of parameters for a function

**Partial Application**

* Uncurried form – parameters bundled into a tuple and passed as a group
* Curried form – parameters passed to a function sequentially
* Functions curry and uncurry are available to transform the forms
* Partial application allows us to set one or more parameters to a specific value, leaving a function that still required at least one parameter
  + Can be used with map, filter, fold, zipWith
* Operator sections are a partial application of an operator
  + (+) (+5) (5/)
  + Use subtract for right section of (-)

**Function Composition**

* Two or more functions can be composed into a single function
* f (g (h x)) 🡪 (f .g. h) x
* Think of how you would create the function the first way, then remove brackets and insert a . to compose it, then pass it to x as a whole

**Type Classes**

* Allows functions to work for some types and not others
* Places restrictions on type variables
* Type classes:
  + Eq - ==, /=
  + Ord - <=, >=, <, >, max, min, compare
  + Enum – related to ..
  + Convert to string: show
  + Numeric: Num - +, -, \*, /
* Create own type classes
  + Class HasLength a where

Len :: a -> Int

* + Create an instance for everything you want in your type (list, 2-tuple, etc)

**Algebraic Types**

* Base types: Int, Float, Char, Bool
* Composite types: Lists, tuples
* Enumerated types – Algebraic type that has a finite list of possible values
  + Can derive default implementation or provide our own
  + Can be passed as parameters or returned as results
* Product Types – Algebraic type that is similar to a tuple
  + One or more values provided to constructor when instance is created
  + Problem: Many things can be represented by same signature, so we want to derive our own instances of Eq, Show, Ord in order to prevent collisions
  + Can have multiple constructors, each with different number of parameters
* Recursive Algebraic Types – a type defined in terms of itself
  + Functions that operate on recursive data structures are often recursive
* Polymorphic Recursive Algebraic Types
  + Example: data BinaryTree a = NilT | Node a (BinaryTree a) (BinaryTree a)
  + Make functions that deal with these types polymorphic as well
  + Every time you use BinaryTree use BinaryTree a

**Lazy Evaluation**

* Evaluation in Haskell is lazy rather than eager
* Expressions are not evaluated until they are needed
* Unused expressions are never evaluated
* Lazy evaluation permits the creation (but not complete evaluation of) infinite data structures
* Composite parameters may only be partially evaluated
* C++ can sometimes avoid fully evaluating expressions
* Exception to eager evaluation: built in logical operators
  + Short circuit evaluation for ‘and’ and ‘or’ operators. If left operand can conclusively determine the outcome of the logical expression, the right operand is never evaluated.
* [2 .. ] is an infinite list beginning at 2

**Type Checking**

* Static vs. Dynamic – when is type checking performed?
  + Static – Before the program starts running (compile time)
  + Dynamic – While the program is running
* Some languages do a mixture of both (Jave, C++)
* Strong vs. Weak – how much type checking is performed?
  + A continuum independent of when type checking is performed
  + Strong must have value match some expected type
  + Weak has more freedom to mix values
* Explicit vs. Implicit – Does the programmer write down the types?
  + Explicit – programmer is responsible for specifying types
  + Implicit – Types are inferred by the type system or aren’t needed
  + Haskell allows explicit, but it is rarely required, will infer types
* Polymorphic type checking – each expression is reduced to a set of possible values, or the declared type incorrect
* Unification – identified the set of types that satisfy two or more types
  + Look at two parallel types and try to find ways to make them merge. Can this type be changed into this? Conflicts = No Unification
* Same literal can appear more than once and have a different type each time
* Type classes must also be considered when performing type checking
  + Make sure type class of a and type class of b match in polymorphism

**Topic A**

* Conjunction – “And”, Denoted by ,
* Disjunction – “Or”, Denoted by ;
* Modus Ponens – A rule of logical inference. Given A rule that says a is true if b is true, and knowledge that b is true, we can conclude that A is true
* Prolog knowledge bases have three types of terms:
  + Constants – A number or an atom, denoted by a lowercase letter or ‘’
  + Variables – Begin with an uppercase letter
  + Compound Terms – f(t1, t2, .., tn).
    - Atom f is the functor, terms t1 – tn are components, number of terms in parentheses is the arity
* Matching – Performed between goal and the knowledge base.
  + Two terms t1 and t2 match if:
    - T1 and t2 are both some constant term
    - T1 is uninstantiated and t2 is any other term (and vice-versa)
    - T1 and t2 are both uninstantiated variables – they will corefer
    - T1 and t2 are compound terms with same functor and arity

**Topic B – Backtracking and Lists**

* A prolog program must be logically correct and procedurally correct
* Prolog evaluates clauses in a defined order, from top to bottom, and left to right within each line
* Once a match is found the left most rule is satisfied and next rule is attempted
* If a term fails, most recent match is undone and backtracking is performed
  + Move back one level and try another value. If all values tried, move back another level … until all combinations have been tested (brute force)
* Recursion – Same functor is used on both sides of the :- symbol
  + Order of clauses matters. Instantiate variables as early as possible.

Lists – Represented as head element followed by the tail of the list [ H | T ]

* Built in predicates for Lists:
  + Length – determines number of atoms in a list
  + Member – determines whether or not a value is present in a list
  + Negation – Reverses the outcome of a predicate. Denoted \+.
  + Append – Concatenate two lists

**Topic C – Structures, Cuts, Math**

* Complex terms can be used as a simple data structure
  + Animal(‘Thika’, elephant, 2500) 🡨 Data structure names Animal
  + Good for finite state machine problems (state(\_, \_, \_,))
* Cuts (!) prevent Prolog from back tracking
  + “If we have reached this point we have found the only answer”. 🡪 ! Last
  + “If we have reached this point, we have found the correct rule”

🡪 ! After rule to prevent backtracking and trying other rules

* + “If prolog has reached this point, there is no result we want”

🡪 Cut is second last and built in predicate fail is last.

* Cuts introduce a point of no return within a current rule and commit prolog to every choice it has made since selecting the current rule
* Making cut last clause will ensure at most one result is returned
* Making cut first clause ensures no further rules are considered

Math

* = is the matching operator
* == is the identical operator
* =:= is the arithmetic comparison operator
* Force evaluation: is, <, >, =<, >=, =:=, =\=