Haskell

```
len [] acc = 0
len (x:xs) acc = len xs (+ acc 1)

concat [] l = l
concat (x:xs) l = (x:(concat xs l))

rev [] = []
rev (x:xs) = (rev xs) ++ [x]

member x l = if empty(l) then false
   else if x == head(l) then true
   else member x tail(l)

member x [] = False
member x (y:_) = True
member x (y:ys) = member x ys

// generate list of ints from 1 to n
ints 0 = []
ints 1 = [1]
int n | n > 1 = (ints (n-1)) ++ [n]

filter f [] = []
filter f (x:xs) | (f x) = x : (filter f xs)

filter' f l = [x | x <- l, f x]

sums [] [] = []
sums (x:xs) (y:ys) = (x + y) :
(sums xs ys)

sieve (x:xs) = x : (filter (\y -> y
`mod` x /= 0) xs)
sieve (x:xs) = x : (sieve (filter (\y -
> y `mod` x /= 0) xs))
// call as sieve [2..]
```

PROLOG

X is 2 + 2: is means equals
Can use greater than, less than
Multiplication is *
\+ equals not equal \+ (3 = 10) returns true
5+4 =:= 4+5 checks for same answer
5+4 =\= checks for inequality
or operator is the semicolon ;
// is int division

```
concat([], L, L).
concat([X|Xs], L2, [X|L3]) :-
concat(Xs, L2, L3).

prefix(L1, L3) :- concat(L, L, L3).
suffix(S, L) :- concat(P, S, L).

thrice(L, L3) :- concat(L, L, L2),
concat(L, L2, L3).

rev([], []).
rev([X|L], R) :- rev(L, R2),
concat(R2, [X], R).

palindrome(L) :- rev(L, L).

delete(X, L, Result) :- concat(L1,
[X|L2], L), concat(L1, L2, Result).

member(X, [X|_]).
member(X, [_|Tail]) :- member(X,
Tail).

nonmember(X, []).
nonmember(X, [_|Tail]) :-
nonmember(X, Tail).

sublist(B, ABC) :-
  concat(AB, C, ABC),
  concat(A, B, AB).

// order constraints on colour
immediate after selecting colours
to improve efficiency as follows
map(A, B, C, D) :-
  colour(A ), colour(B), A \= B,
  colour(C), A \= C, B \= C,
  colour(D), D \= C, D \= B.

// backtracks sooner (once any
difs are violated)
mapBetter(A, B, C, D) :-
  dif(A, B), dif(A, C), dif(B, C),
  dif(D, B), dif(D, C),
  color(A), color(B), color(C),
color(D).

insert(X,[Y|T],[Y|NT]) :-
  X > Y,
  insert(X,T,NT).
insert(X,[Y|T],[X,Y|T]) :-
  X =< Y.
insert(X,[],[X]).

is_sorted([]).
is_sorted([_]).
is_sorted([X,Y|T]) :-
  X =< Y,
  is_sorted([Y|T]).

quicksort([], []).
quicksort([Head|Tail], Sorted) :-
  partition(Head, Tail, L1, L2),
  quicksort(L1, Sorted1),
  quicksort(L2, Sorted2),
  concat(Sorted1, [Head|
Sorted2]).

partition(Head, [], [], []).
partition(Head, [X|Tail], [X|L], G) :-
  X =< Head,
  partition(Head, Tail, L, G).
partition(Head, [X|Tail], L, [X|G]) :-
  X > Head,
  partition(Head, Tail, L, G).

alldif( [] ).
alldif( [_] ).
alldif( [X1, X2 | Xs] ) :-
  dif( X1, X2 ),
  alldif( [X2|Xs] ),
  alldif( [X1|Xs] ).

% father/mother/ancestor/
parent(X,Y) is true if X is a father/
mother/ancestor/parent of Y
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

ancestor(X,Z) :- parent(X,Y),
parent(Y,Z).
ancestor(X,Z) :- parent(X,Y),
ancestor(Y,Z).

son(X,Y) :- parent(Y,X), male(X).
daughter(X,Y) :- parent(Y,X),
female(X).
sibling(X,Y) :- parent(Z,X),
parent(Z,Y), X \= Y.
cousin(X,Y) :- parent(U,X),
parent(V,Y), sibling(U,V).
```

CSP

No shared memory
Synchronous, unbuffered, unidirectional (holds only 1 item), 1 : 1 channels
send: c!v (sends v through c)
receive: c?v (receive v through c)
<-c (receive on c in Go)
c<- (send through c in Go)

Go

Channels are first class (can send channels over channels)
Uni/Multidirectional, un/buffered, many : many channels
c := make(chan int, buffer_size)
i := <- c // receive into i from channel c
c <- i // send i into channel c

Go compiler gc built in C released in 2009 then rewritten in Go in 2015 for Go v1.5. Bootstrapping improved security (no longer relied on insecure C pointers). Original compiler only targeted 4 architectures. GccGo (front end for GCC) in 2011 allowed 45 architectures.

General Concepts

Lazy evaluation (call by name): expressions only evaluated when absolutely necessary, allows infinite data structures
Non strict evaluation order: left to right, like in lambda calculus. Strict is normal BEDMAS.

Higher order function: takes a function as parameter(s) and/or returns a function as result

Activation records (stack frames): implement recursion, manage local variables
Tail recursion allows the compiler to use jmp instructions rather than expensive stack frames

Static typing: type determined at compile time. Noncompliant programs are rejected.
Dynamic typing: types determined at run time. Can potentially cause runtime errors.

Static binding (lexical scoping): free variables are bound to outer blocks (static chaining looks at enclosing blocks until variable can be resolved).
Dynamic binding: free variables are bound to most recently used versions in control flow. Dynamic chaining refers to caller.
Ex: p() { var i; q(); } q() { i // same i as i in p }

Recursion implementation: static - points at original call. dynamic - points are record immediately before.

Compiler architecture

Lexical analysis: Initial part of reading and analyzing program text: text is read and divided into tokens, each of which corresponds to symbol in the programming language
Syntax analysis: takes list of tokens and arranges into a tree structure (syntax tree) – often called parsing
Type checking: analyze the syntax tree to determine if the program violates certain consistency requirements (e.g. variable used but not declared)
Intermediate code generation: program translated to a simple machine-independent intermediate language
Register allocation: symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code
Machine code generation: intermediate language is translated to assembly language (textual representation of machine code) for a specific machine architecture
Assembly and linking: assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined
First 3 phases = front end of compiler, last 3 phases = backend

Variable lifetime:
static: variable lifetime == program lifetime
dynamic: allocated and freed by programmer at any time. stored in heap.
automatic: push/pop stack
memory: [stack ->   <- heap | global (static)].

Type equivalence:
name equivalence: types are same if they have same name (i.e. two struct types a and b both made of same composition are not equal)
structural equivalence: types are same if they are composed of same base types (i.e. two differently named structs can be equal)

Type coercion: implicitly substitute an object of one type for another with same value.
Ex: 5 + 4.0 -> 5.0 + 4.0
Type conversion: same but explicit.
Ex: (float)5 + 4.0 -> 5.0 + 4.0
Non-converting type cast: change type, but keep bit pattern (reinterprets value)

Polymorphism: only one implementation of function required for inputs of different types (type determined at runtime decides interpreter behaviour). Fundamentally unification.

Concurrency issues: shared memory - race conditions (solution: lock critical section to ensure mutual exclusion), spin lock (busy waiting), semaphores, mutexes, message based communicating processes (channels)

Synchronous: you know for sure when data will arrive, but must wait for blocking tasks.

Asynchronous: don't have to wait for blocking processes to complete, but can't rely on data arrival.

Type inference Go vs. Haskell: Go is always right to left (and only available in using :=). Haskell uses unification to infer

FSMs: boxes represent state, arrows represent transitions (events occurring). Remember to clearly indicate starting and terminating states.

Message passing: Erlang, Go
First class functions: JavaScript, Haskell

<u>Simula</u>
First OO language

<u>Fortran</u>
1950's IBM team by John Backus
"Made up language as they went along" – seriously flawed
Quicker and more reliable development, less machine dependence since register and machine instructions are abstracted away
Became standard language in science and engineering and is only now being replaced by other languages

<u>Cobol and PL/I</u>
Cobol language developed on the 1950's for business data processing
Designed by a committee of US Department of Defense reps, computer manufacturers and commercial organizations such as insurance companies
Supposed to be a short-range solution until a better design could be created—instead became the most widespread language in its field -> good for simple calculations on vast numbers of complex data records (why it was good for business)
IBM later created the language PL/I: had all features of Fortran, Cobol and Algol and has now replaced Fortran and Cobol on many IBM computers

<u>Algol</u>
Originally designed by an international team for general and scientific applications
First published 1958 – revised version=Algol 60 was extensively used in computer science research and implemented on many computers, 3rd version wasn't very popular
First to do BNF
Two languages derived from Algol are Jovial (used by US Air Force for real-time systems) and Simula (one of first simulation languages)--- most famous descendant is Pascal (wanted to create a language that could be used to demonstrate ideas about type declarations and type checking)

<u>Pascal</u>
Advantage: original Pascal compiler was written in Pascal and thus could easily be ported to any computer
Disadvantage: Pascal language is too small- the standard language has no facilities whatsoever for dividing a program into modules on separate files, and thus cannot be used for programs larger than several thousand lines
Wirth recognized that modules were an essential part of any practical language and developed the Modula language

<u>C</u>
Developed by Dennis Ritchie of Bell Laboratories in early 1970's as an implementation language for the UNIX operating system
Operating systems were written in assembly language because thought that high level languages were too complex
Designed to be close to assembly language (extremely flexible)
Easy to write programs with bugs because unsafe constructs are not checked by the compiler as they would be in Pascal.
Standardized in 1989 by ANSI

<u>C++</u>
Created by Bjarne Stroustrup from Bell Laboratories in 1980's: used C as basis of C++
Extended C to include support for object-oriented programming similar to that provided by the Simula language

<u>Ada</u>
Based on Pascal
1977- US DoD standardized it
Different because Ada subject to intense review and criticism before standardization (instead of being standardized after)
Designed to support writing portable programs
Supports error handling and concurrent programming which are traditionally left to (non-standard) OS functions
Difficult language because it supports many aspects of programming that other languages leave to the operating system

<u>Ada 95</u>
Published in 1983
Support for true object-oriented programming including inheritance

Data-oriented Languages:
<u>Lisp</u>
Basic data structure is linked list
Much work on AI was carried out in Lisp
Common Lisp language later developed to enable programs to be ported from one computer to another – a popular dialect of Lisp is CLOS which supports object-oriented programming
3 elementary operations of Lisp are:
car(L) and cdr(L) which extract the head and tail of a list L, respectively
cons(E, L) which creates a new list from an element E and an existing list L

<u>APL</u>
Basic data structures are vectors and matrices – operations work directly on such structures without loops
Requires a special terminal
Difficult to experiment with APL w/o investing in costly hardware

<u>Snobol, Icon</u>
Basic data structure is the string
In Snobol, the basic operation is matching a pattern to a string and as a side effect of the match, the string can be decomposed into substrings
In Icon, the basic operation is expression evaluation where expressions include complex string operations – also has backtracking

<u>SETL</u>
Basic data structure is the set
Programs created resemble logic programs
Notation used is that of set theory: {x | p(x)}

Call by name: same as normal order reduction (always reduces the leftmost outer most beta redex first) except that no redex in a lambda expression that lies within an abstraction (within a function body) is reduced. An actual parameter is passed as an unevaluated expression that is evaluated in the body of the function being executed each time the corresponding formal parameter is referenced. Normal order is ensured by choosing the leftmost redex, which will always be an outermost one with an unevaluated operand
Call by value: same as applicative order reduction (always reduces the left most inner most beta redex first) except no redex in a lambda expression that lies within an abstraction is reduced. This restriction corresponds to the principle that the body of a function is not evaluated until the function is called (in a beta-reduction). Applicative order means the argument to a function is evaluated before the function is applied.

Explicit polymorphism: parametrization is obtained by explicit type parameters in procedure headings, and corresponding explicit applications of type arguments when procedures are called (having parameters of type type)
Implicit polymorphism: type parameters and type applications are not admitted, but types can contain type variables which are unknown, yet to be determined, types

<u>modus ponens</u>: If a knowledge base contains a rule head :- body, and Prolog knows that body follows from the information in the knowledge base, then Prolog can infer head.