Computational Process	
Conditional Expression	In the form: Pred ? (Do if True) : (Do if False)
Function	Act as a blackbox, improves readability when used in a moderate, traceable amount
Program Evaluation	Evaluate Inputs, Evaluate, then substitute In the case of arithmetic operation, follows: (), */,+-
Applicative Order Reduction	Evaluate then substitute Ie f(x + y) to f(z)
Normal Order Reduction	Substitute then Evaluate Ie f(x + y) to (f)(x + y)

Order of Growth with Recurrence Relations (“Out of Syllabus”)	
$T(n) = O(1) + b$	$T(n) = \begin{cases} O(n) & b = T(n-1) \\ O(2^n) & b = 2T(n-1) \\ O(log(n)) & b = T(n/2) \\ O(n) & b = 2T(n/2) \end{cases}$
$T(n) = O(log(n)) + b$	$T(n) = \begin{cases} O(n\,log(n)) & b = T(n-1) \\ O(n) & b = T(n/2) \\ O(n\,log(n)) & b = 2T(n/2) \end{cases}$
$T(n) = O(n^a) + b$	$T(n) = \{O(n^{a+1}) \quad b = T(n-1)$
$T(n) = O(n) + b$	$T(n) = \begin{cases} O(n^2) & b = T(n-1) \\ O(2^n) & b = 2T(n-1) \\ O(n) & b = T(n/2) \\ O(n \log(n)) & b = 2T(n/2) \end{cases}$
$T(n) = O(a^n) + b$	Generalise as $T(n) = O(a^n)$
$T(n) = T(n-1) + T(n-2)$	$T(n) = \theta(\phi^n) \approx \theta(2^n)$
n-choose-k	$T(n) = O\left(\left(\frac{e \cdot n}{k}\right)^k\right), \Omega\left(\left(\frac{n}{k}\right)^2\right)$
General Formulas	
$T(n) = O(a^n) + b$	Generalise as $T(n) = O(a^n)$
$T(n) = aT(n-b) + r$	$T(n) = \begin{cases} r \times n^a & a = 0,1 \\ a^n & a > 1 \end{cases}$
Master Theorem $T(n) = aT(n/b) + r$, $a \in Z_+, b \in Z_{\geq 2}$	$T(n) = \begin{cases} \theta(n\,log(ab)) & r = O(nc), c < log(ab) \\ \theta(nc\,log(n)) & r = \theta(nc), c = log(ab) \\ \theta(r) & r = \Omega(nc), c > log(ab) \end{cases}$
Memoized (Dynamic Programming)	
$T(n) = T(n-1) + T(n-2)$	$T(n) = \theta(n)$
n-choose-k	$T(n) = \theta(n \times k)$
General Formula for Memoized (Dynamic Programming)	
n states/subproblem, k time per state	$T(n) = \begin{cases} O(n * k_T), & k = O(k_T) \\ \theta(n * k_T), & k = \theta(k_T) \\ \Omega(n * k_T), & k = \Omega(k_T) \end{cases}$

Higher Order Functions (Functional Abstraction)	
Scope	Variables can be accessed after declaration within the same environment or its descendants
Scoping Rule	Name occurrence refers to the closest surrounding declaration
return	return keyword causes the function to output the following expression and terminate the remaining further evaluation of the function

Data Abstraction	
Types	$Type(x) = \begin{cases} Number, & x = 1, -5.6, 0.5e-157 \\ Boolean, & x = true, false \\ String, & x = \text{"hello world"} \\ Function, & x = (f \Rightarrow f + x) \\ Pair, & x = pair(?,?)/[?,?] \\ List, & x = pair(?, list)/null \\ Array, & x = [?, \ldots, ?] \\ Null, & x = null \end{cases}$
Box-and-pointer Diagram	Graphical representations of data structures made of pairs, box represent array, segmented by number of items in the array, arrow points from within the box out to another array box

Tree Processing	
Tree(type)	$pair(?\,(type)/tree(type), tree(type))/null$
Tree (Caveat)	No null or pair as datatype, unable to differentiate null and pair from tree
Binary Tree (BT)	$pair(entry, pair(BT, BT))/null$
Binary Search Tree (BST)	$pair(entry, pair(BST < entry, BST > entry))/null$
Binary Search in BST	To search for value v, check entry is v, return if true, else check smaller BST if entry < v, check bigger BST otherwise, O(log n) runtime

Stream	
General Idea	Delay evaluation of subsequent elements of the stream / “list” output
Stream Constructor	Pair(element, () => evaluation of next stream)
Stream (Type)	Pair(element(type), () => evaluation of next stream(type))
Subsets of stream	Empty list, pair whose tail is a nullary function that returns a stream
Infinite Streams	Streams that never terminates ie the nullary function at its tail never returns null Done usually by referencing itself, referencing an element in itself, or referencing another infinite stream
Lazy Definition	Only evaluates what is required, that is, streams are lazy lists
Applications of Laziness	Avoids problem of non-termination by delaying evaluation, enabling a concept of infinite generative data structure

Mutable Data; Array and Loops	
State	Memory of variables and their values
Assignment	Using “let” declaration, allowing mutability of the variable ie manipulation of variable value
Function Parameter Changes in Source 3	Function Parameters are now variables ie they are mutable
Mutable Pairs	Make use of set_head and set_tail to directly mutate the values in the pair
Self-Referencing in Mutable Data	Since variables acts as containers, this may lead to loops when lists or pairs is an element of itself.
Mutable (“Destructive”) List Processing	Making use of existing pairs, we manipulate the elements of the output list by changing the elements within the existing pairs.
Array	Random access, able to access (read/write) each values in the array in O(1) time
Array Processing	Store intermediate values in a constant, or index pointers, read/write values in array directly, abuse the O(1) random access
While Loop	In the form: while (pred) { statement } Checks predicate before each evaluation of the statement. Run statement /loop body if pred is true, else terminate loop.
For Loop	In the form: for (stmt 1; pred; assignment) { statement } stmt1: variable declaration / assignment statement Pred: Evaluate statement / loop body if true, else terminate loop Assignment: evaluated after evaluation of statement / loop body
break	break keyword terminates current iteration and the entire nearest loop
continue	continue keyword terminates current iteration and continue with the nearest loop

Sorting	
Insertion Sort	Insert head of list to the right position in a helper list, continue sort for the rest of the list Time: $O(n^2)/\Omega(n)$ Space: $O(1)$
Selection Sort	Find smallest value of list, remove it and put at the front, continue sort with the rest of the list Time: $\theta(n^2)$ Space: $O(1)$
Merge Sort	Recursively split list in half, then merge the split lists in a sorted manner Time: $\theta(n\,log(n))$ Space: $O(n)$
Quicksort	Recursively use head as pivot and split list to two lists holding larger and smaller than pivot, then merging the smaller list in front, pivot at center, and larger list at the back. Time: $O(n^2)/\Omega(n\,log(n))$ Space: $O(n)/\Omega(log(n))$
Memoization	Uses a wrapper function, whose environment serve to store outputs of previously run function, returns a helper function acting as a substitute to the original function and acts as an access to the wrapper function

	Do not use when: Impure function Repeated function call is not expected Space is limited

Symbolic Evaluation	
Definition	Functions / Expressions with data structures
Symbolic Representation	Converts Function and Expressions to a predetermined data structure using constructors
Constructors	Abstraction of turning functions and expressions to a predetermined data structure
Accessors / Selectors	Abstraction of information extraction from the data structure
Predicates	Abstraction of checks on data structure
Expression Simplification	Abstraction of Evaluation of data structure
Specification	Describe what is done
Implementation	Describe how it is done

Environment	
Environment	A sequence of frames
Frame	Contains the bindings of values to names
Pointer	Points a frame to its enclosing environment
Extending an Environment	Means to add a new frame in an existing frame
Scope	Variables can be accessed after declaration within the same environment or its descendants
Scoping Rule	Name occurrence refers to the closest surrounding declaration
Unbound	Where a variable/name is undeclared or not assigned a value in the current frame and its enclosing environment
Hoisting	Compiler shifts certain declarations at the start of the evaluation of the program. In Source, function declarations are hoisted
Frame Generation	When a constant or variable is declared in a block which extends an environment ie no empty frame
Constant/Variable Declaration	Changes / Adds the binding of name to a value in the current frame
	:= denotes constant, : denotes variable
Assignment	Changes binding of name to the new value, giving an error when the name is a constant, unbound or unassigned ie non-mutable / non-existent
Function Application	1 Frame for the parameter variables, 1 Frame for the body block of the function if there is a declaration in each of the respective frames
Primitive Values	Bindings are drawn inside frames
	Appear when needed, placed inside stash and frames, does not carry identity
Compound Values	Objects are drawn outside frames ie each object carries a unique identity

	Created fresh in environment area, pointed to from stash and frames	
Function Construction	Creates a function object with a pointer to the environment in which it was constructed The name that the object is assigned to during declaration is a constant	
Pair, Array Construction	Creates a pair or array object, not pointed or associated to any environment	
=== returns true if	True,False, Null, Undefined	Identical to itself
	Numbers	Same representation in double-precision floating-point representation
	Strings	Same characters in the same order
	Functions, Pairs, Arrays	Holds the same unique object / identity
While Loop	Loop body is in a new block / frame, hence the loop body extends the environment each evaluation	

CSE Machine	
CSE	Control, Stash, Environment, saves Environment when needed ie declarations made in the frame
Control	Holds statements and expressions that appear in the program that is being executed, and instructions that are generated during program execution, Uses runtime stack memory
Stash	Hold or point to (intermediate) result(s) of computation when instructions in the Control is executed, and hold or point to result of computation when the program execution terminates
Environment	Stores binding of names, Uses heap for memory
Value-Producing Statement	Leaves the result of statement in the Stash, of which, if it is the last statement in the program, will be the output of the program
Sequences	An order of Statements, where ; pushes a pop instruction if the statement is value producing and not the last statement in the sequence
pop instruction	Removes the latest value in the Stash
Conditional Expressions / Statements	Pushes predicate then the branch instruction
branch instruction	Pops Boolean value from Stash, commit to consequent (if true), else to alternative (if false)
Pred1 && Pred2	Equivalent to Pred1 ? Pred2 : false
Pred1 || Pred2	Equivalent to Pred1 ? true : Pred2
Declaration	Not value-producing, result is popped
Assignment	Is value-producing, result not popped
asgn instruction	Assigns name to the top value in Stash
While Loop	Pushes undefined, predicate and while instruction, is value-producing
while instruction	Predicate is true, pushes: pop, body, predicate, same while instruction
	Predicate is false: done
Block	Generate a new frame, set program to new frame, and pushes the sequence within the block and then an env instruction pointing to the previous environment
env instruction	Stores a pointer to the previous environment, to which the program is set to when env instruction is evaluated
Functions	Same as handling 2 Blocks, just more complex. Pushes a closure, function parameters, (marker if required) and a call instruction
closure (function object) in stash	Closures, also known as function object, behave similarly as primitives in the stash, with a pointer to the referenced closure in the environment
call instruction	In the form: call n, where n is the number of parameters expected for the closure. Pops the closure and args from Stash, place args a new frame extending function's environment, and push the function body onto Control
return instruction	Pops instructions or statements in the Control until it reaches a marker, then pops itself and the marker
Implication of Recursive vs Iterative	Recursive causes Control to grow, Iterative causes Environment to grow, since Control memory is more precious that Environment, we typically prefer using Iterative over Recursive
CPS Implication	Wraps statements within functions, shifting memory from Control to Environment

Metacircular Evaluator	
General Idea	Using the language to mimic itself
Parsing	Lexical Analysis, Token, Syntactic Analysis
Lexical Analysis	Split Characters into meaningful symbols
Tokens	Holds values, and operands for analysis
Syntactic Analysis	Checks and converts the series of Tokens to a syntax / parse tree
Evaluator	Evaluates the parse tree
Control	List of statements and expressions
Stash	List of intermediate values
Environments	Treated as a list of pairs, frames as pairs
Function Objects	Treated as a list

Backus-Naur Form (Subsection)

```
stmt ::= expr
       | stmt1 … stmtn
       | { stmt }
       | const name = expr
       | function name
( params ) block
       | return expr
bin-op ::= + | - | * | /

expr ::= expr bin-op expr
       | number
       | ( expr )
       | true | false
       | expr1 ? expr2 : expr3
       | expr(expr1, …, exprn)
       | params => block
```