Computational Process	
Processes	A set of activities that interact to produce a result
Computational Processes	A set of activities in a computer, designed to achieve a desired result
Task and Design Method	We are concerned about how this design happen and use programs to prescribe how the computational process unfolds
People in Focus	Programs play a central role in the human communication during their construction and operation
Motto	Programming is communicating computational process.
Elements of Programming	Primitives, Combination, Means of Abstraction
Abstraction	Naming, Functional (Naming should reflect its use / purpose)
Declaration	Pre-declared Names, Constant/Variable Declarations, Parameters of function declarations and lambda expressions, Function name of function declarations
Declaration Mandatory	All names in Source must be declared
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)
Recursion	Base Case Evaluate Rec(n – 1), then evaluate Rec(n)
Deferred Operations	Accumulation of Recursive calls
Iterative	Base Case Evaluate Iter(n), then evaluate Iter(n – 1)
Wishful Thinking	Assume that f(n – 1) is correct

Order of Growth	
Dimensions of Performance	Time Complexity, Space Complexity
Time Complexity	How long program run given an input of a large size n
Space Complexity	How much memory for the program to run given an input of a large size n
Order of Growth	Describes Time Complexity or Space Complexity
Big O Notation	Upper bound of Complexity $\exists k \in R^+, n_0 \in R$ $\left(\forall n \in R_{\geq n_0}\left(r(n) \leq k \cdot g(n)\right)\right)$ $\Rightarrow$ Function r has a order of growth $O\left(g(n)\right)$
Big Omega Notation	Lower bound of Complexity $\exists k \in R^+, n_0 \in R$ $\left(\forall n \in R_{\geq n_0}\left(k \cdot g(n) \leq r(n)\right)\right)$ $\Rightarrow$ Function r has a order of growth $\Omega\left(g(n)\right)$
Big Theta Notation	Average Complexity $\exists k_1, k_2 \in R^+, n_0 \in R$ $\left(\forall n \in R_{\geq n_0}\left(k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n)\right)\right)$ $\Rightarrow$ Function r has a order of growth $\theta\left(g(n)\right)$
Constants and Minor Terms	Overruled at large n, hence does not matter $1 < log\, n < n < n\, log\, n < n^2 < n^3 < 2^n$

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)	
Lambda Expression	In the form: (Param) => Exp
Function	In the form: function name(param) { return ? ; }

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	$Type(x) = \begin{cases} Number, & x = 1, -5.6, 0.5e-157 \\ Boolean, & x = true, false \\ String, & x = \text{"hello world"} \\ Function, & x = (f \Rightarrow f + x) \\ Pair, & x = pair(?,?)/[?,?] \\ List, & x = pair(?, list)/null \\ Array, & x = [?, \ldots, ?] \\ Null, & x = null \end{cases}$
Data Abstraction	Constructor and Selector Operations
Data Abstraction Barrier	
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

List Processing	
length (xs)	Return length of list xs, number of elements in xs
append (xs, ys)	Return list containing elements of xs, then ys
reverse (xs)	Return list of elements of xs in reversed order
map(f, xs)	Return list of elements of xs with f applied
filter(pred, xs)	Return list of elements of xs where pred is true
accumulate (op, init, xs)	Return output of op after applying op(x, ys) on elements of xs from the back, initialised with init

Tree Processing	
Tree(type)	$pair\left(?\,(type)/tree(type), tree(type)\right)/null$
Tree (Caveat)	No null or pair as datatype, unable to differentiate null and pair from tree
Binary Tree (BT)	$pair\left(entry, pair(BT, BT)\right)/null$
Binary Search Tree (BST)	$pair\left(entry, pair(BST < entry, BST > entry)\right)$ $/null$
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
CSE Machine	Prior to this, we have been using Substitution Model, which immediately substitute the value of variables into the code. Since variables are now mutable, that is, the value of variables can change during the evaluation of the code, we need use a CSE Machine model to keep track of variable states.
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
	Time: refer to Order of Growth section Space: m, where m is the number of combinations of possible inputs
	Do not use when: Impure Function: Output of function can change between calls, ie accessing values in a mutable data structure Repeated function call is not expected Space is limited

Continuation-Passing Style (Out of Syllabus)	
General Idea	Convert recursive functions to iterative by passing deferred operations as a function as an extra parameter/argument

Programming Language Processing	
Tombstone Diagrams T-Diagrams	Illustrate the relationships of programs, interpreters and compilers
Program	Denoted with an oval head "Prog name", rectangular body "ProgLang", which means the program named "Prog name" written in "ProgLang" programming language
Processor	Denoted with a rectangle with a triangular extrusion at the bottom, with "ProcLang", which means the processor executes codes with the language "ProcLang"
Evaluator/Interpreter	Denoted as a rectangle, "SrcLang" at top and "TgtLang" at bottom, which means the interpreter takes in code in source language "SrcLang", and execute the code in the target language "TgtLang"
Compiler	Denoted as a T-Shape, with "SrcLang" on the left, "TgtLang" on the right, and arrow from "SrcLang" to "TgtLang" and "CompLang" at the bottom, which means the compiler, written in "CompLang", converts a "SrcLang" code to a "TgtLang" code
Tool Chains	Combinations of interpreter and compilers

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
Global Environment	Stores the primitive and pre-declared functions or constants
Program Environment	The implicit environment block that extends the global environment in which the user program runs
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

<table>
<tr><td rowspan="2">Function Application</td><td>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</td></tr>
<tr></tr>
<tr><td rowspan="2">Primitive Values</td><td>Bindings are drawn inside frames</td></tr>
<tr><td>Appear when needed, placed inside stash and frames, does not carry identity</td></tr>
<tr><td rowspan="2">Compound Values</td><td>Objects are drawn outside frames ie each object carries a unique identity</td></tr>
<tr><td>Created fresh in environment area, pointed to from stash and frames</td></tr>
<tr><td>Function Construction</td><td>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</td></tr>
<tr><td>Pair, Array Construction</td><td>Creates a pair or array object, not pointed or associated to any environment</td></tr>
<tr><td rowspan="4">=== returns true if</td><td>True,False, Null, Undefined | Identical to itself</td></tr>
<tr><td>Numbers | Same representation in double-precision floating-point representation</td></tr>
<tr><td>Strings | Same characters in the same order</td></tr>
<tr><td>Functions, Pairs, Arrays | Holds the same unique object / identity</td></tr>
<tr><td>While Loop</td><td>Loop body is in a new block / frame, hence the loop body extends the environment each evaluation</td></tr>
<tr><td>For Loop
(Out of Syllabus)</td><td>Creates during initialisation:
1 Frame for stmt1, that is variable declaration or assignment statement, points to current environment
Creates during each loop body evaluation:
1 frame storing a copy of loop variable, extending the frame initialised for the loop
1 frame storing a declaration of variable with the same name and value as loop variable, extending the frame above
1 frame storing declaration of variable in the loop body, extending the frame above</td></tr>
</table>

<table>
<tr><th colspan="2">CSE Machine</th></tr>
<tr><td>CSE</td><td>Control, Stash, Environment, saves Environment when needed ie declarations made in the frame</td></tr>
<tr><td>Control</td><td>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</td></tr>
<tr><td>Stash</td><td>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</td></tr>
<tr><td>Environment</td><td>Stores binding of names, Uses heap for memory</td></tr>
</table>

<table>
<tr><td>Value-Producing Statement</td><td>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</td></tr>
<tr><td>Sequences</td><td>An order of Statements, where ; pushes a pop instruction if the statement is value producing and not the last statement in the sequence</td></tr>
<tr><td>pop instruction</td><td>Removes the latest value in the Stash</td></tr>
<tr><td>Conditional Expressions / Statements</td><td>Pushes predicate then the branch instruction</td></tr>
<tr><td>branch instruction</td><td>Pops Boolean value from Stash, commit to consequent (if true), else to alternative (if false)</td></tr>
<tr><td>Pred1 && Pred2</td><td>Equivalent to Pred1 ? Pred2 : false</td></tr>
<tr><td>Pred1 || Pred2</td><td>Equivalent to Pred1 ? true : Pred2</td></tr>
<tr><td>Declaration</td><td>Not value-producing, result is popped</td></tr>
<tr><td>Assignment</td><td>Is value-producing, result not popped</td></tr>
<tr><td>asgn instruction</td><td>Assigns name to the top value in Stash</td></tr>
<tr><td>While Loop</td><td>Pushes undefined, predicate and while instruction, is value-producing</td></tr>
<tr><td>while instruction</td><td>Predicate is true, pushes:
pop, body, predicate, same while instruction
Predicate is false: done</td></tr>
<tr><td>Block</td><td>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</td></tr>
<tr><td>env instruction</td><td>Stores a pointer to the previous environment, to which the program is set to when env instruction is evaluated</td></tr>
<tr><td>Functions</td><td>Same as handling 2 Blocks, just more complex. Pushes a closure, function parameters, (marker if required) and a call instruction</td></tr>
<tr><td>closure (function object) in stash</td><td>Closures, also known as function object, behave similarly as primitives in the stash, with a pointer to the referenced closure in the environment</td></tr>
<tr><td>call instruction</td><td>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</td></tr>
<tr><td>return instruction</td><td>Pops instructions or statements in the Control until it reaches a marker, then pops itself and the marker</td></tr>
<tr><td>Implication of Recursive vs Iterative</td><td>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</td></tr>
<tr><td>CPS Implication</td><td>Wraps statements within functions, shifting memory from Control to Environment</td></tr>
</table>

<table>
<tr><th colspan="2">Metacircular Evaluator</th></tr>
<tr><td>General Idea</td><td>Using the language to mimic itself</td></tr>
</table>

<table>
<tr><td>Parsing</td><td>Lexical Analysis, Token, Syntactic Analysis</td></tr>
<tr><td>Lexical Analysis</td><td>Split Characters into meaningful symbols</td></tr>
<tr><td>Tokens</td><td>Holds values, and operands for analysis</td></tr>
<tr><td>Syntactic Analysis</td><td>Checks and converts the series of Tokens to a syntax / parse tree</td></tr>
<tr><td>Evaluator</td><td>Evaluates the parse tree</td></tr>
<tr><td>Backus-Naur Form (BNF)</td><td>Expresses possible values types can hold</td></tr>
<tr><td>Control</td><td>List of statements and expressions</td></tr>
<tr><td>Stash</td><td>List of intermediate values</td></tr>
<tr><td>Environments</td><td>Treated as a list of pairs, frame as pair of lists</td></tr>
<tr><td>Function Objects</td><td>Treated as a list</td></tr>
<tr><th colspan="2">Backus-Naur Form (Subsection)</th></tr>
<tr><td>stmt ::= expr
| stmt1 ... stmtn
| { stmt }
| const name = expr
| function name
(params) block
| return expr

bin-op ::= + | - | * | /</td><td>expr ::= expr bin-op expr
| number
| (expr)
| true | false
| expr1 ? expr2 : expr3
| expr(expr1, ..., exprn)
| params => block</td></tr>
</table>

<table>
<tr><th colspan="2">Type, Classes and Objects</th></tr>
<tr><td>Constant Declaration</td><td>In the form: const name: type = ?(type);</td></tr>
<tr><td>Variable Declaration</td><td>In the form: let name: type = ?(type);</td></tr>
<tr><td>Function Declaration</td><td>In the form: function name(param: param_type): return_type {... return ?(return_type) }</td></tr>
<tr><td>void type</td><td>Function does not return anything</td></tr>
<tr><td>any type</td><td>Can hold any types (self-explanatory)</td></tr>
<tr><td>Function Type Annotation</td><td>(param_type) => return_type</td></tr>
<tr><td>Union Type</td><td>type1 | type2 | ... | typen</td></tr>
<tr><td>Pair Declaration</td><td>In the form: name: Pair<head_type, tail_type> = pair(?(head_type), ?(tail_type))</td></tr>
<tr><td>List Declaration</td><td>In the form: List<type> = list(?(type))</td></tr>
<tr><td>Param Type Validity</td><td>All param satisfies types</td></tr>
<tr><td>List Type Validity</td><td>Any element satisfies type</td></tr>
<tr><td>Array Declaration</td><td>In the form: name: type[] = [?(type)];</td></tr>
<tr><td>Stream Declaration</td><td>In the form: name: Stream<type> = pair(?(type), () => ?(Stream<type>))</td></tr>
<tr><td>Class</td><td>A special abstraction for objects or variables that holds specific and unique properties, methods, and variables</td></tr>
<tr><td>_name</td><td>Variable name is intended to be used only within the class, <s>but may still be used outside of class</s></td></tr>
<tr><td>__name</td><td>Differentiates from another variable with the same name in another class, ie private from other classes</td></tr>
<tr><td>Class Constructor</td><td>A blueprint for the class</td></tr>
<tr><td>new</td><td>Creates a new object of the class based on the class constructor</td></tr>
</table>

Dot Operator "."	Access the object's properties or methods directly