Concept 1: Programming Language Syntax

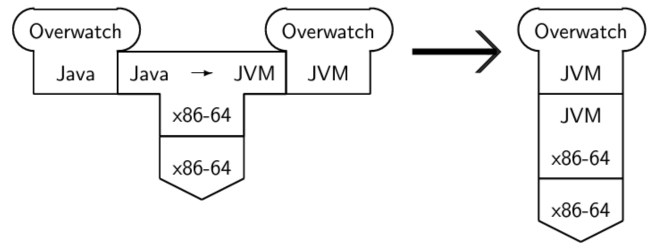
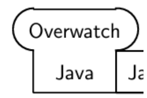
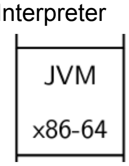
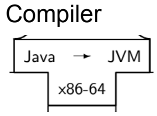
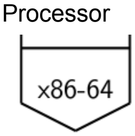
Programming Language	
Definition	The set of programs that are considered “allowed”
Purpose	Formal framework to instruct computers to execute a computational process
Human-made	Programming languages are designed by people (individual, research groups, companies) for people (programmers)

Syntax	
Definition	Describes the structure of programs: their components and the rules in which components can be combined
Phonology	Not important Usually written, not spoken
Morphology	Not important Usually simple categories, e.g. keywords vs names
Pragmatics	Not important Purpose of programs is clear
Token level	Mirroring natural language, individual “words” form the basic syntactic building blocks
Program level	Larger structures are built out of tokens, following rules

Macros	
Preprocessing of programs	Expand names and abbreviations before the program is run
Macros in C	Recursive macros are allowed but not expanded

Assemblers	Provide access to machine code in a human-readable fashion
Parser	Parsing is part of the syntactic analysis of programs Carried out before program execution

Concept 2: Control

Interpreters & Compilers	
 Compiling Overwatch from Java to JVM code, and running the JVM code on a JVM interpreter running on an x86-64	
Program 	The Program Overwatch is written in Java
Interpreter 	Program that executes program Interpreter for JVM written in x86-64
Compiler 	Program that translates from one language to another language Java-to-JVM compiler written in x86-64
Processor 	The computer's processor

Programming Paradigms		
Programming	Functional	Imperative
Semantics	Simple	Complex
Functions return	Expression evaluation	Explicit return
Name reference	Values	Locations
Assignment	None	Changes the stored value

Concept 3: Types

Types in programming	
Purpose	Prevent bugs due to errors Types are to make errors occur earlier, before they cause “more damage”
Type-safe	A language is type-safe, if well-typed programs don’t apply forbidden operations to values
Memory-safe	A language is memory-safe, if well-typed programs don’t access memory in ways that are not consistent with the data types in the programs
Optimization	Type-safe languages allow omission of runtime type checks

SML	
Data Types	`datatype Color = Red | Green | Blue;`

Tuples	`type Point = real * real;`
	`val p = ( 0.5, 0.75 );`
	`val x_of_p = #1 p;`

Function types	`fun twice (f : int-> int, x : int) : int = f (f (x));`
	
	`type IntFunction = int-> int;`
	`fun evaluate (g: IntFunction, value: int) = g value;`
	
	`fun square (x: int) = x * x;`
	`val f = square;`
	
	`evaluate(f, 3); (* evaluates to 9 *)`

Recursive types	`datatype IntTree = Empty` `| Node of int * IntTree * IntTree`

Generic types	`fun map (F, nil) = nil | map(F, x::xs) = F(x) :: map(F,xs)`
	
	`val map = fn : ( ’a-> ’b) * ’a list-> ’b list`

Concept 4: Data & Memory Management

Parameter Passing	
Pass-by-value	The value is copied into a new memory location used during function execution
Pass-by-reference	A reference to the data structure is passed as argument Function has read and write access to the original data structure

Mutability	
Constants	Names for which assignment is not allowed. (const, final, etc.)
Variables	Names for which assignment is allowed

Memory Allocation	
Static Allocation	Assign fixed memory location for every identifier Size of data structure must be known at compile-time. Recursive functions are not possible. Data structures such as closures cannot be created dynamically.
Stack Allocation	Keep track of information on function invocations on runtime stack Size of locals can depend on arguments Recursion possible Only objects with known compile time size can be returned from functions Difficult to manipulate recursive data structures
Heap Allocation	Data structures may be allocated and deallocated in any order Complex pointer structures will evolve at runtime Management of allocated memory becomes an issue

Heap Management Techniques	
Reference Counting	Each object has a counter that tracks how many references point to it. When the counter drops to zero, the object is deallocated immediately. Advantages ● Incrementality ● Locality ● Immediate reuse Disadvantages ● Runtime overhead ● Cannot reclaim cyclic data structures
Mark-Sweep Garbage Collection	Mark live nodes Sweep to free unmarked nodes Advantages: ● Handles cyclic references Disadvantages ● Requires traversal

Copying Garbage Collection	Use only half of the available memory for allocating nodes Once this half is filled up, copy only the live memory contained in the first half to the second half Reverse the roles of the halves and continue Advantages: ● Handles cyclic references ● Compaction (memory layout) ● Performs better at low residency Disadvantages ● Double the memory required ● Copying overhead when moving live objects

Concept 6: Object-oriented programming

Knowledge Representation View of Objects	
Aggregation	Has-a relationship Class A contains Class B, but B can exist independently
Classification	Grouping relationship Organizes entities with similar properties into categories
Specialization	Is-a relationship Subclass inherits and extends a parent class's properties Usually, the concept of inheritance (class extension) achieves specialization in object-oriented languages.

	Procedural Languages	Object-oriented Languages
Binding	Early Binding (Static Binding)	Late Binding (Dynamic Binding)
Determine function at	Compile-time	Runtime

Concept 7: Concurrency

Granularity Level of Concurrency	
Lowest	Parallel write access to the same memory location; (undefined behavior)
Middle	Interleaving execution with atomic operations
Highest	Threads are executed atomically

Overhead	Introducing synchronization primitives adds overhead to the system.
Race Conditions	Two or more threads can access shared data and they try to change it at the same time. The last thread to access the shared resource can overwrite data changes made by the previous threads, leading to unpredictable results.
Deadlocks	A situation where two or more threads are blocked forever, each waiting for the other to release a resource.
Livelocks	A scenario where two or more threads are active but are effectively doing nothing useful since they keep responding to each other's actions. It's a form of infinite loop.
Starvation	Starvation occurs when a thread cannot access the resources it needs because other "greedy" threads are monopolizing the resources. The starving thread may never get to execute or may do so too infrequently.
Mutual Exclusion	A code section that can only be executed by a single thread at a time

	Semaphores	Monitors
	Uses `wait` and `signal` operations Both operations are executed atomically	Uses `wait` and `notify` operations Every object is associated with its own queue
Level of Abstraction	Low-level primitive	High-level construct
Mutual Exclusion	Requires manual implementation	Built-in
Encapsulation	None	Encapsulates shared resources
Ease of Use	More complex, error-prone	Easier, structured
Flexibility	High	Moderate
Prone to Errors	Deadlocks, priority inversion	Less prone to errors

Basics
Choose interleaving execution of threads at the level of virtual machine instructions
Use a virtual machine for Source as the starting point
Threads are running independently, each with their own set of registers

Implementing Interleaving
Switching execution from thread to thread, also called “time-slicing”
Keep a queue of threads in the machine, each with its own registers
Machine picks a thread from the queue, and executes a certain number of instructions in that thread
Then, it suspends the execution of the thread, and starts execution of the next thread in the queue
The process of saving and re-installing registers is called context switching

Execution of concurrent_execute
Expect closure on stash
Initialize control and stash of the new thread to be empty stacks
Call closure using new control and stash
Push true on stash of old thread

Concept 8: Parallelism

Sequential	Execute one task at a time Tasks run in the order they are listed
Concurrent	Tasks appear to run at the sametime Tasks may or may not run in parallel
Parallel	Tasks run simultaneously, on different hardware units

Parallelism types	
Task parallelism	Distribute **different tasks** across cores Each core performs a **unique** operation
Data parallelism	Distribute **data** across cores Each core performs a **similar** operation

Parallelism in Computer Architecture	
SIMD	A single machine instruction operate simultaneously on multiple data points e.g. vector operations
SPMD	
Pipeline	Instructions are arranged in a sequence such that later instructions can start execution before earlier instructions finish
Speculative	Multiple instruction sequences are executed speculatively without knowing which will be needed e.g. branches of conditionals

Granularities of Parallelism		
Intracore	Pipelining, speculative parallelism,SIMD	Pipelining of microcode instructions
Core level	Across multiple cores on a single chip, SPMD on GPUs	Handled by low-level parallel programming features
Multi processor	Processors have access to a shared memory while each processor has their own cache. OS assigns threads or processes to processors.	Handled by concurrency mechanisms (threads, synchronization), and high-level parallel programming language features
Multi server	Distributed databases (sharding)	MPI using dedicated compilers such as `mpiCC`

Implicit Parallelism	
Side effects	This works for pure functional languages (no side effects), and if there are no data dependencies
LISP	Function calls (synchronization) `(f a b c d e)` (6 processes) Let `(let ((a e1) (b e2) (c e3)))` (3 processes) Conditionals `(cond (p1e1) (p2 e2) (p3 e3)))` (3 processes)

Prolog	Intra-clause parallelism
	`q:-p1,p2,...,pn` (n processes)
	
	`p(X) :- q(X).`
	`p(X) :- r(X).`

Explicit Parallelism	
Multilisp	Conditionals (synchronization) `(pcall f a b c d e)` (six processes) Futures `(pcall f (future a) (future b))` allows execution of f to proceed before the values of a and b have been computed
Haskell	

Lazy Evaluation	Haskell evaluates expressions only when they are needed
	
	`-- definition of f`
	`f x _ = x * 10`
	`-- use of f`
	`f (1 + 2) (3 * 4)`
	`-- 3 * 4 is not evaluated`

Infinite lists	Haskell's lazy evaluation naturally allows handling of infinite data structures
	
	`-- Infinite list of all positive integers`
	`let xs = [1..]`
	`-- Takes first 3 elements`
	`let ys = take 3 xs`
	`-- [1,2,3]`
	`print ys`

`seq`	Each result in their second operand and instruct the compiler to compute the value to the term left of seq
	
	`strictSumAndLength :: [Int]-> (Int, Int)`
	`strictSumAndLength xs =`
	`    let s = sum xs`
	`    in s ‘seq‘ (s, length xs)`

`par`
`pseq`
	Each result in their second operand and instruct the compiler to sequentialize or parallelize the program execution
	
	`parallelSum :: [Int]-> [Int]-> Int`
	`parallelSum xs ys =`
	`    let sumX = sum xs`
	`        sumY = sum ys`
	`    in sumX ‘par‘ (sumY ‘pseq‘ (sumX +`
	`sumY))`

Concept 9: Logic Programming

Query	

Assertion	Represents information in a database
	
	`assert(salary(list("Bitdiddle", "Ben"),`
	`60000))`

Matching	An assertion matches a query if we can instantiate the query's logic variables with data and obtain the assertion.
	
	`// Query input:`
	`address($x, $y)`
	
	`// Query input:`
	`supervisor($x, $x)`

Query processing	● Find all assignments to variables in the query pattern that satisfy the pattern ○ The kind of information specified in the pattern needs to match the kind of information in an assertion ○ The assertion must result from the pattern by instantiating the pattern variables with values ● System responds to query by listing all instantiations of the query pattern with the variable assignments that satisfy it Special case ● If the pattern has no variables, the query reduces to a determination of whether that pattern is in the database. The empty assignment satisfies that pattern for that database.
Rule	Represents a large set of assertions `rule(conclusion, body)` where conclusion is a pattern and body is any query.