Vocabulary

For each vocabulary item:

1. know its meaning
2. be able to compare/contrast it with related terms
3. be able to understand its use in sentences
4. be able to use it appropriately in sentences

Introduction: computability, static analysis, compile-time, runtime, expressiveness, efficiency

Computability: termination, nontermination, diverge, partial function, partial recursive functions, computable, Turing machine, Turing-Complete, \( \lambda \)-calculus, undecidability, and the Halting Problem.

Compilers, etc.: compiler, interpreter, source language, machine code, native code, “compiled language,” “interpreted language,” virtual machine (a.k.a. managed runtime environment), lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, backend code generation, common subexpression elimination, copy propagation, dead-code elimination, loop optimization, in-lining function calls

Formal languages, BNF, and parse trees formal language, grammar, production rule, derivation, terminal symbol, nonterminal symbols, recursively enumerable, context free, regular languages, Chomsky Hierarchy, BNF (Backus-Naur Form), parse tree, precedence, associativity.

\( \lambda \)-calculus: \( \lambda \)-calculus, \( \lambda \)-expression = \( \lambda \)-term, abstraction, application, free variable, bound variable, \( \alpha \)-renaming, \( \beta \)-reduction, reducible expression (redex), normal form, currying, Church-Rosser Theorem, confluence, \( Y \) (fixed-point operator), \( \Omega \)

Paradigms declarative, imperative, procedural, object-oriented (OOP), functional, logic, Lisp, Scheme, Haskell, ML, OCaML, Prolog, Java, Scala, side-effect, mutation, referential transparency

Algol/ML Algol 60, Algol 68, Pascal, ML, Standard ML, CaML, Objective CaML (OCaML), F\#, Standard ML of New Jersey, REPL, unit type, tuple, record, list, constructor, reference cell, dereference, list constructor (::)

types type, type error, type safety, type checking, dynamic (runtime) type checking, static (compile-time) type-checking, type inference, Hindley-Milner type inference, polymorphism, parametric polymorphism, implicit parametric polymorphism, explicit parametric polymorphism, ad-hoc polymorphism, overloading), subtype polymorphism
**scopes/activation records** block, local variable, parameter, non-local (global) variable, activation record (= stack frame), stack discipline, inline block, scope, lifetime, control link (a.k.a. dynamic link), actual parameters, formal parameter, environment pointer (= frame pointer = $fp frame pointer on MIPS, rbp on AMD64), access link (a.k.a. static link), first-order functions, higher-order functions, closure, pass-by-value, pass-by-reference

**control structures** spaghetti code, structured control

**object oriented languages** object-oriented design, semantics, dynamic lookup, abstraction, subtyping, inheritance, dynamic lookup, abstraction, design pattern

**Simula** pass-by-reference, instance, class prefixing, prefix class, coroutine

**SmallTalk** object, class, subclass, selector, message, method, instance variable, class variable

**C++** member function, data member, virtual function (function with dynamic lookup), base class (superclass), derived class (subclass), abstract function, abstract class, multiple inheritance, diamond inheritance virtual base class virtual base class

**Java** reference variable, primitive type, virtual machine, garbage collection, access modifier, class, abstract class, interface [keyword/Java entity], single inheritance, mtable, itable, invokevirtual, invokeinterface, invokestatic, class loader

**Objectives**

1. Introduction (Michell, Ch. 1; Lecture Notes #1)
   (a) be able to list four major programming languages created around 1950–1960 (Algol, Cobol, Fortran, and Lisp) and for each one be able to describe the problem domain that language was trying to support (i.e., portable description of algorithms, business information processing, scientific computing, and symbolic processing (artificial intelligence)) and also at least one language feature that was influenced by that domain (e.g., recursion and nest blocks, base-10 arithmetic and verbose ‘readable’ syntax, arithmetic expressions with infix operators, lists—actually, cons-cells—as a primitive data type).
   (b) be able to explain what it means for a problem to computable
   (c) be able to explain what the difference between compile-time and run-time is
   (d) be able to explain why, in language design, one often has to make trade-offs between expressiveness and efficiency

2. Computability (Mitchell, Ch. 2; Lecture Notes #2)
   (a) understand the meaning of a partial function and its relevance to programming
      i. be able to explain the difference between a partial and total function
      ii. be able to determine whether a simple function is a partial or total function
iii. be able to explain the relationship between partial functions and nontermination
(b) understand and be able to apply the Turing-completeness and the Turing-Church thesis to reasoning about the functions that are computable in principle by a particular programming languages
i. be able to recall \( \lambda \)-calculus and the Turing Machine as two separate formal notions for describing computable functions
ii. be able to describe informally the some of the kinds of language features (recursion with parameters or loops + variable assignment) that can make a language Turing-complete; noting that basically everything considered a programming language is Turing-complete.
iii. be able to explain the Church-Turing thesis (i.e., there exists a Church’s \( \lambda \)-calculus term, a Turing Machine, and a programs in any Turing-complete language that computes any computable function)
(c) understand and be able to apply the notion of undecidability
i. be able to explain the definition of undecidability
ii. understand that some problems are undecidable
iii. be able to explain the proof of the Halting Problem’s undecidability

3. Compilers, Interpreters, and the JVM (Mitchell 4.1; Lecture Notes #3)
(a) be able to explain what compilation and interpretation are, what the difference between them is, and what the advantages of each are
(b) be able to explain what is meant by a ‘compiled language’ or an ‘interpreted language’
(c) be able to explain how Java’s implementation uses compilation and interpretation
(d) be able to list the six typical compilation phases listed in Mitchell and describe what each of them does
(e) be able to describe five common optimizations.
(f) be able to why the undecidability of the Halting Problem means that static (=compile time) analysis of runtime behaviors will never be precise

4. Formal languages, BNF, and parse-trees (Mitchell 4.1; Lecture Notes #4)
(a) be able to, given a formal grammar for a language, derive a valid sentence in that language, and write its derivation
(b) be able to list the language classes, regular, LL(1), LR(1), context-free, context-sensitive, and recursively enumerable, in order from least to most general
(c) be able derive the parse tree for a sentence in a language described by a BNF grammar
(d) be able to identify (simple) ambiguous grammars
(e) be able to identify the precedence and associativity implied by a grammar or parse tree
(f) be able to draw abstract syntax trees (AST's) for arithmetic expressions written in infix or prefix notation

5. λ-calculus (Mitchell 4.2; Lecture Notes #5)

(a) be able to read and interpret λ-calculus terms, specifically:
   i. be able to write an AST for a given λ-calculus term
   ii. be able to identify the free/bound variables in a λ-term
   iii. be able to determine if two λ-terms are α-equivalent (and perform α-renaming)
   iv. be able to identify occurrences of abstraction and application in a λ-calculus term
(b) be able to 'evaluate' λ-calculus terms
   i. be able to identify reducible expressions (redexes)
   ii. be able to determine if a λ-calculus term is in its normal form,
   iii. be able to apply β-reduction to rewrite reducible expressions, (avoiding variable capture by doing α-renaming, if necessary)
(c) be able to recognize simple λ-terms that do not have a normal form (e.g., Ω)
(d) be able to describe how arbitrary computations could be done using λ-calculus, including functions involving multiple arguments (currying), numbers (Church numerals), unbounded recursion (using the fixed point operator), etc.
(e) be able to write λ-calculus terms (possibly using abbreviations) representing arbitrary computable functions.

6. Language Paradigms (Mitchell 4.4, Ch. 15; Lecture Notes #6)

(a) be able to compare and contrast the declarative (functional/logic) and imperative (procedural/OO) approaches to programming drawing concrete examples from known functional, procedural, OO, and multi-paradigm languages
(b) be able to compare and contrast the procedural and OO paradigms, drawing concrete examples from known languages
(c) be able to compare and contrast the functional and procedural paradigms, drawing concrete examples from known languages
(d) be able to determine the value of Scheme expressions (whose evaluation terminates in a small finite number of steps)
(e) be able to list some of the features commonly found in procedural languages but are not used in the functional paradigm (assignments or side-effects more generally, and while loops)
(f) be able to list several key features of functional programming languages that facilitate programming in a functional style (first-class/higher-order functions, extensive polymorphism list types and operators, structured function returns, and constructors (aggregates) for structured objects)

7. Functional Programming (Mitchell Ch. 3; Odersky et al. Ch. 1–5; Lab Notes on Scheme)
(a) be able to understand simple, idiomatic Scala programs (including var, val, def, object, and class declarations, including type annotations, tuples)
(b) identify shallow errors related to logic/type/syntax of Scala programs
(c) be able to write Scala programs that handle I/O and/or operate on a tree-structure implied by parenthesis nesting

8. Algol and ML (Mitchell, Ch. 5, Lecture Notes #7)
(a) For each of Algol 60, Pascal, C, and ML, be able to describe their historical context, major contributions, and distinctive features.
(b) Be able to comprehend, determine the value of, and identify/read the types of simple ML expressions / short programs using basic types, tuples, lists, datatypes, patterns, reference cells, anonymous functions, optional type annotations, pattern matching, and val/fun bindings.

9. Type Systems and Type Inference (Mitchell, Ch. 6, Lecture Notes #8)
(a) be able to describe and give examples of how types are used to make sure that bit sequences are used consistently
(b) be able to describe how static typing makes languages like predominately statically typed languages like ML/Java/Scala more efficient than predominately dynamically typed languages like Lisp/Scheme/Python/Perl/SmallTalk
(c) be able to describe how dynamic and static typing interact in Java
(d) be able to describe why Java is categorized as type safe but C is not, citing specific language features that illustrate the difference
(e) be able to translate short ML code fragments to lambda-calculus AST's capturing the type relevant interactions
(f) be able to perform Hindley-Milner type inference (on lambda-calculus AST's)

10. Procedure calls, Scopes, and Activation Records (Mitchell, Ch 7, Lecture Notes #9)
(a) be able to describe the components and runtime layout activation records in block structured languages
(b) be able to identify which incremental part(s) of the book’s reference implementation are necessary to support each of the following features: blocks, nested blocks, functions/procedures, functions with static scope, and functions as arguments and results
(c) be able to identify the circumstances in which stack disciple can or cannot be used for activation records
(d) be able to draw/evolve the runtime layout of call stacks (with environment pointers and activation records), closures, function code, etc., as applicable, for C and ML programs at different points in their execution assuming the book’s reference implementation of procedure calls
(e) be able to trace (simulate by hand) the execution of programs in an Algol-, C-, or ML-like programming language with static and dynamic scoping for non-local variables in nested functions.

11. Control Structures (Mitchell, Ch. 8, Lecture Notes #10)

(a) be able to translate code using high-level control structures (specifically, block-nested “if” and “while” statements) into a language into labels and gotos (possibly controlled by single-line if-statements)

(b) be able to draw control flow graphs

(c) be able to apply ML exception typing rules to determine the types of (and well-typedness) of expressions involving exceptions

12. Object-Oriented Language Overview (Mitchell, Ch. 10, Lecture Notes #12)

(a) be able to describe the "four basic concepts" found in object-oriented language

(b) be able to describe how OO features allow programs to be organized in ways not possible in functional/procedural languages

13. Simula (Mitchell, Ch. 11, Lecture Notes #13)

- be able to describe how Simula’s classes/objects are in fact procedures/activation records

14. SmallTalk (Mitchell, Ch. 11, Lecture Notes #14)

(a) be able to describe in what ways SmallTalk can be considered a more ‘dynamic’ language (than C++, Java, or Scala)

(b) be able to diagram the memory layouts SmallTalk uses for method and field lookup

15. C++ (Mitchell, Ch. 12, Lecture Notes #15)

(a) be able to describe the difference between virtual and non-virtual functions in C++

(b) be able to explain how C++ object layouts and vtables work

(c) be able to explain why C++’s virtual function dispatch is more efficient than SmallTalk’s message dispatch

(d) be able to diagram the memory layout associated with C++ classes and objects (single inheritance case).

(e) be able to describe issues with multiple inheritance

(f) be able to describe the difference between virtual and non-virtual base classes

16. Java (Mitchell, Ch. 13, Lecture Notes #16)

(a) be able to list the major design goals of Java

(b) be able to describe some of the ways these goals impacted the design of the Java language and implementation
(c) be able to compare and contrast Java’s design goals with those of C++ and SmallTalk and be able to give examples of how those differing goals are reflected in the design.

(d) be able to compare and contrast the Java language mechanisms for inheritance and subtyping with inheritance and subtyping in C++

(e) be able to explain the major cases when dynamic lookup is used (instance method invocation on an object reference typed by a class or interface) and not used (static method invocation) in Java.