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

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

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,

Objectives

1. 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 logic 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)

2. 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

3. Algol and ML (Mitchell, Ch. 5)

(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.

4. Type Systems and Type Inference (Mitchell, Ch. 6)

(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)

5. Procedure calls, Scopes, and Activation Records (Mitchell, Ch 7)

(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.

6. Control Structures (Mitchell, Ch. 8)

(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