Programming Language Principles

1. Definitions of common programming language features
2. implementation of those features
3. Their significance/use in various paradigms
4. examples

References and Values

- **value model** => a variable is a named container for a value
- location expression = l-value
- value expression = r-value
- **reference model =>** a variable is a named reference to a value
- all variables are l-values

Type Inference

Orthogonality
- features can be used in any combination, the combinations all make sense, and the meaning of a given feature is consistent (regardless of the other features it is combined with)

Coercion
- compiler **converts** a value of one type into a value of another type when that second type is required by the surrounding context
  - Represent a weakening of type security (allow types to be mixed w/o explicit intent)

Aliasing

- two or more names that refer to a single object in a given scope

Overloading

- allowing a name to refer to more than one object in a given scope
- implementation (in symbol table): lookup routine returns a list of possible meanings for the requested name. semantic analyzer must choose from among the elements of the list based on context.

Polymorphism

generally: a single subroutine accepts **unconverted** arguments of different types

- **Genericity:** explicit parametric polymorphism
  - usually implemented by creating multiple copies of the polymorphic code, one specialized for each needed concrete type

  - programmer can define classes with type parameters eg Hashmap <String, Integer>
- **Inheritance:** subtype polymorphism
  - implemented by creating a single copy of the code, and by inserting sufficient 'metadata' in the representation of objects that the code can tell when to treat them differently
  - used in object oriented languages. Allows a variable X of type T to refer to any type derived from T.
  - Can be implemented entirely at compile time, given simple model of inheritance
- **Implicit Parametric Polymorphism:**
  - programmer does not need to specify type
  - typically the interpreter/compiler will examine the arguments to a given function and determine if they support a given operaion
  - relies on type inference
  - can be implemented in either of the above ways (ie single code sometimes, multiple code when necessary)

Casting

- allows programmer to change type w/o changing the underlying implementation (reinterpret the bits)

Scope

- Scope is the textual region where a binding is active
- **Static Scope**: bindings between names and objects determined at compile time by inspection
  - Current binding is found in the matching declaration whose block most closel surrounds a given point in the program
- **Dynamic Scope** : the current binding for a given name is the one encountered most recently during execution and not yet destroyed by returning from its scope
  - Bindings cannot be determined by a compiler
    - Result: No program fragment that makes use of nonlocal names is guaranteed a predictable referencing environment
    - Advantage: facilitates the customization of subroutines, but usually other ways to get same effect
- **Scope Implementation**
  - Symbol table used in statically scoped programs
    - Maps names to the information the compiler has
  - W/Dynamic scoping, an interpreter must perform operations that correspond to a symbol table
    - Tend to use an association list or a central reference table

Binding

- **Deep binding**: bind the reference environment at the time the subroutine is passed as a parameter (early binding) -> usually default w/static scoping
  - Implemented with closures: an explicit representation of the referencing environment along with a reference to the subroutine
  - If association lists are used, then referencing environment can be represented by a top-of-stack

pointer
    - When the subroutine is called, its saved pointer replaces the current referencing environment pointer
  - If central reference table used, may have to copy the entire main array of the central table and the first entry on each of the lists.
    - May be able to reduce amount to be copied by compiler/interpreter determining which names may be used by the subroutine and then copying the first entries of those lists
- A running program may have multiple instances of an object declared in a recursive subroutine. A closure captures the current instance of every object. All of these instances will be found when the subroutine is executed.
- **Shallow binding**: bind reference when the subroutine is actually called (late binding)

OOP Principles:

Five Principle View:
- **Encapsulation**
  - gathering state and behavior of an AT together, and allowing direct access to elements of AT state only by that AT's behaviors. Encapsulation relies on Information Hiding.
- **Inheritance**
  - A relation among classes corresponding to acquisition by a child of characteristics of its parent(s).
  - Characteristics include elements of state (attributes) and of behavior (methods).
  - Child is said to belong to derived class (subclass).
  - Parent is said to belong to base class (superclass).
  - These relations induce a hierarchy which can be represented by a digraph and whose members form a *family*.
- **Abstract Data Types**
  - a named aggregation of relevant aspects (of state and behavior) of a phenomenon or concept, with a behavioral interface.
  - The type of any element of AT state is either primitive (e.g., integer) or some AT.
  - AT's reduce or hide complexity;
  - AT's help map problem domain to solution domain;
  - AT's facilitate communication and reuse.
  - In OOP an AT augments the set of native types.
  - AT's rely on encapsulation.
- **Information Hiding**
  - publish interface and conceal implementation, through
    (a) non-public elements of state (attributes), enforcing encapsulation; and,
    (b) non-disclosure of the implementation of behavior (source code of methods), protecting intellectual property.
  - Makes objects and algorithms invisible to portions of the system that do not need them
  - Reduces the amount of info required to understand any given portion of the system
  - Interfaces should be as simple as possible
  - Any design decision that is likely to change should be hidden inside a single module
  - Other benefits:
    - Reduces name conflicts

    - Safeguards the integrity of data abstractions
    - Compartmentalizes runtime errors
- **Polymorphism**
    - generally: the ability of a single subroutine to accept **unconverted** arguments of different types
    - Subtype, Explicit Parametric (Generics), and Implicit Parametric

Three Principle View

- **dynamic method binding –** new version of an abstraction displays newly refined behavior, even in a context that expects an earlier version
- **inheritance –** new abstractions refine/extend earlier ones
- **encapsulation –** the ability to group data and subroutines, operate on them together in one place and hid irrelevant detail

Functional Languages

Features of functional languages not in imperative languages:

- First class function values and higher-order functions
- Extensive polymorphism
- List types and operators
- Recursion
- Structure function returns
- Constructors (aggregates) for structured objects
- Garbage collection

More details:

- First class subroutine - subroutine whose behavior is determined dynamically
- Higher-order functions - takes a function as an argument, or returns a function as a result
- Polymorphism important in functional languages, b/c it allows a function to be used on as general a class of arguments as ossible
- Recursion is important b/c (in the absence of side effects) is the only method of repetition
- A pure functional language must provide completely general aggregates -- cannot build a strucutred object via assignment to subcomponents
- Functional languages employ a heap for all dynamically allocated data
- referential transparency => expression yields the same value at any time

Imperative

- program built on an ordered series of changes to variables in memory
- expressions => produce a values
- statements => executed solely for side effect

- Heavy use of:
  - assignments
  - sequencing
  - selection (if-then-else)
  - iteration (some recursion)
- better portability, library packages, interfaces to other languages, debugging and profiling tools than functional languages

Java

- uses a value model for built in types (ie int, boolean variables are value containers)
- reference model for user-defined types (classes) (variables are location containers)
- all instances of a given generic share the same code at runtime