Review

Programming Languages
CS 214

Summary (Scope)

Dynamic-scoped languages bind identifiers at *run-time*
- Traversing *control-links* down the run-time stack.

Static-scoped languages bind locals at *compile-time*
- But binding for non-locals still occurs at run-time via *access links*

Static- and dynamic-scoping behave exactly the same, except when resolving accesses to non-local variables.
– If you never use non-locals, you'll never have scope problems, regardless of which kind of scope a language uses!

Summary (Object Oriented)

Object-oriented analysis and design is a way to build a system made up of a hierarchy of classes that reflects *real-world relationships*.

- A *subclass* inherits the attributes (data + operations) of its *superclass*.
- *Run-time binding* (or *dynamic dispatch*) ensures that when a message is sent to an object, the message is delivered to that object *first*:
  - If its class defines that message, that definition is invoked;
  - Otherwise, the message is sent "upward" in the hierarchy to the parent class, where the process is repeated.
  - If the message reaches the root class without finding a definition, a run-time error occurs.

This is called *polymorphism*, because the same message: *handle msg* may produce very different behaviors, depending on the receiver.

Summary (ii) (Object Oriented)

"OO" languages differ in how easy/simple they make OOP:

language/ binding	Ada	C++	Java	Clojure
compile time (static)	*default*	*default*	`final` (method)	*not supported*
run time (dynamic)	`tagged T` (type), and `T'Class` (handle)	`virtual` (method)	*default*	*all methods*

"OO" languages thus lie on an OO continuum:

Ada Clojure C++ Java

Summary (Types 2)

A type consists of *data* and *operations*.

The set constructors: *product*, *function*, and *Kleene closure* provide a formal way to represent type construction, using *product* and *Kleene closure* to represent the *data*, and *function* to represent the *operations* on the new type.

Enumerations let us use *real-world values* for type data.

Subranges constrain the values of existing data types.

Arrays and *vectors* are sequences stored in *adjacent memory locations* that permit *O(1)* time access to any value.

Lists are sequences stored in *dynamically allocated nodes*, that require *O(n)* time (average) to access a random value.

Summary (ii) (Types 2)

Aggregates store *multiple values of arbitrary types*.

Pointers store *addresses*, permit us to build linked nodes.

A *type system* performs type-checking using *structural equivalence* or a version of *name equivalence*.

Type-checking may be:

- *Static*: done at *compile-time*; and/or
- *Dynamic*: done at *run-time*.

The more type-checking a language requires, the stronger its type-system, and the fewer type-errors slip past.

Ada has a very strong type-system.

Summary (encapsulation)

To achieve the goals of abstract data typing, we need:

- *Encapsulation*: data and operations in 1 syntactic unit;
- *Data hiding*: the ability to restrict access to ADT data.

Three different mechanisms have evolved for doing so:

- *The* ***module***: a container for storing a data-type and its operations
- *The* ***class***: a type-constructor that stores data and operations.
- *The* ***Function***: the fulfillment of a contract, maintaining data within its own closure.

The class is better for ***flexibility***, ***reusability***, and ***maintenance***,

the module is better for ***time-efficiency*** and ***performance***,

Functions are better for ***Multithreading*** and ***performance***.

Containers are packages or classes that store data-values.

- *Generic packages* and *templates* support strongly typed containers.
- *Generic containers* can be built in most modern languages.

Summary (Concurrency)

- *Concurrent computations* consist of multiple entities:
  - *Processes* in Smalltalk, MPI
  - *Tasks* in Ada
  - *Threads* in C/C++, C#, Java, Go, Python, Ruby, Scala, …
- On a shared-memory multiprocessor:
  - The *Semaphore* was the first synchronization primitive
    - *Java* provides a *Semaphore* class for synchronizing threads
  - *Locks* and *condition variables* separate a semaphore's mutual-exclusion usage (locks) from its synchronization usage (c.v.s)
  - *Monitors* are higher-level, self-synchronizing objects
    - *Java* classes have an associated (simplified) monitor
- On a distributed system:
  - *Ada* tasks provide self-synchronizing *entry procedures*
  - *Erlang*, *Scala*, *MPI* support message-passing between processes

Summary (ii) (Concurrency)

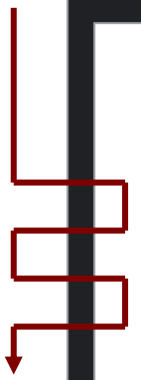
Comparing Monitors and Tasks/Threads (and Coroutines):

	Has Its Own Thread	Has Its Own Execution State
Monitor	No	No
Task/Thread	Yes	Yes
Coroutine	No	Yes

A *coroutine* (Simula, Lua) is two or more procedures that *share a single thread*, each exercising mutual control over the other:

```
procedure A;
begin
-- do something
resume B;
-- do something
resume B;
-- do something
-- …
end A;
```

```
procedure B;
begin
-- do something
resume A;
-- do something
resume A;
  -- …
end B;
```



Summary (Formal Languages)

The Chomsky Hierarchy names four "levels" of language, plus the weakest machine able to recognize at each level:

3 Regular Expressions	→	Finite State Machine
2 Context Free Languages	→	Pushdown Automata
1 Context Sensitive Languages	→	Linear Bounded Automata
0 Unrestricted Languages	→	Turing Machine

The TM is the most powerful of the machines, able to

- recognize any language capable of being recognized.
- compute any function capable of being computed.

The RAM is a computational model that is

- as powerful as the TM
- more convenient than the TM for studying HLL constructs.