Abstract Data Types (Scala)

H. Conrad Cunningham

21 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good of April 2022 is a recent version of Firefox from Mozilla.

1 Abstract Data Types

1.1 Introduction

These notes examine how to specify, design, and implement abstract data types in Scala.

The goals of these notes are to:

- review the key concepts of modular design and programming (e.g., modules, interfaces, information hiding, and contracts)
- explore how to specify abstract data types using a constructive approach based on an abstract model of the data type
- illustrate how to use Scala features (e.g traits and classes) to define appropriate interfaces and implement information-hiding modules
- examine the design and implementation of a nontrivial abstract data type (i.e., a doubly labelled directed graph)

The concepts and terminology in this chapter are mostly general. They are applicable to most any language. Here we look specifically at Scala. (I have implemented basically the same data abstraction module in Haskell and Elixir.)

1.2 Modular Design and Programming

In the provocative 1986 essay "No Silver Bullet—Essence and Accidents in Software Engineering," software engineering pioneer Fred Brooks asserts that "building software will always be hard" because software systems are inherently *complex*, must *conform* to all sorts of physical, human, and software interfaces, must *change* as the system requirements evolve, and are inherently *invisible* entities [3] A decade later Brooks again observes, "The best way to attack the essence of building software is not to build it at all" [4]. That is, software engineers should reuse both software and, more importantly, software designs.

What was true in the 1980s is still true today. Although software development tools and practices have evolved, removing some of the "accidental" properties of software development, the essential difficulties remain. Complexity continues to increase. The interfaces to which software must conform continue to change quickly and increase in number. The requirements on software continue to evolve, driven by inexorable changes in the environment and increasing penetration of computerized processes into new aspects of society. Globalization generates new requirements, which arise from both new opportunities and new competition.

We often develop software systems in multiperson teams. In many cases, the teams are geographically distributed, perhaps even across national boundaries. Communication among team members adds complexity to software development.

How should we approach software development in this contemporary context?

As a starting point, let us again look to a software engineering pioneer: David Parnas. Parnas focuses on how to decompose a system into modules to achieve robustness with respect to change and potential reuse of software. He stresses the clarity of thought more than the sophistication of languages and tools.

Although Parnas and his colleagues published their ideas on *modular specification* in the 1970s and 1980s [2,10–13], the ideas are as relevant today as they were when first published.

1.2.1 What is a module?

Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [14]. This is a *software engineering* view of a module.

In a programming language, a "module" may also be a program unit defined with a construct or convention. This is a *programming language* view of a module.

Ideally, a language's module features should support the software engineering module methods.

1.2.2 Information-hiding modules and secrets

According to Parnas, the goals of *modular design* are to [10]:

1. enable programmers to understand the system by focusing on one module at a time (i.e., *comprehensibility*).
2. shorten development time by minimizing required communication among groups (i.e., *independent development*).
3. make the software system flexible by limiting the number of modules affected by significant changes (i.e., *changeability*).

Parnas advocates the use of a principle he called *information hiding* to guide decomposition of a system into appropriate modules (i.e., work assignments). He points out that the connections among the modules should have as few information requirements as possible [10].

In the Parnas approach, an information-hiding module:

- forms a *cohesive* unit of functionality *separate* from other modules
- *hides* a design decision—its *secret*—from other modules
- *encapsulates* an aspect of system likely to change (its secret)

Aspects likely to change independently of each other should become secrets of separate modules. Aspects unlikely to change can become interactions (connections) among modules.

This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented

interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

Information hiding has been absorbed into the dogma of contemporary object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [15].

The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [10,12,13]. These are important aspects that may change as the system evolves.

1.2.3 Contracts: Preconditions and postconditions

Let's review the concepts of precondition and postcondition, which introduced in the notes on *Recursion Styles, Correctness, and Efficiency (Scala Version)* [5].

The *precondition* of a function is what the caller (i.e., the client of the function) must ensure holds when calling the function. A precondition may specify the valid combinations of values of the arguments. It may also record any constraints on any "global" state that the function accesses or modifies.

If the precondition holds, the supplier (i.e., developer) of the function must ensure that the function terminates with the *postcondition* satisfied. That is, the function returns the required values and/or alters the "global" state in the required manner.

We sometimes call the set of preconditions and postconditions for a function the *contract* for that function. These concepts underlie Meyer's *design by contract* approach to software development [9].

1.2.4 Interfaces

It is important for information-hiding modules to have well-defined and stable interfaces.

According to Britton, Parker, and Parnas, an *interface* is a "set of assumptions ... each programmer needs to make about the other program ... to demonstrate the correctness of his own program" [2].

A module's interface includes the type signatures (i.e., names, arguments, and return values), preconditions, and postconditions of all public operations (e.g., functions).

As we see later, the interface also includes the *invariant* properties of the data values and structures manipulated by the module and the definitions of any new data types exported by the module. An invariant must be part of the precondition of public operations except operations that construct elements of the data type (i.e., constructors). It must also be part of the postcondition of

public operations except operations that destroy elements of the data type (i.e., destructors).

In Scala, we often use a `trait`, a `class`, or an `object` to implement the module.

1.2.5 Abstract interfaces

An *abstract interface* is an interface that does not change when one module implementation is substituted for another [2,14]. It concentrates on module's essential aspects and obscures incidental aspects that vary among implementations.

Information-hiding modules and abstract interfaces enable us to design, build, and test software systems with multiple versions. The information-hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces.

We can reuse the software design across several similar systems. We can reuse an existing module implementation when appropriate. When we need a new implementation, we can create one by following the specification of the module's abstract interface.

In Scala, we often use a `trait` (or an `abstract class`) to specify an abstract interface in concrete form so that we can use it for several "modules".

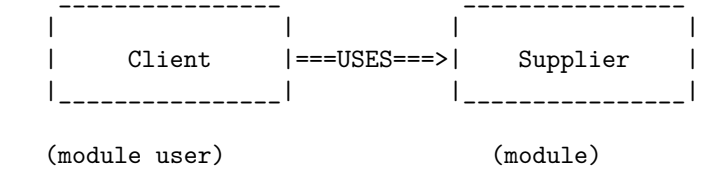
1.2.6 Client-supplier relationship

The design and implementation of information-hiding modules must be approached from two points of view simultaneously [9]:

***supplier*:** the developers of the module—the providers of the services
***client*:** the users of the module—the users of the services (e.g., the designers of other modules)

The *client-supplier relationship* is as represented in the following diagram:

TODO: Provide a better drawing below.



The supplier's concerns include:

- efficient and reliable algorithms and data structures
- convenient implementation
- easy maintenance

The clients' concerns include:

- accomplishing their own tasks
- using the supplier module without effort to understand its internal details
- having a sufficient, but not overwhelming, set of operations.

As we have noted previously, the *interface* of a module is the set of features (i.e., public operations) provided by a supplier to clients.

A precise description of a supplier's interface forms a *contract* between clients and supplier.

The client-supplier contract:

1. gives the responsibilities of the client

   These are the conditions under which the supplier must deliver results—when the *preconditions* of the operations are satisfied (i.e., the operations are called correctly).

2. gives the responsibilities of the supplier

   These are the benefits the supplier must deliver—make the *postconditions* hold at the end of the operation (i.e., the operations deliver the correct results).

The contract:

- protects the client by specifying how much must be done by the supplier
- protects the supplier by specifying how little is acceptable to the client

If we are both the clients and suppliers in a design situation, we should consciously attempt to separate the two different areas of concern, switching back and forth between our supplier and client "hats".

1.2.7 Design criteria for interfaces

What else should we consider in designing a good interface for an information-hiding module (e.g., a Scala `trait`, `class`, or `object`)?

In designing an interface for a module, we should also consider the following criteria. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

Note: These are general principles; they are not limited to Scala or functional programming. In object-oriented languages, these criteria apply to class interfaces, whether the instances are immutable or mutable.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The module should describe a single abstraction.

- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the module.
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the module.
- **Consistency:** Provide a set of operations that are internally consistent in
  - naming convention (e.g., in use of prefixes like "set" or "get", in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g., order and type of arguments),
  - behavior (i.e., make operations work similarly).

  Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.

  The operations should be consistent with good practices for the specific language being used.
- **Reusability:** Do not customize modules to specific clients, but make them general enough to be reusable in other contexts.
- **Robustness with respect to modifications:** Design the interface of an module so that it remains stable even if the implementation of the module changes. (That is, it should be an abstract interface for an information-hiding module as we discussed above.)
- **Convenience:** Where appropriate, provide additional operations (e.g., beyond the complete primitive set) for the convenience of users of the module. Add convenience operations only for frequently used combinations after careful study.

We must trade off conflicts among the criteria. For example, we must balance:

- completeness versus simplicity
- reusability versus simplicity
- convenience versus consistency, simplicity, no redundancy, and atomicity

We must also balance these design criteria against efficiency and functionality.

1.3 Terminology

The remainder of these notes use the term *abstract data type* to refer to a data abstraction. A data abstraction "module" defines and exports a user-defined type and a set of operations on that type. The type is abstract in the sense that its concrete representation is hidden; only the module's operations may manipulate the representation directly.

For convenience, these notes sometimes use acronym *ADT* to refer to an abstract data type. The term abstract data type or acronym ADT should not be confused with algebraic data type, which we have discussed previously. We specify an algebraic data type with rules on how to compose and decompose them—primarily with syntax. We specify an abstract data type with rules about how the operations behave in relation to one another—primarily with semantics.

1.4 Case Study: Doubly Labelled Digraph

In these notes, we develop a family of *doubly labelled digraph* data structures.

As a *graph*, the data structure consists of a finite set of *vertices (nodes)* and a set of *edges*. Each edge connects two vertices.

Note: Some writers require that the set of vertices be nonempty, but here we prefer to allow an empty graph to have no vertices. (But the question remains whether such a graph with no vertices is pointless concept.)

As a *directed graph* (or *digraph*), each pair of vertices has at most one edge connecting them; the edge has a direction from one of the edges to the other.

As a *doubly labelled* graph, each vertex and each edge has some user-defined data (i.e., labels) attached.

These notes draw on the discussion of digraphs and their specification in Chapters 1 and 10 of the Dale and Walker book *Abstract Data Types* [7].

1.5 Use Case

For what purpose can we use a doubly labelled digraph data structure?

One concrete use case is to represent the game world in an implementation of an adventure game.

For example, in the Wizard's Adventure Game from Chapter 5 of reference [1], the game's rooms become vertices, passages between rooms become edges, and descriptions associated with rooms or passages become labels on the associated vertex or edge (as shown in Figure 1.1).

Aside: By using a digraph to model the game world, we disallow multiple passages directly from one room to another. By changing the graph to a *multigraph*, we can allow multiple directed edges from one vertex to another.

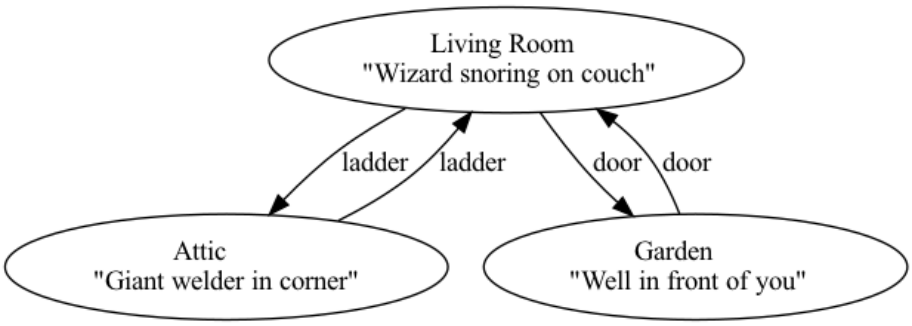


Figure 1.1: Labelled digraph for Wizard's Adventure game

The Adventure game must create and populate the game world initially, but it does not typically modify the game world during play. It maintains the game state (e.g., player location) separately from the game world. A player moves from room to room during play; the labelled digraph gives the static structure and descriptions of the game world.

1.6 Defining Abstract Data Types

How can we define an abstract data type?

The behavior of an ADT is defined by a set of operations that can be applied to an *instance* of the ADT (e.g., a Scala object).

Each operation of an ADT can have inputs (i.e., parameters) and outputs (i.e., results). The collection of information about the names of the operations and their inputs and outputs is the *interface* of the ADT.

1.6.1 Specification

To specify an ADT, we need to give:

1. the *name* of the ADT
2. the *sets* (or domains) upon which the ADT is built. These include the type being defined and the auxiliary types (e.g., primitive data types and other ADTs) used as parameters or return values of the operations.
3. the *signatures* (syntax or structure) of the operations
   - name
   - input sets (i.e., the types, number, and order of the parameters)
   - output set (i.e., the type of the return value)
4. the *semantics* (or meaning) of the operations

1.6.2 Operations

We categorize an ADT's operations into four groups depending upon their functionality:

- A *constructor* (sometimes called a creator, factory, or producer function) constructs and initializes an instance of the ADT.
- A *mutator* (sometimes called a modifier, command, or setter function) returns the instance with its state changed.
- An *accessor* (sometimes called an observer, query, or getter function) returns information from the state of an instance without changing the state.
- A *destructor* destroys an instance of the ADT.

We normally list the operations in that order.

If we use immutable data structures, then a mutator returns a distinct new instance of the ADT with a state that is a modified version of the original instance's state. That is, we take an applicative (or functional or referentially transparent) approach to ADT specifications.

If we use mutable data structures, then a mutator can change the state of an instance in place. That may be more efficient, but it tends to be less safe. It also tends to make concurrent use of an abstract data type more problematic.

Technically speaking, a destructor is not an operation of the ADT. We can represent the other types of operations as functions on the sets in the specification. However, we cannot define a destructor in that way. But destructors are of pragmatic importance in the implementation of ADTs, particularly in languages that do not have automatic storage reclamation (i.e., garbage collection).

1.6.3 Approaches to semantics

There are two primary approaches for specifying the semantics of the operations:

- The *axiomatic* (or *algebraic*) approach gives a set of logical rules (properties or axioms) that relate the operations to one another. The meanings of the operations are defined implicitly in terms of each other.
- The *constructive* (or *abstract model*) approach describes the meaning of the operations explicitly in terms of operations on other abstract data types. The underlying *model* may be any well-defined mathematical model or a previously defined ADT.

In some ways, the axiomatic approach is the more elegant of the two approaches. It is based in the well-established mathematical fields of abstract algebra and category theory. Furthermore, it defines the new ADT independently of other ADTs. To understand the definition of the new ADT it is only necessary to understand its axioms, not the semantics of a model.

However, in practice, the axiomatic approach to specification becomes very difficult to apply in complex situations. The constructive approach, which builds a new ADT from existing ADTs, is the more useful methodology for most practical software development situations.

In these notes, we use the constructive approach. We use contracts—preconditions, postconditions, and invariants—to specify the semantics of the operations.

1.6.4 Invariants

A module that implements an ADT must ensure that the objects it creates and manipulates maintain their integrity—always have a valid structure and state.

These properties are *invariant* for the ADT operations. An invariant for the data abstraction can help us design and implement such objects.

***Invariant*:** A logical assertion that must always be true for every "object" created by the public constructors and manipulated only by the public operations of the data abstraction.

An invariant is a precondition of all operations except constructors and a postcondition of all operations except destructors.

Often, we separate an invariant into two parts.

***Interface invariant*:** An invariant stated in terms of the public features and abstract properties of the "object".

***Implementation (representation) invariant*:** A detailed invariant giving the required relationships among the internal features of the implementation of an "object"

An interface invariant is a key aspect of the *abstract interface* of an ADT module. It is useful to the users of the module, as well to the developers.

It is important to note that an invariant is not required to hold:

- for private operations (e.g., functions, procedures, or methods) within an implementation of an ADT
- at any point within the implementation of an operation except the beginning and the end

1.7 Specification of Labelled Digraph ADT

Now let's look at a constructive specification of the doubly labelled digraph.

First, we specify the ADT as an implementation-independent abstraction. The *secret* of the ADT module is the data structure used internally to implement the doubly labelled digraph.

Then, we examine two implementations of the abstraction:

- using Scala lists to represent the vertex and edge sets
- using a Scala `Map` to map a vertex to the set of outgoing edges from that vertex

Before we specify the ADT, let's define the mathematical notation we use. We choose notation that can readily be used in comments in program.

1.7.1 Notation

We use the the plain-text specification notation to describe the abstract data type's model and its semantics. The following document summarizes this notation:

- **Plain Text Specification Notation (PTSN)** HTML
  PDF

TODO: Make sure the specification in this chapter and source code use the PTSN.

1.7.2 Sets

The abstract data type being defined is named `Digraph`.

We specify that this abstract data type be represented by a Scala generic type

```
Digraph[VertexType,VertexLabelType,EdgeLabelType]
```

which has three type parameters (i.e., sets):

1. Type (or set) `VertexType` denotes the possible vertices (i.e., vertex identifiers) in the `Digraph`.
2. Type (or set) `VertexLabelType` denotes the possible labels on vertices in the `Digraph`.
3. Type (or set) `EdgeLabelType` denotes the possible labels on edges in the `Digraph`.

Given this ADT defines a digraph, edges can be identified by ordered pairs (tuples) of vertices.

Values of the above types, in particular the labels, may have several components.

Values the above types, in particular the labels, may have several components.

1.7.3 Signatures

We define the following operations on the Labelled Digraph ADT. We specify these as methods on a Scala trait `Digraph`. The methods have an implicit argument `this`, which is an object from a concrete class that extends the trait (i.e., the receiver or target object for the method).

For conciseness, we use short generic parameter names `A`, `B`, and `C` for `VertexType`, `VertexLabelType`, and `EdgeLabelType`, respectively.

```
trait Digraph[A,B,C] { // A = VertexType, B = VertexLabelType,
                       // C = EdgeLabelType
```

TODO: Probably should use better generic parameters than A, B, C.

1.7.3.1 Constructors Given the primary use case described above, we require a *zero-parameter constructor* that creates an empty graph. A concrete class that extends the trait may include other constructors.

1.7.3.2 Mutators Given the primary use case described above, we specify mutators to add a new vertex (`addVertex`) and add a new edge between existing vertices (`addEdge`{.scala).

```
def addVertex(nv:A, nl:B): Digraph[A,B,C]
def addEdge(v1:A, v2:A, nl:C): Digraph[A,B,C]
```

We also specify mutators to remove a vertex (`removeVertex`), remove an edge (`removeEdge`), update the labels on a vertex (`updateVertex`, and update the label on an edge (`updateEdge`). (Note: In the identified use case, these are likely used less often than the mutators that add new vertices and edges. But we include them for completeness.)

```
def removeVertex(ov:A): Digraph[A,B,C]
def removeEdge(v1:A, v2:A): Digraph[A,B,C]
def updateVertex(ov:A, nl:B): Digraph[A,B,C]
def updateEdge(v1:A, v2:A, nl:C): Digraph[A,B,C]
```

Note: To remove a vertex requires us to remove any edges attached to that vertex.

1.7.3.3 Accessors We specify query functions to check whether the labelled digraph is empty (`isEmpty`), has a given vertex (`hasVertex`), and has an edge between two vertices (`hasEdge`).

```
def isEmpty: Boolean
def hasVertex(v:A): Boolean
def hasEdge(v1:A, v2:A): Boolean
```

We specify accessors to retrieve the label associated with a given vertex (`getVertex`) and with a given edge (`getEdge`).

```
def getVertexLabel(ov:A): B
def getEdgeLabel(v1:A, v2:A): C
```

Given the identified use case, we also specify accessors to return a list of all vertices in the graph (`allVertices`) and of the pairing of all vertices with their labels (`allVerticesLabels`).

```
def allVertices: List[A]
def allVerticesLabels: List[(A,B)]
```

Similarly, we specify accessors to return a list of all outgoing edges from a given vertex (`fromEdges`) and of the pairing of all outgoing edges with their edge labels (`fromEdgesLabels`). Here we identify outgoing edge by the "to" vertex for that edge.

```
def fromEdges(v1:A): List[A]
def fromEdgesLabels(v1:A): List[(A,C)]
```

Question: What other operations might be useful?

1.7.3.4 Destructors We do not specify any separate destructors. We rely on the garbage collection of the objects. (In some cases, concrete classes that implement abstract data types may need to implement explicit destructors to deallocate resources such as open files, network connections, etc.)

1.7.4 Semantics

We *model* the state of the instance of the Labelled Digraph ADT with an abstract value `G` such that `G = (V,E,VL,EL)` with `G`'s components satisfying the following *Labelled Digraph Properties*.

- `V` is a finite subset of values from the set `VertexType`. `V` denotes the vertices (or nodes) of the digraph.
- Any two elements of `V` can be compared for *equality*.

  Some implementations of the `Digraph` may further require that the set `V` be:

  - totally ordered—supporting < and the other relational operators
  - hashable
- `E` is a binary relation on the set `V`. A pair `(v1,v2) IN E` denotes that there is a directed edge from `v1` to `v2` in the digraph.

  Note that this model allows at most one (directed) edge from a vertex `v1` to vertex `v2`. It allows a directed edge from a vertex to itself.

  Also, because vertices can be compared for equality, any two edges can also be compared for equality.
- `VL` is a total function from set `V` to the set `VertexLabelType`.
- `EL` is a total function from set `E` to the set `EdgeLabelType`.

1.7.4.1 Interface invariant We define the following interface invariant for the Labelled Digraph ADT:

Any valid labelled digraph instance `G`, appearing in either the arguments or return value of a public ADT operation, must satisfy the Labelled Digraph Properties.

1.7.4.2 Constructive semantics We specify the various ADT operations below using their type signatures, preconditions, and postconditions. Along with the interface invariant, these comprise the (implementation-independent) specification of the ADT (i.e., its abstract interface).

In these assertions, for a digraph object `this` that satisfies the invariants, `G(this)` denotes its abstract model `(V,E,VL,EL)` as described above. The value `Result` denotes the return value of method.

Given this family of implementations uses immutable graph objects, every operation, both accessor and mutator, has a postcondition conjunct

```
G(this) = G(this')
```

where `this` denotes the receiver object before the operation and `this'` denotes the receiver object after the operation.

For an implementation allowing mutable graph objects, this requirement may be relaxed for some mutators. However, it should still hold for all accessors.

- A parameterless constructor creates and returns a new empty instance of the graph ADT.
  - Precondition:

    ```
    true
    ```

  - Postcondition:

    ```
    G(Result) == ({},{},{},{})
    ```

    For a Scala class constructor, the `Result` is a new instance of the data type created by the constructor. Thus the postcondition is equivalent to `G(this') == ({},{},{},{})`.

- Mutator method

  ```
  def addVertex(nv:A, nl:B): Digraph[A,B,C]
  ```

  inserts vertex `nv` with label `nl` into graph `this` and returns the resulting graph.
  - Precondition:

    ```
    G(this) = (V,E,VL,EL) && nv NOT_IN V
    ```

  - Postcondition:

    ```
    G(Result) == (V UNION {nv}, E, VL UNION {(nv,nl)}, EL)
    ```

Note: The set operation `VL UNION {(nv,nl)}` redefines the function (i.e., a type of relation) `VL` so that vertex `nv` now maps to vertex label `nl` (where there was no mapping beforehand).

If the precondition is true, then the method must terminate with the postcondition being true. The specification does not say what must occur if the precondition is false.

Consider the following questions:

- Can we remove the `nv NOT_IN V` conjunct in the precondition? (That is, can we *weaken* the precondition in this way?)
- If so, what are ways we can modify the postcondition to handle the new input states? (That is, how do we *strengthen* the postcondition by adding new conjuncts for the new input states?)
- Would it be appropriate to redefine the operation's signature and semantics to return an object of type `Option[Diagraph[A,B,C]]`? of some `Either` type? What are advantages and disadvantages of this kind of change?

- Mutator method

  ```
  def removeVertex(ov:A): Digraph[A,B,C]
  ```

  deletes vertex `ov` from graph `this` and returns the resulting graph.

  - Precondition:

    ```
    G(this) =  (V,E,VL,EL) && ov IN V
    ```

  - Postcondition:

    ```
    G(Result) == (V', E', VL', EL')
        where V'  = V  - {ov}
              E'  = E  - {(ov,*),(*,ov)}
              VL' = VL - {(ov,*)}
              EL' = EL - {((ov,*),*),((*,ov),*)}
    ```

    This operation also removes all edges attached to the vertex removed.

    Question: Can we remove the `ov IN V` conjunct in the precondition? If so, what are ways we can modify the postcondition to handle the new input states?

- Mutator method

  ```
  def updateVertex(ov:A, nl:B): Digraph[A,B,C]
  ```

  changes the label on vertex `ov` in graph `this` to be `nl` and returns the resulting graph.

  - Precondition:

```
G(this) =  (V,E,VL,EL) && ov IN V
```

– Postcondition:

```
G(Result) == (V - {ov}, E, VL', EL)
    where VL' = (VL - {(ov,VL(ov))}) UNION {(ov,nl)}
```

Question: Can we remove the `ov IN V` conjunct in the precondition? If so, what are ways we can modify the postcondition to handle the new input states?

- Mutator method

```
def addEdge(v1:A, v2:A, nl:C): Digraph[A,B,C]
```

inserts an edge from vertex `v1` to vertex `v2` with label `nl` in graph `this` and returns the resulting graph.

– Precondition:

```
G(this) = (V,E,VL,EL) && v1 IN V && v2 IN V &&
(v1,v2) NOT_IN E
```

– Postcondition:

```
G(Result) == (V, E', VL, EL')
    where E'  = E  UNION {(v1,v2)}
          EL' = EL UNION {((v1,v2),nl)}
```

Question: Can we weaken the precondition?

- Mutator method

```
def removeEdge(v1:A, v2:A): Digraph[A,B,C]
```

deletes the edge from vertex `v1` to vertex `v2` from graph `this` and returns the resulting graph.

– Precondition:

```
G(this) =  (V,E,VL,EL) && (v1,v2) IN E
```

– Postcondition:

```
G(Result) == (V, E - {(v1,v2)}, VL, EL - { ((v1,v2),*) }
```

Question: Can we weaken the precondition?

- Mutator method

```
def updateEdge(v1:A, v2:A, nl:C): Digraph[A,B,C]
```

changes the label on the edge from vertex `v1` to vertex `v2` in graph `this` to have label `nl` and then returns the resulting graph.

– Precondition:

```
G(this) = (V,E,VL,EL) && (v1,v2) IN E
```

– Postcondition:

```
G(Result) == (V, E, VL, EL')
    where EL' == (EL - {((v1,v2),*)}) UNION {((v2,v2),nl)
```

Question: Can we weaken the precondition?

- Accessor method

  ```
  def isEmpty: Boolean
  ```

  returns **true** if and only if graph **this** is empty.

  – Precondition:

  ```
  G(this) = (V,E,VL,EL)
  ```

  – Postcondition:

  ```
  Result == (V == {} && E == {})
  ```

- Accessor method

  ```
  def hasVertex(ov:A): Boolean
  ```

  returns **true** if and only if `ov` is a vertex of graph **this**.

  – Precondition:

  ```
  G(this) = (V,E,VL,EL)
  ```

  – Postcondition:

  ```
  G(Result) == ov IN V
  ```

- Accessor method

  ```
  def hasEdge(v1:A, v2:A): Boolean
  ```

  returns **true** if and only if there is an edge from a vertex `v1` to a vertex `v2` in graph **this**.

  – Precondition:

  ```
  G(this) = (V,E,VL,EL)
  ```

  – Postcondition:

  ```
  Result == (v1,v2) IN E
  ```

- Accessor method

  ```
  def getVertex(ov:A): B
  ```

  returns the label from vertex `ov` in graph **this**

  – Precondition:

  ```
  G(this) = (V,E,VL,EL) && ov IN V
  ```

– Postcondition:

```
Result == VL(ov)
```

Question: Can we weaken the precondition?

- Accessor method

```
def getEdge(v1:A, v2:A): C
```

returns the label on the edge from vertex `v1` to vertex `v2` in graph `this`.

– Precondition:

```
G(g) = (V,E,VL,EL) && (v1,v2) IN E
```

– Postcondition:

```
Result == EL((v1,v2))
```

Question: Can we weaken the precondition?

- Accessor method

```
def allVertices: List[A]
```

returns a List of all the vertices in graph `this`.

– Precondition:

```
G(this) = (V,E,VL,EL)
```

– Postcondition:

```
(ForAll ov:: ov IN Result <=> ov IN V) &&
length(Result) == size(V)
```

This returns the set `V` as a Scala `List` without duplicates. The order of the elements in the list is not specified. (Remember that the set `VertexType` is not necessarily partially or totally ordered, but it does support equality comparisons)

- Accessor method

```
def fromEdges(v1:A): List(A)
```

returns a sequence of all vertices `v2` such that there is an edge from vertex `v1` to vertex `v2` in graph `this`.

– Precondition:

```
G(this) = (V,E,VL,EL) && v1 IN V
```

– Postcondition:

```
(ForAll v2:: v2 IN Result <=> (v1,v2) IN E) &&
length(Result) == (# v2 :: (v1,v2) IN E)
```

Question: Can we remove the precondition conjunct `v1 IN V`?

Method call `fromEdges(v1)` probably should return `List()` when `v1` does not appear in `this`. This will make it can work well with the Wizard's Adventure game. To do so, we can redefine the precondition and postcondition to specify appropriate behavior.

- Revised Precondition (weaker):

```
G(this) = (V,E,VL,EL)
```

- Revised Postcondition (stronger):

```
(v1 NOT_IN V => Result == List())
&&
(v1 IN V =>
    (ForAll v2:: v2 IN Result <=> (v1,v2) IN E) &&
    length(Result) == (# v2 :: (v1,v2) IN E) )
```

- Accessor method

```
def allVerticesLabels: List[(A,B)]
```

returns a sequence of all pairs `(v,l)` such that `v` is a vertex and `l` is it's label in graph `this`.

- Precondition:

```
G(this) = (V,E,VL,EL)
```

- Postcondition:

```
(ForAll v, l:: (v,l) IN Result <=> (v,l) IN VL) &&
length(Result) == size(VL)
```

- Accessor method

```
def fromEdgesLabels(v1:A): List[(A,C)]
```

returns a List of all pairs `(v2,l)` such that there is an edge `(v1,v2)` labelled with `l` in graph `this`.

- Precondition:

```
G(this) = (V,E,VL,EL) && v1 IN V
```

- Postcondition:

```
(ForAll v2, l :: (v2,l) IN Result <=> ((v1,v2),l) IN EL)
&& length(Result) == (# v2 :: (v1,v2 ) IN E)
```

Question: Can we remove the precondition conjunct `v1 IN V`?

Method call `fromEdgesLabels(v1)` probably should return `List()` when `v1` does not appear in `this`. This will make it can work well with the

Wizard's Adventure game. To do so, we can redefine the precondition and postcondition to specify appropriate behavior.

- Revised Precondition (weaker):

```
G(this) = (V,E,VL,EL)
```

- Revised Postcondition (stronger):

```
(v1 NOT_IN V => Result == List())
&&
(v1 IN V =>
  (ForAll v2, l :: (v2,l) IN Result <=> ((v1,v2),l) IN EL)
  && length(Result) == (# v2 :: (v1,v2 ) IN E) )
```

1.7.5 Abstract interface in Scala

We specify the abstract interface for the family of Scala implementations of the `Digraph` ADT using a generic Scala trait `Digraph[A,B,C]`, defined in file `Digraph.scala`.

The trait defines the interface to the mutator and accessor methods. The construtor and destructor methods are left for the implementing class to implement. We document the semantics as comments in the Scala source code.

1.8 List Implementation

This section gives an implementation of the ADT that uses Scala lists to represent the vertex and edge sets.

Conceptually, the approach taken here is a variation on the *adjacency matrix* representation of a graph. However, unlike the typical presentation in an algorithms and data structures course, the approach here does not restrict the set of vertices to be from a finite range of integers.

1.8.1 Labelled digraph representation

We represent the List implementation of the Labelled Digraph ADT as a a Scala class `DigraphList[A,B,C]` that extends `Digraph[A,B,C]`. (Remember that type variable `A` is `VertexType`, `B` is `VertexLabelType`, and `C` is `EdgeLabelType`.)

We use the following *primary constructor* for this Scala class.

```
class DigraphList[A,B,C] private  // private constructor
    (private val vs: List[(A,B)],
     private val es: List[(A,A,C)])
    extends Digraph[A,B,C] { ... }
```

Above we use Scala features that we have not used before.

- We declare a Scala constructor parameter (i,e. object field) as `var`, `val`, or neither.
  - `var` means that the Scala compiler automatically generates a mutable instance variable with both public getter and setter methods.
  - `val` means that the Scala compiler automatically generates an immutable instance variable with a public getter method.
  - Neither means that any variable that the Scala compiler generates will be immutable and will not have automatically generated public getter or setter methods. If the parameter is not used outside the initialization of the class, then no instance variable is generated.
- We can add the modifier `private` to Scala constructor parameters: `private var` or `private val`. This means the compiler will not generate public getter and setter methods.
- We can declare the primary constructor as `private`. That means it can only be called from inside the class. (In such a case, we may need to have an auxiliary constructor that is public.)

Now let's consider how we use these fields to represent the abstract data type.

In an instance `DigraphList(vs,es)`:

- `vs` is a list of tuples `(v,vl)` where
  - `v` has type `VertexType` (i.e., `A`) and represents a vertex of the digraph
  - `vl` has type `VertexLabelType` (i.e., `B`) and is the label associated with vertex `v`
  - a vertex `v` occurs at most once in `vs` (i.e., `vs` encodes a function from vertices to vertex labels)

  Note: A list list `vs` is sometimes called an *association list* because it associates a key (e.g., `v`{.scala]) with its value (e.g., `vl`).
- `es` is a list of tuples `(v1,v2,el)` where
  - `v1` and `v2` are vertices occurring in `vs`, representing a directed edge from `v1` to `v2`
  - `el` has type `EdgeLabelType` (i.e., `C`) and is the label associated with edge `(v1,v2)`
  - an edge `(v1,v2)` occurs at most once in `es` (i.e., `es` encodes a function from edges to edge labels)

In terms of the abstract model, `vs` encodes `VL` directly and, because `VL` is a total function on `V`, it encodes `V` indirectly. Similarly, `es` encodes `EL` directly and `E` indirectly.

Of course, there are many other ways to represent the graph as lists. This representation is biased for a context where, once built, the labelled digraph is relatively static and the most frequent operations are the retrieval of labels

attached to vertices or edges. That is, it is biased toward the Adventure game use case

1.8.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labelled

> *Any Scala Digraph value `DigraphList(vs,es)` with abstract model `G = (V,E,VL,EL)`, appearing in either the implicit or explicit parameters or return value of an operation, must also satisfy the following:*

```
(ForAll v, l :: (v,l) IN vs  <=> (v,l) IN VL ) &&
(ForAll v1, v2, m :: (v1,v2,m) IN es  <=> ((v1,v2),m) IN EL )
```

1.8.3 Scala implementation

See the Digraph List implementation at `DigraphList.scala` and some testing code at `Test_DigraphList.scala`.

In addition to the private primary constructor discussed above, `DigraphList` implements a public zero-parameter auxiliary constructor that returns a new empty instance of the Digraph ADT. This is the public constructor required by the `Digraph` specification.

```
def this() {
    this(Nil,Nil)
}
```

This constructor has the following contract, as required by the abstract specification:

- Precondition: `true`
- Postcondition: `G(this') == ({},{},{},{})`

Auxiliary constructors in Scala must call another constructor, eventually calling the primary constructor.

As a convenience, `DigraphList` also implements a public copy constructor that constructs this instance to have same state as an existing instance.

```
def this(dg: DigraphList[A,B,C]) {
    this(dg.vs,dg.es)
}
```

This constructor has the following contract:

- Precondition: `G(dg) = (V,E,VL,EL)`
- Postcondition: `G(this') == G(dg)`

1.8.4 Improvements to the list implementation

Based on the list-based design and implementation above, what improvements should we consider?

Here are some possibilities.

1. The current list implementations of methods such as `removeVertex` and `updateVertex` do some unnecessary work with respect to the implementation invariant. This could be eliminated.

2. The data representation (i.e., implementation invariant) could be changed to allow, for example, multiple occurrences of vertices in the vertex list. This would avoid the checks of `hasVertex` in `addVertex` and `updateVertex`. Then, as it does above, `removeVertex` needs to remove all occurrences of the vertex.

   Other functions would need to be modified accordingly so that they only access the first occurrence of a vertex (especially the `allVertices` and `allVerticesLabels` methods).

   A similar change could be made to the list of edges.

3. Most of the methods throw a `sys.error` exception when the vertex they reference does not exist.

   A better Scala functional design would be to redefine these functions to return an `Option` or `Either` value. This would eliminate most of the `hasVertex` checks and make the functions defined on all possible inputs.

   Alternatively, we could redefine the methods to give other meaningful behavior in those circumstances, as suggested by some of the questions in the ADT specification.

   Either approach would require changes to the overall Labelled Digraph ADT specification and its abstract interface.

4. New methods could be added to the Labelled Digraph ADT—such as an equality check on graphs or functions to apply various graph algorithms.

   Alternatively, these methods could be a separate abstraction layered on top of the existing ADT specification.

5. Existing methods could be eliminated. For example, if the graph is only constructed and used for retrieval, then the remove and update functions could be eliminated.

1.9 Map Implementation

This section gives an implementation of the ADT that uses an instance of `scala.collection.immutable.HashMap` to map a vertex to the set of outgoing edges from that vertex,

The approach taken here is also a variation on the *adjacency list* representation of a graph.

1.9.1 Labelled digraph representation

We represent the Map implementation of the Labelled Digraph ADT as a a Scala class `DigraphMap[A,B,C]` that extends `Digraph[A,B,C]`. (Remember that type variable `A` is `VertexType`, `B` is `VertexLabelType`, and `C` is `EdgeLabelType`.)

We use the following primary constructor for this Scala class.

```
import scala.collection.immutable.HashMap

class DigraphMap[A,B,C] private
    (private val m: HashMap[A,(B,List[(A,C)])])
    extends Digraph[A,B,C] { ... }
```

This implementation represents a labeled digraph as an instance of the Scala class `DigraphMap(m)`, where `m` is a Scala `Map` implemented as a hash array mapped trie (i.e., `HashMap`).

Note: The `HashMap` collection requires that its keys be hashable.

An instance of `DigraphMap(m)` corresponds to the abstract model as follows:

- The keys for `Map m` are from `VertexType` (i.e., `A`).
- `Map m` is defined for all keys `v1` in vertex set `V` and undefined for all other possible keys.
- For some vertex `v1`, the value of `m` at key `v1` is a pair `(l,es)` where
  - `l` is an element of `VertexLabelType` (i.e., `B`) and is the label associated with `v1`, that is, `l == VL(v1)`.
  - `es` is the list of all tuples `(v2,el)` such that `(v1,v2) IN E`, `el IN EdgeLabelType` (i.e., `B`{.scala]), and `el == EL((v1,v2))`. That is, `(v1,v2)` is an edge and `el` is its label.

1.9.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the map-based version of the Labelled Digraph ADT:

> *Any Scala Digraph value `DigraphMap(m)` with abstract model `G = (V,E,VL,EL)`, appearing in either the implicit or explicit parameters or return value of an operation, must also satisfy the following:*

```
(ForAll v1, l, es ::
    ( m(v1) defined && m(v1) == (l,es) ) <=>
    ( VL(v1) == l &&
    (ForAll v2, el :: (v2,el) IN es <=>
```

```
                                EL((v1,v2)) == el) ) )
```

1.9.3 Scala implementation

See the Digraph Map implementation at `DigraphMap.scala` and some testing code at `Test_DigraphMap.scala`.

As with the Digraph List implementation, the Map implementation has two auxiliary constructors—a zero-parameter constructor and a copy constructor.

1.9.4 Improvements to the map implementation

All the improvements suggested for the list-based implementation apply to the map-based implementation except for the first.

For large graphs, the map-based implementation should perform better than the list-based implementation.

For large graphs with many outgoing edges on each vertex, it might be useful to implement the edge-list itself with a `HashMap`.

Alternatively, a `HashMap` could use use pairs `(v1,v2)` for its keys. This would give a more direct analog to an array-based adjacency matrix representation. The vertex labels could be represented separately by a list or map that associates a vertex with its label.

1.10 What Next?

TODO

1.11 Chapter Source Code

TODO

1.12 Exercises

TODO

1.13 Mealy Machine Simulator Project

1.13.1 Project introduction

In this project, you are asked to design and implement Scala modules to represent Mealy Machines and to simulate their execution.

This kind of machine is a useful abstraction for simple controllers that listen for input events and respond by generating output events. For example in an automobile application, the input might be an event such as “fuel level low” and the output might be command to “display low-fuel warning message”.

In the theory of computation, a *Mealy Machine* is a *finite-state automaton* whose output values are determined both by its current state and the current input. It is a *deterministic finite state transducer* such that, for each state and input, at most one transition is possible.

Appendix A of the Linz textbook [8] defines a Mealy Machine mathematically by a tuple

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$$

where

Q is a finite set of internal states
Σ is the input alphabet (a finite set of values)
Γ is the output alphabet (a finite set of values)
$\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
$\theta : Q \times \Sigma \longrightarrow \Gamma$ is the output function
q_0 is the initial state of M (an element of Q)

In an alternative formulation, the transition and output functions can be combined into a single function:

$$\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$$

We often find it useful to picture a finite state machine as a *transition graph* where the states are mapped to vertices and the transition function represented by directed edges between vertices labelled with the input and output symbols.

1.13.2 Mealy Machine project exercises

1. Specify, design, and implement a general representation for a Mealy Machine as a set of Scala definitions implementing an abstract data type. It should hide the representation of the machine behind an abstract interface and should have, at least, the following public operations.

   - Constructor `MealyMachine(s)` creates a new machine with initial (and current) state `s` and no transitions.
   - Mutator method `addState(s)` adds a new state `s` to this machine and returns an `Either` wrapping the modified machine or an error message.
   - Mutator method `addTransition(s1,in,out,s2)` adds a new transition to this machine and returns an `Either` wrapping the modified machine or an error message. From state `s1` with input `in`, the modified machine outputs `out` and transitions to state `s2`.
   - Mutator method `addResets` adds all reset transitions to this machine and returns the modified machine. This operation makes the transition function a total function by adding any missing transitions from a state back to the initial state.

- Mutator method `setCurrent(s)` sets the current state of this machine to `s` and returns an `Either` wrapping the modified machine or an error message.
- Accessor method `getCurrent` returns the current state of this machine.
- Accessor method `getStates` returns a list of the elements of the state set of this machine.
- Accessor method `getInputs` returns a list of the input set of this machine.
- Accessor method `getOutputs` returns a list of the output set of this machine.
- Accessor method `getTransitions` returns a list of the transition set of this machine. Tuple `(s1,in,out,s2)` occurs in the returned list if and only if, from state `s1` with input `in`, the machine outputs `out` and moves to state `s2`.
- Accessor method `getTransitionsFrom(s)` returns an `Either` wrapping a list of the set of transitions enabled from state `s` of this machine or an error message.

Note: It is possible to use a Labelled Digraph ADT module in the implementation of the Mealy Machine. A state is a vertex of the graph, transition is an edge of the graph, and an `(in,out)` is a label for an edge.

2. Given the above implementation for a Mealy Machine ADT, design and implement a separate Scala module that simulates the execution of a Mealy Machine. It should have, at least, the following public operations.
   - Mutator `move(m,in)` moves machine `m` from the current state given input `in` and returns an `Either` wrapping a tuple `(mm,out)` or an error message. The tuple gives the modified machine `mm` and the output `out`.
   - Mutator method `simulate(m,ins)` simulates execution of machine `m` from its current state through a sequence of moves for the inputs in list `ins` and returns an `Either` wrapping a tuple `(mm,outs)` or an error message. The tuple gives the modified machine `mm` after the sequence of moves and the output list `outs`.
3. Implement a Scala abstract data type that uses a different representation for the Mealy Machine. Make sure the simulator module still works correctly.

1.14 Acknowledgements

I specified the Doubly Labelled Digraph ADT for Assignment #1 in CSci 556 (Multiparadigm Programming) in Spring 2015. This assignment was motivated

by underlying data structure for the Wizard's Adventure Game from [1]. I developed the list- and map-based Haskell implementations as my solution for that assignment. In addition, I developed a list-based implementation in Elixir and used it to implement the Adventure game. In Spring 2016, I developed list- and map-based Scala implementations for CSci 555 (Functional Programming).

In Spring 2016, I defined the Mealy Machine Simulator as a programming assignment in the Scala-based CSci 555 (Functional Programming) course.

In Spring 2017, I created a Labelled Digraph ADT document by adapting and revising comments from the Haskell implementations of the Labelled Digraph abstract data type. I also included some content from my notes on Data Abstraction [6]. I also revised the Mealy Machine assignment description to create a Mealy Machine Exercise document.

In 2018, I merged and revised these documents to become a new Chapter 23, Data Abstraction Revisited, in the textbook *Exploring Languages with Interpreters and Functional Programming (ELIFP)*. In 2022, I reordered a few chapters of ELIFP; the Data Abstraction Revisited chapter became new Chapter 22.

In 2019, I adapted the Data Abstraction Revisited chapter and parts of the Data Abstraction chapter (7) of ELIFP and the Scala-based source code comments to develop these notes.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed matreials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

1.15 Terms and Concepts

TODO: Update

Data abstraction; module, goals of modularization (comprehensibility, independent development, changeability), information hiding, secret, encapsulation, contract (precondition, postcondition, invariant), interface, abstract interface; client-supplier relationship, design criteria for interfaces; abstract data type (ADT), instance; specification of ADTs using name, sets, signatures, and semantics; constructor, accessor, mutator, and destructor operations; axiomatic and constructive semantics; abstract model, interface and implementation invariant; using mathematical concepts to model the data abstraction; graph, digraph, labelled graph, multigraph, set, sequence, total and partial functions,

relation; adventure game; use of Scala trait and generics to define ADT interface, Scala constructors (private, auxiliary), builtin List and Map (HashMap) data structures; adjacency matrix, adjacency list.

Mealy Machine, simulator, finite-state automaton (machine), deterministic finite state transducer, state, transition, transition graph.

1.16 References

[1] Conrad Barski. 2011. *Land of Lisp: Learn to program in Lisp, one game at a time*. No Starch Press, San Francisco, California, USA.

[2] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[3] Frederick P. Brooks, Jr. 1987. No silver bullet: Essence and accident in software engineering. *IEEE Computer* 20, 4 (1987), 10–19.

[4] Frederick P. Brooks, Jr. 1995. 'No Silver Bullet' refired. In *The mythical man month* (Anniversary). Addison-Wesley, Boston, Massachusetts, USA.

[5] H. Conrad Cunningham. 2019. *Recursion concepts and terminology: Scala version*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/RecursionStyles/Scala/RecursionStylesScala.html

[6] H. Conrad Cunningham. 2022. *Notes on data abstraction*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DataAbstraction.html

[7] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications*. D. C. Heath, Lexington, Massachusetts, USA.

[8] Peter Linz. 2017. *Formal languages and automata* (Sixth ed.). Jones & Bartlett, Burlington, Massachusetts, USA.

[9] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[10] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[11] David L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[12] David L. Parnas. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 1 (1979), 128–138.

[13] David L. Parnas. 1979. The modular structure of complex systems. *IEEE Transactions on Software Engineering* SE-11, 13 (1979), 128–138.

[14] David L. Parnas. 2001. Some software engineering principles. In *Software fundamentals: Collected papers by David L. Parnas*, Daneil M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.

[15] David M. Weiss. 2001. Introduction: On the criteria to be used in decomposing systems into modules. In *Software fundamentals: Collected papers by David L. Parnas*, Daniel M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.