CSci 450: Functional Programming
Fundamental Concepts

H. Conrad Cunningham

18 January 2017

Contents

Acknowledgements: I adapted and revised much of this work in Summer and Fall 2016 from my previous materials:

- Evolving Computer Hardware Affects Programming Languages from my notes *Effect of Computing Hardware Evolution on Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages
- Programming Paradigms from chapter 1 of my *Notes on Functional Programming with Haskell*
- History of Programming Languages from my notes *History of Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages. Those remarks drew on the following:
  - O'Reilly History of Programming Languages poster
  - Wikipedia article History of Programming Languages
- Abstraction from my notes on *Data Abstraction*
- Motivating Functional Programming from chapter 1 of my *Notes on Functional Programming with Haskell*
- Review of Relevant Mathematics from chapter 2 of my *Notes on Functional Programming with Haskell*

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of January 2017 is a recent version of Firefox from Mozilla.

TODO:

- Update the module objectives to match the expanded role of this module.
- Elaborate descriptions of "other paradigms"
- Focus the object orientation better for this course?
- Better integrate the discussion of abstraction.
- Exercises?

Fundamental Concepts

Module Introduction

The goal of this module is motivate the concept of functional programming and relate it to other approaches to programming and to the relevant mathematical concepts.

Upon successful completion of this module, students should be able to:

1. describe the evolution of computers and programming languages since the 1940's and its impact upon contemporary programming language design and implementation
    - Corresponding Course Outcome 8: relate the concepts and methods for functional programming in the chosen language to their previous programming knowledge and experiences
2. distinguish among the primary programming paradigms, in particular to be able to relate functional programming to their previous knowledge of imperative programming
    - Corresponding Course Outcome 1: describe the concepts of the functional programming paradigm
    - Corresponding Course Outcome 8: relate the concepts and methods for functional programming in the chosen language to their previous programming knowledge and experiences
    - Corresponding Course Outcome 9: be more confident as computer programmers using a variety of languages and techniques
3. understand the abstraction and mathematical concepts that underlie the functional programming paradigm
    - Corresponding Course Outcome 1: describe the concepts of the functional programming paradigm
    - Corresponding Course Outcome 8: relate the concepts and methods for functional programming in the chosen language to their previous programming knowledge and experiences
    - Corresponding Course Outcome 9: be more confident as computer programmers using a variety of languages and techniques
    - Corresponding Course Outcome 10: appreciate the ability to exploit the abstraction and mathematical properties of a purely functional programs and languages
    - Corresponding Course Outcome 13: exploit the mathematical nature of purely functional programs to prove properties, transform programs,

and synthesize (i.e., derive) programs from their specifications

Students entering the course should already have sufficient familiarity with the relevant mathematical concepts from the prerequisite courses. However, they may need to relate the mathematics with the programming constructs in functional programming.

Before we look at Haskell programming, let's consider the development of programming languages over the past 70 years, examine the primary programming programming paradigms, consider key concepts, and review the related mathematical concepts.

Evolving Computer Hardware Affects Programming Languages

To put our study in perspective, let's examine the effect of computing hardware evolution on programming languages by considering a series of questions.

When were the first "modern" computers developed? That is, programmable electronic computers.

Although the mathematical roots of computing go back more than a thousand years, it is only with the invention of the programmable electronic digital computer during the World War II era of the 1930s and 1940s that modern computing began to take shape.

One of the first computers was the ENIAC (Electronic Numerical Integrator and Computer), developed in the mid-1940s at the University of Pennsylvania. When construction was completed in 1946, the ENIAC cost about $500,000. In today's terms, that is more than $5,000,000. It weighed 30 tons, occupied as much space as a small house, and consumed 160 kilowatts of electric power.

The ENIAC and most other computers of that era were designed for military purposes, such as calculating firing tables for artillery or breaking codes. As a result, many observers viewed the market for such devices to be quite small. The observers were wrong!

Electronics technology has improved greatly in 70 years. Today, a computer with the capacity of the ENIAC would be smaller than a coin from our pockets, would consume little power, and cost just a few dollars on the mass market.

How have computer systems and their use evolved over the past 70 years?

- Contemporary processors are much smaller and faster. They use much less power, cost much less money, and operate much more reliably.
- Contemporary "main" memories are much larger in capacity, smaller in physical size, and faster in access speed. They also use much less power, cost much less money, and operate much more reliably.

- The number of processors per machine has increased from one to many. First, channels and other co-processors were added, then multiple CPUs. Today, computer chips for common desktop and mobile applications have several processors–cores–on each chip, plus specialized processors such as graphics processing units (GPUs) for data manipulation and parallel computation. This trend toward multiprocessors will likely continue given that physics dictates limits on how small and fast we can make computer processors; to continue to increase in power means increasing parallelism.
- Contemporary external storage devices are much larger in capacity, smaller in size, faster in access time, and less expensive to construct.
- The number of computers available per user has increased from much less than one to many more than one.
- Early systems were often locked into rooms, with few or no direct connections to the external world and just a few kinds of input/output devices. Contemporary systems may be on the user's desktop or in the user's backpack, be connected to the internet, and have many kinds of input/output devices.
- The range of applications has increased from a few specialized applications (e.g., code-breaking, artillery firing tables) to almost all human activities.
- The cost of the human staff to program, operate, and support computer systems has probably increased somewhat (in constant dollars).

How have these changes affected programming practice?

- In the early days of computing, computers were very expensive and the cost of the human workers to use them relatively less. Today, the opposite holds. So we need to maximize human productivity.
- In the early days of computing, the slow processor speeds and small memory sizes meant that programmers had to control these precious resources to be able to carry out most routine computations. Although we still need to use efficient algorithms and data structures and use good coding practices, programmers can now bring large amounts of computing capacity to bear on most problems. We can use more computing resources to improve productivity to program development and maintenance. The size of the problems we can solve computationally has increased.
- In the early days of computing, multiple applications and users usually had to share one computer. Today, we can often apply many processors for each user and application if needed. Increasingly, applications must be able to use multiple processors effectively.
- Security on early systems meant keeping the computers in locked rooms and restricting physical access to those rooms. In contemporary networked systems with diverse applications, security has become a much more difficult issue with many aspects.

- Currently, industry can devote considerable hardware and software resources to the development of production software.

The first higher-level programming languages began to appear in the 1950s. IBM released the first compiler for a programming language in 1957–for the scientific programming language Fortran. Although Fortran has evolved considerably during the past 60 years, it is still in use today.

How have the above changes affected programming language design and implementation over the past 60 years?

- Contemporary programming languages often use automatic memory allocation and deallocation (e.g., garbage collection) to manage a program's memory. Although programs in these languages may use more memory and processor cycles than hand-optimized programs, they can increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.
- Contemporary programming languages are often implemented using an interpreter instead of a compiler that translates the program to the processor's machine code–or be implemented using a compiler to a virtual machine instruction set (which is itself interpreted on the host processor). Again they use more processor and memory resources to increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.
- Contemporary programming languages should make the capabilities of contemporary multicore systems conveniently and safely available to programs and applications. To fail to do so limits the performance and scalability of the application. Think Erlang, Scala, and Clojure versus C, C++, and Java.
- Contemporary programming languages increasingly incorporate declarative features (higher order functions, recursion, immutable data structures, generators, etc.). These features offer the potential of increasing programming productivity, increasing the security and reliability of programs, and more conveniently and safely providing access to multicore processor capabilities. Think Scala, Clojure, and Java 8 and beyond versus C, C++, and older Java.
- etc.

As we study programming and programming languages in this and other courses, we need to keep the nature of the contemporary programming scene in mind.

Now let's consider the concept of programming paradigm and the major paradigms.

Primary Programming Paradigms

According to Timothy Budd, a *programming paradigm* is "a way of conceptualizing what it means to perform computation, of structuring and organizing how tasks are to be carried out on a computer." [from: Timothy A. Budd. *Multiparadigm Programming in Leda*, Addison-Wesley, 1995, page 3]

Historically, computer scientists have classified programming languages into one of two primary paradigms: *imperative* and *declarative*.

In recent years, many imperative languages have added more declarative features, so the distinction between languages has become blurred. However, the concept of programming paradigm is still meaningful.

Imperative paradigm

A program in the imperative paradigm has an *implicit state* (i.e., values of variables, program counters, etc.) that is modified (i.e., side-effected or mutated) by *constructs* (i.e., commands) in the source language.

As a result, such languages generally have an explicit notion of *sequencing* (of the commands) to permit precise and deterministic control of the state changes.

Imperative programs thus express *how* something is to be computed.

Consider the following fragment of Java code:

```
int count =  0;
int maxc  = 10;
while (count <= maxc) {
    System.out.println(count) ;
    count = count + 1
}
```

In this fragment, the program's state includes at least the values of the variables `count` and `maxc`, the sequence of output lines that have been printed, and an indicator of which statement to execute next (i.e., location or program counter).

The assignment statement changes the value of `count` and the `println` statement adds a new line to the output sequence. These are *side effects* of the execution.

Similarly, Java executes these commands in sequence, causing a change in which statement will be executed next. The purpose of the `while` statement is to cause the statements between the braces to be executed zero or more times. The number of times depends upon the values of `count` and `maxc` and how the values change within the `while` loop.

We call this state *implicit* because the aspects of the state used by a particular statement are not explicitly specified; the state is assumed from the context of

the statement. Sometimes a statement can modify aspects of the state that are not evident from examining the code fragment itself.

The Java variable `count` is *mutable* because its value can change. After the declaration, `count` has the value 0. At the end of the first iteration of the `while` loop, it has value 1. After the `while` loop exits, it has a value 10. So a reference to `count` yields different values depending upon the state of the program at that point.

The Java variable `maxc` is also `mutable`, but this code fragment does not change its value.

Imperative languages are the "conventional" or "von Neumann languages" discussed by John Backus in his 1977 Turing Award address. (See the section with excerpts from that address.) They are well suited to traditional computer architectures.

Most of the languages in existence today are primarily imperative in nature. These include Fortran, C, C++, Java, C#, Python, Lua, and JavaScript.

Declarative paradigm

A program in the declarative paradigm has *no implicit* state. Any needed state information must be handled explicitly.

A program is made up of *expressions* (or terms) that are *evaluated* rather than commands that are executed.

Repetitive execution is accomplished by *recursion* rather than by sequencing.

Declarative programs express *what* is to be computed (rather than how it is to be computed).

The declarative paradigm is often divided into two types: *functional* (or applicative) and *relational* (or logic).

Functional paradigm

In the functional paradigm the underlying model of computation is the mathematical concept of a *function*.

In a computation, a function is applied to zero or more arguments to compute a single result; that is, the result is deterministic (or predictable).

Consider the following Haskell code. (Don't worry about the details of the language for now. We study the syntax and semantics of Haskell in the first parts of this course.)

```
counter :: Int -> Int -> String
counter count maxc
```

```
    | count <= maxc = show count ++ "\n" ++ counter (count+1) maxc
    | otherwise     = ""
```

This fragment is similar to the Java fragment above. This Haskell code defines a function `counter` that takes integer arguments `count` and `maxc` and returns a string consisting of a sequence of lines with the integers from `count` to `maxc` such that each would be printed on a separate line. (It does not print the string.)

In the execution of a function call, `counter` references the *values* of `count` and `maxc` corresponding to the explicit arguments of the function call. These values are not changed within the execution of that function call. However, the values of the arguments can be changed as needed for a subsequent *recursive* call of `counter`.

We call the state of `counter` *explicit* because it is passed in arguments of the function call. These parameters are *immutable* (i.e., their values cannot change) within the body of the function. That is, any reference to `count` or `maxc` within a call gets the same value.

In a pure functional language like Haskell, the names like `count` and `maxc` are said to be *referentially transparent*. In the same context (such as the body of the function), they always have the same value. Although a name may need to be defined before it is used, the order of the execution of the expressions within a function body does not matter.

There are no "loops". The functional paradigm uses recursive calls to carry out a task repeatedly.

In most programming languages that support functional programming, functions are treated as *first class* values. That is, like other data types, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions.

A function that can take functions as arguments or return functions in the result is called a *higher-order function*. A function that does not take or return functions is thus a *first-order function*. Most imperative languages do not fully support higher-order functions.

The higher-order functions in functional programming languages enable regular and powerful abstractions and operations to be constructed. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

Purely functional languages include Haskell, Idris, Miranda, Hope, and Backus' FP.

Hybrid functional languages with significant functional subsets include Scala, F#, OCaml, SML, Erlang, Elixir, Lisp, Clojure, and Scheme.

Mainstream imperative languages such as Java (beginning with version 8), C#, Python, Ruby, Groovy, Rust, and Swift have recent feature extensions that make

them hybrid languages as well.

Relational (or logic) paradigm

In the relational (logic) paradigm, the underlying model of computation is the mathematical concept of a *relation* (or a *predicate*).

A computation is the (nondeterministic) association of a group of values—with backtracking to resolve additional values.

Consider the following SWI-Prolog code. (Don't worry about the details of the language.)

```
counter(X,Y,S) :- count(X,Y,R), atomics_to_string(R,'\n',S).

count(X,X,[X]).
count(X,Y,[])     :- X  > Y.
count(X,Y,[X|Rs]) :- X < Y, NX is X+1, count(NX,Y,Rs).
```

This fragment is somewhat similar to the Java and Haskell fragments above. It can be used to generate a string with the integers from `X` to `Y` where each integer would be printed on a separate line. (As with the Haskell fragment, it does not print the string.)

This program fragment defines a *database* consisting of four *clauses*.

The clause `count(X,X,[X]).` defines a *fact*. For any *variable* value `X` and a list `[X]` consisting of the single value `X` is asserted to be true.

The other three clauses are *rules*. The left-hand-side of `:-` is true if the right-hand-side is also true. For example,

```
count(X,Y,[]) :- X > Y.
```

asserts that

```
count(X,Y,[])
```

is true when `X > Y`. The empty brackets denote and empty list of values.

As a logic or relational language, we can *query* the database for any missing components. For example,

```
count(1,1,Z).
```

yields the value `Z = [1]`. However,

```
count(X,1,[1]).
```

yields the value `X = 1`. If more than one answer is possible, the program can generate all of them in some nondeterministic order.

So, in some sense, where imperative and functional languages only run a computation in one direction and give a single answer, Prolog can potentially run a computation in multiple directions and give multiple answers.

Example relational languages include Prolog, Parlog, and miniKanren.

Most Prolog implementations have imperative features such as the cut and the ability to assert and retract clauses.

Other Programming Paradigms

Note: I added the "Other Programming Paradigms" section to use this module with a Programming Languages Organization course. It is not fully integrated into the module. It can be omitted for a Haskell-based class on functional programming.

The imperative-declarative taxonomy described above divides programming styles and language features on how they handle state and how they are executed.

The computing community often speaks of other paradigms – procedural, modular, object-oriented, concurrent, parallel, language-oriented, scripting, reactive, and so forth. The definitions of these "paradigms" may be quite fuzzy and vary significantly from one writer to another. Sometimes a term is chosen for "marketing" reasons – to associate a language with some trend even though the language may be quite different from others in that paradigm – or to make a language seem different and new even though it may not be significantly different.

These paradigms tend to divide up programming styles and language features along different dimensions than the primary taxonomy above. Often the languages we are speaking of are subsets of the imperative paradigm.

Procedural

For example, *procedural* languages are imperative languages built around the concept of subprograms – procedures and functions. Programmers divide a program's behavior into these program units that call each other. Subprograms may be nested inside of other subprograms to control the range of the program where the name of the subprogram is known.

Languages like C, Fortran, Pascal, Lua, and Python are primarily procedural languages, although most have evolved to support other styles.

Modular

Similarly, *modular programming* refers more to a design method for programs and program libraries than to languages. It means to decompose a program

into packages of functionality that can be developed separately. Key design and implementation details are hidden inside the module – the principle of *information hiding*. The interactions among modules is kept at a minimum – exhibit a low degree of coupling. A language that provides constructs for defining modules, packages, namespaces, or compilation units can assist in writing modular programs.

Object oriented

The dominant paradigm since the early 1990s has been the *object-oriented paradigm*. Because this paradigm is likely familiar with most readers, let's examine it in more detail.

We discuss object orientation in terms of an object model. Our *object model* includes four basic components:

1. objects (i.e., abstract data structures)
2. classes (i.e., abstract data types)
3. inheritance (hierarchical relationships among abstract data types)
4. subtype polymorphism

Note: Some writers consider *dynamic binding* a basic component of object orientation. Here we consider it an implementation technique for subtype polymorphism.

Objects

An *object* is a characterized by three essential characteristics:

a. state
b. operations
c. identity

An object is a separately identifiable entity that has a set of operations and a state that records the effects of the operations. An object is typically a *first class* entity that can be stored in variables and passed to or returned from subprograms.

The *state* is the collection of information held (i.e., stored) by the object.

- It can change over time.
- It can change as the result of an operation performed on the object.
- It cannot change spontaneously.

The various components of the state are sometimes called the *attributes* of the object.

An *operation* is a procedure that takes the state of the object and zero or more arguments and changes the state and/or returns one or more values. Objects permit certain operations and not others.

If an object is *mutable*, then an operation may change the stored state so that a subsequent operation on that object acts upon the modified state; the language is thus imperative.

If an object is *immutable*, then an operation cannot change the stored state; instead it returns a new object with the modified state.

Identity means we can distinguish between two distinct objects (even if they have the same state and operations).

As an example, consider an object for a student desk in a simulation of a classroom. Student desks are distinct from each other. The relevant *state* might be attributes like location, orientation, person using, items in the basket, items on top, etc. The relevant *operations* might be state-changing operations (*mutators*) such as "move the desk", "seat student", or "remove from basket" or might be state-observing operations (*accessors*) such as "is occupied" or "report items on desktop".

A language is *object-based* if it supports objects as a language feature.

Object-based languages include Ada, Modula, Clu, C++, Java, Scala, C#, and Smalltalk. Pascal (without module extensions), Algol, Fortran. and C are not inherently object-based.

Some writers require that an object have additional characteristics, but these notes consider these as important but *non-essential* characteristics of objects:

c. encapsulation
d. independent lifecycle

The state may be *encapsulated* within the object – that is, not be directly visible or accessible from outside the object.

The object may also have an *independent lifecycle* – that is, the object may exist independently from the program unit that created it.

We do not include these as essential characteristics because they do not seem required by the object metaphor. There are languages that use a modularization feature to enforce encapsulation separately from the object (or class) feature. Also, there are languages that may have local "objects" within a function or procedure.

Classes

A *class* is a template or factory for creating objects.

- A class describes a collection of related objects (i.e., *instances* of the class).

- Objects of the same class have common operations and a common set of possible states.
- The concept of class is closely related to the concept of *type*.

A class description includes definitions of:

- operations on objects of the class
- the possible set of states

As an example, again consider a simulation of a classroom. There might be a class `StudentDesk` from which specific instances can be created as needed.

An object-based language is *class-based* if the concept of class occurs as a language feature and every object has a class.

Class-based languages include Clu, C++, Java, Scala, C#, Smalltalk, Ruby, and Ada 95. Ada 83 and Modula are not class-based.

At their core, JavaScript and Lua are object-based but not class-based.

In statically typed, class-based languages such as Java, Scala, C++, and C# classes are treated as types. Instances of the same class have the same type.

However, some dynamically typed languages may have a more general concept of type: If two objects have the same set of operations, then they have the same type regardless of how the object was created. Languages such as Smalltalk and Ruby have this characteristic – sometimes informally called *duck typing*. (If it walks like a duck and quacks like a duck, then it is a duck.)

Inheritance

A class C *inherits* from class P if C's objects form a subset of P's objects.

- Class C's objects must support all of the class P's operations (but perhaps are carried out in a special way).
- Class C may support additional operations and an extended state (i.e., more information fields).
- Class C is called a *subclass* or a *child* or *derived class*.
- Class P is called a *superclass* or a *parent* or *base* class.
- Class P is sometimes called a *generalization* of class C; class C is a *specialization* of class P.

The importance of inheritance is that it encourages sharing and reuse of both design information and program code. The shared state and operations can be described and implemented in base classes and shared among the subclasses.

As an example, again consider the student desks in a simulation of a classroom. The `StudentDesk` class might be derived (i.e., inherit) from a class `Desk`, which in

turn might be derived from a class `Furniture`. In diagrams, it is the convention to draw arrows from the subclass to the superclass.

```
Furniture <-- Desk <-- StudentDesk
```

The simulation might also include a `ComputerDesk` class that also derives from `Desk`.

```
Furniture <-- Desk <-- ComputerDesk
```

In Java and Scala, we can express the above inheritance relationships using the `extends` keyword as follows.

```
class Furniture  // extends cosmic root class for references by default
{   ...          // (i.e., java.lang.Object or scala.AnyRef)
}

class Desk extends Furniture
{   ...
}

class StudentDesk extends Desk
{   ...
}

class ComputerDesk extends Desk
{   ...
}
```

Both `StudentDesk` and `ComputerDesk` objects will need operations to simulate a `move` of the entity in physical space. The `move` operation can thus be implemented in the `Desk` class and shared by objects of both classes. Invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` will be bound to the general `move` in the `Desk` class.

The `StudentDesk` class might inherit from a `Chair` class as well as the `Desk` class.

```
Furniture <-- Chair <-- StudentDesk
```

Some languages support *multiple inheritance* as shown above for `StudentDesk` (e.g., C++, Eiffel). Other languages only support a single inheritance hierarchy.

Because multiple inheritance is both difficult to use correctly and to implement in a compiler, the designers of Java and Scala did not include multiple inheritance of classes as features. Java has a single inheritance hierarchy with a top-level class named `Object` from which all other classes derive (directly or indirectly). Scala is similar, with the corresponding top-level class named `AnyRef`.

```
class StudentDesk extends Desk, Chair  // NOT VALID in Java
{   ...
```

```
}
```

To see some of the problems in implementing multiple inheritance, consider the above example. Class `StudentDesk` inherits from class `Furniture` through two different paths. Do the data fields of the class `Furniture` occur once or twice? What happens if the intermediate classes `Desk` and `Chair` have conflicting definitions for a data field or operation with the same name?

The difficulties with multiple inheritance are greatly decreased if we restrict ourselves to inheritance of class *interfaces* (i.e., the signatures of a set of operations) rather than a supporting the inheritance of the class *implementations* (i.e., the instance data fields and operation implementations). Since interface inheritance can be very useful in design and programming, the Java designers introduced a separate mechanism for that type of inheritance.

The Java `interface` construct can be used to define an interface for classes separately from the classes themselves. A Java `interface` may inherit from (i.e., `extend`) zero or more other `interface` definitions.

```
interface Location3D
{   ...
}

interface HumanHolder
{   ...
}

interface Seat extends Location3D, HumanHolder
{   ...
}
```

A Java `class` may inherit from (i.e., `implement`) zero or more interfaces as well as inherit from (i.e., `extend`) exactly one other `class`.

```
interface BookHolder
{   ...
}

interface BookBasket extends Location3D, BookHolder
{   ...
}

class StudentDesk extends Desk implements Seat, BookBasket
{ ...
}
```

This definition requires the `StudentDesk` class to provide actual implementations for all the operations from the `Location3D`, `HumanHolder`, `BookHolder`, `Seat`, and `BookBasket` interfaces. The `Location3D` operations will, of course, need

to be implemented in such a way that they make sense as part of both the `HumanHolder` and `BookHolder` abstractions.

The Scala `trait` provides a more powerful, and more complex, mechanism than Java's original `interface`. In addition to signatures, a `trait` can define method implementations and data fields. These traits can be added to a class in a controlled, linearized manner to avoid the semantic and implementation problems associated with multiple inheritance of classes. This is called *mixin* inheritance.

Java 8 generalized interfaces to allow default implementations of methods.

Most statically typed languages treat subclasses as *subtypes*. That is, if `C` is a subclass of `P`, then the objects of type `C` are also of type `P`. We can *substitute* a `C` object for a `P` object in all cases.

However, the inheritance mechanism in languages in most class-based languages (e.g., Java) does not automatically preserve substitutability. For example, a subclass can change an operation in the subclass to do something totally different from the corresponding operation in the parent class.

Polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. Polymorphism appears in several forms in programming languages. We will discuss these more later.

Subtype polymorphism (sometimes called *polymorphism by inheritance*, *inclusion polymorphism*, or *subtyping*) means the association of an operation invocation (i.e., procedure or function call) with the appropriate operation implementation in an inheritance (subtype) hierarchy.

This form of polymorphism is usually carried out at run time. That implementation is called *dynamic binding*. Given an object (i.e., class instance) to which an operation is applied, the system will first search for an implementation of the operation associated with the object's class. If no implementation is found in that class, the system will check the superclass, and so forth up the hierarchy until an appropriate implementation is found. Implementations of the operation may appear at several levels of the hierarchy.

The combination of dynamic binding with a well-chosen inheritance hierarchy allows the possibility of an instance of one subclass being substituted for an instance of a different subclass during execution. Of course, this can only be done when none of the extended operations of the subclass are being used.

As an example, again consider the simulation of a classroom. As in our discussion of inheritance, suppose that the `StudentDesk` and `ComputerDesk` classes are derived from the `Desk` class and that a general `move` operation is implemented as a part of the `Desk` class. This could be expressed in Java as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}

class ComputerDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}
```

As we noted before, invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` instance will be bound to the general `move` in the `Desk` class.

Extending the example, suppose that we need a special version of the `move` operation for `ComputerDesk` objects. For instance, we need to make sure that the computer is shut down and the power is disconnected before the entity is moved.

To do this, we can define this special version of the `move` operation and associate it with the `ComputerDesk` class. Now a call to `move` a `ComputerDesk` object will be bound to the special `move` operation, but a call to `move` a `StudentDesk` object will still be bound to the general `move` operation in the `Desk` class.

The definition of `move` in the `ComputerDesk` class is said to *override* the definition in the `Desk` class.

In Java, this can be expressed as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}

class ComputerDesk extends Desk
```

```
{   ...
    public void move(...)
    ...
}
```

A class-based language is *object-oriented* if class hierarchies can be incrementally defined by an inheritance mechanism and the language supports polymorphism by inheritance along these class hierarchies.

Object-oriented languages include C++, Java, Scala, C#, Smalltalk, and Ada 95. The language Clu is class-based, but it does not include an inheritance facility.

Other object-oriented languages include Objective C, Object Pascal, Eiffel, and Oberon 2.

Prototype based

Classes and inheritance are not the only way to support relationships among objects in object-based languages. Another approach of growing importance is the use of *prototypes*.

A *prototype-based* language does not have the concept of class as defined above. It just has objects. Instead of using a class to instantiate a new object, a program copies (or clones) an existing object – the *prototype* – and modifies the copy to have the needed attributes and operations.

Each prototype consists of a collection of *slots*. Each slot is filled with either a data attribute or an operation

This cloning approach is more flexible than the class-based approach.

In a class-based language, we need to define a new class or subclass to create a variation of an existing type. For example, we may have a `Student` class. If we want to have students who play chess, then we would need to create a new class, say `ChessPlayingStudent`, to add the needed data attributes and operations.

In a class-based language, the boundaries among categories of objects specified by classes should be *crisply defined*. That is, an object is in a particular class or it is not. Sometimes this crispness may be unnatural.

In a prototype-based language, we simply clone a student object and add new slots for the added data and operations. This new object can be a prototype for further objects.

In a prototype-based language, the boundaries between categories of objects created by cloning may be fuzzy. One category of objects may tend to blend into others. Sometimes this fuzziness may be more natural.

Consider categories of people associated with a university. These categories may include `Faculty`, `Staff`, `Student`, and `Alumnus`. Consider a *student* who

gets a BSCS degree, then accepts a *staff* position as a programmer and stays a student by starting an MS program part-time, and then later teaches a course as a graduate student. The same person who started as a student thus evolves into someone who is in several categories later. And he or she may also be a chess player.

Instead of static, class-based inheritance and polymorphism, some languages exhibit prototype-based *delegation*. If the appropriate operation cannot be found on the current object, the operation can be delegated to its prototype, or perhaps to some other related, object. This allows dynamic relationships along several dimensions. It also means that the "copying" or "cloning" may be partly logical rather than physical.

Prototypes and delegation are more basic mechanisms than inheritance and polymorphism. The latter can often be implemented (or perhaps "simulated") using the former.

Self, JavaScript, and Lua are prototype-based languages.

History of Programming Languages

From the instructor's perspective, key languages and milestones in the history of programming languages include the following.

1950's

- Fortran, 1957; imperative, first compiler, math-like language for scientific programming, developed at IBM by John Backus, influenced most subsequent languages, enhanced versions still in use today (first language learned by the instructor in 1974)
- Lisp, 1958; mix of imperative and functional features, code and data have same format (i.e., the language is *homoiconic*), related to Church's lambda calculus theory, recursion, syntactic macros, automatic storage management, higher order functions, developed at MIT by John McCarthy, influenced most subsequent languages/research, enhanced versions still in use today
- Algol, 1958, 1960; imperative, nested block structure, lexical scoping, BNF invented to define syntax, call by name parameter passing, developed by an international team from Europe and the USA, influenced most subsequent languages
- COBOL, 1959; imperative, key designer Grace Hopper, business/accounting programming, still in use today

1960's

- Simula; 1962, 1967; imperative, original purpose for discrete-event simulation, developed in Norway by Ole-Johan Dahl and Kristen Nygaard,

first object-oriented language (in Scandinavian school of object-oriented languages), influenced subsequent object-oriented languages

- Snobol, 1962; string processing, patterns as first class data, backtracking on failure, developed at AT&T Bell Laboratories by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky
- PL/I, 1964; IBM-designed language to merge scientific (Fortran), business (COBOL), and systems programming (second language learned by the instructor in 1975)
- BASIC, 1964; interactive computing in early timesharing and microcomputers environments, developed at Dartmouth College by John G. Kemeny and Thomas E. Kurtz
- Algol 68, 1968; ambitious and rigorously defined successor to Algol 60, greatly influenced computing science theory and subsequent language designs, not widely implemented because of its complexity

1970's

- Pascal, 1970; simplified Algol family language designed by Niklaus Wirth (Switzerland) because of frustration with complexity of Algol 68, structured programming, one-pass compiler, important for teaching in 1980s and 1990s, Pascal-P System virtual machine implemented on many early microcomputers (Pascal used by UM CIS in CS1 and CS2 until 1999)
- Prolog, 1972; first and most widely used logic (relational) programming language, originally developed by a team headed by Alain Colmerauer (France), rooted in first-order logic, most modern Prolog implementations based on the Edinburgh dialect (which ran on the Warren Abstract Machine), used extensively for artificial intelligence research in Europe, influenced subsequent logic languages and also Erlang
- C, 1972; systems programming language for Unix operating system, widely used today, developed by Dennis Ritchie at AT&T Bell Labs, influenced many subsequent languages
- Smalltalk, 1972; ground-up object-oriented programming language, message-passing between objects (in American school of object-oriented languages), developed by Alan Kay and others at Xerox PARC, influenced many subsequent object-oriented languages
- ML, 1973; mostly functional, polymorphic type system on top of Lisp-like language, pioneering statically typed functional programming, algebraic data types, developed by Robin Milner at the University of Edinburgh, influenced subsequent functional programming languages
- Scheme, 1975; minimalist dialect of Lisp with lexical scoping, tail call optimization, first-class continuations, developed by Guy Steele and Gerald Jay Sussman at MIT, influenced subsequent languages/research

- Icon, 1977; structured programming successor to Snobol, developed by a team led by Ralph Griswold at the University of Arizona, uses goal-directed execution based on success or failure of expressions

1980's

- C++, 1980; C with Simula-like classes, developed by Bjarne Stroustrup (Denmark)
- Ada, 1983; designed by US DoD-funded committee as standard language for military applications, design led by Jean Ichbiah and a team in France, statically typed, block structured, modular, synchronous message passing, object-oriented extensions in 1995 (instructor studied this language while working in the military aerospace industry 1980-83)
- Eiffel, 1985; object-oriented language designed with strong emphasis on software engineering concepts such as design by contract and command-query separation, developed by Bertrand Meyer (France)
- Objective C, 1986; C with Smalltalk-like messaging, developed by Brad Cox and Tom Love at Stepstone, selected by Steve Jobs' NeXT systems, picked up by Apple when NeXT absorbed, key language for OS X and iOS
- Erlang, 1986; message-passing concurrency on functional programming base (actors), fault-tolerant/real-time systems, dynamic typing, virtual machine, originally used in telephone switches, developed by Joe Armstrong, Robert Virding and Mike Williams at Ericsson (Sweden)
- Self, 1986; dialect of Smalltalk, developed by David Ungar and Randall Smith while at Xerox PARC, Stanford University, and Sun Microsystems, first prototype-based object-oriented language (which influenced JavaScript, Lua), used virtual machine with just-in-time compilation (JIT) which influenced Java HotSpot
- Perl, 1987; dynamic programming language originally focused on providing powerful text-processing facilities based around regular expressions, developed by Larry Wall

1990's

- Haskell, 1990; purely functional language with non-strict semantics (i.e., lazy evaluation) and strong static typing, developed by an international committee of functional programming researchers, widely used in research community
- Python, 1991; dynamically typed, multiparadigm language, developed by Guido van Rossum (Netherlands), increasing in use
- Ruby, 1993; dynamically typed, object-oriented, supports reflective/metaprogramming and internal domain-specific languages, developed by Yukihiro "Matz" Matsumoto (Japan), popularized by Ruby on Rails web framework, influenced subsequent languages

- Lua, 1993; minimalistic language designed for embedding in any environment supporting standard C, developed by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (Brazil), dynamic typing, lexical scoping, first-class functions, garbage collection, tail recursion optimization, pervasive table/metatable data structure, facilities for prototype object-oriented programming, coroutines, used as scripting language in games, etc.
- R, 1993; statistical computing and graphics, open-source implementation of the language S, developed by Ross Ihaka and Robert Gentleman (New Zealand)
- Java, 1995; class-based object-oriented programming, statically typed, virtual machine, version 8 has functional programming features (higher order functions, streams); developed by Sun Microsystems, now Oracle
- JavaScript, 1995 (standardized as ECMAScript); designed for embedding in web pages, developed by Brendan Eich at Netscape in 12 days to meet a deadline, internals influenced by Scheme and Self but using a Java-like syntax, dynamic typing, first-class functions, prototype-based object-oriented programming, became popular quickly before language design made clean
- PHP, 1995; originally developed by Rasmus Lerdorf (Canada), evolved organically, server-side scripting language for dynamic web applications,
- OCaml (originally Objective Caml), 1996; a dialect of ML that adds object-oriented constructs, focusing on performance and practical use, developed by a team lead by Xavier Leroy (France)

2000's

- C#, 2001; class-based object-oriented programming, statically typed, mostly imperative, language that runs on Microsoft's Common Language Infrastructure (in some sense, Microsoft's response to Sun's Java)
- F#, 2002; OCaml re-envisioned for Microsoft's Common Language Infrastructure (.Net), developed by a team led by Don Syme at Microsoft Research in the UK, replaces OCaml's object and module systems with .Net concepts,
- Scala, 2003; hybrid functional and object-oriented language that runs on the Java Virtual Machine, developed by Martin Odersky's team at EPFL in Switzerland
- Groovy, 2003; dynamically typed, object-oriented "scripting" language that runs on the Java Virtual Machine, originally proposed by James Strachan
- miniKanren, 2005; a family of relational programming languages, developed by Dan Friedman's team at Indiana University, implemented as an extension

to other languages (originally in Scheme, most popular current usage probably in Clojure)

- Clojure, 2007; Lisp dialect developed by Rich Hickey, runs on Java Virtual Machine, Microsoft Common Language Runtime, and JavaSript platform, emphasis on functional programming, concurrency (e.g., software transactional memory), and immutable data structures

2010's

- Idris, 2011 (1.0 release planned 2017); eagerly evaluated, Haskell-like functional language with dependent types, incorporating ideas from proof assistants, intended for practical programming, developed by Edwin Brady (UK)
- Julia, 2012; dynamic programming language designed to address high-performance numerical and scientific programming
- Elixir, 2012 (1.0 release 2014); functional concurrent programming language, dynamic strong typing, metaprogramming, protocols, Erlang actors, runs on Erlang Virtual Machine, influenced by Erlang, Ruby, and Clojure, developed by a team led by Jose Valim (Brazil)
- Elm, 2012 (0.17 release May 2016); Haskell-like functional programming language that compiles to JavaScript and supports reactive-style programming, developed by Evan Czaplicki (original version for his senior thesis at Harvard)
- Rust, 2012 (1.0 release 2015); systems programming language that incorporates contemporary language concepts and focuses on safety and performance, meant to replace C and C++, developed originally at Mozilla Research by Graydon Hoare
- Swift, 2014; Apple's replacement for Objective C that incorporates contemporary language concepts and focuses on program safety; "Objective C without the C"
- etc.

Abstraction

As computing scientists and computer programmers, we should remember the maxim:

> *Simplicity is good; complexity is bad.*

The most effective weapon that we have in the fight against complexity is *abstraction*. What is abstraction?

Abstraction is *concentrating on the essentials and ignoring the details.*

Sometimes abstraction is described as *remembering the "what" and ignoring the "how"*.

Kinds of abstraction

Large complex systems can only be made understandable by decomposing them into modules. When viewed from the outside, from the standpoints of users, each *module* should be simple, with the complexity hidden inside.

We strive for modules that have simple interfaces that can be used without knowing the implementations. Here we use *interface* to mean any information about the module that other modules must assume to be able to do their work correctly.

Two kinds of abstraction are of interest to computing scientists: *procedural abstraction* and *data abstraction*.

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

When we develop an algorithm following the top-down approach, we are practicing procedural abstraction. At a high level, we break the problem up into several tasks. We give each task a name and state its requirements, but we do not worry about how the task is to be accomplished until we expand it at a lower level of our design.

When we code a task in a programming language, we will typically make each task a subprogram (procedure, function, subroutine, method, etc.). Any other program component that calls the subprogram needs to know its interface (name, parameters, return value, assumptions, etc.) but does not need to know the subprogram's internal implementation details. The internal implementation can be changed without affecting the caller.

In data abstraction, the focus is on the problem's data rather than the tasks to be carried out.

Procedures and functions

Generally we make the following distinctions among subprograms:

- A *procedure* is (in its pure form) a subprogram that takes zero or more arguments but does not return a value. It is executed for its effects, such as changing values in a data structure within the program, modifying its reference or value-result arguments, or causing some effect outside the program (e.g., displaying text on the screen or reading from a file).

- A *function* is (in its pure form) a subprogram that takes zero or more arguments and returns a value but that does not have other effects.
- A *method* is a procedure or function associated with an object or class in an object-oriented program. Some object-oriented languages use the metaphor of message-passing. A method is the feature of an object that receives a message. In an implementation, a method is typically a procedure or function associated with the object; the object may be in implicit parameter of the method.

Of course, the features of various programming languages and usual practices for their use may not follow the above pure distinctions. For example, a language may not distinguish between procedures and functions. One term or another may be used for all subprograms. Procedures may return values. Functions may have side effects. Functions may return multiple values. The same subprogram can sometimes be called either as a function or procedure.

Nevertheless, it is good practice to maintain the distinction between functions and procedures for most cases in software design and programming.

In Haskell, the primary unit of procedural abstraction is the pure function. Haskell also groups functions and other declarations into a program unit called a `module`. A `module` explicitly exports selected functions and keep others hidden.

Motivating Functional Programming: John Backus

John W. Backus (December 3, 1924 – March 17, 2007) was a pioneer in research and development of programming languages. He was the primary developer of Fortran while a developer at IBM in the mid-1950s. Fortran is the first widely used high-level language. Backus was also a participant in the international team that designed the influential languages Algol 58 and Algol 60 a few years later. The notation used to describe the Algol 58 language syntax–Backus-Naur Form (BNF)–bears his name. This notation continues to be used to this day.

In 1977, ACM bestowed its Turing Award on Backus in recognition of his career of accomplishments. (This award is sometimes described as the "Nobel Prize for computer science".) The annual recipient of the award gives an address to a major computer science conference. Backus's address was titled "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs".

Although functional languages like Lisp go back to the late 1950's, Backus's address did much to stimulate research community's interest in functional programming languages and functional programming over the past 40 years.

The next subsection gives excerpts from Backus' Turing Award address published as the article "Can Programming Be Liberated from the von Neumann Style? A

Functional Style and Its Algebra of Programs" (*Communications of the ACM*, Vol. 21, No. 8, pages 613–41, August 1978).

Excerpts from Backus's Turing Award Address

Programming languages appear to be in trouble. Each successive language incorporates, with little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. ... Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs. ... In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived of it ... [in the 1940's], it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this ... concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent form the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must either be generated by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent, and so on.

Surely there must be a less primitive way of making big changes in the store than

by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. ...

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our ... old belief that there is only one kind of computer is the basis our our belief that there is only one kind of programming language, the conventional–von Neumann–language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-at-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements in the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of

programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. ...

Aside on the disorderly world of statements

Backus states that "the world of statements is a disorderly one, with few mathematical properties". Even in 1977 this was a bit overstated since Dijkstra's work on the weakest precondition (*wp*) calculus and other work on axiomatic semantics had already appeared.

However, because of the referential transparency property of purely functional languages, reasoning can often be done in an equational manner within the context of the language itself. We examine this convenient approach later in the course.

In contrast, the *wp*-calculus and other axiomatic semantic approaches must project the problem from the world of programming language statements into the world of predicate calculus, which is much more orderly. We leave this study to other courses (such as CSci 550, Program Semantics and Derivation).

Perspective from four decades later

In his Turing Award Address, Backus went on to describe FP, his proposal for a functional programming language. He argued that languages like FP would allow programmers to break out of the von Neumann bottleneck and find new ways of thinking about programming.

FP itself did not catch on, but the widespread attention given to Backus' address and paper stimulated new interest in functional programming to develop by researchers around the world. Modern languages like Haskell developed partly from the interest generated.

In the past 15 years, the software industry has become more interested in functional programming. Some functional programming features now appear in most mainstream programming languages (e.g., in Java 8). This interest seems to driven primarily by two concerns:

- managing the complexity of large software systems effectively
- exploiting multicore processors conveniently and safely

The functional programming paradigm is able to address these concerns because of such properties such as referential transparency, immutable data structures, and composability of components. We look at these aspects later in the course.

Review of Relevant Mathematics

This section reviews the mathematical concepts of functions and a few other mathematical concepts used in this course. The concept of function in functional programming corresponds closely to the mathematical concept of function.

Natural numbers and ordering

Several of the examples in this course use natural numbers.

For this course, we consider the set of *natural numbers* N to consist of 0 and the positive integers.

Inductively, $n \in N$ if and only if one of the following holds

- $n = 0$
- There exists $m \in N$ such that $m = S(n)$

where S is the *successor* function, which returns the next element.

Furthermore,

- No element is the successor of more one other natural number.
- 0 is not the successor of any natural number. That is, it is the least (base) element.

The natural numbers thus form a *totally ordered* set in conjunction with the binary relation $\leq$ (less or equal). That is, the relation $\leq$ satisfies the following properties on set N:

- $n \leq n$ for all $n \in N$ (*reflexivity*)
- $m \leq n$ and $n \leq m$ implies $m = n$ (*antisymmetry*)
- $m \leq n$ and $n \leq p$ implies $m \leq p$ (*transitivity*)
- Either $m \leq n$ or $n \leq m$ for all $m, n \in N$ (*trichotomy*)

It is also a *partial ordering* because it satisfies the first three properties above.

For all $m, n \in N$, we can define the other ordering relations in terms of $=$, $\neq$, and $\leq$ as follows:

- $m < n$ (less) to mean $m \leq n$ and $m \neq n$. We say that m is smaller (or simpler) than n.
- $m > n$ (greater) to mean $n \leq m$ and $n \neq m$. We say that m is larger (or more complex) than n.
- $m \geq n$ (greater or equal) to mean the same as $n \leq m$

Functions

As we have studied in mathematics courses, a *function* is a mapping from a set A into a set B such that each element of A is mapped into a unique element of B.

- The set A (on which f is defined) is called the *domain* of f.
- The set of all elements of B into which f maps elements of A is called the *range* (or *codomain*) of f, and is denoted by $f(A)$.

If f is a function from A into B, then we write:

$$f : A \to B$$

We also write the equation

$$f(a) = b$$

to mean that the *value* (or *result*) from *applying* function f to an element $a \in A$ is an element $b \in B$.

If a function

$$f : A \to B$$

and $A \subseteq A'$, then we say that f is a *partial function* from A' to B and a *total function* from A to B. That is, there are some elements of A' on which f may be undefined.

Recursive functions

Informally, a *recursive function* is a function defined using recursion.

In computing science, *recursion* is a method in which an "object" is defined in terms of smaller (or simpler) "objects" of the same type. A recursion is usually defined in terms of a recurrence relation.

A *recurrence relation* defines an "object" x_n as some combination of zero or more other "objects" x_i for $i < n$. Here $i < n$ means that i is smaller (or simpler) than n. If there is no smaller object, then n is a base object.

For example, consider a recursive function to compute the sum s of the first n natural numbers.

We can define a recurrence relation for s with the following equations:

$s(n) = 0$, *if* $n = 0$
$s(n) = n + s(n-1)$, *if* $n \geq 1$

For example, consider $s(3)$,

$s(3) = 3+s(2) = 3+(2+s(1)) = 3+(2+(1+s(0))) = 3+(2+(1+0)) =$
6

Mathematical induction on natural numbers

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments.

We can define *fact* using the product operator as follows:

$fact(n) = \prod_{i=1}^{i=n} i$

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

$fact'(n) = 1$, *if* $n = 0$
$fact'(n) = n \times fact'(n-1)$, *if* $n \geq 1$

It is, of course, easy to see that the recurrence relation definition is equivalent to the previous definition. But how can we prove it?

To prove that the above definitions of the factorial function are equivalent, we can use *mathematical induction* over the natural numbers.

Mathematical induction: To prove a logical proposition $P(n)$ holds for any natural number n, we must show two things:

- For the *base case* $n = 0$, show that $P(0)$ holds.
- For the *inductive case* $n = m + 1$, show that, if $P(m)$ holds for some natural number m, then $P(m + 1)$ also holds.

The $P(m)$ assumption is called the *induction hypothesis*.

Now let's prove that the two definitions *fact* and *fact'* are equivalent.

Prove For all natural numbers n, $fact(n) = fact'(n)$.

Base case $n = 0$**.**

	$fact(0)$
=	{ definition of *fact* (left to right) }
	$(\Pi i : 1 \leq i \leq 0 : i)$
=	{ empty range for Π, 1 is the identity element of $\times$ }
	1
=	{ definition of *fact'* (first leg, right to left) }
	$fact'(0)$

Inductive case $n = m + 1$**.**
Given induction hypothesis $fact(m) = fact'(m)$, prove $fact(m+1) = fact'(m+1)$.

	$fact'(m + 1)$
=	{ definition of *fact* (left to right) }
	$(\Pi i : 1 \leq i \leq m + 1 : i)$
=	{ $m + 1 > 0$, so $m + 1$ term exists, split it out }

	$(m+1) \times (\Pi i : 1 \leq i \leq m : i)$
=	{ definition of *fact* (right to left) }
	$(m+1) \times$ *fact*(m)
=	{ induction hypothesis }
	$(m+1) \times$ *fact'*(m)
=	{ $m+1 > 0$, definition of *fact'* (second leg, right to left) }
	fact'$(m+1)$

Therefore, we have proved *fact*(n) = *fact'*(n) for all natural numbers n. QED

In the inductive step above, we explicitly state the induction hypothesis and assertion we wish to prove in terms of a different variable name (m instead of n) than the original statement. This helps to avoid the confusion in use of the induction hypothesis that sometimes arises.

We use an equational style of reasoning. To prove that an equation holds, we begin with one side and prove that it is equal to the other side. We repeatedly "substitute equals for equal" until we get the other expression.

Each transformational step is justified by a definition, a known property of arithmetic, or the induction hypothesis.

The structure of this inductive argument closely matches the structure of the recursive definition of *fact'*.

What does this have to do with functional programming? Many of the functions we will define in this course have a recursive structure similar to *fact'*. The proofs and program derivations that we do will resemble the inductive argument above.

Recursion, induction, and iteration are all manifestations of the same phenomenon.

Operations

A function

$$\oplus : (A \times A) \to A$$

is called a *binary operation on A*. We usually write binary operations in *infix* form:

$$a \oplus a'$$

We often call a two-argument function of the form

$$\oplus : (A \times B) \to C$$

a binary operation as well. We can write this two argument function in the equivalent *curried* form:

$$\oplus : A \to (B \to C)$$

The curried form shows a multiple-paramenter function in a form where the function takes the arguments one at a time, returning the resulting function with one fewer arguments.

Let $\oplus$ be a binary operation on some set A and x, y, and z be elements of A. We can define the following kinds of properties.

- Operation $\oplus$ is *closed* on A if and only if $x \oplus y \in A$ for any $x, y \in A$. That is, the operation is a total function on its domain.
- Operation $\oplus$ is *associative* if and only if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for $x, y, z \in A$.
- Operation $\oplus$ is *commutative* (also called *symmetric*) if and only if $x \oplus y = y \oplus x$ for $x, y \in A$.
- An element e of set A is
  - a *left identity* of $\oplus$ if and only if $e \oplus x = x$ for any $x \in A$
  - a *right identity* of $\oplus$ if and only if $x \oplus e = x$ for any $x \in A$
  - an *identity* of $\oplus$ if and only if it is both a left and a right identity.

  An identity of an operation is called a *unit* of the operation.
- An element z of set A is
  - a *left zero* of $\oplus$ if and only if $z \oplus x = z$ for any $x \in A$
  - a *right zero* of $\oplus$ if and only if $x \oplus z = z$ for any $x \in A$
  - a *zero* of $\oplus$ if and only if it is both a right and a left zero
- If e is the identity of $\oplus$ and $x \oplus y = e$ for some x and y, then
  - x is a *left inverse* of y
  - y is a *right inverse* of x.

  Elements x and y are inverses of each other if $x \oplus y = e = y \oplus x$.

For example, the addition operation + on natural numbers is closed, associative, and commutative and has the identity element 0. It has neither a left or right zero element and the only element with a left or right inverse is 0. If we consider the set of all integers, then all elements also have inverses.

Also, the multiplication operation * on natural numbers (or on all integers) is closed, associative, and commutative and has identity element 1 and zero element 0. Only value 1 has a left or right inverse.

However, the subtraction operation on natural numbers is not closed, associative, or commutative and has neither a left nor right zero. The value 0 is subtraction's right identity, but subtraction has no left identity. Each element is its own right and left inverse. If we consider all integers, then the operation is also closed.

Algebraic structures

We can also define *algebraic structures* for binary operations on a set with certain properties.

- If $\oplus$ is an *associative* operation closed on A, then $\oplus$ and A are said to form a *semigroup*.
- A semigroup that also has an *identity* element is called a *monoid*.
- A monoid that is also *commutative* is a *commutative monoid*.
- If every element of a monoid has an inverse then the monoid is called a *group*.
- If a monoid or group is also commutative, then it is said to be *Abelian*.

For example, addition on natural numbers forms a commutative monoid and on integers forms an Abelian group.

(TODO) Exercises

TODO