CSci 658: Software Language Engineering
Programming Paradigms

H. Conrad Cunningham

17 February 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall

P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

Programming Paradigms

Introduction

TODO: Add text and objectives

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" [Budd 1995 (p. 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 [Hudak 1989].

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 [Backus 1978]. (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 [Hudak 1989].

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

TODO: Perhaps introduce IO program for printing result of counter.

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 an upcoming chapter.)

```
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 two 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, but it inserts newline character at the end of each line.)

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. A name must be defined before it is used, but otherwise 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. (The implementation technique for first-order functions usually involves creation of a *lexical closure* holding the function and its environment.)

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, Elm, 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*) [Hudak 1989].

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 list `[X]` consisting of the single value `X`, `count(X,X,[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 an 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

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

TODO: Expand

Procedural languages, for example, 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

TODO: Expand

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 separate 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 (called *mutator*, setter, or command operations) such as "move the desk", "seat student", or "remove from basket" or might be state-observing operations (called *accessor*, getter, observer, or query operations) 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:

d. encapsulation
e. 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.

TODO: Give examples of languages without d or e – e.g., Lua, Oberon, C++.

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 (e.g., ⟵) 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
{   ...   }      // (java.lang.Object, 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

TODO: Give example

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.

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" [Backus 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 work by Hoare on *axiomatic semantics*, by Dijkstra on *weakest precondition (wp) calculus*, and by others 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 these notes.

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 21st Century, 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 these notes.

Conclusions

TODO: Add

Exercises

TODO: Add

Acknowledgements

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

- Primary Programming Paradigms from chapter 1 of my *Notes on Functional Programming with Haskell*, which I wrote originally in the mid-1990s for the Gofer dialect of Haskell but later updated to the 1998 and 2010 Haskell standards. I updated this content to expand the discussion of the paradigms to include examples.
- Object-Oriented programming paradigm from my notes *Introduction to Object Orientation*, which I wrote originally for the first UM C++ (CSci 490) and Java-based (CSci 211) classes in 1996 but expanded and adapted for other courses.
- Motivating Functional Programming from chapter 1 of my *Notes on Functional Programming with Haskell*.

In 2017, I continued to develop this material as a part of Chapter 1 of my Haskell-based programming languages "textbook". I added new material on other paradigms, particularly on the Prototype-based paradigm (first drafted in Fall 2016).

In Spring 2018, I pulled this Programming Paradigms document back out of the Fundamentals chapter to continue development. I wanted a less cluttered discussion of paradigms, particularly the object-oriented paradigm.

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

References

TODO: Complete and update

[**Backus 1978**] John Backus. 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 (ACM Turing Award Lecture, 1977).

[**Bird 1988**] Richard S. Bird and Phillip L. Wadler. *Functional Programming*, Prentice Hall, 1988.

[**Budd 1995**] Timothy A. Budd. *Multiparadigm Programming in Leda*, Addison-Wesley, 1995.

[**Budd 2000**] Timothy Budd. *Understanding Object-Oriented Programming with Java*, Updated Edition, Addison Wesley, 2000.

[**Craig 2007**] Iain D. Craig. *Object-Oriented Programming Languages*, Springer 2007. (Especially chapter 1 "Introduction" and chapter 3 "Prototype and Actor Languages")

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Horstmann 1995**] Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995. (Especially chapters 3-6 on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants" which influenced my views on object-oriented design and programming)

[**Hudak 1989**] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys*, Vol. 21, No. pp. 359-411, 1989. (Especially the "Introduction"w section.)

Terms and Concepts

TODO: Complete and update

Programming language paradigms (imperative, declarative, functional, relational or logic language), program state, implicit versus explicit state, execution of commands versus evaluation of expressions, abstraction, procedural abstraction, data abstraction, procedure, function, method