Functional Programming

This document was generated on 2016-09-26 at 23:14.

Contents

We now turn our attention to *procedural abstraction*, a strategy for decomposing complex programs into smaller pieces of code in the form of *functions* (also called *procedures* or *subroutines*; there are subtle differences in how these terms are used in various contexts, but for our purposes, we will treat them as synonyms). A function encapsulates some computation behind an interface, and as with any abstraction, the user of a function need only know what the function does and not how it accomplishes it. A function also generalize computation by taking in arguments that affect what it computes, and the result of the computation is the function's *return value*.

In this chapter, We start by discussing aspects of functions that are relevent to all procedural languages. We then take a closer look at *functional programming*, a programming paradigm that models computation after mathematical functions.

Functions

We first consider various schemes that are used for passing data to functions in the form of parameters and arguments. We make a distinction between the parameters that appear in a function definition, which are also called *formal parameters*, and the actual values that are passed to the function when it is called. The latter are often called *actual parameters*, but we will use the term *argument* to refer to these values and the shorthand *parameter* for formal parameters.

Keyword Arguments

Some languages allow, or even require, parameter names to be provided when calling a function. This strategy is called *named parameters* or *keyword arguments*.

Keyword arguments generally allow arguments to be provided in a different order than the parameter list of a function. In Python, for example, a keyword argument can be used for any parameter. Consider the following code:

```
def foo(x, y):
    print(x, y)
```

Calling `foo()` without keyword arguments passes the first argument as the first parameter, and the second argument as the second parameter:

```
>>> foo(1, 2)
1 2
```

The arguments can be reordered using the parameter names:

```
>>> foo(y = 1, x = 2)
2 1
```

A handful of languages require names to be provided for all or most arguments by default, as well as requiring that they be given in the same order as the parameters. The following is an example in Swift 3:

```
func greet(name: String, withGreeting: String) {
  print(withGreeting + " " + name)
}

greet(name: "world", withGreeting: "hello")
```

Calling `greet()` with the arguments in reverse order is erroneous.

Swift is also rare in that it allows different argument and parameter names to be specified for a parameter, meaning that the name that is provided for an argument when calling a function can differ from the internal name of the parameter used in the body of the function.

Default Arguments

In some languages, a function declaration or definition may be provided with a *default argument* value that allows the function to be called without that argument. This can be an alternative to overloading, where separate function definitions are written to handle the cases where an argument is present or missing.

The following is an example in Python:

```
def power(base, exponent = 2):
    return base ** exponent
```

The `power()` function can be called with a single argument, in which case the default argument `2` is used to compute the square of the number. It can also be called with two arguments to compute an arbitrary power:

```
>>> power(3)
9
>>> power(3, 4)
81
```

Parameters that have default arguments generally must appear at the end of the parameter list. Languages differ on when they evaluate the default argument, and in which environment. The most common strategy is to evaluate a default argument every time a function is called, but in the definition environment (static scope). Python is rare in that it only evaluates default arguments once, when the function definition statement is executed.

Variadic Functions

A language may provide a mechanism for a function to be called with a variable number of arguments. This feature is often referred to as *varargs*, and functions that make use of it are *variadic*. The mechanism may provide type safety, or it permit unsafe uses that result in erroneous or undefined behavior. A variadic parameter generally must appear at the end of a parameter list, and it matches arguments that remain once the non-variadic parameters are matched. Usually, only a single variadic parameter is allowed.

In languages that provide safe variadic functions, a common mechanism for doing so is to automatically package variable arguments into a container, such as an array or tuple. For example, the following Python function computes the product of its arguments:

```
def product(*args):
    result = 1
    for i in args:
        result *= i
    return result
```

The `*` in front of a parameter name indicates a variadic parameter, and the variable arguments are passed as a tuple bound to that name. The function above iterates over the elements of the tuple, updating the total product. In order to call `product()`, 0 or more arguments must be provided:

```
>>> product()
1
>>> product(1, 2, 3)
6
```

Python also provides variadic keyword arguments, which are packaged into a dictionary. Placing `**` in front of a parameter specifies that it is a variadic keyword parameter, and such a parameter must be the last one. As an example, the following function has both a non-keyword variadic parameter and a variadic keyword parameter, printing out the tuple corresponding to the former and the dictionary for the latter:

```
def print_args(*args, **kwargs):
    print(args)
    print(kwargs)

>>> print_args(3, 4, x = 5, y = 6)
(3, 4)
{'x': 5, 'y': 6}
```

Finally, Python allows a sequence or dictionary to be "unpacked" using the `*` or `**` operator, allowing the unpacked values to be used where a list of values is required. For example, the following unpacks a list to make a call to `product()`:

```
>>> product(*[1, 2, 3])
6
```

In Python, a variadic parameter can match arguments with any type, since Python is dynamically typed. In statically typed languages, variadic parameters are usually restricted to a single type, though the type may be polymorphic. For example, the following is a variadic method in Java:

```
public static void print_all(String... args) {
  for (String s : args) {
    System.out.println(s);
  }
}
```

The arguments to `print_all()` must be `Strings`, and they are packaged into a `String` array. Java also allows a single `String` array to be passed in as an argument:

```
print_all("hello", "world");
print_all(new String[] { "good", "bye" });
```

C and C++ also have a mechanism for variadic arguments, but it poses significant safety issues. In particular, it provides no information about the number of arguments and their types to the function being called. The following is an example of a function that adds its arguments:

```
#include <stdarg.h>

int sum(int count, ...) {
  va_list args;
  int total = 0;
  int i;
  va_start(args, count);
  for (i = 0; i < count; i++) {
    total += va_arg(args, int);
  }
  va_end(args);
  return total;
}
```

In this function, the first argument is assumed to be the number of remaining arguments, and latter are assumed to have type `int`. Undefined behavior results if either of these conditions are violated. Another strategy is to use a format string to determine the number and types of arguments, as used in `printf()` and similar functions. The lack of safety of variadic arguments enables vulnerabilities such as format string attacks.

C++11 provides variadic templates that are type safe. We will discuss them later in the course.

Parameter Passing

Another area in which languages differ is in the semantics they provide, and the mechanism, they use, for communicating arguments between a function and its caller. A function parameter may be unidirectional, used for only passing input to a function or only passing output from a function to its caller, or it may be bidirectional. These cases are referred to as *input*, *output*, and *input/output* parameters. A language need not support all three parameter categories.

Different parameter passing techniques, or *call modes*, are used by languages, affecting the semantics of arguments and parameters as well as what parameter categories they support. The following are specific call modes used by different languages:

- *Call by value*. A parameter represents a new variable in the frame of a function invocation. The argument value is copied into the storage associated with the variable. Call-by-value parameters only provide input to a function.

- *Call by reference*. An l-value must be passed as the argument, and the parameter aliases the object that is passed in. Any modifications to the paramater are reflected in the argument object. Thus, call by reference parameters provide both input and output. In C++, reference parameters provide call by reference, and they may be restricted to just input by declaring them `const`.

  Call by reference is sometimes used to refer to passing objects indirectly using pointers. The following function swaps object values using pointers:

  ```
  void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
  }

  int x = 3, y = 4;
  swap(&x, &y);
  ```

  Technically speaking, the arguments and parameters are separate pointer objects that are passed by value. The effect, however, emulates call by reference, enabling both input and output to be done through a parameter.

- *Call by result*. A parameter represents a new variable that is not initialized with a value from the caller. The caller specifies an l-value for the argument, and when the function call terminates, the final value of the parameter is copied to the l-value. Thus, call by result only provides output parameters.

- *Call by value-result*. This is the combination of call by value and call by result. The argument value is copied into a new variable corresponding to the parameter, and then upon return from the function, the value of the parameter is copied back to the l-value provided by the caller. This differs from call by reference in that copies are made upon entry and exit to the function. This can be seen the same l-value is passed to multiple parameters, as in the following example using C-like syntax:

```
int foo(int x, int y) {
  x++;
  return x - y;
}

int z = 3;
print(foo(z, z));
```

In this code, `x` and `y` are new variables that are initialized to the value of `z`, i.e. 3. The increment of `x` does not affect `y`, since they are separate variables, so the call to `foo()` returns 1. Thus, 1 is printed. (The final value of `z` depends on the semantics of the language as to whether it is copied from `x` or `y`.) If call by reference were used instead, then `x` and `y` would alias the same object, and the call to `foo()` would return 0.

- *Call by name.* In this mode, a full expression can be provided as an argument, but it is not evaluated at the time a function is called. Instead, the parameter name is replaced by the expression wherever the name occurs in the function, and the expression is evaluated at the time that it is encountered in the body. This is a form of *lazy evaluation*, where an value is not computed until it is needed.

  There is a subtle issue that arises in call by name. Consider the following code that uses C-like syntax but call by name:

  ```
  void bar(int x) {
    int y = 3;
    print(x + y);
  }

  int y = 1;
  bar(y + 1);
  ```

  If we replace the occurrence of the parameter `x` in `bar()` with the argument expression, we get `y + 1 + y` as the argument to `print()`. If this is evaluated in the environment of `bar()`, the result is 7. This is undesirable, since it means that the implementation detail of a local declaration of `y` changes the behavior of the function.

  Instead, the argument expression should be evaluated in the environment of the argument. This requires passing both the argument and its environment to the function invocation. Languages that use call by name often use a compiler-generated local function, called a *thunk*, to encapsulate the argument expression and its environment. This thunk is then passed to the invoked function, and it is the thunk is called whenever the parameter is encountered.

Call by value is the call mode used by most modern languages, including C, C++ (for non-reference parameters), Java, and Python. Programmers often mistakenly believe the latter two languages use call by reference, but in reality, they combine call by value with reference semantics. This combination is sometimes called "call by object reference." The following example illustrates that Python is call by value:

```
def swap(x, y):
    tmp = x
    x = y
    y = tmp

>>> x, y = 1, 2
>>> swap(x, y)
>>> x, y
(1, 2)
```

The erroneous `swap()` function merely changes the values of the local variables, which changes the objects they refer to, without affecting the variables used as arguments. This demonstrates that the storage for the global `x` and `y` is distinct from that of the parameters, so Python does not use call by reference. In fact, Python cannot even emulate call by reference in the manner that C and C++ pointers do.

Introduction to Scheme

In this section, we introduce a high-level programming language that encourages a functional style. Our object of study, a subset of the R5RS Scheme language, employs a very similar model of computation to Python's, but uses only expressions (no statements), specializes in symbolic computation, and employs only immutable values.

Scheme is a dialect of Lisp, the second-oldest programming language that is still widely used today (after Fortran). The community of Lisp programmers has continued to thrive for decades, and new dialects of Lisp such as Clojure have some of the fastest growing communities of developers of any modern programming language. To follow along with the examples in this text, you can download a Scheme interpreter or use an online interpreter.

Expressions

Scheme programs consist of expressions, which are either simple expressions, call expressions, or special forms. A simple expression consists of a literal or a symbol. A call expression consists of an operator expression followed by zero or more operand sub-expressions, as in Python. Both the operator and operand are contained within parentheses:

```
> (quotient 10 2)
5
```

Scheme exclusively uses prefix notation. Operators are often symbols, such as `+` and `*`. Call expressions can be nested, and they may span more than one line:

```
> (+ (* 3 5) (- 10 6))
19
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

Call expressions are compound forms that include arbitrary subexpressions. The evaluation procedure of call expressions proceeds from left to right: first the operator and operand expressions are evaluated in order, and then the function that is the value of the operator is applied to the arguments that are the values of the operands.

The `if` expression in Scheme is a *special form*, meaning that while it looks syntactically like a call expression, it has a different evaluation procedure. The general form of an `if` expression is:

```
(if <predicate> <consequent> <alternative>)
```

To evaluate an `if` expression, the interpreter starts by evaluating the `<predicate>` part of the expression. If the `<predicate>` evaluates to a true value, the interpreter then evaluates the `<consequent>` and returns its value. Otherwise it evaluates the `<alternative>` and returns its value. The `<alternative>` may be elided.

Numerical values can be compared using familiar comparison operators, but prefix notation is used in this case as well:

```
> (>= 2 1)
#t
```

The boolean values `#t` (or `true`) and `#f` (or `false`) in Scheme can be combined with boolean special forms, which have evaluation procedures as follows:

- `(and <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to `false`, the value of the `and` expression is `false`, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to true values, the value of the `and` expression is the value of the last one.
- `(or <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to a true value, that value is returned as the value of the `or` expression, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to `false`, the value of the `or` expression is `false`.
- `(not <e>)` The value of a not expression is `true` when the expression `<e>` evaluates to `false`, and `false` otherwise.

Definitions

Values can be named using the `define` special form:

```
> (define pi 3.14)
> (* pi 2)
6.28
```

New functions (usually called *procedures* in Scheme) can be defined using a second version of the `define` special form. For example, to define squaring, we write:

```
(define (square x) (* x x))
```

The general form of a procedure definition is:

```
(define (<name> <formal parameters>) <body>)
```

The `<name>` is a symbol to be associated with the procedure definition in the environment. The `<formal parameters>` are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The `<body>` is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied. The `<name>` and the `<formal parameters>` are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined square, we can now use it in call expressions:

```
> (square 21)
441

> (square (+ 2 5))
49

> (square (square 3))
81
```

User-defined functions can take multiple arguments and include special forms:

```
> (define (average x y)
    (/ (+ x y) 2))
> (average 1 3)
2
> (define (abs x)
    (if (< x 0)
        (- x)
       x))
> (abs -3)
3
```

Scheme supports local function definitions with static scope. We will defer discussion of this until we cover higher-order functions.

Anonymous functions are created using the `lambda` special form. A `lambda` is used to create a procedure in the same way as `define`, except that no name is specified for the procedure:

```
(lambda (<formal-parameters>) <body>)
```

The resulting procedure is just as much a procedure as one that is created using `define`. The only difference is that it has not been associated with any name in the environment. In fact, the following expressions are equivalent:

```
> (define (plus4 x) (+ x 4))
> (define plus4 (lambda (x) (+ x 4)))
```

Like any expression that has a procedure as its value, a lambda expression can be used as the operator in a call expression:

```
> ((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

Compound values

Pairs are built into the Scheme language. For historical reasons, pairs are created with the `cons` built-in function, and the elements of a pair are accessed with `car` and `cdr`:

```
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
```

Recursive lists are also built into the language, using pairs. A special value denoted `nil` or `'()` represents the empty list. A recursive list value is rendered by placing its elements within parentheses, separated by spaces:

```
> (cons 1
        (cons 2
              (cons 3
                    (cons 4 nil))))
(1 2 3 4)
> (list 1 2 3 4)
(1 2 3 4)
> (define one-through-four (list 1 2 3 4))
> (car one-through-four)
1
> (cdr one-through-four)
(2 3 4)
> (car (cdr one-through-four))
2
> (cons 10 one-through-four)
(10 1 2 3 4)
> (cons 5 one-through-four)
(5 1 2 3 4)
```

Whether a list is empty can be determined using the primitive `null?` predicate. Using it, we can define the standard sequence operations for computing `length` and selecting elements:

```
> (define (length items)
    (if (null? items)
        0
        (+ 1 (length (cdr items)))))
> (define (getitem items n)
    (if (= n 0)
        (car items)
        (getitem (cdr items) (- n 1))))
> (define squares (list 1 4 9 16 25))
> (length squares)
5
> (getitem squares 3)
16
```

Symbolic Data

All the compound data objects we have used so far were constructed ultimately from numbers. One of Scheme's strengths is working with arbitrary symbols as data.

In order to manipulate symbols we need a new element in our language: the ability to *quote* a data object. Suppose we want to construct the list `(a b)`. We can't accomplish this with `(list a b)`, because this expression constructs a list of the values of `a` and `b` rather than the symbols themselves. In Scheme, we refer to the symbols `a` and `b` rather than their values by preceding them with a single quotation mark:

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

In Scheme, any expression that is not evaluated is said to be *quoted*. This notion of quotation is derived from a classic philosophical distinction between a thing, such as a dog, which runs around and barks, and the word "dog" that is a linguistic construct for designating such things. When we use "dog" in quotation marks, we do not refer to some dog in particular but instead to a word. In language, quotation allow us to talk about language itself, and so it is in Scheme:

```
> (list 'define 'list)
(define list)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists:

```
> (car '(a b c))
a

> (cdr '(a b c))
(b c)
```

The full Scheme language contains additional features, such as mutation operations, vectors, and maps. However, the subset we have introduced so far provides a rich functional programming language capable of implementing many of the ideas we have discussed so far.

Recursion

Recursion is a mechanism for repetition that makes use of functions and function application. It involves a function calling itself directly or indirectly, usually with arguments that are in some "smaller" than the previous arguments. A recursive computation terminates when it reaches a *base case*, an input where the result can be computed directly without making any recursive calls.

It is sufficient for a language to provide recursion and condtionals in order for it to be Turing complete.

Activation Records

On a machine, recursion works due to the fact that each invocation of a function has its own activation record that maps its local variables to values. Consider the following recursive definition of factorial:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Calling `factorial(4)` results in five invocations of `factorial()`, with arguments from 4 down to 0. Each has its own activation record with its own binding for the parameter `n`:

```
factorial(4):   n --> 4
factorial(3):   n --> 3
factorial(2):   n --> 2
factorial(1):   n --> 1
factorial(0):   n --> 0
```

Thus, when `n` is looked up while executing the body of `factorial()`, each invocation obtains its own value of `n` without being affected by the other activation records.

An activation record requires more than just storage for parameters and local variables in order for function invocation to work. Temporary values also need to be stored somewhere, and since each invocation needs its own storage for temporaries, they are generally also placed in the activation record. An invocation also needs to know where to store its return value, usually in temporary storage in the frame of the caller. Finally, a function needs to know how to return execution to its caller. Details are beyond the scope of this course, but included in this information is the instruction address that follows the function call in the caller and the address of the caller's activation record.

The set of temporary objects can be conservatively determined statically, so the size of an activation record, as well as the placement of objects within it, can be determined at compile time. For `factorial()` above, temporary storage is required for `n - 1` as well as the result of the recursive call to `factorial()`. The location of the latter in the caller is used by a recursive call to store its return value. Depending on the implementation, the invocation of `factorial(0)` may still have space for these temporary objects in its activation record even though they will not be used.

Tail Recursion

A recursive computation uses a separate activation record for each call to a function. The amount of space required to store these records is proportional to the number of active function calls. In `factorial(n)` above, when the computation reaches `factorial(0)`, all `n + 1` invocations are active at the same time, requiring space in `O(n)`. Contrast this with the following iterative implementation that uses constant space:

```
def factorial_iter(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

The space requirements of the recursive version of `factorial()`, however, is not intrinsic to the use of recursion but is a result of how the function is written. An invocation of `factorial(k)` cannot complete until the recursive call to `factorial(k - 1)` does, since it has to multiply the result by `k`. The fact that the invocation has work that needs to be done after the recursive call requires its its activation record to be retained during the recursive call, leading to the linear space requirement.

Consider an alternative recursive computation of factorial:

```
def factorial_tail(n, partial_result = 1):
    if n == 0:
        return partial_result
    return factorial_tail(n - 1, n * partial_result)
```

Observe that the `factorial_tail()` function does not do any work after the completion of its recursive call. This means that it no longer needs the storage for parameters, local variables, or temporary objects when the recursive call is made. Furthermore, since `factorial(n, k)` directly returns the result of the recursive call `factorial(n - 1, n * k)`, the latter can store its return value in the location meant for the return value of `factorial(n, k)` in the caller of `factorial(n, k)`, and it can return execution directly to that caller. Thus, an optimizing implementation can reuse the space for the activation record of `factorial_tail(n, k)` for `factorial_tail(n - 1, n * k)` since the activation record of the former is no longer required.

This process can be generalized to any function call, not just recursive calls. A function call is a *tail call* if its caller directly returns the value of the call without performing any additional computation. A function is *tail recursive* if all of its recursive calls are tail calls. Thus, `factorial_tail()` is tail recursive.

A tail-recursive computation uses only a constant number of activation records, so its space usage matches that of an equivalent iterative computation. In fact, many functional languages do not provide constructs for iteration, since they can be expressed equivalently using tail recursion. These languages often require that implementations perform *tail-call optimization*, reusing the space for activation records where possible.

Since a tail call requires that no computation be performed after it returns, calls that syntactically appear to be tail calls may not be when implicit computation may occur at the end of a function. A specific example of this is scope-based resource management, as in the example below:

```
int sum(vector<int> values, int index, int partial_result = 0) {
  if (values.size() == index) {
    return 0;
  }
  return sum(values, index + 1, partial_result + values[index])
}
```

While it appears that this code does not do computation after the recursive call, the local `vector<int>` object has a destructor that must run after the recursive call completes. Thus, the recursive call to `sum()` is not a tail call, and this computation is not tail recursive.

Another situation that prevents tail-call optimization is when a function contains a function definition within it, in languages that use static scope and support the full power of higher-order functions. The nested function requires access to its definition environment, so that environment must be retained if the nested function can be used after the invocation of its enclosing function completes.

Higher-Order Functions

Recall that a first-class entity is one that supports the operations that can be done on other entities in a language, including being passed as a parameter, returned from a function, and created dynamically. In a language in which functions are first class, it is possible to write *higher-order functions* that take in another function as a parameter or return a function. Other languages may also support higher-order functions, even if functions are not first-class entities that can be created at runtime.

Function Objects

In some languages, it is possible to define objects that aren't functions themselves but provide the same interface as a function. These are known as *function objects* or *functors*. In general, languages enable functors to be written by allowing the function-call operator to be overridden. Consider the following example in C++:

```
class Counter {
public:
  Counter : count(0) {}
  int operator()() {
    return count++;
  }
private:
  int count;
};
```

The `Counter` class implements a functor that returns how many times it has been called. Multiple `Counter` objects can exist simultaneously, each with their own count:

```
Counter counter1, counter2;
cout << counter1() << endl;  // prints 0
cout << counter1() << endl;  // prints 1
cout << counter1() << endl;  // prints 2
cout << counter2() << endl;  // prints 0
cout << counter2() << endl;  // prints 1
cout << counter1() << endl;  // prints 3
```

Functors allow multiple instances of a function-like object to exist, each with their own state that persists over the lifetime of the functor. This is in contrast to functions, where automatic objects do not persist past a single invocation, and static objects persist over the entire program execution.

Python also allows functors to be written by defining the special `__call__` method:

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self):
        self.count += 1
        return self.count - 1
```

In general, additional parameters can be specified when overloading the function-call operator, emulating functions that can take in those arguments.

Some languages do not allow the function-call operator itself to be overridden but specify conventions that allow functor-like objects to be defined and used. For example, the `Predicate` interface in Java is implemented by functor-like objects that take in an argument and return a boolean value:

```
interface Predicate<T> {
  boolean test(T t);
  ...
}

class GreaterThan implements Predicate<Integer> {
  public GreaterThan(int threshold) {
    this.threshold = threshold;
  }
  public boolean test(Integer i) {
    return i > threshold;
  }
  private int threshold;
}
```

Code that uses these functor-like objects calls the `test()` method rather than calling the object directly:

```
GreaterThan gt3 = new GreaterThan(3);
System.out.println(gt3.test(2));    // prints out false
System.out.println(gt3.test(20));   // prints out true
```

Separate interfaces are provided for common patterns in the `java.util.function` library package.

Functions as Parameters

A higher-order function may take another function as a parameter. We first examine languages that only have top-level functions and allow a pointer or reference to a function to be passed as an argument. We then examine how passing a function as an argument can effect the environment in which the function's code is executed.

Function Pointers

In some languages, functions can be passed as parameters or return values but cannot be created within the context of another function. In these languages, all functions are defined at the top level, and only a pointer or reference to a function may be used as a value. Consider the following example in C, a language that provides function pointers:

```
void apply(int *array, size_t size, int (*func)(int)) {
  for (; size > 0; --size, ++array) {
    *array = func(*array);
  }
}

int add_one(int x) {
  return x + 1;
}

int main() {
  int A[5] = { 1, 2, 3, 4, 5 };
  apply(A, 5, add_one);
  printf("%d, %d, %d, %d, %d\n", A[0], A[1], A[2], A[3], A[4]);
  return 0;
}
```

The `apply()` function takes in an array, its size, and a pointer to a function that takes in an `int` and returns an `int`. It applies the function to each element in the array, replacing the original value with the result. The `add_one()` function is passed as an argument to `apply()` (C automatically converts a function to a function pointer), and the result is that each element in `A` has been incremented.

Binding Policy

In the code above, there are three environments associated with the `add_one()` function: its definition environment, the environment where it was referenced (in `main()`), and the environment where it was called (in `apply()`). Depending on the semantics of the language, any of these three environments may be components of the environment in which the body of `add_one()` is executed.

Recall that in static scope, the code in a function has access to the names in its definition environment, whereas in dynamic scope, it has access to the names in the environment of its use. Considering dynamic scope, is the non-local environment of a function the one where the function was referenced or the one where it was called? The following is an example where this distinction is relevant:

```
int foo(int (*bar)()) {
  int x = 3;
  return bar();
}

int baz() {
  return x;
}

int main() {
  int x = 4;
  print(foo(baz));
}
```

In dynamic scope, a function has access to the environment of its use. In the example above, however, the result is different of the use environment of `baz()` is where the function was referenced or where it was called. In the former case, the non-local environment of `baz()` is the environment of `main()`, and the `x` in the body of `baz()` would refer to the one defined in `main()`. This is known as *deep binding*. In the latter case, the non-local environment of `baz()` is the environment of `foo()`, and `x` in `baz()` would refer to the one defined in `foo()`. This is called *shallow binding*. Both approaches are valid, and the binding policy of a language determined which one is used.

Binding policy can also make a difference when static scope is used in the case of functions defined locally inside of a recursive function. However, deep binding is universally used in languages with static scope, so that the environment established at the time of a function's definition is the one the function has access to.

Nested Functions

A key feature of functional programming is the ability to define a function from within another function, allowing the dynamic creation of a function. In languages with static scoping, such a nested function has access to its definition environment, and the combination of a function and its definition environment is called a *closure*. Variables used in the nested function but defined in the enclosing environment are said to be *captured* by the closure. If a nested function is returned or otherwise leaks from the enclosing function, the environment of the enclosing function generally must persist after the function returns, since bindings within it may be accessed by the nested function.

As an example, consider the following higher-order function in Python that returns a nested function:

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold
    return greater_than
```

The `make_greater_than()` function takes in a threshold value and constructs a nested function that determines if its input is greater than the threshold value. The `threshold` variable is located in the activation record of `make_greater_than()` but is captured by `greater_than()`. Since the latter is returned, the activation record must persist so that invocations of `greater_than()` can access the binding for `threshold`.

Observe that each time `make_greater_than()` is called, a different instance of `greater_than()` is created with its own enclosing environment. Thus, different invocations of `make_greater_than()` result in different functions:

```
>>> gt3 = make_greater_than(3)
>>> gt30 = make_greater_than(30)
>>> gt3(2)
False
>>> gt3(20)
True
>>> gt30(20)
False
>>> gt30(200)
True
```

Languages that are not purely functional may allow modification of a captured variable. For example, the following defines a data abstraction for a bank account using nested functions:

```
def make_account(balance):
    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance
    def withdraw(amount):
        nonlocal balance
        if 0 <= amount <= balance:
            balance -= amount
            return amount
        else:
            return 0
    return deposit, withdraw
```

The `nonlocal` statements are required in Python, since it assumes that assignments are to local variables by default. We can then use the created functions as follows:

```
>>> deposit, withdraw = make_account(100)
>>> withdraw(10)
10
```

```
>>> deposit(0)
90
>>> withdraw(20)
20
>>> deposit(0)
70
>>> deposit(10)
80
>>> withdraw(100)
0
>>> deposit(0)
80
```

We will return to data abstraction using functions later.

Decorators

A common pattern in Python is to transform a function (or class) by applying a higher-order function to it. Such a higher-order function is called a *decorator*, and Python has specific syntax for decorating functions:

```
@<decorator>
def <name>(<parameters>):
    <body>
```

This is largely equivalent to:

```
def <name>(<parameters>):
    <body>

<name> = <decorator>(<name>)
```

The decorated function's definition is executed normally, and then the decorator is called on the function. The result of this invocation is then bound to the name of the function.

As an example, suppose we wanted to trace when a function is called by printing out the name of the function as well as its arguments. We could define a higher-order function that takes in a function and returns a new nested function that first prints out the name of the original function and its arguments and then calls it:

```
def trace(fn):
    def tracer(*args):
        if len(args) == 1:
            print('{0}({1})'.format(fn.__name__, args[0]))
        else:
            print('{0}{1}'.format(fn.__name__, args))
        return fn(*args)
    return tracer
```

Here, we make use of variadic arguments to pass any number of arguments to the original function. We can then use decorator syntax to apply this to a function:

```
@trace
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

Now whenever a call to `factorial()` is made, we get a printout of the arguments:

```
>>> factorial(5)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)
120
```

Notice that the recursive calls also call the transformed function. This is because the name `factorial` is now bound to the nested tracer function in the enclosing environment of `factorial()`, so looking up the name results in the tracer function rather than the orignal one. A side effect of this is that we have *mutual recursion* where a set of functions indirectly make recursive calls through each other. In this case, the tracer calls the original `factorial()`, which calls the tracer.

Lambda Functions

Nested function definitions allow the construction of functions at runtime, fulfilling one of the requirements for functions to be a first-class entity. So far, however, we've only seen nested function definitions that are named, introducing a binding into the definition environment. This is in contrast to other first-class entities, such as data values, that can be created without being bound to a name. Just like it can be useful to construct a value without a name, such as when passing it as an argument or returning it, it can be useful to construct an unnamed functions. These are called *anonymous* or *lambda* functions.

Lambda functions are ubiquitous in functional languages, but many common imperative languages also provide some form of lambda functions. The syntax and capabilities differ between different languages, and we will examine a few representative examples.

Scheme

Lambdas are a common construct in the Lisp family of languages, those languages being primarily functional, and Scheme is no exception. The `lambda` special form constructs an anonymous function:

```
(lambda (<parameters>) <body>)
```

A function definition using the `define` form can then be considered a shorthand for a variable definition and a `lambda`:

```
(define (<name> <parameters>) <body>)
  -->
(define <name> (lambda (<parameters>) <body>))
```

As an example, consider the following function that creates and returns an anonymous function that adds a given number to its argument:

```
(define (make-adder n) (lambda (x) (+ x n)))
```

This is simpler and more appropriate than an equivalent definition that only uses `define`:

```
(define (make-adder n) (define (adder x) (+ x n)) adder)
```

We can then call the result of `make-adder` on individual arguments:

```
> (define add3 (make-adder 3))
> (add3 4)
7
> (add3 5)
8
> ((make-adder 4) 5)
9
```

Nested functions in Scheme use static scope, so the anonymous function has access to the variable `n` in its definition environment. It then adds its own argument `x` to `n`, returning the sum.

Scheme is not purely functional, allowing mutation of variables and compound data. Nested functions, whether anonymous or not, can modify variables in their non-local environment. The following function creates a counter function that returns how many times it has been called:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (- count 1))))
```

The `set!` form mutates a variable to the given value. We can then use the `make-counter` function as follows:

```
> (define counter (make-counter))
> (counter)
0
> (counter)
1
> (counter)
2
```

Python

Python supports anonymous functions with the `lambda` expression. This takes the following form:

```
lambda <parameters>: <body expression>
```

The syntax of lambda expressions in Python produce a constraint on anonymous functions that is not present in named nested functions: the body must be a single expression, and the value of that expression is automatically the return value of the function. In practice, this limitation is usually not a problem, since lambdas are often used in functional contexts where statements and side effects may not be appropriate.

The following is a definition of the `greater_than()` higher-order function that uses a lambda:

```
def make_greater_than(threshold):
    return lambda value: value > threshold
```

As can be seen in this example, simple nested functions that are used in only a single place can be written more succinctly with a lambda expression than with a definition statement.

While lambda functions in Python have access to their definition environment, they are syntactically prevented from modifying bindings in the non-local environment.

Java

Java does not allow nested function definitions, but it does have syntax for what it calls "lambda expressions." In actuality, this construct constructs an anonymous class with a method corresponding to the given parameters and body, and the compiler infers the base type of this class from the context of its use.

The following example uses a lambda expression to construct a functor-like object:

```
public static IntPredicate makeGreaterThan(int threshold) {
  return value -> value > threshold;
}
```

We can the use the result as follows:

```
IntPredicate gt3 = makeGreaterThan(3);
System.out.println(gt3.test(2));     // prints out false
System.out.println(gt3.test(20));    // prints out true
```

Java allows a lambda to take in any number of arguments, and providing types for the parameters is optional. The body can be a single expression or a block containing arbitrary statements.

On the other hand, Java places a significant restriction on lambda expressions. A lambda can only access variables in its definition environment that are never reassigned, and it cannot modify them itself. This is because lambdas are not implemented as closures, but rather as functor-like objects that store "captured" variables as members.

C++

Like Java, C++ has lambda expressions, but they provide more functionality than those in Java. A programmer can specify which variables in the definition environment are captured, and whether they are captured by value or by reference. The former creates a copy of a variable, while the latter allows a captured variable to be modified by the lambda.

The simplest lambda expressions are those that do not capture anything from the enclosing environment. Such a lambda is equivalent to a function written at the top level, and C++ even allows a captureless lambda to be converted to a function pointer. For example, the following code passes a lambda function to a higher-order function that takes in a function pointer:

```
int max_element(int *array, size_t size, bool (*less)(int, int)) {
  assert(size > 0);
  int max_so_far = array[0];
  for (size_t i = 1; i < size; i++) {
    if (less(max_so_far, array[i])) {
      max_so_far = array[i];
    }
  }
  return max_so_far;
}

int main() {
  int array[5] = { 3, 1, 4, 2, 5 };
  cout << max_element(array, 5,
                      [](int a, int b) {
                        return a > b;
                      })
       << endl;
}
```

The code constructs a lambda function that returns true if the first element is bigger than the second, and passing that to `max_element()` finds the minimum rather than the maximum element.

Lambdas that capture variables, whether by value or by reference, have state that is associated with a specific evaluation of a lambda expression, and this state can differ between different calls to the enclosing function. As a result, such a lambda is not representable as a top-level function. Instead, C++ implicitly defines a functor type for a capturing lambda. Evaluating a capturing lambda expression constructs an instance of this functor type, with the captured values and references stored as non-static members. Since the functor type is implicitly defined, type deduction with the `auto` keyword is usually used where the type of the functor is required.

The following is an example that uses a lambda to define a greater-than functor:

```
auto make_greater_than(int threshold) {
  return [=](int value) {
    return value > threshold;
  };
}

int main() {
  auto gt3 = make_greater_than(3);
  cout << gt3(2) << endl;                        // prints 0
  cout << gt3(20) << endl;                       // prints 1
  cout << make_greater_than(30)(20) << endl;    // prints 0
}
```

The = in the capture list for the lambda specifies that all variables from the enclosing environment that are used by the lambda should be captured by value. The code above is equivalent to the following that explicitly uses a functor:

```
class GreaterThan {
public:
  GreaterThan(int threshold_in) : threshold(threshold_in) {}
  bool operator()(int value) const {
    return value > threshold;
  }
private:
  int threshold;
};

auto make_greater_than(int threshold) {
  return GreaterThan(threshold);
}
```

An enclosing variable may also be captured by reference. However, a variable that is captured by reference does **not** have its lifetime extended. The reasoning for this is twofold. The first, practical reason is that most C++ implementations use stack-based management of automatic variables, and when a function returns, its activation record on the stack is reclaimed. Requiring that a variable live past its function invocation prevents activation records from being managed using a stack. The second, more fundamental reason is that the RAII (i.e. scope-based resource management) paradigm in C++ requires that when an automatic variable goes out of scope, the destructor for its corresponding object is run and the object reclaimed. Relaxing this requirement would result in undesirable effects similar to those of finalizers in garbage-collected languages.

The end result is that a lambda functor that captures by reference should not be used past the existence of its enclosing function invocation. The following counter definition is therefore erroneous:

```
auto make_counter() {
  int count = 0;
  return [&]() {
    return count++;
  };
}
```

The lifetime of the `count` variable ends when `make_counter()` returns, so that calling the lambda functor afterwards erroneously uses a dead object.

An alternative is to capture `count` by value, which stores a copy as a member of the lambda, and then mark the lambda as `mutable`, allowing its members to be modified:

```
auto make_counter() {
  int count = 0;
  return [=]() mutable {
    return count++;
  };
}
```

This definition is equivalent to the `Counter` functor we defined in Function Objects.

Common Patterns

We now take a look at some common computational patterns in functional programming. We will look at how to abstract these patterns as higher-order functions, as well as how to use them with lambda functions.

Sequence Patterns

A number of functional patterns operate over sequences. These patterns take in a sequence and a function and apply the function to elements of the sequence, producing a new sequence or value as a result. Since these are functional patterns, the original sequence is left unchanged.

Map The *map* pattern takes a sequence and a function and produces a new sequence that results from applying the function to each element of the original sequence. For example, the following adds one to each element of a Scheme list:

```
> (map (lambda (x) (+ x 1)) '(1 2 3))
(2 3 4)
```

We can define the `map` higher-order function as follows:

```
(define (map func lst)
  (if (null? lst)
      lst
      (cons (func (car lst))
            (map func (cdr lst)))))
```

Applying `map` to an empty list results in an empty list. Otherwise, `map` applies the given function to the first item in the list and recursively calls `map` on the rest of the list.

Python has a built-in `map()` function that takes in a function and an iterator and returns an iterator that results from applying the function to each item in the original iterator. We will see how to define this function later.

Reduce In the *reduce* pattern, a two-argument function is applied to the first two items in a sequence, then it is applied to the result and the next item, then to the result of that and the next item, and so on. A reduction may be left or right associative, but the former is more common. Often, if only a single item is in the sequence, that item is returned without applying the function. Some definitions allow an initial value to be specified as well for the case in which the sequence is empty.

The following examples compute the sum and maximum element of a Scheme list:

```
> (reduce-right (lambda (x y) (+ x y)) '(1 2 3 4))
10
> (reduce-right (lambda (x y) (if (> x y) x y)) '(1 2 3 4))
4
```

We can define a right-associative reduction as follows, which assumes that the given list has at least one element:

```
(define (reduce-right func lst)
  (if (null? (cdr lst))
      (car lst)
      (func (car lst) (reduce-right func (cdr lst)))))
```

Python includes a left-associated `reduce()` function in the `functools` module.

Filter The *filter* pattern uses a predicate function to filter items out of a list. A *predicate* is a function that takes in a value and returns true or false. In filter, elements that test true are retained while those that test false are discarded.

The following example filters out the odd elements from a list:

```
> (filter (lambda (x) (= (remainder x 2) 0)) '(1 2 3 4))
(2 4)
```

The following is a definition of `filter`:

```
(define (filter pred lst)
  (if (null? lst)
      lst
      (if (pred (car lst))
          (cons (car lst) (filter pred (cdr lst)))
          (filter pred (cdr lst)))))
```

Python provides a built-in `filter()` function as well.

Any The *any* pattern is a higher-order version of `or` (disjunction). It takes a predicate and applies the predicate to each successive item in a list, returning the first item that tests true. If no item tests true, then false is returned. Some languages use the name *find* for this pattern rather than *any*.

The following examples search a list for an even value:

```
> (any (lambda (x) (= (remainder x 2) 0)) '(1 2 3 4))
2
> (any (lambda (x) (= (remainder x 2) 0)) '(1 3))
#f
```

A short-circuiting `any` function can be defined as follows:

```
(define (any pred lst)
  (if (null? lst)
      #f
      (if (pred (car lst))
          (car lst)
          (any pred (cdr lst)))))
```

The *every* pattern can be similarly defined as the higher-order analogue of conjunction.

Composition

Programs often compose functions, applying a function to the result of applying another function to a value. Wrapping these two function applications together in a single function enables both operations to be done with a single call. For example, the following multiplies each item in a list by three and then adds one:

```
> (map (compose (lambda (x) (* 3 x))
                (lambda (x) (+ x 1)))
       '(3 5 7))
(12 18 24)
```

We can define `compose` as follows:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

Partial Application and Currying

Partial application allows us to specify some arguments to a function at a different time than the remaining arguments. Supplying *k* arguments to a function that takes in *n* arguments results in a function that takes in $n-k$ arguments.

As an example, suppose we want to define a function that computes powers of two. In Python, we can supply 2 as the first argument to the built-in `pow()` function to produce such a function. We need a partial-application higher-order function such as the following:

```
def partial(func, *args):
    def newfunc(*nargs):
        return func(*args, *nargs)
    return newfunc
```

We can then construct a powers-of-two function as follows:

```
>>> power_of_two = partial(pow, 2)
>>> power_of_two(3)
8
>>> power_of_two(7)
128
```

Python actually provides a more general implementation of `partial()` that works for keyword arguments as well in the `functools` module. C++ provides partial application using the `bind()` template in the `<functional>` header.

A related but distinct concept is *currying*, which transforms a function that takes in *n* arguments to a sequence of *n* functions that each take in a single argument. For example, the `pow()` function would be transformed as follows:

```
>>> curried_pow(2)(3)
8
```

The curried version of the function takes in a single argument, returning another function. The latter takes in another argument and produces the final value. Since the original `pow()` takes in two arguments, the curried function chain has length two.

We can define currying for two-parameter functions as follows in Python:

```
def curry2(func):
    def curriedA(a):
        def curriedB(b):
            return func(a, b)
        return curriedB
    return curriedA
```

Then we can call `curry2(pow)` to produce a curried version of `pow()`.

We can also define an "uncurry" operation that takes in a function that must be applied to a sequence of *n* arguments and produce a single function with *n* parameters. The following does so for a sequence of two arguments:

```
def uncurry2(func):
    def uncurried(a, b):
        return func(a)(b)
    return uncurried
```

```
>>> uncurried_pow = uncurry2(curried_pow)
>>> uncurried_pow(2, 3)
8
```

Some functional languages, such as Haskell, only permit functions with a single parameter. Functions that are written to take in more than one parameter are automatically curried.