3. High-Level Functional Programming

3.1 Introduction

This chapter focuses on *functional programming*, a very basic, yet powerful, paradigm in which computation is represented solely by applying functions. It builds upon the information structures in the preceding chapter, in that those structures are representative of the kinds of structures used as data in functional programming. Some additional characteristics of the functional programming paradigm are:

- Variables represent values, rather than memory locations.
- In particular, there are no assignment statements; all bindings to variables are done in terms of *definitions* and function arguments.
- Data are never modified once created. Instead, data are created from existing data by functions.
- Each occurrence of a given functional expression in a given context always denotes a single value throughout its lifetime.

This last point is sometimes called “referential transparency” (although it might have been equally well called “referential opacity”) since one cannot see or discern any differences in the results of two different instances of the same expression.

A term often used for modifying the value of a variable is “side effect”. In short, in a functional language there is no way to represent, and thus cause, side effects.

Some advantages that accrue from the functional style of programming are:

- Debugging a program is simpler, because there is no dependence on sequencing among assignment statements. One can re-evaluate an expression many times (as in debugging interactively) without fear that one evaluation will have an effect on another.
- Sub-expressions of a program can be processed *in parallel* on several different processors, since the meaning of an expression is inherent in the expression and there is no dependence on expression sequencing. This can produce a net speed-up of the execution, up to a factor of the number of processors used.
- Storage is managed by the underlying system; there is no way to specify allocation or freeing of storage, and the attendant problems with such actions are not present.

One does not need a functional language to program in a functional style. Most of the principles put forth in this section can be applied to ordinary imperative languages, provided that the usage is disciplined. Please note that while functional-programming has much to recommend it, we are not advocating its exclusive use. It will later be combined with other programming paradigms, such as object-oriented programming. It is difficult to present functional programming in that context initially, because the key ideas tend to become obscured by object-oriented syntax. For this reason we use the language rex rather than other more common languages. Once the ideas are instilled, they can be applied in whatever language the reader happens to be working.

3.2 Nomenclature

Before starting, it is helpful to clarify what we mean by *function*. We try to use this term mostly in the mathematical sense, rather than giving the extended meaning as a synonym for *procedure* as is done in the parlance of some programming languages.

> A *function* on a set (called the *domain* of the function) is an entity that associates, with each member of the set, a single item.

The key word above is *single*, meaning *exactly one*. A function never associates two or more items with a given member of the domain, nor does it ever fail to associate an item with any member of the domain.

We say the function, given a domain value, *yields* or *maps to* the value associated with it. The syntax indicating the element associated with a domain element x, if f represents the function, is $f(x)$. However this is only one possible syntax of many; the key idea is the association provided.

Examples of functions are:

- The *add1* function: Associates with any number the number + 1. (The domain can be any set of numbers).
- The *multiply* function: Associates with any pair of numbers the product of the numbers in the pair. (The domain is a set of pairs of numbers.)
- The *reverse* function: Associates with any list of elements another list with elements in reverse order.
- The *length* function: Associates with any list a number giving the length of the list.
- The *father* function: Associates with any person the person's father.

- The *zero* function: Associates with any value in its domain the value 0.

Note that there is no requirement that two different elements of the domain can't be associated with a single value. That is, *f*(*x*) could be the same as *f*(*y*), even though *x* and *y* might be bound to different values. This occurs, for example, in the case of the multiply function: multiply(3, 4) gives the same value as multiply(4, 3). Functions that prohibit *f*(*x*) from being the same as *f*(*y*) when *x* and *y* are bound to different values are called *one-to-one* functions. Functions such as the *zero* function, which associates the same value with *all* elements of the domain are called *constant functions*.

A related definition, where we will tend to blur the distinction with function as defined, is that of partial function:

> A *partial function* on a set is an entity that associates, with each member of a set, *at most one* item.

Notice that here we have replaced "single" in the definition of "function" with "at most one". In the case of a partial function, we allow there to be *no* item associated with some members of the set. In this book, the same syntax is used for functions and partial functions. However, with a partial function *f*, it is possible to have no value *f*(*x*) for a given *x*. In this case, we say that *f*(*x*) *is undefined*.

An example of a partial function that is not a function is:

> The *divide* function: It associates with any pair of numbers the first number divided by the second, except for the case where the second is 0, in which case the value of the function is undefined.

An example of a partial function on the integers is a *list*:

> A list may be viewed as a partial function that returns an element of the list given its index (0, 1, 2, 3, ...). For any integer, there is at most one element at that index. There is no element if the index is negative or greater than N-1 where N is the length of the list. Finally, there must be no "holes", in the sense that the partial function is defined for all values between 0 and N-1.

We will use the common notation

> f: A $\rightarrow$ B

to designate that f is a partial function on set A, and that every value f(a) for a in A is in the set B.

Evidently, any partial function is a function over a sufficiently selective domain, namely the set of values for which the partial function is defined. Another way to remove the *partial* aspect is by defining a *special element* to indicate when the result would have otherwise been undefined. For example, in the case of divide by 0, we could use `Infinity` to indicate the result (rex does this). However, it is important to note than in computational systems, there are cases where this scheme for removing the *partial* nature of partial function can only be done for the sake of mathematical discussion; that is, we cannot, in some cases, *compute* the fact that the result will be undefined. The phenomenon to which we allude is *non-termination* of programs. While we often would like to think of a program as representing a function on the set of all possible inputs, for some inputs the program might not terminate. Moreover, it cannot always be detected when the program will not terminate. So in general, programs represent partial functions at best.

Two of the main ways to represent a partial function for computational purposes are: *by equation* and *by enumeration*. When we say "by equation", we mean that we give an equation that defines the function, in terms of constants and simpler functions. Examples in rex are:

```
f(x) = x*5;

g(x, y) = f(x) + y;

h(x) = g(f(x), 9);
```

Here `*`, `+`, `5`, and `9` are symbols that are "built-in" to rex and have their usual meaning (multiplication, addition, and two natural numbers). The semi-colon simply tells rex that the definition stops there (rather than, say, continuing on the next line).

When we say "by enumeration", we mean giving the set of pairs, the left elements of which are the domain elements and the right elements the corresponding values of the function. For example, we enumerate the *add1* function on the domain {0, 1, 2, 3, 4} by the list:

```
[ [4, 5], [3, 4], [2, 3], [1, 2], [0, 1] ]
```

Here we are treating a list as a set. Any reordering of the list would do as well.

While we can apply a function defined by equation by simply juxtaposing it with its arguments, e.g.

```
rex > f(1);
5

rex > g(2, 3);
13
```

we cannot do this in rex with a function enumerated by a list of pairs. We must instead pass the list to another function, such as one that we call `assoc` (short for *associate*), which will be described a bit later.

Likewise, we can go from a computed version of a function to an (internally) tabulated version using an idea known as *caching*. Caching allows rex to possibly bypass re-computing the function by first consulting the table. If the domain value is not present in the table, it will compute it, then put it in the table.

Pragmatically, to cause caching to occur in rex, we issue a one-time *directive*, such as:

```
rex > sys(on, cache(f#1));
```

The # sign is used to identify the number of arguments to this particular function (since different functions with different numbers of arguments can use the same function name, the selection being determined by the number of arguments). So here we are causing caching of the one-argument function with name `f`.

Another case of enumeration of a function occurs when the domain consists of consecutive natural numbers from 0 to some value. In this case, we can list the corresponding function values *in order* and apply the list. For example, in the preceding example of the domain-limited `add1` function, we could list the value (leaving the domain implicit) as:

```
[1, 2, 3, 4, 5]
```

This list can be applied by merely juxtaposing it with an argument:

```
rex > [1, 2, 3, 4, 5](3);
4
```

Exercises

1. ••• Specify, by equation, a function that gives the area of a triangle given the lengths of the sides as arguments. (Use Heron's formula, which may be found in various mathematics references).

2. •• Specify, by enumeration, a function that gives the number of days in each month in the year (assume 28 for February).

3.3 Using High-Level Functions

In the previous chapter, we used some of rex's built-in functions such as `length` and `type` to demonstrate properties of information structures. Later we will show how to construct some functions on our own. First, however, we want to get more practice in simply using functions. This will help in thinking at a relatively high level about information structures and functions that use them and create them.

Suppose we wish to create a list of numbers over a given range, where each number is 1 greater than its predecessor. To do this, we can use the function `range` that takes the lower and upper limit of the range as its arguments:

```
rex > range(1, 10);
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

rex > range(1.5, 9.5);
[1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5]
```

If we want the increment to be other than 1, we can use a three-argument version of `range` that specifies the increment:

```
rex > range(0, 20, 2);
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

rex > range(20, 0, -2);
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
```

As can be seen above, in rex the same function name can be used to designate different functions, based upon the number of arguments. The number of arguments is called the *arity* of the function (derived from using *n*-ary to mean *n* arguments). The use of one name for several different functions, the choice of which depends on the arity or the type of arguments, is called *overloading* the function's name.

Function `scale` multiplies the elements of an arbitrary list to produce a new list.

```
rex > scale(5, [1, 2, 20]);
[5, 10, 100]
```

As examples of functions on lists, we have `prefix`, which returns a specified-length prefix of a sequence:

```
rex > prefix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[1, 2, 3, 5]
```

`antiprefix`, which returns the sequence with the prefix of that length taken off:

```
rex > antiprefix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[7, 11, 13, 17, 19, 23]
```

and `suffix`, which returns the last so many elements of the list.

```
rex > suffix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[13, 17, 19, 23]
```

Function `remove_duplicates` returns a list of the elements in the argument list, with subsequent duplicates removed:

```
rex > remove_duplicates([1, 2, 3, 2, 3, 4, 3, 4, 5]);
```

```
[1, 2, 3, 4, 5]
```

Function `zip` interlaces elements of two lists together, as if they were the two sides of a zipper:

```
rex > zip([1, 3, 5, 7], [2, 4, 6]);
[1, 2, 3, 4, 5, 6, 7]
```

Function `reverse` reverses elements of a list:

```
rex > reverse([1, 2, 3, 4, 5]);
[5, 4, 3, 2, 1]
```

Function `append` appends the elements of the second list to those of the first:

```
rex > append([1, 2, 3, 4], [5, 6]);
[1, 2, 3, 4, 5, 6]
```

Sorting Lists

Another high-level function is `sort`, which sorts its argument list into ascending order:

```
rex > sort(["peas", "beans", "oats", "barley"]);
[barley, beans, oats, peas]
```

When we do this with functional programming, we are *not modifying* the original list. Instead, we are creating a new list from the original. To verify this using rex:

```
rex > L = ["peas", "beans", "oats", "barley"];
1

rex > M = sort(L);
1

rex > M;
[barley, beans, oats, peas]

rex > L;
[peas, beans, oats, barley]
```

We can see that `L` has not changed from the original.

Function `sort` will also work when the list's *elements* are lists. It uses the idea of *lexicographic ordering* to compare any two elements. Lexicographic ordering is like dictionary ordering extended to any ordered set, such as the set of numbers. For example, the empty list `[ ]` is < than any other list. One non-empty list is < another provided that the first element of one is < the other or the first elements are equal and the rest of the first list is < the rest of the second. For example:

```
rex > [1,2,3] < [1,2,4];
```

```
1

rex > [1,2,3] < [1,4];
1

rex > [1,2] < [1,1,2];
0
```

Notice that it is this same principle that allows us to sort a list of words, as if the words were lists of characters.

In rex, numbers are < strings, and strings are < lists, but this is somewhat arbitrary. The purpose of it is for certain algorithms such as removing duplicates, where an ordering is helpful.

Finding Things in Lists

A couple of frequently-used functions that can be used to tell us things about the contents of lists are `member` and `assoc`. Function `member` is a predicate that tells whether a specific item occurs in a list: `member(E, L)` returns `1` if `E` occurs in `L` and returns `0` otherwise. For example,

```
rex > member(99, range(90, 100));
1

rex > member(99, range(90, 95));
0
```

Function `assoc` applies to a special kind of list called an *association list*, a list of lists, each with at least one element. Usually this will be a list of pairs representing either a dictionary or a binary relation. The purpose of `assoc` is to find an element in an association list having a specified first element. If the list is a dictionary, we can thus use `assoc` to find the meaning of a given word. As an example,

```
rex > assoc(5, [ [4, 16], [5, 25], [6, 36], [7, 49] ]);
[5, 25]
```

Note that by definition of an association list, if an element is found, the returned value will always be non-empty (because each list is supposed to have at least one element in it). Thus we can use the empty list as a returned value to indicate no element was found:

```
rex > assoc(3, [ [4, 16], [5, 25], [6, 36], [7, 49] ]);
[ ]
```

Typically the association list represents an enumerated *partial function*, which, as we stated earlier, is a binary relation such that no two different pairs in the relation have the same first element. For example,

```
[ [5, “apple”], [6, “banana”], [10, “grapefruit”] ]
```

is a partial function, whereas

```
[ [5, “apple”], [6, “banana”], [10, “grapefruit”], [6, “orange”]
]
```

is not, because in the latter there are two pairs with first component 6. However, even if the list were not a partial function, `assoc` treats it as if it were by only returning the *first* pair in the list that matches the argument and ignoring the rest of the pairs. If one wanted to verify that the pair was unique, one could use function `keep` to find out.

Implementing Ordered Dictionaries

An *ordered dictionary* associates with each item in a set (called the set of keys) another item, the “definition” of that item. An implementation of an ordered dictionary is a list of ordered pairs, where a pair is a list of two elements, possibly of different types. The first element of each pair is the thing being defined, while the second is its. For example, in the following ordered dictionary implementation, we have musical *solfege* symbols and their definitions. Each definition is a list of words:

```
[ ["do", ["a", "deer", "a", "female", "deer"]],
  ["re", ["a", "drop", "of", "golden", "sun"]],
  ["me", ["a", "name", "I", "call", "myself"]],
  ["fa", ["a", "long", "long", "way", "to", "run"]],
  ["so", ["a", "needle", "pulling", "thread"]],
  ["la", ["a", "note", "that", "follows", "sol"]],
  ["ti", ["a", "drink", "with", "jam", "and", "bread"]]
]
```

Exercises

1. •• Using the functions presented in this section, along with arithmetic, give a one-line rex definition of the following function:

   `infix(I, J, L)` yields elements `I` through `J` of list `L`, where the first element is counted as element 0. For example,

   ```
   rex > infix(1, 5, range(0, 10));
   [1, 2, 3, 4, 5]
   ```

2. •• An *identity* on two functional expressions indicates that the two sides of the expression are equal for all arguments on which one of the two sides is defined. Which of the following identities are correct?

```
append(L, append(M, N)) == append(append(L, M), N)

reverse(reverse(L)) == L

reverse(append(L, M)) == append(reverse(L), reverse(M))

sort(append(L, M)) == append(sort(L), sort(M))

reverse(sort(L)) == reverse(L)

sort(reverse(L)) == sort(L)

[A | append(L, M)] == append([A | L], M)

reverse([A | L]) == [A | reverse(L)]

reverse([A | L]) == [reverse(L) | A]

reverse([A | L]) == append(reverse(L), [A])
```

3. ••• Two functions of a single argument are said to *commute* provide that for every argument value `X`, we have `f(g(X)) == g(f(X))`. Which pairs of functions commute?

   `sort` and `remove_duplicates`

   `reverse` and `remove_duplicates`

   `sort` and `reverse`

3.4 Mapping, Functions as Arguments

An important concept for leveraging intellectual effort in software development is the ability to use functions as arguments. As an example of where this idea could be used, the process of creating a list by doing a common operation to each element of a given list is called *mapping* over the list. Mapping is an extremely important way to think about operations on data, since it captures many similar ideas in a single high-level thought. (The word mapping is also used as a noun, as a synonym for function or partial function; this is not the use for the present concept.) Later on we will see how to define our own mapping functions. One of the attractive uses of high-level functions is that we do not over-specify how the result is to be achieved. This leaves open many possibilities for the compiler to optimize the performance of the operation.

The function `map` provides a general capability for mapping over a single sequence. For example, suppose we wish to use mapping to create a list of squares of a list of numbers. The first argument to the function `map` is itself a *function*. Assume that `sq` is a function that squares its argument. Then the goal can be accomplished as follows:

```
rex > map(sq, [ 1, 7, 5, 9 ]);
[1, 49, 25, 81]
```

Likewise, one way to create a list of the cubes of some numbers would be to *define* a function `cube` (since there isn't a built in one) and supply that as an argument to `map`. In rex, a function can be defined by a single equation, as shown in the first line below. We then supply that user-defined function as an argument to map:

```
rex > cube(X) = X*X*X;
1

rex > map(cube, [1, 7, 5, 9]);
[1, 343, 125, 729]
```

There is also a version of `map` that takes three arguments, the first being a function and the latter two being lists. Suppose that we wish to create a list of pairs of components from two argument lists. Knowing that function `list` will create a list of its two given arguments, we can provide `list` as the first argument to `map`:

```
rex > map(list, [1, 2, 3], [4, 5, 6]);
[[1, 4], [2, 5], [3, 6]]
```

A function such as `map` that takes a function as an argument is called a *higher-order*function.

3.5 Anonymous Functions

A powerful concept that has a use in conjunction with `map` is that of *anonymous function*. This is a function that can be created by a user or programmer, but which does not have to be given a name.

As a simple example of an anonymous function, suppose we wish to create a list by adding 5 to each element of a given list. We could do this using `map` and an "add 5" function. The way to define an "add 5" function without giving it a name is as follows:

```
(X) => X + 5
```

This expression is read:

> *the function that, with argument* `X`*, returns the value of* `X + 5`.
>
> The "arrow" `=>` identifies this as a *functional expression*.

We apply the anonymous function by giving it an argument, just as with any other function. Here we use parentheses around the functional expression to avoid ambiguity:

```
((X) => X + 5)(6)
```
function *argument*

Here the function designated by `(X) => X + 5` is applied to the actual argument `6`. The process is that formal argument `X` gets bound to the actual argument `6`. The body, with this identification, effectively becomes `6 + 5`. The value of the original expression is therefore that of `6 + 5`, i.e. `11`.

Here is another example, an anonymous function with two arguments:

```
((X, Y) => Y - X)(5, 6)
```

The function designated by `(X, Y) => Y - X` is applied to the actual pair of arguments `(5, 6)`. Formal argument `X` is bound to `5`, and `Y` to `6`. The result is that of `6 - 5`, i.e. `1`.

A common use of such anonymous expressions is in conjunction with functions such as `map`. For example, in order to cube each element of a list `L` above, we provided a function `cube` as the first argument of the function `map`, as in `map(cube, L)`. In some cases, this might be inconvenient. For example, we'd have to disrupt our thought to think up the name "`cube`". We also "clutter our name-space" with yet another function name. The use of such an anonymous function within `map` then could be

```
rex > map( (X) => X*X*X, range(1, 5));
[1, 8, 27, 64, 125]
```

Anonymous functions can have the property that identifiers mentioned in their expressions can be given values apart from the parameters of the function. We call these values *imported* values. In the following example

```
((X) => X + Y)(6)
```

`Y` is not an argument. It is assumed, therefore, that `Y` has a value defined from its context. We call this a *free variable* as far as the functional expression is concerned. For example, `Y` might have been given a value earlier. The function designated by `(X) => X + Y` is applied to the argument `6`. The result of the application is the value of `6 + Y`. We need to know the value of `Y` to simplify it any further. If `Y` had the value `3`, the result would be `9`. If `Y` had been given no value, the result would be undefined (rex would complain about `Y` being an *unbound variable*).

Here is an application of the idea of imported values. Consider defining the function `scale` that multiplies each element of a list by a factor. Since this is a `map` -like concept, the hope is we could use `map` to do it. However, to do so calls for an anonymous function:

```
scale(Factor, List) = map( (X) => Factor*X,  List);
```

Here `X` represents a "typical" element of the list, which is bound to values in the list as map applies its function argument to each of those elements. The variable `Factor`, on the other hand, is not bound to values in the list. It gets its value as the first argument to function `scale` and that value is *imported* to the anonymous function argument of `map`.

As a still more complex anonymous function example, consider:

```
((F) => F(6))((X) => X * 3)
```

Here the argument `F` to the functional expression `(F) => F(6)` is itself a functional expression `(X) => X * 3`. We identify the formal argument `F` with the latter, so the body becomes `((X) => X * 3)(6)`. We can then simplify this expression by performing the application indicated, with `X` identified with `6`, to get `6*3`, which simplifies to `18`.

In computer science, another, less readable, notation is often used in place of the `=>` notation we use to define anonymous functions. This is called "lambda notation", "lambda abstraction", or Church's **lambda calculus**. Instead of the suggestive `(X) => Y*X - 1`, lambda notation prescribes `λX.(Y*X - 1)`. In other words, the prefix `λ` takes the place of the infix `=>`.

3.6 Functions as Results

An interesting aspect about anonymous functions is that they can be returned as *results*. Consider

```
(Y) => ( (X) => X + Y )
```

This is read

> "the function that, with argument `Y`, returns:
>
> > the function that, with argument `X`, returns the value of `X + Y`"

In other words, the first function mentioned returns a function as its value. The second outer parentheses can be omitted, as in

```
(Y) => (X) => X + Y
```

because grouping around `=>` is to the right. Obviously we are again applying the idea of an *imported* variable, since the value of `Y` is not an argument but rather is imported to the inner expression.

When we apply such a function to a number such as `9`, the result is a function, namely the function represented by

```
(X) => X + 9
```

What happens, for example, when we map a function-returning function over a list of numbers? The result is a *list of functions*:

```
rex > map( (Y) => (X) => X + Y, [5, 10, 15] );
[(X) => (X+5), (X) => (X+10), (X) => (X+15)]
```

While the idea of a list of functions might not be used that often, it is an interesting to exercise the concept of one function returning another. The point here is to get used to thinking of functions as whole entities in themselves.

Suppose we wanted to apply each of the functions in the list above to a single value, say `9`. We could use `map` to do that, by mapping a function with a *function* argument:

```
(F) => F(9)
```

is, of course, the function that with argument `F` returns the result of applying `F` to `9`. When we map this function over the previous result, here's what we get:

```
rex > L = map((Y) => (X) => X + Y, [5, 10, 15]);
1

rex > map( (F)=>F(9),  L);
[14, 19, 24]
```

Consider the problem of making a list of *all possible pairs* of two given lists. For example, if the lists were:

`[1, 2, 3, 4]` and `[96, 97, 98]`

then we want the result to be something like

```
[ [1, 96], [1, 97], [1, 98],  [2, 96], [2, 97], [2, 98],
  [3, 96], [3, 97], [3, 98],  [4, 96], [4, 97], [4, 98]]
```

Note that this is quite different from the result of

```
map(list, L)
```

discussed earlier. We solve this problem by first considering how to make all pairs of a *given* element, say `X`, with each element of the second list, say `M`. This can be accomplished by using map:

```
map((Y) => [X, Y], M)
```

Now we make a function that does this mapping, taking `X` as its argument:

```
(X) => map((Y) => [X, Y], M)
```

Now consider mapping this function over the first list `L`:

```
map((X) => map((Y) => [X, Y], M), L)
```

For the present example, this doesn't quite do what we want. The result is a list of lists of pairs rather than a list of pairs:

```
[ [ [1, 96], [1, 97], [1, 98] ],
  [ [2, 96], [2, 97], [2, 98] ],
  [ [3, 96], [3, 97], [3, 98] ],
  [ [4, 96], [4, 97], [4, 98] ] ]
```

Instead of the outer application of `map`, we need to use a related function `mappend` (`map`, then `append`) to produce the list of all the second level lists appended together:

```
mappend((X) => map((Y) => [X, Y], M), L)
```

Let's try it:

```
rex > L = [1, 2, 3, 4];
1

rex > M = [96, 97, 98];
1

ex > mappend((X) => map((Y) => [X, Y], M), L);

[ [1, 96], [1, 97], [1, 98], [2, 96], [2, 97], [2, 98],
  [3, 96], [3, 97], [3, 98], [4, 96], [4, 97], [4, 98] ]
```

We can package such useful capabilities as functions by using the expression in a function-defining equation.

```
pairs(L, M) = mappend((X) => map((Y) => [X, Y], M), L);
```

3.7 Composing Functions

Suppose we wish to construct a function that will take two other functions as arguments and return a function that is the *composition* of those functions. This can be accomplished using the following rule:

```
compose(F, G) = (X) => F(G(X));
```

We could alternatively express this as a rule using a *double layer of arguments*:

```
compose(F, G)(X) = F(G(X));
```

Let's try using compose in rex:

```
rex > compose(F, G) = (X) => F(G(X));
1

rex > square(X) = X*X;
1

rex > cube(X) = X*X*X;
1
```

```
rex > compose(square, cube)(2);
64

rex > compose(cube, square)(2);
64

rex > compose(reverse, sort)([3, 5, 1, 2, 6, 7]);
[7, 6, 5, 3, 2, 1]

rex > map(compose(square, cube), range(1, 10));
[1, 64, 729, 4096, 15625, 46656, 117649, 262144, 531441, 1000000]
```

In mathematical texts, the symbol o is often used as an infix operator to indicate composition:

compose(F, G) ≡ F o G

The following diagram suggests how *compose* works. It in effect splices the two argument functions together into a single function.

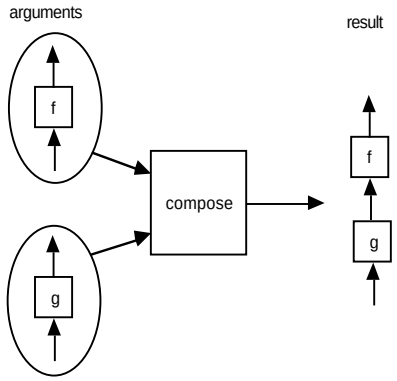

Figure 31: Diagram of the action of the function-composing function compose

3.8 The Pipelining Principle

A key design technique in computer science and engineering involves the *decomposition* of a specified function as a composition of simpler parts. For example, in the UNIX® operating system, single application programs can be structured as functions from a stream of input characters to a stream of output characters. The **composition** of two such functions entails using the output stream of one program as input to another. The result behaves as a program itself, and can be further composed in like fashion. One can regard this style of programming as building a **pipeline** connecting stages, each stage being an application program. Indeed, the operator for constructing such pipelines is known as "pipe" and shown as a vertical bar |. If P, Q, and R are programs, then

P | Q | R

defines the composition that connects the output of P to the input of Q and the output of Q to the input of R. The overall input then is the input to P and the output is the output of R. In terms of the function composition notation introduced earlier, we have R o (Q o P).

The pipelining principle is pervasive in computer science. Not only is it used in the construction of software; it is also extremely important at low levels of processor design, to enable parts of successive instructions to execute simultaneously. In later discussion, we will extend the idea of function composition to allow us to derive function composition networks of arbitrary structure.

3.9 Functions of Functions in Calculus (Advanced)

The failure to differentiate expressions from functions can be the source of confusion in areas such as differential and integral *calculus*, where it is easy to forget that *functions*, not numbers, are generally the main focus of discussion. For example, we are used to seeing equations such as

$$\frac{d}{dx} \; sin\ x = cos\ x$$

What is really meant here is that the result of operating on a function, *sin*, is another function, *cos*. The use of *x* is really irrelevant. It might have been less confusing to state

$$\text{derivative}(sin) = cos$$

and leave the dummy argument x out of the picture. A step in the right direction is to use the "prime" notation, wherein the derivative of a function *f* is shown as *f '*. But then we don't often see written equalities such as

$$sin' = cos$$

even though this, coupled with a proper understanding of functions as entities, would be less confusing than the first equation above.

As an example of the confusion, consider the **chain rule** in calculus, which can be correctly expressed as

$$(f \circ g)'(x) = f'(g(x)) * g'(x)$$

Using the *d/dx* notation, the chain rule cannot be expressed as nicely. If we are willing to *define* the product of two functions to be the function whose value for a given x is the product of the values of the individual functions, i.e.

$$(f * g)(x) = f(x) * g(x)$$

then the chain rule can be nicely expressed as:

$$(f \circ g)' = (f' \circ g) * g'$$

Translated to English, this statement says that the derivative of the composition of two functions is equal the product of the derivative of the second function and the composition of the derivative of the first function with the second function.

Exercises

1. • Create a function that cubes every element of a list, using `map` as your only named function.

2. •• Describe the functions represented by the following functional expressions:

```
a. (X) => X + 1

b. (Y) => Y + 1

c. (X) => 5

d. (F) => F(2)

e. (G, X) => G(X,X)

f. (X, Y) => Y(X)
```

3. •• Argue that `compose(F, compose(G, H))` == `compose(compose(F, G), H)`, i.e. that composition is associative. Written another way, `F o (G o H)` ≡ `(F o G) o H`. This being the case, we can eliminate parentheses and just write `F o G o H`.

4. ••• Express the calculus chain rule for the composition of three functions:

$$(F \circ G \circ H)' = ??$$

5. •• In some special cases, function composition is commutative, that is (F o G) ≡ (G o F). Give some examples of such cases. (Hint: Look at functions that raise their argument to a fixed power.)

6. •• Give an example that shows that function composition is not generally commutative.

7. •• Which functional identities are correct?

```
a. map(compose(F, G), L) ==  map(F, map(G, L))

b. map(F, reverse(L)) == reverse(map(F, L))

c. map(F, append(L, M)) == append(map(F, L), map(F, M))
```

3.10 Type Structure

A concern that occupies many computer scientists is that of the *types* of data and functions that operate on that data. The language we have been using so far, rex, is rather "loose" in its handling of types. This has its purpose: we don't wish to encumber the discussion with too many nuances at one time. Nonetheless, it is helpful to have a way to talk about expected types of data in functions; it helps us understand the specification of the function.

The basic types of most programming languages include:

integer numerals
characters or character strings
floating-point numerals

In order to provide a safe computational system, rex has to be able to discern the type of a datum dynamically: Although a rex variable is not annotated with any type, the basic operations in rex use the type of the data. For example, the + operator applies to integers or floating-point numerals, but not to character strings. Nothing prevents us from trying to use + on strings, but doing so will result in a run-time error that terminates the computation. Thus it is important that the programmer be aware of the type likely to be passed to a function. The rex language includes some built-in predicates for determining the type of data. For example, the predicate *is_number* establishes whether its argument is either an integer or floating point. The predicate *is_integer* establishes whether its argument is integer. The programmer can use these predicates to steer clear of run-time type errors.

It is common to treat data types as sets and to assert the type of functions using the customary domain-range notation on those sets. For example:

f: integer x integer → integer

asserts that function (or partial function) f takes two integer arguments and returns an integer.

In general, A x B, where A and B are sets, means the set of all pairs, the first element drawn from A and the second from B. This is called the *Cartesian product* of the sets. For example, the Cartesian product is computed by the function `pairs` worked out earlier.

In dealing with types, we use | to mean *union*, i.e. to describe elements that can be one of two different types. For example,

g: (integer | float) x integer → float

describes a function *g*, the first argument of which can be an integer or a float.

We often see type equations used to define intermediate classes. For example, we could define the type numeric to be the union of integer and float thus:

numeric = integer | float

We could also use equations to define types for functions:

binary_numeric_functions = (numeric x numeric) → numeric

treating the usual arrow notation as defining a set of functions.

Perhaps more important than the particular choice of basic types is the means of dealing with composite or aggregate types. The fundamental aggregation technique in rex is creating lists, so we could enlist the * notation to represent lists of arbitrary type things. For example,

integer*

could represent the type of lists of integers. Then

integer**

would represent the type of lists of lists of integers, etc. Since rex functions do not, in general, require their argument to be of any specific type, it is helpful to have a designation for the union of all types. This will be called *any*. For example, the type of the function length that counts the number of items in a list, is:

`length`: any* → integer

since this function pays absolutely no attention to the types of the individual elements in the argument list. On the other hand, some functions are best described using type variables. A good example is the function `map` that takes two arguments: a list of elements and a function. The domain of that function must be of the same type as the elements in the list. So we would describe `map` by

`map`: ((A → B) x A*) → B* where A and B are arbitrary types

A function such as `map` that operates on data of many different types is called *polymorphic*.

Function *compose* is a polymorphic function having following type:

((B → C) x (A → B)) → (A → C)

where A, B, and C are any three types.

Notice that although anonymous functions don't have names, we can still specify their *types* to help get a better understanding of them. For example, the type of `(X)=>X*X` is:

numeric → numeric

The set of lists permitting arbitrary nesting, which we have already equated to trees, deserves another type designator. If A is a type, then let's use $A^{\dagger}$ to designate the type that includes A, lists of A, lists of lists of A, and so on, ad infinitum. In a sense, $A^{\dagger}$ obeys the following type equation:

$$A^{\dagger} = A \mid (A^{\dagger})^{*}$$

That is to say $A^{\dagger}$ is the set that contains A and all lists of things of type $A^{\dagger}$. We shall encounter objects of this type again in later chapters when we deal with so called "S expressions".

We should cultivate the habit of checking that types *match* whenever a function is applied. For example, if

f: A → B

then, for any element a of type A, we know that f(a) is an element of type B.

Exercises

1. • Describe the types of the following functions:

   a. `(X) => X + 1`

   b. `(Y) => Y + 1`

   c. `(X) => 5`

   d. `(F) => F(2)`

   e. `(G, X) => G(X,X)`

   f. `(X, Y) => Y(X)`

2. •• Describe the type structure of the following functions that have been previously introduced:

   a. `range`

   b. `reverse`

   c. `append`

3.11Transposing Lists of Lists

In a previous section, we showed how to use `map` to pair up two lists element-wise:

```
rex > map(list, [1, 2, 3], [4, 5, 6]);
[[1, 4], [2, 5], [3, 6]]
```

Another way to get the effect of pairing elements would be to cast the problem as an instance of a more general function `transpose`. This function would understand the representation of matrices as lists of lists, as described in the previous chapter. By giving it a list of the two argument lists above, we get the same result, pairing of corresponding elements in the two lists. What we have done, in terms of the matrix view, is transposed the matrix, meaning exchanging the rows and columns in the two-dimensional presentation. For example, the transpose of the matrix

1	2	3
4	5	6

is

1	4
2	5
3	6

Using a list of lists, the transposition example would be shown as:

```
rex > transpose([ [1, 2, 3], [4, 5, 6] ]);
[[1, 4], [2, 5], [3, 6]]
```

However, `transpose` is more general than a single application of `map` in being able to deal with matrices with more than two rows, for example.

```
rex > transpose([ [1,   2,  3],
                  [4,   5,  6],
                  [7,   8,  9],
                  [10, 11, 12] ] );

[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

The effect could be achieved with a double application of `map`, but to do so is a little tricky.

Exercises

1. • Give the type of function `transpose`.

2. •• Give the type of function `mappend` defined earlier.

3.12 Predicates

By a predicate, we mean a function to be used for discrimination, specifically in determining whether its arguments have a certain property or do not. Here we will use 1 (representing *true*) to indicate that an argument has the property, and 0 (also called *false*) otherwise, although other pairs of values are also possible, such as {yes, no}, {red, black}, etc. The *arity* of a predicate is the number of arguments it takes.

A simple example is the 2-ary predicate that we might call `less`, of two arguments, for example `less(3, 5) == 1`, `less(4, 2) == 0`, etc. Informally, `less(x, y)` is `1` whenever `x` is less than `y`, and `0` otherwise. We are used to seeing this predicate in infix form (with the symbol `<` between the arguments), i.e. `x < y` instead of `less(x, y)`. We could also use the symbol `<` instead of the name `less` in our discussion. Actually this is the name by which rex knows the predicate.

```
rex > map(<, [1, 2, 4, 8], [2, 3, 4, 5]);
[1, 1, 0, 0]
```

When an argument combination makes a predicate true, we say that the combination *satisfies* the predicate. This is for convenience in discourse.

Some functions built into rex expect predicates as arguments. An example is the function `some`: if `P` is a predicate and `L` is a list, then `some(P, L)`, returns 1 if some element of list `L` satisfies predicate `P`, and 0 otherwise. For example, `is_prime` is a predicate that gives the value 1 exactly if its argument is a prime number (a number that has no natural number divisors other than 1 and itself). We can ask whether any member of a list is prime using some in combination with `is_prime`. For example:

```
rex > some(is_prime, [4, 5, 6]);
1

rex > some(is_prime, [4, 6, 8, 10]);
0
```

Here 5 is the only prime. Note that `some` itself is a predicate. It would be called a *higher-order* predicate, because it takes a predicate as an argument. It could also be called a *quantifier*, since it is related to a concept in logic with that name. We shall discuss quantifiers further in a later chapter. A related quantifier is called `all`. The expression `all(P, L)` returns 1 iff all elements of `L` satisfy `P`. For example:

```
rex > all(is_prime, [2, 3, 5, 7]);
1
```

Often we want to know more than just whether some element of a list satisfies a predicate `P`; we want to know the identity of those elements. To accomplish this, we can use the predicate `keep`. The expression `keep(P, L)` returns the list of those elements of `L` that satisfy `P`, in the order in which they occur in `L`. For example:

```
rex > keep(is_prime, [2, 3, 4, 5, 6, 7, 8, 9]);
[2, 3, 5, 7]
```

Note that if sets are represented as lists, `keep` gives a facility like set selection in mathematics.

$$\{x \in S \mid P(x)\}$$

(read “the set of x in S such that P(X) is true”) is analogous to:

```
keep(P, S)
```

where we are representing the set S as a list. Function `keep` gives us a primitive database search facility: the list could be a list of lists representing records of some kind. Then `keep` can be used to select records with a specific property. For example, suppose our database records have the form

[*Employee*, *Manager*, *Salary*, *Department*]

with an example database being:

```
DB = [ ["Jones",   "Smith", 25000, "Development"],
       ["Thomas",  "Smith", 30000, "Development"],
       ["Simpson", "Smith", 29000, "Development"],
       ["Smith",   "Evans", 45000, "Development"]];
```

Then to pose the query “What records correspond to employees managed by Smith with a salary more than 25000, we could ask rex:

```
rex > keep((Record) => second(Record) == "Smith"
                     && third(Record) > 25000, DB);

[[Thomas, Smith, 30000, Development],
 [Simpson, Smith, 29000, Development]]
```

This could be made prettier by using pattern matching, however we have not yet introduced pattern matching in the context of anonymous functions and don’t wish to digress to do so at this point.

The complementary predicate to `keep` is called `drop`. The expression `drop(P, L)` returns the list of elements in `L` that do *not* satisfy `P`:

```
rex > drop(is_prime, [2, 3, 4, 5, 6, 7, 8, 9]);
```

```
[4, 6, 8, 9]
```

Sometimes we are interested in the *first* occurrence of an element satisfying a particular predicate, and might make use of the other occurrences subsequently. The predicate `find` gives us the suffix of list `L` beginning with the first occurrence of an element that satisfies `P`. If there are no such elements, then it will give us the empty list:

```
rex > find(is_prime, [4, 6, 8, 11, 12]);
[11, 12]

rex > find(is_prime, [12, 14]);
[ ]
```

The predicate `find_indices` gives us a list of the *indices* of all elements in a list which satisfy a given predicate:

```
rex > find_indices(is_prime, range(1, 20));
[0, 1, 2, 4, 6, 10, 12, 16, 18]

rex > find_indices(is_prime, range(24, 28));
[ ]
```

Exercises

1. • Suppose `L` is a list of lists. Present an expression that will return the lists of elements in `L` having length greater than 5.

2. ••• Present as many functional identities that you can among the functions `keep`, `find_indices`, `map`, `append`, and `reverse`, excluding those presented in earlier exercises.

3. ••• Show how to use `keep` and `map` in combination to define a function `gather` that creates a list of second elements corresponding to a given first element in an association list. For example,

```
rex > gather(3, [[1, "a"], [2, "b"], [3, "c"], [1, "d"],
                 [3, "e"], [3, "f"], [2, "g"], [1, "h"]]);
[c, e, f]
```

4. ••• Then define a second version of gather that gathers the second components of *all* elements of an association list together:

```
rex > gather ([[1, "a"], [2, "b"], [3, "c"], [1, "d"],
               [3, "e"], [3, "f"], [2, "g"], [1, "h"]]);
[[1, a, d, h], [2, b, g], [3, c, e, f]]
```

3.13 Generalized Sorting

A variation on the function `sort` has an additional argument, which is expected to be a binary predicate. This predicate specifies the comparison between two elements to be used in sorting. For example, to sort a list in reverse order:

```
rex > sort(>, [6, 1, 3, 2, 7, 4, 5]);
[7, 6, 5, 4, 3, 2, 1]
```

The ability to specify the comparison predicate is useful for specialized sorting. For example, if we have a list of lists of the form

[*Person*, *Age*]

and wish to sort this list by age, rather than lexicographically, we could supply a predicate that compares second elements of lists only:

```
(L, M) => second(L) < second(M).
```

Let's try this:

```
rex > Data =
          [["John", 25], ["Mary", 24], ["Tim", 21], ["Susan", 18]];
1

rex > sort((L, M) => second(L) < second(M), Data);
[[Susan, 18], [Tim, 21], [Mary, 24], [John, 25]]
```

In the next chapter, we will show some details for how lists can be sorted.

3.14 Reducing Lists

By *reducing* a list, we have in mind a higher-order function in a spirit similar to `map` introduced earlier. As with `map`, there are many occasions where we need to produce a single value that results from applying a *binary* function (i.e. 2-argument function, not be confused with binary relation or binary number representation introduced earlier) to elements of a list. Examples of reducing include adding up the elements in a list, multiplying the elements of a list, etc. In abstract terms, each of these would be considered reducing the list by a different binary operation.

For completeness, we need to say what it means to reduce the empty list. Typically reducing the empty list will depend on the operator being used. It is common to choose the mathematical *unit* of the operation, if it exists, for the value of reducing the empty list. For example, the sum of the empty list is 0 while the product is 1. The defining characteristic of the unit is that when another value is combined with the unit using the binary operator, the result is that other value:

For any number `x`:

```
0 + x == x

1 * x == x
```

The function `reduce` performs reductions based on the binary operator, the unit or other base value, and the list to be reduced:

```
rex > r = range(1, 10);
1

rex > r;
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

rex > reduce(+, 0, r);
55
```

That is, `55` is the sum of the natural numbers from 1 to 10.

```
rex > reduce(*, 1, r);
3628800
```

That is, `3628800` is the product of the natural numbers from 1 to 10.

Suppose we wished to create a single list out of the elements present in a list of lists. For two lists, the function that does this is called `append`:

```
rex > append([1, 2, 3], [4, 5]);
[1, 2, 3, 4, 5]
```

For a list of lists, we can use the higher-order function `reduce` with `append` as its argument. However, we have to decide what the unit for append is. We are looking for a value U such that for any list L

```
append(U, L) == L
```

That value is none other than the null list `[ ]`. So to append together an arbitrary list of lists L, we can use

```
reduce(append, [ ], L)
```

For example,

```
rex > reduce(append, [ ], [ [1, 2], [3, 4, 5], [ ], [6]] );
[1, 2, 3, 4, 5, 6]
```

Actually there are at least two different ways to reduce a list: since the operator operates on only two things at a time, we can group pairs starting from the right or from the left. Some languages make this distinction by providing two functions, `foldl` and `foldr`, with the same argument types as `reduce`. For example, if the operator is +, then

```
foldl(+, 0, [x0, x1, x2, x3, x4])
```

evaluates in the form

```
((((0 + x0)  + x1) + x2) + x3) + x4
```

whereas

```
foldr(+, 0, [x0, x1, x2, x3, x4])
```

evaluates in the form

```
x0 + (x1 + (x2 + (x3 + (x4 + 0))))
```

We can show this idea with rex by inventing a "symbolic" operator `op` that displays its arguments:

```
rex > op(X, Y) = concat("op(", X, ",", Y, ")");
1

rex > foldl(op, "unit", ["x0", "x1", "x2", "x3", "x4"]);
op(op(op(op(op(unit,x0),x1),x2),x3),x4)

rex > foldr(op, "unit", ["x0", "x1", "x2", "x3", "x4"]);
op(x0,op(x1,op(x2,op(x3,op(x4,unit)))))
```

Note that the base value in this example is not the unit for the operator.

Currently, rex uses the `foldl` version for `reduce`, but this is implementation-defined. For many typical uses, the operator is associative, in which case it does not matter which version is used. If it does matter, then one should use the more specific functions.

Exercises

1. •• Suppose we wish to regard lists of numbers as vectors. The inner product of two vectors is the sum of the products of the elements taken element-wise. For example, the "inner product" of `[2, 5, 7]`, `[3, 2, 8]` is `2*3 + 5*2 + 7*8 ==> 72`. Express an equation for the function `inner_product` in terms of `map` and `reduce`.

2. • Show that the rex predicate `some` can be derived from `keep`.

3. •• Suppose that `P` is a 1-ary predicate and `L` is a list. Argue that

   ```
   some(P, L) == !all((X) => !P(X), L)
   ```

   In other words, `P` is satisfied by some element of `L` if, and only if, the negation of `P` is not satisfied by all elements of `L`.

4. •• Show that the rex predicate `drop` can be derived from `keep`.

5. •• Using `reduce`, construct a version of `compose_list` that composes an arbitrary list of functions. An example of `compose_list` is:

```
compose_list([(A)=>A*A, (A)=>A+1, (A)=>A-5])(10) ==> 36
```

   Hint: What is the appropriate *unit* for function composition?

6. •• Which of the following expressions is of the proper type to reproduce its third argument list `L`?

```
a. foldl(cons, [ ], L)
b. foldr(cons, [ ], L)
```

3.15 Sequences as Functions

In computer science, it is important to be aware of the following fact:

> Every list can be viewed as a partial function on the domain of natural numbers (0, 1, 2, 3, ...).
>
> When the list is infinite, this partial function is a function.

That is, when we deal with a list $[x_0, x_1, x_2, \ldots]$ we can think of this as the following function represented as a list of pairs:

$$[[0, x_0], [1, x_1], [2, x_2], \ldots]$$

In the case that the list is finite, of length N, this becomes a partial function on the natural numbers, but a total function on the domain {0, 1, ..., N-1}.

This connection will become more important in subsequent chapters when we consider arrays, a particular sequence representation that can also be modeled as a function. The thing that it is important to keep in mind is that if we need to deal with functions on the natural numbers, we can equivalently deal with sequences (lists, arrays, etc.).

In rex, special accommodation is made for this idea, namely *a sequence can be applied as if it were a function*. For example, if *x* denotes the sequence [0, 1, 4, 9, 16, 25, ...] then x(2) (*x* applied to 2) is 4, x(3) is 9, etc. Moreover, rex sequences need not be only lists; they can also be arrays. An array applied to an argument gives the effect of array indexing.

One way to build an array in rex is to just give the elements of the array to the function `array`:

`array(`a_0`, `a_1`, ...., `a_{n-1}`)`

(The function array has an arbitrary number of arguments.) Another way is to use the function `make_array`. The latter takes two arguments, a function, say `f`, and a natural number, say `n`, and gives the effect of

```
array(f(0), f(1), ...., f(n-1))
```

One reason we might prefer such an array to a function itself is to avoid re-evaluating the function at the same argument multiple times. Once the function values are "cached" in the array, we can access them arbitrarily many times without recomputing.

Array access is preferred over list access for reasons of efficiency. For arrays, we can get to any element in constant time. For lists, the computer has to "count up" to the appropriate element. This takes time proportional to the index argument value. For this reason, we emphasize the following:

Sequencing through a list L by repeated indexing L(i), L(i+1), L(i+2), ... is to be avoided, for reasons of efficiency.

We already know better ways to do this (using the list decomposition operators).

Exercises

1. ••• Construct a function that composes two functions represented as association lists. For example, the following shows the association list form of a composition:

```
    [ [0, 0], [1, 3], [2, 2], [3, 3] ]
o   [ [0, 3], [1, 2], [2, 1], [3, 0] ]
==> [ [0, 3], [1, 2], [2, 3], [3, 0] ]
```

   (Hint: Use `map`.)

2. ••• Construct a function that composes two functions represented as lists-as-functions, for example:

```
[0, 3, 2, 3] o [3, 2, 1, 0] ==> [3, 2, 3, 0]
```

   (Hint: Use `map`.)

3.16 Solving Complex Problems using Functions

Most computational problems can be expressed in the form of implementing some kind of function, whether or not functional programming is used as the implementation method. In this section, we indicate how functions can provide a way of thinking about decomposing a problem, to arrive at an eventual solution.

As we already observed, functions have the attractive property closure under composition: composing two functions gives a function. Inductively, composing any number of functions will give a function. In solving problems, we want to reverse the composition process:

> Given a specification of a function to be implemented, find simpler functions that can be composed to equal the goal function.

By continuing this process of decomposing functions into compositions of simpler ones, we may arrive at some functions that we can use that are already implemented. There may be some we still have to implement, either by further decomposition or by low-level methods, as described in the next chapter.

A Very Simple Example

Implement a function that, with a list of words as input, produces a list that is sorted in alphabetical order and from which all duplicates have been removed. Our goal function can be decomposed into uses of two functions: `sort` and `remove_duplicates`:

```
goal(L) = remove_duplicates(sort(L));
```

If we were working in rex, then those two functions are built-in, and we'd be done. Alternatively, we could set out to implement those functions using low-level methods.

An Example Using Directed Graphs

Consider the problem of determining whether a graph is acyclic (has no cycles). Assume that the graph is given as a list of pairs of nodes.

Examples of an acyclic vs. a cyclic graph is shown below:

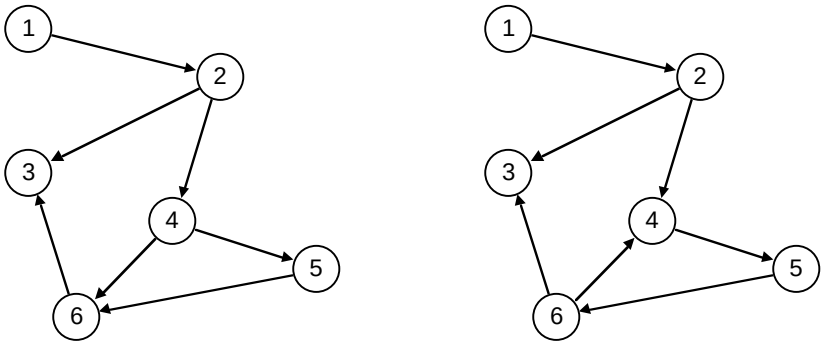

Figure 32: Acyclic vs. cyclic directed graphs

We'd like to devise a function by composing functions that have been discussed thus far. This function, call it `is_acyclic`, will take a list of pairs of nodes as an argument and return a 1 if the graph is acyclic, or a 0 otherwise.

Here's the idea we'll use in devising the function:

> If a graph is acyclic, then it must have at least one leaf.

A *leaf* is defined to be a node with no targets (also sometimes called a “sink”). So if the graph has no leaf, we immediately know it is not acyclic. However, a graph can have a leaf and still be cyclic. For example, in the rightmost (cyclic) graph above, node 3 is a leaf. The second idea we'll use is:

> Any leaf and attached arcs can be removed without affecting whether the graph is acyclic or not.

Removing a leaf may produce new leaves in the resulting graph. For example, in the leftmost (acyclic) graph above, node 3 is a leaf. When it is removed, node 6 becomes a leaf.

The overall idea is this:

> Starting with the given graph, repeat the following process as much as possible:
>
> > Remove any leaf and its connected arcs.

There are two ways in which this process can terminate:

1. All nodes have been eliminated, or

2. There are still nodes, but no more leaves.

In case 1, our conclusion is that the original graph was acyclic. In case 2, it was not. In fact, in case 2 we know that a cycle in the remaining graph exists and it is also a cycle of the original graph.

We now concentrate on presenting these ideas using functions. As a running example, we'll use the graph below, the representation of which is the following list:

```
[ [1, 2], [2, 3], [2, 4], [4, 5], [6, 3], [4, 6], [5, 6] ]
```

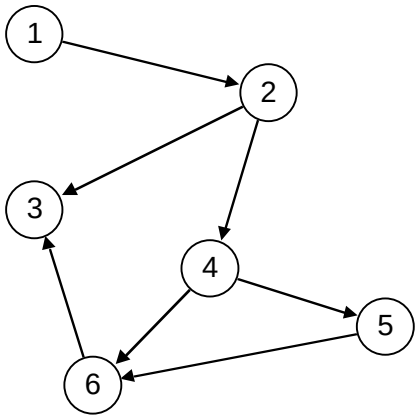

Figure 33: An acyclic graph for discussion

First we need a function that can determine whether there is a leaf. By definition, a leaf is a node with no arcs leaving it. A good place to start would seem to be devising a function that can determine whether a given node of a graph is a leaf, then iterate that function over the entire set of nodes. The following function is proposed:

```
is_leaf(Node, Graph) =

    no( (Pair) => first(Pair) == Node, Graph );
```

The function `no` applies its first argument, a predicate, to a list. If there is an element in the list satisfying the predicate, then there is a leaf. In this case, the predicate is given by the anonymous function

```
(Pair) => first(Pair) == Node
```

that asks the question: *is* `Node` *the first element of* `Pair`? The function `no` asks this question for each element of the list, stopping with 1 when a leaf is found, or returning 0 if no leaf is found.

On our example graph, suppose we try this function with arguments 3 and 4 in turn:

```
rex > graph = [ [1, 2], [2, 3], [2, 4], [4, 5],
                [6, 3], [4, 6], [5, 6] ];
1

rex > is_leaf(3, graph);
1

rex > is_leaf(4, graph);
0
```

Now let's use the `is_leaf` function to return a leaf in the graph, if there is one. Define `find_leaf` as follows:

```
find_leaf(Graph) =

    find((Node) => is_leaf(Node, Graph), nodes(Graph));
```

Here we are assuming that `nodes(Graph)` gives a list of all nodes in the graph. We're going to leave the implementation of this function as an exercise. The result of `find_leaf` will be a list beginning with the first leaf found. Only the first element of this list is really wanted, so we will use `first` to get that element.

Let's try `find_leaf` on the example graph:

```
rex > find_leaf(graph);
[3, 4, 5, 6]
```

indicating that 3 is a leaf, since it is the first element of a non-empty list. We can thus incorporate function `find_leaf` into one that tests whether there is a leaf:

```
no_leaf(Graph) = find_leaf(Graph) == [];
```

To remove a *known* leaf from a graph represented as a list of pairs, we must drop all pairs with the leaf as second element (there are no pairs with the leaf as first element, by definition of "leaf"). Here we use the function `drop` to do the work:

```
remove_leaf(Leaf, Graph) =

    drop((Pair) => second(Pair) == Leaf, Graph);
```

Similar to uses of `no` and `find`, the first argument of `drop` is a predicate. The resulting list is like the original list `Graph`, but with all pairs satisfying the predicate removed.

To test `remove_leaf` in action:

```
rex > remove_leaf(3, graph);
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]
```

Now we work this into a function that finds a leaf and removes it. We'll use the same name, but give the new function just one argument. By the way, here's where we apply `first` to the result of `find`:

```
remove_leaf(Graph) =

    remove_leaf(first(find_leaf(Graph)), Graph);
```

This function in action is exemplified by:

```
rex > remove_leaf(graph);
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]
```

Now we have one issue remaining: the iteration of leaf removal until we get to a stage in which either the graph is empty or no leaf exists. The following scenario indicates what

we'd like to have happen: We create new graphs as long as `no_leaf` is false, each time applying `remove_leaf` to get the next graph in the sequence. The reader is encouraged to trace the steps on a directed graph diagram.

```
rex > graph1;
[[1, 2], [2, 3], [2, 4], [4, 5], [6, 3], [4, 6], [5, 6]]

rex > no_leaf(graph1);
0

rex > graph2 = remove_leaf(graph1);
1

rex > graph2;
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]

rex > no_leaf(graph2);
0

rex > graph3 = remove_leaf(graph2);
1

rex > graph3;
[[1, 2], [2, 4], [4, 5]]

rex > no_leaf(graph3);
0

rex > graph4 = remove_leaf(graph3);
1

rex > graph4;
[[1, 2], [2, 4]]

rex > no_leaf(graph4);
0

rex > graph5 = remove_leaf(graph4);
1

rex > graph5;
[[1, 2]]

rex > no_leaf(graph5);
0

rex > graph6 = remove_leaf(graph5);
1

rex > no_leaf(graph6);
1

rex > graph6;
[]
```

The fact that the final graph is `[ ]` indicates that the original graph was acyclic.

Of course it is not sufficient to apply the transformations manually as we have done; we need to automate this iterative process using a function. Let's postulate a function to do the iteration, since we've not introduced one up until now:

```
iterate(Item, Action, Test)
```

will behave as follows. If `Test(Item)` is `1`, then iteration stops and `Item` is returned. Otherwise, iteration continues with the result being

```
iterate(Action(Item), Action, Test).
```

In other words, `Action` is applied to `Item`, and iteration continues. To accomplish our overall acyclic test then, we would use:

```
is_acyclic(Graph) =

    iterate(Graph, remove_leaf, no_leaf) == [];
```

which reads: "iterate the function `remove_leaf`, starting with `Graph`, until `Graph` is either empty or has no leaf, applying `Action` at each step to get a new `Graph`;. If the result is empty, then `Graph` is acyclic, otherwise it is not." In other words, the functional description succinctly captures our algorithm.

To demonstrate it on two similar test cases:

```
rex > is_acyclic([ [1, 2], [2, 3], [2, 4], [4, 5],
                   [6, 3], [4, 6], [5, 6] ]);
1

rex > is_acyclic([ [1, 2], [2, 3], [2, 4], [4, 5],
                   [6, 3], [6, 4], [5, 6] ]);
0
```

A Game-Playing Example

In this example, we devise a function that plays a version of the game of *nim*: There are two players and a list of positive integers. Each integer can be thought of as representing a pile of tokens. A player’s turn consists of selecting one of the piles and removing some of the tokens from the pile. This results in a new list. If the player removes all of the tokens, then that pile is no longer in the list. On each turn, only one pile changes or is removed. Two players alternate turns and the one who takes the last pile wins.

Example: The current set of piles is [2, 3, 4, 1, 5]. The first player removes 1 from the pile of 5, leaving [2, 3, 4, 1, 4]. The second player removes all of the pile of 4, leaving [2, 3, 1, 4]. The first player does the same, leaving [2, 3, 1]. The second player takes 1 from the pile of 3, leaving [2, 2, 1]. The first player takes the pile of 1, leaving [2, 2]. The second player takes one from the first pile, leaving [1, 2]. The first player takes one from

the second pile, leaving [1, 1]. The second player takes one of the piles, leaving [1]. The first player takes the remaining pile and wins.

There is a strategy for playing this form of nim. It is expressed using the concept of *nim sum*, which we will represent by ⊕. The nim sum of two numbers is formed by adding their binary representations column-wise *without carrying*. For example, 3 ⊕ 5 = 6, since

```
3      = 0 1 1
5      = 1 0 1
-------------
6      = 1 1 0
```

The nim sum of more than two numbers is just the nim sum of the numbers taken pairwise.

The strategy to win this form of nim is: *Give your opponent a list with a nim-sum of 0.* This is possible when you are given a list with a non-zero nim sum, and only then. Since a random list is more likely to have a non-zero nim sum than a zero one, you have a good chance of winning, especially if you start first and know this strategy.

To see why it is possible to convert a *non-zero* nim sum list to a zero nim sum list by removing tokens from one pile only, consider how that sum got the way it is. It has a high-order 1 in its binary representation. The only way it could get this 1 is that there is a number in the list with a 1 in that position. (There could be multiple such numbers, but we know there will be an odd number of them.) Given `s` as the nim sum of the list, we can find a number `n` with a 1 in that high-order position. If we consider `n ⊕ s`, that high-order 1 is added to the 1 `in s`, to give a number with a 0 where there was a 1. Thus this number is less than `n` . Therefore, we can take away a number of tokens, which reduces `n` to `n ⊕ s`. For example, if `n` were 5 and `s` were 6, then we want to leave 5 ⊕ 6 = 3. So what really counts is what we leave, and what we take is determined by that.

To see why the strategy works, note that if a player is given a non-empty list with a nim sum of 0, the player cannot both remove some tokens and leave the sum at 0. To see this, suppose that the player changes a pile with `n` tokens to one with `m` tokens, `m < n`. The nim sum of the original list is `n ⊕ r`, where `r` stands for the nim sum of the remainder of the piles. The new nim sum is `m ⊕ r`. Consider the sum `n ⊕ m ⊕ r`. This sum is equal to `m ⊕ n ⊕ r`, which is equal to `m`, since `n ⊕ r` is 0. So the new nim sum can only be 0 if he leaves 0 in some pile, i.e. he takes the entire pile. But taking a whole pile of size `n` is equivalent to nim-adding `n` to the nim sum, since `r` is the new sum and using the associative property for ⊕:

```
n ⊕ (n ⊕ r) == (n ⊕ n) ⊕ r
            == r
```

since

`(n ⊕ r)` is the original nim sum, assumed to be 0.
`(n ⊕ n) == 0` is a property of ⊕
`(0 ⊕ r) == r` is a property of ⊕

The only way nim-adding `n` to 0 can produce 0 is if `n` itself is 0, which is contradictory.

By always handing over a list with a sum of zero, a player is guaranteed to get back a list with a non-zero sum, up until the point where there is one pile left, which he takes and wins the game.

Now let's see how to make a good nim player as a function `play_nim`. Assume that the argument to our function is a non-empty list with a nim sum of non-zero. Then the player would decompose the problem into:

> Find the nim sum of the list, call it `s`.
>
> Find a pile to which `s` can be nim-added to produce a number less than `s`. Make a new list reflecting that change.

We can show this as:

```
play_nim(L) = make_new_list(nim_sum(L), L);
```

We can construct `nim_sum` by using `reduce` and a function that makes the nim sum of two numbers. Let's call this function `xor`. Then we have

```
nim_sum(L) = reduce(xor, 0, L);
```

The value of `make_new_list(s, L)` must make a list by replacing the *first* element `n` of `L` such that `s` ⊕ `n` < `n` with the value `s` ⊕ `n`. We don't have a function like this in our repertoire just yet, so let's postulate one:

```
change_first(P, F, L)
```

creates a new list from `L` by finding the first element satisfying predicate `P` and applying the function `F` to it, leaving other elements unchanged. We will have to appeal to the next chapter to see how to go further with `change_first`. We also need to drop the element in the case that it is 0. This can be done by our function `drop`:

```
make_new_list(s, L) =
  drop((n) => n == 0,
       change_first((n) => (xor(s, n) < n), (n) => xor(s, n), L));
```

So we have reduced the original problem to that of providing `xor` and `change_first`.

To complete our nim player, we have to provide an action for the case it is given a list with a non-zero sum. How to provide such alternatives will be discussed in the next chapter.

Exercises

1. •• Develop an implementation of the function `nodes` that returns the list of nodes of a graph represented as a list of pairs.

2. ••• Develop an implementation of the function `xor` used in the nim example.

3.17 Conclusion

This chapter has shown how information structures can be transformed using functions. Such techniques provide a powerful collection of viewpoints, even if we do not use the tools exclusively. The language rex was used to illustrate the concepts, however the ideas carry forward into many varieties of language, functional and otherwise. Later on, for example, we show how to express them in Java.

3.18 Chapter Review

Define the following concepts or terms:

acyclic graph test
anonymous function
`assoc` function
association list
composition of functions
definition by enumeration
definition by equation
higher-order function
lists as functions
leaf of a graph
`map` function
`mappend` function
nim sum
pipeline principle
predicate
`reduce` function
satisfy