4. Low-Level Functional Programming

4.1 Introduction

In the previous chapter, we saw how to use various functions over information structures defined in chapter 2. The emphasis in the previous chapter was to develop high-level thinking about working with such structures. The functions with which we worked were assumed to be built into rex. However large a repertoire of functions is provided in a functional language, there will usually be some things that we'd like to do that can't be captured in a manner that is as readable or as efficient as we might like. In the current chapter, we show how to write custom definitions of functions.

4.2 List-matching

Now we illustrate list decomposition using matching within a definition. Consider the form

```
[F | R]
```

(read `F` "followed by" `R`). This form represents a *pattern* or *template* that matches all, and only, non-empty lists. The idea is that identifier `F` matches the first of the list and identifier `R` matches the rest of the list. We can test this by trying a definition:

```
rex > [F | R] = [1, 2, 3];
1
```

The `1` (for *true*) indicates that the definition was successful. We can check that identifiers `F` and `R` are now bound appropriately, `F` to the first element of the list and `R` to the rest of the list:

```
rex > F;
1

rex > R;
[2, 3]
```

A definition of the form

identifier = *expression*;

will always succeed, but it is possible for a definition involving list matching to *fail*. For example,

```
rex > [F | R] = [ ];
0
```

Here the definition fails because we attempt to match `[F | R]` against the empty list. Such a match is impossible, because the empty list has no elements, and therefore no first element. In a similar way, an attempt to apply functions `first` or `rest` to the empty list results in an error:

```
rex > first([ ]);
*** warning: can't select from null list [ ]

rex > rest([ ]);
*** warning: can't take rest of null list [ ]
```

Extended Matching

The list-matching notation may be extended to extract an arbitrary number of initial elements of a list. For example, to extract the first, second, and third elements:

```
rex > [F, S, T | R] = [1, 4, 9, 16, 25, 36];
1
```

As before, the 1 indicates the definition succeeded. We can check that the correct identifications were made:

```
rex > F;
1

rex > S;
4

rex > T;
9
```

This time, however, `R` is bound to the portion of the list after the first three elements:

```
rex > R;
[16, 25, 36]
```

When we match using the vertical bar `|` we can do so only if it is the last punctuation item in a match template. For example, the following attempted match is syntactically *ill-formed*.

```
[F | R, S, T] = [1, 3, 9, 16];
```

rex would report this as a syntax error. On the other hand, the bar would not be used if we wanted to match a list with an exact number of elements:

```
rex > [F, S, T] = [1, 3, 9];
1
```

We must get the number right however, or the match will fail.

```
rex > [F, S, T, X] = [1, 3, 9];
0
```

Also, using an identifier twice with the same left-hand side will fail unless it happens that both occurrences would get bound to the same value.

```
rex > [F, F, S] = [1, 3, 9];
0
```

The above match failed because it is ambiguous whether `F` is getting bound to `1` or `3`.

```
rex > [F, F, S] = [1, 1, 9];
1
```

The above match succeeded despite there being two definitions of `F`, because both definitions are the same. This style is nonetheless regarded as awkward and should be avoided.

We mention these points not because they are so essential in what we will be doing, but because they help further emphasize that = is definition, not assignment. The difference is admittedly subtle: assignment presupposes a memory location containing a value; definition merely identifies a value with an identifier, but there is not necessarily any memory location. So assignment can be treated as definition, if desired, by making sure the location has a value before any use of its value and by not re-assigning the value once established.

Matching in Lists of Lists

The idea of binding variables by matching a template to a list extends naturally to lists of lists. For example, consider the template

```
[ [A, B] | X ]
```

This would match a list of at least one element, the first element of which is a list of exactly two elements. In other words, the only lists it would fail to match would be the empty list and a list that did not begin with an element that is a pair. Let's test this idea using rex:

```
rex > [ [A, B] | X] = [ [1, 2], 3];
1

rex > A;
1

rex > B;
2

rex > X;
[3]
```

```
rex > [ [A, B] | X] = [1, 2, 3];
0
```

We see that the match failed in the last attempt; the list does not begin with an element that is a pair.

Exercises

5. •• For all possible pairs of pattern vs. list below, which patterns match which lists, and what bindings are defined as a result of a successful match? For those pairs that don't match, indicate why.

patterns	**lists**
`[F	R]`	`[1, 2, 3]`
`[F, S	R]`	`[1, [2, 3] ]`
`[F, S, T]`	`[ [1], 2, 3]`
`[ [F], S]`	`[1, 2	[3]]`
`[[F, S]	R]`	`[ [1, 2], 3]`
`[F, S, T	R]`	`[1, 2, [3, 4] ]`

6. •• For the patterns above, give a word description for the lists that they match.

7. ••• Give an algorithm for determining whether a pattern matches a list. It should be something like the equality checking algorithm.

4.3 Single-List Inductive Definitions

In chapter 2 we mentioned the *fundamental list-dichotomy*: A list is either:

- *empty*, i.e. has no elements, or
- *non-empty*, i.e. has a first element and a rest

A great many, but not all, low-level definitions are structured according to this dichotomy. In defining a function that takes a list as argument, we:

- Define what the function does on the empty list.
- Define what the function does on a typical non-empty list.

Typically in the second case, the non-empty list is expressed in terms of its first and rest: `[F | R]`. The definition in the second case would generally use the function's value for the argument `R` to define the value for argument the larger argument `[F | R]`. This type of definition is called *inductive* or *recursive*. The difference between these two terms is primarily one of viewpoint. In inductive definitions, we think in terms of *building up* from definitions on simpler structures to definitions on the more complex ones, while in recursive definitions, we think in terms of *decomposing* a complex structure into simpler ones.

Let's consider the definition of the function `length` that returns the length of its list argument. Using the list-dichotomy, we must say what length does with an empty list. The obvious answer is to return 0. We express this as a *rewrite rule*:

```
length( [ ] ) => 0;
```

It is called a *rewrite* rule because whenever we see `length( [ ] )` we can just as well rewrite it as `0`. The symbol

=>

is read "rewrites as". Thanks to the idea of referential transparency, we are evaluating the expression for a value, not for an effect. This rule is called the *basis* of the induction, since it does not convert to an expression involving `length`.

The other part of the dichotomy is a non-empty list. We express this using the generic form `[F | R]`. What is the length of a list of this form. The answer is it is 1 more than the length of `R`. So we give a second rule:

```
length( [F | R] ) => length(R) + 1;
```

Again this is a rewrite rule because whenever we see `length(L)` where `L` is a non-empty list, we can effectively replace it with `length(R) + 1` where `R` is the rest of the list. Because this rule appeals to the definition of `length` for its final value, it is called the *induction rule* rather than the basis.

For example, `length([2, 3, 5, 7])` is replaceable with `length([3, 5, 7]) + 1`. By continuing this replacement process, using one of the two rules each time, and evaluating the final result as a sum, we can a number that is the actual length of the list. This result is called *irreducible*, because it contains no further function applications that could be rewritten.

```
   length([2, 3, 5, 7])
=> (length([3, 5, 7])             + 1)
=> ((length([5, 7])          + 1) + 1)
=> (((length([7])       + 1) + 1) + 1)
=> ((((length([ ]) + 1) + 1) + 1) + 1)
=> ((((          0 + 1) + 1) + 1) + 1)
=> (((              1  + 1) + 1) + 1)
=> ((                    2  + 1) + 1)
=> (                         3  + 1)
=>                               4
```

Here we have assumed that the + operator is grouped from left to right, so we can only rewrite the + expressions when a numeric value for the left argument is known. In evaluating the + expressions, we are assuming very simple properties. These could be expressed more rigorously themselves using rewrite rules.

Sometimes we do not wish to see the rewrite sequence in this much detail. We use the symbol

==>

to represent a collapsed sequence of intermediate steps. As a relation, ==> represents the *transitive closure* of the relation =>, as described earlier. For example, we could outline the major plateaus in the above derivation as:

```
    length([2, 3, 5, 7])

==> ((((length([ ]) + 1) + 1) + 1) + 1)

=>  ((((          0 + 1) + 1) + 1) + 1)

==> 4
```

Now let's try another low-level definition, this time for the function `append`. Recall that `append` takes two arguments, both lists, and produces the result of appending the second argument to the first. However, in the spirit of functional programming, neither argument is modified, so the term *append* is slightly misleading; nothing gets appended to the first list in place; instead a new list is created. For example,

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

We are in a section on single-list definitions, yet `append` has two arguments. What gives? Regardless of the number of arguments a function has, we should first look for the possibility of using only *one* of the list arguments on which to apply the fundamental list dichotomy. This argument, if it exists, is called the *inductive argument*. In the case of `append`, the first argument rather than the second turns out to be the right choice for the inductive one. Let us see why.

To append a list `M` to an empty list gives the list `M`. This is expressible by the a rule:

```
append( [ ], M ) => M;
```

To append a list `M` to non-empty list, one that matches `[A | L]` say, we observe that the first element of the result list must be the binding of `A`. Furthermore the rest of the result can be obtained by appending `M` to the shorter list `L`, which is the rest of the original list.

This works out perfectly, in that the parts we get by decomposing the original list are exactly what we need to construct the new one.

```
append( [A | L], M ) => [A | append(L, M)];
```

We can check this by a specific example: In evaluating

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

we see how the arguments, or *actual parameters*, of the expression match up to the *formal parameters* of the rule:

```
          actual parameters
           ↓             ↓
append( [1,  2, 3], [4, 5] )
         ↑   ↑        ↑
         ↓   ↓        ↓
append( [A | L],      M     ) => [A | append(L, M)];
         ↑   ↑        ↑
         formal parameters
```

Formal parameter `A` matches the first element `1` of list `[1, 2, 3]`. Formal parameter `L` matches the rest of that list, `[2, 3]`. Formal parameter `M` matches the entire list `[4, 5]`. When we rewrite, or replace the expression with the right-hand side of the rule, the formal parameters carry their values along and the expression that results is constructed from those values:

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

becomes

```
[1 | append([2, 3], [4, 5])]
```

simply by substituting the value of each variable for the value itself. In the computer science literature, this type of rewriting is sometimes called the ***copy rule*** or ***beta reduction***. Rewriting is certainly more complicated to describe than it is to use. Once we have worked through a number of examples, its use should come fairly naturally. Note that the idea of matching formal and actual parameters is not very language specific; some variation of this idea occurs in most computer languages, although not all languages support decomposing lists by pattern matching.

Continuing the above example, we are thus left with another expression containing `append`. Hence another rewrite is requested, this time with a different set of actual parameters. The process of rewriting continues until we get to apply the first rule for `append`, which leaves us with no further instances of `append` to be rewritten.

```
   append([1, 2, 3], [4, 5])
=> [1 | append([2, 3], [4, 5]) ]
=> [1 | [2 | append([3], [4, 5]) ] ]
```

```
== [1, 2 | append([3], [4, 5]) ]
=> [1, 2 | [3 | append([ ], [4, 5]) ] ]
== [1, 2, 3 | append([ ], [4, 5]) ]
=> [1, 2, 3 | [4, 5] ]
== [1, 2, 3, 4, 5]
```

Here the == steps just recall that these are two ways of writing equivalent expressions. The main difference between this series of rewrites and the earlier `length` series is that these construct a *list* from outside in, whereas `length` constructs a number.

What would have happened had we chose to use the second argument instead as the induction variable? The basis would still have been okay:

```
append( L, [ ] ) => L;
```

However, there is a snag when we get to the induction rule:

```
append( L, [A | M] ) => ??
```

There is no elegant way to use the constructs we have here to achieve the result. The single element A does not start the desired resulting list. Being able to recognize the correct variable for induction is a skill that will develop as we do more exercises.

A confusion often held by newcomers is the difference between the following two expressions:

`append(L, M)`   *vs.*   `[L | M]`

The expression on the right produces a new list starting with `L` as an element of the result list. This element does not have to be a list, although it could be if the result is going to be a list of lists. In contrast, the expression on the left *always* needs a list for `L` and the elements of `L`, not `L` itself, are elements of the result list. Now let's see what happens if we use the right-hand expression in place of append in one of the preceding examples.

```
rex > [ [2, 3, 5, 7] | [11, 13] ];
[[2, 3, 5, 7], 11, 13]
```

This is quite different from what `append` produces, a list of six elements:

```
rex > append( [2, 3, 5, 7], [11, 13] );
[2, 3, 5, 7, 11, 13]
```

4.3 Rules Defining Functions

In the previous section, we presented two rules for the function `append` that creates a new list having the elements of the second argument appended to the first:

```
append( [ ], M ) => M;
append( [A | L], M ) => [A | append(L, M)];
```

A question that is proper to ask is *in what sense does a set of rules define a function?* For the case of `append`, we can answer this question using the same reasoning that leads to the construction of the rules in the first place: `append` is a function on the set of all lists if it prescribes a correct rewriting for all pairs of lists. Examination of the rules reveals that the second argument plays a very minor role: It is never decomposed. The only thing that is required is that it be a list, in order that the first rule make sense when that argument is returned as the result (the result is supposed to be a list). So we can focus on the first argument, the inductive one.

An arbitrary list is either the empty list or a non-empty list. The space of all lists is exhausted by these two possibilities. The case of the first argument being empty is handled by the first rule, and the other case, an infinite set of possibilities, is handled by the second rule. This reasoning leads us to conclude that *there will never be a pair of lists for which no rule is applicable*. This is certainly a good start toward defining a function. Furthermore, for a given pair of lists, *only one rule is applicable*; there is never ambiguity as to which rule to choose. So this tells we have at least a *partial* function.

But there is another issue in establishing that the rules give us a function. What we have argued above is that there will always be a rule for a given pair of lists. We haven't shown that, once we apply the rule, the ensuing series of rewrites will always *terminate*. In the case of `append`, here is the way to show termination: Notice that if the first rule applies, no expression in need of rewriting is introduced. This is the *basis* of the definition. In other words, for a list of length 0, the rewrite series will terminate. Next consider the case of a non-empty first argument. We see that the length of the argument of `append` on the right-hand side is *one less* than the length on the left-hand side, thanks to having taken away the first element `A` of the first list. In other words, every application of the second rule effectively *shrinks* the first argument by one, figuratively speaking (because we are not modifying the actual argument in any way). Thus, no matter what length we start with in that argument, it will keep shrinking as further rule applications occur. But when it shrinks to length 0, i.e. the empty list, it can shrink no further. The first rule then applies and rewriting terminates.

What we have just described is a narrative version of an *inductive argument*, or proof by induction. More succinctly, it could be captured as follows:

Claim:

For every finite list `L`, `append(L, M)` produces a terminating sequence of rewrites.

Proof:

(Basis): For `L` being `[ ]`, `append([ ], M)` generates a terminating rewrite sequence, since there are no further rewrites according to the first rule.

> (Induction Step): Assume that `append(L, M)` generates a terminating rewrite sequence. Consider the case of an argument one longer, `append([A | L], M)`. According to the second rule, the continuation of the rewriting sequence is based on the sequence for `append(L, M)`. This sequence terminates by assumption, therefore the sequence for `append([A | L], M)` also terminates, it being one step longer.

We will not go through such proofs for most of the functions to be presented. However, it is important that the reader get comfortable with the logic of the proof, in order to be able to construct sound sets of rules.

4.4 Lists vs. Numbers

Some of our functions involve lists, some involve only numbers, and some, such as `length`, involve combinations of both. A useful viewpoint for reasoning is that *natural number* (numbers in the set {0, 1, 2, 3, ...}) can be thought of as special cases of lists. Consider an arbitrary element, say •. Then the number *n* can be thought of as a list of *n* of these elements. For example,

0 is `[ ]`
1 is `[•]`
2 is `[•, •]`
3 is `[•, •, •]`
...

Representing numbers in this way is sometimes called the *1-adic* representation. This is, no doubt, the representation for numbers used in the stone age (think of each • as a stone). We are not proposing a return to that age for actual calculation, but we do suggest that this analogy provides a basis for reasoning about numbers and for analogies between various functions. For example, if the function `append` is restricted to lists of stones, then it becomes the *addition* function.

A very important theory, known as *recursive function theory*, starts out by defining functions on natural numbers in this very way. While we do not intend to make explicit use of recursive function theory in this book, the ideas are useful as exercises in creating rule sets, so we pursue it briefly. Recursive function theory typically starts with a *successor function,* which in list terms would be defined by one rule:

```
successor(L) => [• | L];
```

In other words, successor adds one to its argument. We can then define addition by using successor and recursion. But rather than showing an argument lists as `[A | L]`, we would show it as `L+1`. (Since all elements of the list are the same, the identity of A is unimportant.) The definition of `add` would be presented:

```
add(0, N) => N;

add(M+1, N) => add(M, N) + 1;
```

The `M+1` on the left is another form of pattern matching available in rex, and can be viewed as a direct translation of `append` specialized to lists of one type of element:

```
append([ ], N) => N;

append([• | M], N) => [• | append(M, N)];
```

On the right-hand side, the `+1` indicates application of the successor function.

Reasoning about such definitions typically uses induction, in the same way we reasoned about `append`. This style of definition is used to build up a *repertoire* of functions. For example, having defined `add`, we can then define `multiply`:

```
multiply(0, N) => 0;

multiply(M+1, N) => add(multiply(M, N), N);
```

Reasoning that `add` and `multiply` are functions for all natural numbers is essentially the same as reasoning that `append` terminates.

Defining subtraction in this way is a little tricky. If you don't believe me, try it before reading on. We start with a simpler function, `predecessor`. Informally, the predecessor of a number is the number minus 1, but since 0 has no predecessor in the natural numbers, we make its predecessor 0 for sake of convention and completeness. Likewise, we define `subtract`, when the first argument is smaller than the second, to be 0. This is known in the literature as *proper subtraction*. In the spirit of building up definitions from nothing but successor, we can't appeal to a comparison operator yet.

```
predecessor(0) => 0;

predecessor(M+1) => M;
```

Now we can define `subtract` using the second argument as an induction variable:

```
subtract(M, 0) => M;

subtract(M, N+1) => subtract(predecessor(M), N);
```

Actually, we could use a different set of rules and bypass `predecessor`:

```
subtract(M, 0) => M;

subtract(0, N) => 0;

subtract(M+1, N+1) => subtract(M, N);
```

However, this rule set is trickier since it uses two induction variables simultaneously. Doing so can be more error-prone unless you have a very clear idea of what you are doing.

Note also the following point about the second set of subtract rules: This is our first set of rules where there was overlap between the applicability of the rules. In particular, `subtract`(0, 0) could invoke both the second and the first rules. Fortunately in the present case, the result is the same either way. In order to avoid possible misinterpretations in the future, and to actually make rule definitions simpler, we adopt the following convention:

> **rex rule-ordering convention:**
>
> In rex, the rules are tried in top-to-bottom order. The first applicable rule is used, and subsequent rules, while they might have been applicable on their own, are not considered if an earlier rule applies.

As a simple example where this makes a difference, consider defining a function `is_zero` that tests whether its argument is 0:

```
is_zero(0) => 1;
is_zero(N+1) => 0;
```

Under the rule-ordering convention, we could have used:

```
is_zero(0) => 1;
is_zero(N) => 0;
```

since the second rule will never be used if the argument is 0, thanks to the rule-ordering convention. Similarly, define `non_zero`:

```
non_zero(0) => 0;
non_zero(N) => 1;
```

Having defined `subtract`, we can define `less_than_or_equal` (for natural numbers):

```
less_than_or_equal(M, N) => is_zero(subtract(M, N));
```

We can define equality in many ways, for example using rex's argument matching capabilities:

```
equal(M, M) => 1;
equal(M, N) => 0;
```

The second rule is valid only by virtue of the rule-ordering convention; two different variables can be bound to the same value.

If this type of matching were not available, we could still construct an equality predicate in other ways, e.g.

```
equal(M, N) => non_zero(multiply(less_than_or_equal(M, N),
                                 less_than_or_equal(N, M));
```

In other words, two numbers are equal if, and only if, each is less than or equal to the other.

Convention: Henceforth, we will use the typical symbols for the functions we have defined, rather than their "spellings", e.g. `+` instead of `add`, `*` instead of `multiply`, `<=` instead of `less_than_or_equal`, `==` instead of `equal`, etc. Keep in mind that the built-in `-` is ordinary signed subtraction, rather than proper subtraction as defined above.

Exercises

1 • Give rules that define the function `zero` that invariably returns the result 0. Similarly, show that for any number you can name, e.g. `five`, `one_thousand`, etc., you can define a function that invariably returns that number, without actually using the number directly in the definition.

2 •• Give rules that define the function `power` that raises a number to a power, using `multiply` and recursion. For example, `power(2, 3)` would ultimately rewrite to 8.

3 •• Give an inductive argument that shows that the rules given for `length` establish a function on the set of lists.

4 ••• Define rules that define the function `superpower` that bears the same relation to `power` as `power` does to `multiply`. For example, `superpower(2, 3)` ==> 16 and `superpower(2, 4)` ==>65536.

5 •••• Continuing in the above vein, we could define `supersuperpower`, `supersupersuperpower`, and so on, ad infinitum. Give rules for a three-argument function that effectively takes the number of "super"s as a first argument and applies the corresponding function to the remaining arguments. The function you have defined is a version of what is commonly known as "Ackermann's function".

4.5 Guarded Rules

One purpose in preferring a sequential list of rules to a single comprehensive rule is clarity and readability. In some cases, however, clarity is best served by conditioning the applicability of a rule on other than the *form* of the arguments. The concept of a **guard** is useful to provide this additional clarity. A rule is *guarded* if it has the form

lhs => *guard* ? *body*;

format of a guarded rule

Here *guard* ? *body* is an expression for the *rhs* as before. The question-mark separator is what distinguishes this form. Both *guard* and *body* are terms that can ultimately rewrite to values. The rule as a whole is called a **guarded rule**. The meaning of a guarded rule is as follows:

> The rule is considered applicable only if the arguments match as before, *and then* only if the value of *guard* ultimately rewrites to 1 (true). In this case the *lhs* rewrites to the value of *body*.

If the condition of applicability does not hold, then the rule is unusable and we must appeal to later rules to rewrite a given term. Note: the rule ordering convention is still in effect; a later rule is applied only if all previous rules don't apply.

An example of guarded rules occurred in our first rex example, the function for testing whether a number is prime. Here is another example.

Euclid's Algorithm

Euclid's algorithm is an algorithm for finding the greatest common divisor (*gcd*) of two natural numbers. The rules are:

```
gcd(0, Y) => Y;
gcd(X, Y) => X <= Y ? gcd(Y-X, X);
gcd(X, Y) => gcd(Y, X);
```

Euclid's Algorithm

The second rule is guarded, using the `<=` (less than or equal) operator of rex. By convention, the third rule, which contains no guard, is applicable only if the first two rules are not applicable, i.e. only in the case that `X` is not 0 and `X` is greater than `Y`.

There are ways to speed up the computation, e.g. by using the operator `%` (remainder or *modulus*). This amounts to repeated subtraction, in place of division.

Let us trace the rewrite behavior of these rules on a test case, `gcd(18, 24)`. Since 18 factors into 2*3*3 and 24 factors into 2*2*2*3, we can anticipate the result will be 2*3 = 6.

```
gcd(18, 24) ==>
gcd(6, 18)  ==>
gcd(12, 6)  ==>
gcd(6, 12)  ==>
gcd(6, 6)   ==>
gcd(0, 6)   ==>
6
```

Why does Euclid’s algorithm work? The rationale for the algorithm is based on two observations:

> The actual greatest common divisor of the two arguments never changes.
>
> Each time one of the second or third rules is applied, one of the arguments will soon decrease.

The first fact is due to some reasoning about division: If a number *Z* evenly divides both *X* and *Y*, and *X* <= *Y*, then *Z* also divides *Y* - *X*. So if the first rule becomes applicable, we see that *Y* is the greatest common divisor, since it is obviously the largest divisor of both *Y* and 0.

The second fact may be seen from the rule structure: If *X* <= *Y*, (and *X* is not 0, otherwise the first rule would have been used) then *Y* - *X* is clearly less than *Y*. On the other hand, if *X* > *Y*, then, based on the third rule, the second rule will be tried with *X* and *Y* reversed.

Because one of the arguments is bound to decrease and stop at 0, we have that the term will eventually be reduced to a case where the first rule applies, i.e. Euclid’s algorithm always terminates.

Exercises

1 ••• Continue the development of recursive function theory by defining the following, using guards where it is helpful:

```
mod(M, N)
```

is the remainder after dividing `M` by `N` (use the convention that `mod(M, 0)` is 0. (In rex, mod`(M, N)` is available as `M % N`, also read “M modulo N”, or “M mod N”.)

```
div(M, N)
```

is the quotient obtained by dividing `M` by `N` (again with `div(M, 0)` defined to be 0). (In rex, `div(M, N)` is available as `M / N`.)

2 ••• Show how Euclid’s algorithm can be “sped up” if `mod` were available as a primitive function.

4.6 Conditional Expressions

Although not absolutely essential due to the rule notation, for convenience rex allows the Java language notation for conditional expressions:

```
C ? A : B
```

is an expression that has the value A if C rewrites to a number other than 0, otherwise the value is B. This is an extension of the guard idea, providing an immediate alternative in case the guard is false, rather than requiring resolution through another rule. Although the same effect can be achieved with guards and additional rules, the conditional expression is a self-contained unit. As an example, an alternative set of rules for *gcd* would be

```
gcd(0, Y) => Y;
gcd(X, Y) => X <= Y ? gcd(Y-X, X) : gcd(Y, X);
```

4.7 Equations

As noted in the previous chapter, rex supports the notion of defining functions by equations as well as rules. By an *equation*, we mean a single expression that captures all cases. While it would certainly be adequate to give a single rule instead of an equation, using an equation has a signal of finality about it: there will be *no further rules* defining this function. Also, in terms of the rex implementation we provide, an equation will execute more efficiently since there will be no pattern-matching superstructure. Finally, the handling of equations vs. rules is different in the interactive environment provided: If an equation for a function is re-entered, it will be taken as a *re-definition*, whereas if a rule for a function is re-entered, it will be taken as an *additional* rule, rather than as a replacement.

Typically, conditional expressions are used to simulate what would have been separate rules. For example, a single equation defining *gcd* of the previous section would be:

```
gcd(X, Y) = X == 0 ? Y : X <= Y ? gcd(Y-X, X) : gcd(Y, X);
```

4.8 Equational Guards

The rex language allows a guard to consist of an equation that binds a variable to a value for use in the expression that follows. Such variables are used in the *rhs* in the same way any *lhs* variable would be used. A basic *equational guard* takes the form:

Var = *Expression*,

The meaning of this equation is that *Var* is bound to *Expression*. This simple form of equational guard always succeeds. However, equational guards that involve "matching" might not, as will be explained momentarily.

The main use of the simple form of equational guard above would be to give a name to the value of a complicated expression, for one of the following purposes:

- to *avoid multiple evaluations* of the expression, even though its value is used multiple times:

  ```
  f(X) => Y = sqrt(X), g(Y, Y);
  ```

  Here `sqrt` is supposed to be a function that is relatively expensive to evaluate.

- to *document* the meaning of the expression by giving the variable a descriptive name:

  ```
  f(X, Y) => First_Vowel = find(vowel, X),
             g(First_Vowel, Y);
  ```

  Here the expression on the *rhs* of the equation for `First_Vowel` could have been substituted directly as the argument of `g`. However, then the documentary aspect of the name would be lost.

- `to` *`redefine the scope`* `of variable X, which` gives an argument variable a value different from the one it had in the function call. This can be used to provide "wrappers" for expressions that we wish to leave intact but for which we don't wish to use the given argument variables.

  ```
  f(X, Y) => X = g(X, Y), X*Y;
  ```

  Here the `X` used in expression `X*Y` is not the argument `X`, but rather the value of `g(X, Y).`

Equational guards can involve binding multiple variables in a single equation through the use of the list notation.

```
[X, Y, Z] = [1, 2, 3],
```

is a guard that binds each of `X`, `Y`, and `Z` simultaneously. If the result of an evaluation is a list, then this type of guard can be used to select elements from the list, in the manner used in argument pattern matching:

```
[X, Y, Z] = g(W),
```

means that `g(W)` is expected to return a list of three elements, and the variables on the *lhs* get bound to these elements. Here is one place an equational guard can fail: If the list returned does not have exactly three elements. In general, a match must be possible with the *lhs* and the list returned. Similarly,

```
[X, Y | Z] = g(W),
```

matches a list with *at least two* elements. Variable `Z` is bound to the list after the first two elements.

Equational guards may also be cascaded:

$$lhs_1 = Expression_1,\ lhs_2 = Expression_2,\ \ldots,\ lhs_N = Expression_N,$$

If any of the left-hand sides fails, the guard is considered to have failed, in which case rex will try the next rule, if there is one. If there are no more rules, then the function returns a distinguishable failure value. When other functions operate on such values, they typically return failure values themselves.

Example – Computing *mod* from first principles

One way to compute the *mod* or remainder function is as follows:

```
mod(0, K) => 0;

mod(N+1, K) => mod(N, K)+1 == K ? 0 : mod(N, K) + 1;
```

Here we define `mod` by induction on the first variable, basing the value of `mod(N+1, K)` on the value of `mod(N, K)`. The unpleasant part of this definition is that potentially the *rhs* sub-expression `mod(N, K)+1` must be computed twice. A way to avoid this would be to introduce a variable, say `R`, to stand for the value of `mod(N, K)+1` then use the value of `R` a second time if necessary. The following alternate set of rules accomplishes this:

```
mod(0, K) => 0;
```

$$\texttt{mod(N+1, K) => }\underbrace{\texttt{R = mod(N, K)+1}}_{\uparrow\ \textit{equational guard defining } \texttt{R}}\texttt{, (R == K ? 0 : R);}$$

The use of equational guards provides a style found in mathematically-oriented texts. It is often convenient to introduce variables to stand for large expressions. So the text would read:

Let R = … *some expression*… .

Then later on either the text or another expression can use R to represent the value of that expression.

A similar style often used in writing is "… R …, *where* R = … *some expression*… ". Both forms are especially convenient when R is referred to more than once. Some programming languages provide a **let** construct or a **where** construct to achieve the same end. The construct **letrec** ("let recursive") is also used when the variable is defined recursively in terms of itself.

4.9 More Recursion Examples

As much as possible, we would like to use powerful concepts such as recursion to simplify our work. When a problem involves structures such as lists and numbers that can be arbitrarily large, often the only reasonable way to get a handle on the problem is to deal with simple cases directly, and deal with the general case by breaking it down into simpler cases that we have assumed can be handled. The tool of recursion can work like magic in the hands of the knowledgeable. Therefore, the **recursion manifesto** is

Let recursion do the work for you.

We applied this principle in several previous examples, but it is time now to really exercise it

Range: Creating a List

The function `range` synthesizes a list from two numbers, *M* <= *N*. Specifically,

`range(M, N)` yields [M, M+1, …, N].

If M > N, the result is specified to be the empty list. Rules that define *range* are:

```
range(M, N) => M > N ? [ ];

range(M, N) => [M | range(M+1, N)];
```

Scale: Transforming a List

Suppose we wish to multiply each element in a list of numbers by a constant K, returning a new list. A function `scale` is to be devised such that `scale(K, L)` is this new list. Here we let recursion work for us, by decomposing into the empty and non-empty list cases and only coding the scaling of the first element in the latter.

```
scale(K, [ ]) => [ ];
scale(K, [ A | L ] ) => [ K*A | scale(K, L) ];
```

Illustration:

```
scale(3, [7, 5, 2])            =>
[ 21 |  scale(3, [5, 2])]      =>
[ 21, 15 | scale(3, [2]) ]     =>
[ 21, 15, 6 | scale(3, [ ]) ] =>
[21, 15, 6 | [ ] ]             =>
[21, 15, 6]
```

Note: The philosophy of "functional programming" (which is what we do in rex) is that we never modify lists in place. We only create new lists, possibly out of existing lists. But the original list remains intact as long as needed.

We mention the above philosophy explicitly, as it is quite possibly foreign, depending on the manner to which one is exposed to programming.

The Map Functions

In the previous chapter, we showed an alternate definition of `scale,` which used `map`. But how would `map` be defined from first principles? It is essentially the same pattern as `scale`, except that the first argument is a function, not a number:

```
map(F, [ ] ) => [ ];

map(F, [A | X]) => [ F(A) | map(F, X) ];
```

For mapping over two lists simultaneously the rules are:

```
map(G, [ ], _) => [ ];

map(G, _, [ ] ) => [ ];

map(G, [A | X], [B | Y]) => [ G(A, B) | map(G, X, Y) ];
```

mapping a function across a pair of lists

The presence of two basis cases allows us to deal with the case where the lists are not the same `length`. As soon as one list is reduced to [ ], the recursion will stop, so the length of the result will be that of the shorter of the two argument lists.

Reducing a List

Quite often we have need to compute the result of a binary (i.e. two-argument) operator being applied to "reduce" a list to a single item. An example would be to "add up" the elements of a list of numbers. The rules specialized to the add operator might be:

```
add_up( [ ] ) => 0;
add_up( [A | X] ) => A + add_up(X);
```

The same technique could be used for multiplying the elements of a list, or to applying any binary function H to a list in the same pattern. We do need to specify a base value for the case of the empty list. The general form of the rules for reducing a list using operator `H` are:

```
reduce( _, Base, [ ] ) => Base;

reduce( H, Base, [A | X] ) => H(A, reduce(H, Base, X));
```

reducing a list by a function `H`*, together with a base value*

This set of rules "biases" the reduction to the right, i.e. the result of

$$\texttt{reduce(H, Base, [}X_0\texttt{, }X_1\texttt{, …, }X_{N-1}\texttt{])}$$

will be that of

$$\texttt{H(}X_0\texttt{, H(}X_1\texttt{, …, H(}X_{N-1}\texttt{, Base) …))}$$

Horner's Rule

Consider the requirement of evaluating polynomials

$$a_0*x^n + a_1*x^{n-1} + \ldots + a_{n-1}*x^1 + a_n*x^0$$

where we are given a list with low-order coefficient first $[a_n, a_{n-1}, \ldots, a_1, a_0]$ and a value x. A method that is commonly used to reduce the number of multiplications is to evaluate the polynomial using the following scheme, known as *Horner’s Rule*:

$$(\ldots((a_0*x + a_1)*x + \ldots + a_{n-1})*x + a_n$$

This has the computational advantage of not computing the large powers separately. Instead they are folded into the multiply-add’s that have to be done anyway. This elegant scheme is concisely represented by the following rex recursion:

```
horner(X, [ ]) => 0;

horner(X, [A | L]) => A + X * horner(X, L);
```

Principle of Radix Representation

This is actually an application of Horner’s rule. Here we assume that a list represents a radix numeral for a number, least-significant digit first. We wish to construct rules for a function *value* that computes the number. The radix r representation of a number is a sequence of digits

$$d_{n-1}\ d_{n-2} \ldots d_2\ d_1\ d_0$$

where each d_i is in a digit in the set {0, 1, …, r-1}. The number represented by the sequence is the value of the expression

$$d_{n-1}*r^{n-1} + d_{n-2}*r^{n-2} + \ldots + d_2*r^2 + d_1*r^1 + d_0*r^0$$

For example, if r = 2, we have the binary representation, where each d_i is either 0 or 1. A numeral such as

1 1 0 1

represents the number designated by the decimal numeral 13, since

$$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 == 13$$

and

$$1*10^1 + 3*10^0 == 13$$

It is important to notice that the expression for the value can also be computed another way, in a nested fashion using *Horner's Rule*:

$$((\ldots ((0 + d_{n-1})*r + d_{n-2})*r + \ldots + d_2)*r + d_1)*r + d_0$$

The function `value` will accept a list of digits [d_0, d_1, d_2, …, d_{n-2}, d_{n-1}] and return the value. It will use the Horner’s rule version of the expression. The idea is that we can compute values of a sequence by multiplying by r the value of all but the first element of the sequence (treated as a numeral with one fewer digit) then adding the remaining digit. In other words, notice that the sub-expression of the above

$$(\ldots ((0 + d_{n-1})*r + d_{n-2})*r + \ldots + d_2)*r + d_1$$

looks a lot like the *original* expression. The only difference is that the d subscripts have been "shifted" by one position:

$$\begin{tikzcd}
((\ldots((0 + d_{n-1})*r + & d_{n-2} \arrow[d, leftrightarrow] &)*r + \ldots + & d_2 \arrow[d, leftrightarrow] &)*r + & d_1 \arrow[d, leftrightarrow] &)*r + & d_0 \arrow[d, leftrightarrow] \\
((\ldots((0 + & d_{n-1} &)*r + \ldots + & d_3 &)*r + & d_2 &)*r + & d_1
\end{tikzcd}$$

That is,

$$\text{value}([\, d_0, d_1, d_2, \ \ldots, \ d_{n-2}, d_{n-1} \,]) ==$$

$$d_0 + r * \text{value}([\, d_1, d_2, \ \ldots, \ d_{n-2}, d_{n-1} \,])$$

In order to set things up to apply recursion, we only need to identify the sequence `[ Digit | Digits ]` with $[d_0, d_1, d_2, \ \ldots, \ d_{n-2}, d_{n-1}]$ to convert this equation into a rex rule, adding a new variable `Radix` for the radix:

```
value( [ Digit | Digits ], Radix ) =>

         Digit + Radix * value( Digits, Radix );

value( [ ], Radix ) => 0;
```

Radix Interpretation of a list of digits, Least-significant first

Here we have added a basis rule for the empty sequence.

Let us check this with the binary numeral 1 1 0 1, which we said has a value of 13. The list representation, least-significant digit first, will be [1, 0, 1, 1]. By the rules:

```
   value( [1, 0, 1, 1], 2 )
=> 1 + 2 * value( [0, 1, 1], 2 )
=> 1 + 2 * (0 + 2 * value( [1, 1], 2 ))
=> 1 + 2 * (0 + 2 * (1 + 2 * value( [1], 2 )))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 2*value( [ ], 2 ))))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 2*0)))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 0)))
=> 1 + 2 * (0 + 2 * (1 + 2 * 1))
=> 1 + 2 * (0 + 2 * (1 + 2))
=> 1 + 2 * (0 + 2 * 3)
=> 1 + 2 * (0 + 6)
=> 1 + 12
=> 13
```

Now consider the inverse function for radix conversion: Given a number, produce a list of its digits in a given radix notation. For now, we will develop the list least-significant digit first. We can either apply `reverse` to the result, or use a later technique to get the digits in most-significant digit first order. We can find the least-significant digit by using the integer remainder function *mod*: `N % M` is the remainder that occurs when `N` is divided by `M` using integer division /. We can find the remaining digits by applying the same process to the quotient of the number and the radix. We are letting recursion do the work for us. We use an auxiliary function digits1 so that our function `digits` handles the case `0` in such a way that the result is `[0]` rather than an empty list. The rules are thus:

```
digits(0, Radix) => [0];
digits(N, Radix) => digits1(N, Radix);

digits1(0, Radix) => [ ];
digits1(N, Radix) => [ (N % Radix) | digits1(N / Radix, Radix) ];
```

Forming the digit list of a natural numbering a given radix

The "Radix" Principle

Although most of the computation with which we will be concerned is "digital" in nature, we use the term "radix principle" to connote algorithmic techniques that rely specifically on the data being representable by a series of digits, such as in radix representation. Some algorithms rely on this fact for efficiency, while others do not. The radix principle makes it possible to do arithmetic with reasonable efficiency. If, for example, all arithmetic were done using tally representation, not much useful computation would get done. We already saw how much space saving was afforded by using radix notation instead of tallies. Now consider the time saved in doing arithmetic with a radix representation instead of tallies. The addition of two *arbitrarily-long* binary numerals, represented as lists, least-significant digit first, can be expressed using the following rules:

```
add_bin(X, Y) => add_bin(X, Y, 0);

add_bin([ ], X, Carry) => add_digit(X, Carry);
add_bin(X, [ ], Carry) => add_digit(X, Carry);

add_bin([A | X], [B | Y], C) =>
  Sum_Digit = (A+B+C) % 2,
  Carry = (A+B+C) / 2,
  [Sum_Digit | add_bin(X, Y, Carry)];

add_digit([ ], 0) => [ ];
add_digit([ ], 1) => [1];
add_digit([A | X], C) =>
  Sum_Digit = (A+C) % 2,
  Carry = (A+C) / 2,
  [Sum_Digit | add_digit(X, Carry)];
```

Adding two arbitrarily-long binary numerals, least-significant digit first.

The core of these rules is the function `add_bin` of three arguments, two sequences and a carry bit. If one of the sequences is empty, then `add_digit` takes over to finish off the addition. Otherwise there is a rule for each combination of first digits in each sequence and for the carry. These rules produce a digit of the resulting sequence, followed by a recursive call to `add_bin` with a new carry value.

Notice that we could, if desired, avoid using the `%` and `/` functions by enumerating the eight different possibilities of digits for `A`, `B`, and `C`.

Examples of other techniques that we will study that make use of, or rely on, the radix principle are:

Fast multiplication by the "Russian peasants' principle"

radix sort, described in *Complexity of Computing*

multiplexors, described in *Computing Logically*

barrel shifters (described in *Finite-State Computing*)

Fast Fourier Transform

The radix principle represents an idea that should not be overlooked when developing efficient algorithms. Here we will show the Russian peasants' principle. Suppose we wish to raise a number to an integer power N. The simple way to do this is to multiply the number by itself N times. This thus requires N multiplications. A more clever way to achieve the same end is to repeatedly square the base number, and selectively multiply some of the results to achieve a final product. Repeatedly squaring the original number gives us powers of two of that number. That is:

$N, N^2, (N^2)^2, ((N^2)^2)^2, \ldots$

is the same as

$N^1, N^2, N^4, N^8, \ldots$

To achieve an arbitrary power of N, say N^k, we can represent k in binary. Then we select the powers of N to powers of 2 according to the bits in the binary representation that are 1. For example, if k were 19, its binary representation would be 10011. The corresponding powers of N are then N^{16}, N^2, N^1. When we multiply these together, we get the desired product $N^{16+2+1} = N^{19}$.

Fortunately, we do not have to put these powers into a list. We can simply multiply in the ones we need as we decompose k into binary. The Russian peasants' approach, expressed in rex, would be:

```
power(X, 0) => 1;

power(X, Y) => even(Y) ? power(X*X, Y/2);

power(X, Y) => X * power(X*X, Y/2);
```

where function even tests whether its argument is even or odd. The binary decomposition of the second argument is taking place by dividing it by 2 on each recursion step.

A further example of the radix principle occurs after the following brief presentation of sorting techniques.

Insertion Sorting a List

Arranging a list so that the elements appear in increasing order is known as "sorting" the list. As explained above, in functional programming, the original list is not disturbed. Instead a new list is created that has the same elements in the appropriate order. There are dozens of ways to do it. Here are a few:

Function `insertion_sort` sorts a list by repeatedly inserting an element where it belongs:

To sort an empty list, return the empty list:

```
insertion_sort([ ]) => [ ];
```

To sort a non-empty list, `insertion_sort` all but the first element (using recursion), then insert the first element into its proper place:

```
insertion_sort([F | R]) => insert(F, insertion_sort(R));
```

Overall, recursion does most of the work in `insertion_sort`. However, we still need to define `insert`.

Inserting an element into its proper place in an empty list just gives the list with one element:

```
insert(A, [ ]) => [A];
```

To insert an element into a non-empty list, compare the element with the first element of that list. The resulting list starts with one or the other, and recursion takes care of inserting the other element in the remaining list:

```
insert(A, [B | X]) =>    // note: [B | X] is assumed to be ordered
  A < B ?
    [A, B | X]
  : [B | insert(A, X)];
```

This is a fine example of letting recursion do the work for you.

Selection Sorting a List

An example of a different sort is `selection_sort`which sorts a list by repeatedly selecting the minimum of the unsorted remaining elements and putting it next.

To sort an empty list, return the empty list

```
    selection_sort([ ]) => [ ];
```

To sort a non-empty list, an equational guard comes in handy. First get a list with the minimum as the first element, and the rest of the elements as the rest of that list. Call this list `[M | R]`. Return the minimum `M` followed by the result of sorting `R`, the rest of that list (letting recursion do that work):

```
    selection_sort(L) =>
        [M | R] = select_min(L),
        [M | selection_sort(R)];
```

Function `select_min` is designed to work only on *non-empty* lists L It brings the minimum of the list to the first position.

The minimum of a list of one element is at the first

```
    select_min([A]) => [A];
```

For a list with at least two elements, retain the first element and apply `select_min` to the remaining elements, then return a new list with the retained element and the first element of the result properly ordered:

```
    select_min([A | L]) =>
        [B | R] = select_min(L),
        (A < B ? [A, B | R] : [B, A | R]);
```

Merge Sorting a List

Merge sorting is another way to sort. We will show later that it has substantially fewer rewrite steps than either of the sorts introduced prior. By "merging", we mean operation of creating, from two sequences already in order, a longer sequence containing the elements of both sequences. This can be done easily by examining only the first elements of residual unmerged sequences and choosing the smaller one for output, until both sequences have been decimated.

Our implementation of function `merge_sort` works in the following way:

To sort:

> A non-empty list to be sorted is made into a list of 1-element lists. These lists are merged together a pair at a time using `merge_pairs`. This gives us lists of length at most 2. Then the process is repeated, merging pairs of those lists to get half as many lists that are twice as long. This is done in successive stages until only one list is left. That list is the sorted list.

To merge two lists:

> If the either list to be merged is empty, return the other list.
>
> Otherwise, compare the first elements of each list. Return a new list starting with the smaller element and followed by the result of merging the remaining elements.

The `merge_sort` function, expressed in rex, is given below:

First the initial list is transformed to a list of 1-element lists then those lists are merged repeatedly.

```
merge_sort(List) = repeat_merge( map((X) => [X], List ) );
```

Function `repeat_merge` merges pairs in a list of lists until there is only one list left.

```
repeat_merge([A]) => A;              // only one list left

repeat_merge(Lists) =>               // more than one list left
  repeat_merge( merge_pairs(Lists) );
```

Function `merge_pairs` merges pairs of lists in a list until none is left. It is similar to a `map` application, except that the function being mapped (`merge`) is called on successive pairs from a single list rather than on pairs from two different lists.

```
merge_pairs([ ]) => [ ];                  // no more lists

merge_pairs([A]) => [A];                  // only one list

merge_pairs([A, B | L]) => [merge(A, B) | merge_pairs(L)];
```

Function `merge` creates a single ordered list from two ordered lists.

```
merge(L, [ ]) => L;

merge([ ], M) => M;

merge([A | L], [B | M]) =>
```

```
    A <= B ? [A | merge(L, [B | M])] : [B | merge([A | L], M)];
```

Below is a coarse trace of `merge_sort` in operation:

```
merge_sort([5, 1, 2, 7, 0, 4, 3, 6]) ==>

repeat_merge([ [5], [1], [2], [7], [0], [4], [3], [6] ]) ==>

repeat_merge(merge_pairs([ [5], [1], [2], [7], [0], [4], [3], [6]
]))

repeat_merge([ [1, 5], [2, 7], [0, 4], [3, 6] ]) ==>

repeat_merge(merge_pairs([ [1, 5], [2, 7], [0, 4], [3, 6] ])) ==>

repeat_merge([ [1, 2, 5, 7], [0, 3, 4, 6] ]) ==>

repeat_merge(merge_pairs([ [1, 2, 5, 7], [0, 3, 4, 6] ])) ==>

repeat_merge([ [0, 1, 2, 3, 4, 5, 6, 7] ]) ==>

[0, 1, 2, 3, 4, 5, 6, 7]
```

Radix Sorting a List

We conclude the set of sorting examples with a method based on the radix principle. For this method, we assume that the numbers are non-negative integers. Sorting is based on comparing bits of the numbers, from lowest to highest. As splitting and regrouping is done for each bit, the numbers remain sorted on lower-order bits. Sorting is complete after the numbers are regrouped on the highest order bit.

```
// To sort, we sort based on the number of bits,
// from lowest order to highest

radix_sort(L) = radix_sort(0, numBits(L)-1, L);


// Sort on the Ith bit, then on the remaining bits

radix_sort(I, N, L) = I > N ? L : radix_sort(I+1, N, split(I,
L));


// split the list into two based on the Ith bit,
// then append the results

split(I, L) = append(drop((X)=>bit(I, X), L),
                     keep((X)=>bit(I, X), L));


// bit(I, X) gives the I-th bit of X

bit(I, X) = I == 0 ? X%2 : bit(I-1, X/2);
```

```
// find the maximum number of bits across all numeral in list

numBits(L) = ceilLog2(reduce(max, -Infinity, L));


// find the number of bits required to represent a numeral

ceilLog2(N) = N == 0 ? 0 : 1 + ceilLog2(N/2);
```

Further discussion of sorting methods appears in the chapter on Computational Complexity.

Exercises

Wherever possible, adhere to the recursion manifesto in the following:

1 • Give a set of rules for a function that computes the list of squares of each of a list of numbers. (This could be done with `map`, but do it from scratch instead.)

2 •• Give a set of rules for computing the sum of a list of numbers; for computing the product of a list of numbers. (This could be done with `reduce`, but do it from scratch instead.)

3 •• Using your function `days_of_month` constructed in an earlier exercise, give rules for the function `total_days` that takes as an argument a list of months and returns the sum of the days in those months.

4 •• Give a set of rules for computing the average of a list of numbers (use 0 for the average of an empty list).

5 •• Indicate two different ways to compute the sum of the squares of a list of numbers.

6 •• Give rules that define the function `flip`, that operates on lists, and exchanges successive pairs of elements. If there is an odd number of elements, the last element is left as is. For example:

```
flip([1, 2, 3, 4, 5, 6, 7]) ==> [2, 1, 4, 3, 6, 5, 7]
```

Suggestion: Use a rule that matches on the first two elements, rather than just one:

```
flip([A, B | L]) => … ;
```

7 ••• Give rules for the function `at_least` that tells whether a list has at least a certain number of elements. For example:

```
at_least(3, [1, 2, 3]) ==> 1

at_least(3, [1, 2]) ==> 0
```

Avoid counting all of the elements of the list. This is unnecessarily inefficient for large lists.

8 ••• Like the previous problem, except `at_most`.

9 •• The function `select` has the property of selecting the I^{th} element of a list, $I >= 0$, or returning the value of a special parameter `Other` if there is no such element (the list is not long enough). That is,

`select(I, [X0, X1, …, XN-1], Other)` ==> X_I if $I < N$

`select(I, [X0, X1, …, XN-1], Other)` ==> Other if $I >= N$

Give a set of rules for *select*.

10 •• The function `find_index` has the property of computing the index of the first occurrence of a given element within a list. If there is no such occurrence, -1 is returned. For example,

```
find_index('d', ['a', 'b', 'c', 'd', 'e']) ==> 3
find_index('a', ['a', 'b', 'c', 'd', 'e'] ==> 0
find_index('g', ['a', 'b', 'c', 'd', 'e']) ==> -1
```

Give a complete set of rules for `find_index`.

11 ••• Give rules for a function `remove_duplicates` that removes all duplicates in a list. For example

```
remove_duplicates([1, 2, 1, 3, 1, 2, 3]) ==> [1, 2, 3]
```

12 •• Give rules for a function that gives the value of a list representing the 2-adic representation of a number, least-significant digit first, using the digits 1 and 2.

13 ••• Give rules for a function that gives the list representation of a number in 2-adic form, least-significant digit first.

14 ••• Give rules for a function that produces the list of prime factors of a natural number. For example

```
factors(72) ==> [2, 2, 2, 3, 3]
```

15 •• Using functions above, give rules for a function that produces the unique prime factors of a natural number. For example

```
    unique_factors(72) ==> [2, 3]
```

16 •• Give rules for the function `subst` that makes substitutions in a list. Specifically, `subst(A, L, R)` returns a new list that is like list `L` except that whenever `A` would have occurred as a member of `L`, `R` occurs instead.

17 •• By adding an extra argument, and assuming integer functions `mod` and `div`, generalize the function `add_bin` to a function that adds in an arbitrary radix.

18 ••• Devise a function that will *multiply* two numbers represented as a list of bits, least-significant-bit first. Notice that this function has some advantage over the standard multiplication function found in most programming languages, namely that it will work for arbitrarily-large numbers.

19 ••• Sometimes we use numeral systems of mixed radix. For example, in referring to time within a given month, we could use expressions of the form D:H:M:S for days, hours, minutes, and seconds. H ranges from 0 to 24, M from 0 to 59, and S from 0 to 59. To compute the number of seconds from the start of the day corresponding to a given time, we'd compute:

```
    S + 60*(M + 60*(H+24*D)).
```

Generalize this mixed radix computation by giving rules for a function *value* that takes as arguments two lists, one giving the ranges and another giving the ordinal numbers within these ranges. For example, in the current case we would call

```
    value( [S, M, H, D], [1, 60, 60, 24] )
```

20 ••• Devise a function that will *divide* one number represented in binary by another, yielding a quotient and a remainder. This function should return the pair of two items as a list. Do the division digit-by-digit, don't convert to another form first.

21 •• The function `keep` takes two arguments: a predicate and a list: `keep(P, L)` is the list of those items in `L` such that `P` is true for the item. For example,

```
    keep(odd, [1,3,2,4,6,7]) ==> [1,3,7]
```

Provide rex rule definitions for `keep`.

22 •• The function `drop` is like function keep above, except that the items for which `P` is not true are kept. For example,

```
    drop(odd, [1,3,2,4,6,7]) ==> [2,4,6]
```

Provide rex rule definitions for `drop`.

23 •• The function `select` takes two arguments: a list of 0's and 1's and another list, usually of the same length. `select(S, L)` is the list of those items in `L` for which the corresponding element of `S` is 1. For example,

```
select([1,0,0,1,1,0], [1,3,2,4,6,7]) ==> [1,4,6]
```

Provide rex definitions for `select`.

24 •• *Iterated function systems* are used for producing so-called "fractal" images. These entail applying a function repeatedly to an initial seed argument. Let

$F^N(X)$

denote

`F(F(F…(F(X))…))`
N applications of `F`

including the definition:

$F^0(X) = X.$

Give rewrite rules for the function `iterate` informally defined by:

`iterate(N, F, X)` ==> $F^N(X)$

25 ••• Restate the definition of *Ackermann's function* using *iterate*.

26 ••• By **indefinite iteration** we mean iteration that stops when some condition is true, rather than by iterating a pre-determined number of times. The condition for stopping is best expressed as a predicate, say P. Give the rewrite rules for a function

```
iterate(P, F, X)
```

defined to compute

$F^{\mathbf{n}}(X)$

where **n** is the least value of N such that $P(F^N(X)) == 1$.

27 ••• Give the rules for a function that transposes a matrix represented as a list of lists. Your function can assume that each "row" of the matrix has the same number of elements without checking. You might, however, wish to construct a

separate function that checks the integrity of the matrix. Check your definition carefully and show that the types match up.

28 ••• Referring to the previous problem, if you understand matrix addition and multiplication, construct functions that carry out these operations.

29 •••• If you understand matrix inversion, construct a function that carries out this operation.

30 •• Show how to use enumeration to define the function `radix` without using the `%` and `/` functions.

4.10 Accumulator-Argument Definitions

Consider the problem of specifying a function that can reverse a list, for example:

```
reverse([1, 2, 3]) ==> [3, 2, 1]
```

The newcomer will typically try to approach this problem inductively by creating something like:

```
reverse( [ ] ) => [ ];                     // not recommended

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

While this pair of rules does achieve its purpose, it is clumsier than necessary when it comes to execution by rewriting. This clumsiness translates into taking much longer in execution. This particular rule set requires a number of rewrites proportional to $n^2/ 2$ to reverse a list of length n, whereas it is possible to do it in rewrites proportional to only n. Here's an illustration for a list of length 4:

```
   reverse([1, 2, 3, 4])
=> append(reverse([2, 3, 4]), [1])
=> append(append(reverse([3, 4], [2]), [1])
=> append(append(append(reverse([4]), [3]), [2]), [1])
=> append(append(append(append(reverse([ ]), [4]), [3]), [2]), [1])
=> append(append(append(append([ ], [4]), [3]), [2]), [1])
=> append(append(append([4], [3]), [2]), [1])
=> append(append([4 | append([ ], [3])], [2]), [1])
=> append(append([4 | [3]], [2]), [1])
=> append(append([4, 3], [2]), [1])
=> append([4 | append([3], [2])], [1])
=> append([4, 3 | append([ ], [2])], [1])
=> append([4, 3 | [2]], [1])
=> append([4, 3, 2], [1])
=> [4 | append([3, 2], [1])]
=> [4, 3 | append([2], [1])]
=> [4, 3, 2 | append([ ], [1])]
=> [4, 3, 2 | [1] ]
=> [4, 3, 2, 1]
```

This clumsiness can be avoided by using the technique of an *accumulator*. An accumulator is an “extra” argument that serves to accumulate the result. In the case of `reverse`, what is accumulated is a list that ends up being the answer. For the reverse function, the reversal of the list is accomplished by moving the elements from one list to another. They are thus accumulated in an order that is the reverse of the order on the original list. We use a two-argument function `reverse`, then define a one-argument version in terms of it. In the first rule, when the original list is empty, we return the accumulated last:

```
      reverse( [ ], R ) => R;
                    ↑      ↑
  accumulator argument     the accumulator is returned
```

In the second rule, when the list is non-empty, we continue with the rest of the list and accumulate the first of the list on an already-started list:

```
      reverse( [A | L], R ) => reverse( L, [A | R]);
                        ↑                  ↑
      accumulator argument                 the accumulator accumulates
```

Let us verify that this results in far fewer rewriting steps for the previous list example:

```
         reverse( [1, 2, 3, 4], [ ])
      => reverse( [2, 3, 4], [1])
      => reverse( [3, 4], [2, 1])
      => reverse( [4], [3, 2, 1])
      => reverse( [ ], [4, 3, 2, 1])
      => [4, 3, 2, 1]
```

In general, using the non-accumulator definition will require a number of steps that is about one-half the square of the number of steps in the accumulator definition. Thus using an accumulator provides a significant saving in computation time. We shall see how to perform such an analysis in more detail in the chapter on Complexity.

4.11 Interface vs. auxiliary functions

In order to make a one-argument reverse function, we may define it in terms of the two-argument version presented in the previous section. The function in terms of this one by specifying an additional argument:

```
      reverse(L) = reverse(L, [ ]);
```

We can give a description of what the two-argument `reverse` does: It appends the second list to the reverse of the first. This jibes with the rule above: appending `[ ]` to the reverse of the first list is exactly the reverse of the first list, since appending `[ ]` to anything gives that thing.

To simply reverse a list, the one-argument `reverse` function is what we should provide to the user. Thus it is called an *interface* function. The two-argument reverse is called the *auxiliary* or "helper" function. This is not to say that a user would never have need for the auxiliary itself, but this would only happen if she wanted to do a combination of reversal and appending, which seems to be a less frequent need.

In many cases, we can build reversal into our function definitions rather than call upon reverse after the fact. For example, it was natural to produce the digits of the radix representation of a number least-significant digit first. If we wanted them most-significant first instead, we could either call reverse on the result, or we could just design the function to handle it directly using an accumulator argument. Here's how it would look for the radix representation. Note that we use an interface function for two purposes: to handle the special case of argument 0, and to call the auxiliary function with `[ ]` as the accumulator value.

```
digits(0, Radix) => [0];
digits(N, Radix) => digits1(N, Radix, [ ]);

digits1(0, Radix, Tail) => Tail;
digits1(N, Radix, Tail) =>
    digits1(N / Radix, Radix, [N % Radix | Tail]);
```

Function `digits` *gives the digits of the first argument represented in the radix of the second, most-significant digit first.*

Notice that the third argument of `digits1` is an accumulator. As we divide the original number successively by `Radix`, we are determining digits of higher powers of the radix, that get tacked on to the left-end of the list. When the number is finally decimated (reduced to 0), in the basis for digits1, the accumulated list is returned.

4.12 Tail Recursion

The type of recursion displayed by reverse using an accumulator, where there are no further operations to be performed on the rewritten result, is called **tail-recursion**. Tail-recursive rules have the desirable property that they reduce storage overhead resulting from nested function calls.

Below we show the distinction using `nrev` to denote the "naive" first attempt at constructing the `reverse` function vs. `rev2` to show the version with an accumulator argument. In `rev2`, there is nothing else to be done when the right-hand side returns its result. This is tail-recursion.

```
nrev([ ]) => [ ];
nrev([A | L]) => append(nrev(L), [A]);
                 ↑
                 due to this call, this rule is not tail-recursive

reverse( L ) = rev2(L, [ ]);
rev2( [ ], M ) => M;
rev2( [A | L], M ) => rev2( L, [A | M] );
                      ↑
                      this rule is tail-recursive
```

comparative forms of list reversal, non-tail-recursive vs. tail-recursive

While tail-recursive rules are desirable for efficiency, they can be less readable unless one is on the lookout for them. Therefore it is sometimes a good idea to have a non-tail-recursive reference version of a function on hand if a tail-recursive version is being used.

Consider trying to give a tail-recursive formulation for *factorial*:

```
factorial(0) => 1;

factorial(N) => N * factorial(N-1);
```

As with the naive `reverse` example, there is a tendency to build-up unfinished work, in this case multiplies, outside the principal expression being rewritten:

```
factorial(4) ==> 4*factorial(3) ==> 4*3*factorial(2) ==> …
```

The unresolved multiplications represent work to which we will have to return when we finally get to use the first rule. How would we express this function using tail-recursion? As it turns out, we cannot do so with only one argument: We need to add an *accumulator* argument to carry the accumulated product and the original argument value as it diminishes. This can be accomplished by using an auxiliary function of two arguments and defining the interface function in terms of it:

```
factorial(N) = factorial(N, 1);

factorial(0, M) => M;

factorial(N, M) => factorial(N-1, N*M);
```

Here we have "overloaded" the name `factorial` to use it for two distinct functions, one with one argument and the other with two arguments. Now consider evaluating factorial(4):

```
factorial(4)     ==>
factorial(4, 1)  ==> factorial(3, 4*1)  ==>
factorial(3, 4)  ==> factorial(2, 3*4)  ==>
factorial(2, 12) ==> factorial(1, 2*12) ==>
factorial(1, 24) ==> factorial(0, 1*24) ==>
factorial(0, 24) ==> 24
```

While the tail-recursive factorial has a cleaner evaluation sequence, its rules are more complicated due to the introduction of a second function. These rules don't appear to be as natural as the original ones.

Exercises

1 ••• Construct an alternate set of rules for `reduce` that biases the reduction to the left, i.e.

```
reduce(H, Base, [X0, X1, …, XN-1])  ==>
      H(…H(H(Base, X0), X1), …, XN-1)
```

This function is often differentiated from the original one by calling this one `foldl` (fold-left) and the other `foldr` (fold-right).

2 •• The function `mappend` combines `map` and `append` in the following way. Suppose `f` is a function of one argument that returns a list for arguments drawn from a list `L`. Then `mappend(f, L)` is the list of the values of `f(A)`, for `A` in `L`, appended together.

For example, if `f(1) ==> [10, 11]`, `f(2) ==> [12, 13]`, and `f(3) ==> [14, 15]`, then

```
mappend(f, [1, 2, 3]) ==> [10, 11, 12, 13, 14, 15]
```

This is in contrast with `map`:

```
map(f, [1, 2, 3]) ==> [[10, 11], [12, 13], [14, 15]]
```

Give rules that define `mappend`.

3 •• Give another set of rules for the function `length` that computes the length of a list. This time, use an accumulator argument so that the rules are tail-recursive.

4 ••• Using an accumulator argument, but not using explicit list reversal, give rules for a function that converts from binary to a natural number when the binary is represented as a list *most significant digit first.*

5 ••• Give rules for the function `qsort` (abbreviation for "Quicksort") that sorts a list by the following recursive method:

If the list has one or no element, the result is just the list itself.

If the list has more than one element, use the first element to split the list into two: one list of elements less than the first element, and a list of the remaining elements. qsort each of those lists and append the results together so that the ordering is correct.

Once you have your function working, replace the use of `append` with an appropriate accumulator argument.

4.13 Using Lists to Implement Sets

It is common to use lists in the computer to represent sets. In order to represent a set, we disregard order of the elements. We must also ensure that there are no duplicate elements. The empty list [ ] is naturally used to represent the empty set. To add a new member to a set, we only need use the list constructor [|]. However, we must be sure that the member is not already present. The function `member` is such that `member(A, S)` ==> 1 if A is in the set and 0 otherwise.

```
member(_, [ ]) => 0;          // since the empty set can have no member

member(A, [A | S]) => 1;      // A is the first member in the list

member(A, [_ | S]) => member(A, S);

                              // A is not the first member, but could come later
```

To add a member known not to be in the set:

```
add_new_member(A, S) => [A | S];
```

To add a member in general, we use a guarded rule:

```
add_member(A, S) => member(A, S) ? S;   // already a member, no change

add_member(A, S) => add_new_member(A, S);
```

To form the union of two sets, we can use a process similar to `append`. However, we must take care not to duplicate any elements. We assume that there are no duplicates in either argument set.

```
union([ ], T) => T;

union([A | S], T) => add_member(A, union(S, T));
```

Power Set Example

The power set of a set S is the set of all of subsets of S. Suppose we wished to give rules for computing the power set (as a list) from a given set (list). For example,

```
subsets([a, b, c]) ==>
```

```
        [[ ], [a], [a, b], [a, c], [a, b, c], [b], [b, c], [c]]
```

The type of *power*, by the way, is $A^* \rightarrow A^{**}$, since it takes a list of arbitrary things and returns a list of lists of those things.

Below we have worked through the reasoning of this problem. Thinking inductively …

Basis: `subsets( [ ] )` is `[ [ ] ]` since the empty set has only one subset: itself. This gives the following rule:

```
      subsets( [ ] ) => [ [ ] ];
```

Induction: How can we get `subsets([A | L])` from `subsets(L)`?

For one thing, `subsets(L)` is *contained in* `subsets([A | L])`, i.e. `subsets([A | L])` will be something appended to `subsets(L)`:

```
      subsets( [ A | L ] ) => append(subsets(L), ???);
```

What is missing? `subsets(L)` are those subsets of `[A | L]` that don't contain `A`. We need the ones that do contain `A`. But these are just like `subsets(L)` except that `A` has been added to each set.

So for `???` above we can use

```
      add_to_each(A, subsets(L))
```

Now we have to define `add_to_each`. We can give rules for it alone, or we can recognize that `add_to_each` is just a "map" application:

```
      add_to_each(_, [ ]) => [ ];

      add_to_each(A, [E | S]) => [ [A | E] | add_to_each(A, S)];

      subsets( [ ] ) => [ [ ] ];

      subsets( [ A | L ] ) =>
            append( subsets(L), add_to_each(A, subsets(L)));
```

The first alternative eliminates the function `add_to_each` by using an anonymous function in conjunction with `map`. Noting that

```
      add_to_each(A, L) == map( (S) => [A | S], L);
```

we can replace the `add_to_each` expression with the `map` expression in the second rule for subsets.

A second alternative is to replace the multiple uses of `subsets(L)` with an equational guard, giving us a new second rule:

```
subsets( [ A | L ] ) =>
      SoL = subsets(L),
      append( SoL, map( (X) => [A | X], SoL));
```

Finally, we can get rid of the call to `append` by using a version of `map` called `map_tail` that has an accumulator argument:

```
subsets( [ ] ) => [ [ ] ];

subsets( [ A | L ] ) =>
      SoL = subsets(L),
      map_tail( (X) => [A | X], SoL, SoL);

map_tail( F, [ ], Acc ) => Acc;

map_tail(F, [A | X], Acc) => map_tail(F, X, [ F(A) | Acc ]);
```

Exercises

1 • Trace through the series of rewrites for `union( [1, 2, 3, 4], [2, 4, 5, 6])`.

2 •• Give a set of rules for finding the *intersection* of two sets.

3 •• The *difference* of two sets, `difference(S, T)`, is defined to be the elements that are in S but that are not in T. Give a set of rules for the function `difference`.

4 •• Give a set of rules for testing two sets for equality, recalling that the elements need not be listed in the same order. One possibility is to use the `difference` function defined above. What other ways are there?

5 •• Define rules for a function `includes` so that `includes(S, T) ==> 1` if set S includes set T and `includes(S, T) ==> 0` otherwise. [Hint: Use the function `difference`.]

6 ••• Show that definitions for set operations can be simplified if we assume that the elements of each list always occur in a specific order and we can test that order between any two elements.

7 •••• Earlier we derived a way to compute the set of all pairs of elements of two sets represented as lists. Extend this idea to a function that computes the set of all n-tuples from a *list* of n sets. Calling this function tuples, we would have

```
tuples( [ [1, 2], [3, 4, 5], [6, 7] ] ) ==>

[[1, 3, 6], [1, 3, 7], [1, 4, 6], [1, 4, 7], [1, 5, 6],
 [1, 5, 7], [2, 3, 6], [2, 3, 7], [2, 4, 6], [2, 4, 7],
 [2, 5, 6], [2, 5, 7]]
```

Note that:

```
tuples( [ ] ) ==> [ [ ] ]
```

since there is exactly one tuple of no elements, the empty tuple. Also,

```
tuples( [ [ ] ] ) ==> [ ]
```

since there are no tuples that contain an element of the empty set.

4.14 Searching Trees

Consider the problem of determining whether an element satisfying a given property occurs in a tree. The property could entail specifying the exact identity or it could specify some characteristic of the element. Both of these cases are subsumed by specifying a predicate `P` that is satisfied exactly by the elements of interest.

This type of problem arises routinely in computer science. For example, if the tree is a directory structure, we might want to search for a specific file or sub-directory name, or for a file with specific ownership or permission properties, or with a last-write date before a certain date.

Let us assume that our trees are specified as lists, with the root as the first element and its major sub-trees as the remaining elements. For example, the following tree would be specified as the list

[1, [2, [4], [5], [6]], [3, [7, [9]], [8]]]

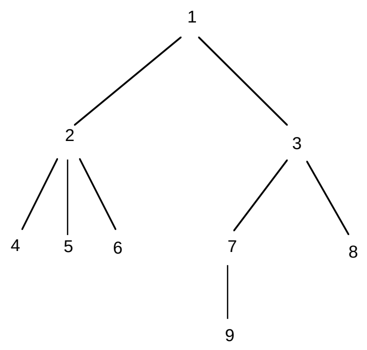


Figure 34: A tree for searching

The following simple recursive algorithm searches a tree for a node with property P:

To find a node with property P in a tree:

- If the root has property P, then return 1 (for success).
- Search the sub-trees of the root until one returns 1.
- If no sub-tree has returned success, then return 0 (for failure).

Let’s cast these ideas as rex rules:

```
find_df(P, [Root | Subtrees]) => P(Root) ? 1;

find_df(P, [Root | Subtrees]) => find_in_subtrees(P, Subtrees);


find_in_subtrees(P, [ ]) => 0;

find_in_subtrees(P, [Tree | Trees]) =>
    find_df(P, Tree) ?
          1
        : find_in_subtrees(P, Trees);
```

Depth-first search of a tree for a node with property `P`.

Here `find_in_subtrees` iterates over the sub-trees performing a search on each until either there is success or until there are no more trees.

This is an example of **mutual recursion**. There are two functions, `find` and `find_in_subtrees` and each calls the other. Each function has a different set of responsibilities: `find` checks the root of its tree, while `find_in_subtrees` checks each sub-tree. The latter is essentially an iterative process. Both functions are tail-recursive.

Often mutual recursion can be replaced with use of one or more higher-order functions. For example, the following definition using the function `some` is equivalent, but more succinct:

```
find_df(P, [Root | Subtrees]) => P(Root) ? 1;

find_df(P, [Root | Subtrees]) => some((T)=>find_df(P, T), Subtrees);
```

The type of search exhibited above is called **depth-first search**. If there are several nodes in the tree satisfying the property, the first one is detected is the one that is encountered on a trajectory that plunges downward before it traverses laterally. The pattern of depth-first search in this case is shown below:

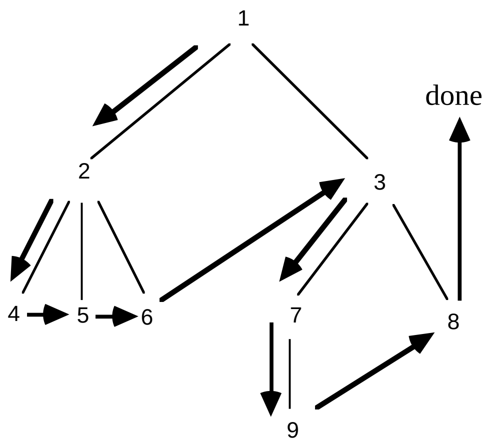


Figure 35: Depth-first search of a tree

A depth-first search establishes an ordering of the nodes, the order in which P is applied to test the node in the search. In this example, the ordering is [1, 2, 4, 5, 6, 3, 7, 9, 8]. This ordering is known as the **depth-first ordering**.

A complementary style of search is known as **breadth-first search**. Here the nodes are checked in order of increasing distance from the root. In order to accomplish this kind of search, we need to simulate a structure called a *queue*, which holds subtrees in the order the nodes are encountered until they can be revisited later. The algorithm is then as follows:

To find a node with property P in a tree, breadth-first:

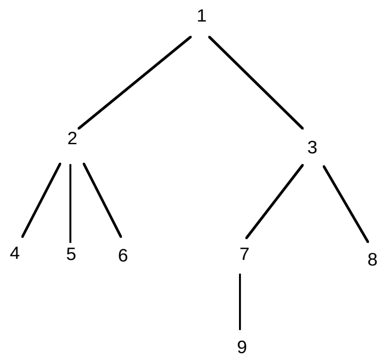
- Start with the tree as the only element in the queue.
- Repeat the following until success, or until queue is empty:
    - Consider the first tree in the queue. If the root of the tree satisfies P, then return success.
    - Add each of the sub-trees to the rear of the queue.
- (Queue is empty). Return failure.

Let’s make this algorithm more precise by presenting it in rex. Here we are using the same tree representation as before: The tree is represented as a list, with the first element of the list being the root and the rest of the list being the sub-trees.

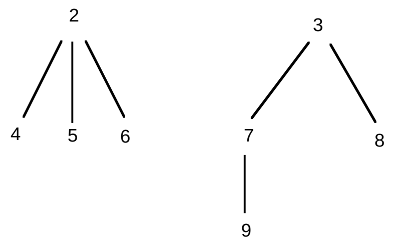
```
find_bf(P, Tree) => find_in_queue(P, [Tree]);

find_in_queue(P, [ ]) => 0;

find_in_queue (P, [[Root | Subtrees] | Trees]) =>
    P(Root) ?
      1
    : find_in_queue(P, append(Trees, Subtrees));
```

Breadth-first search of a tree for a node with property `P.`

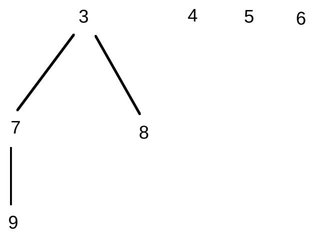
The following is a simulation of this algorithm, using the previous example tree. Suppose we are searching for a node equal to 7. Since the queue is a sequence of trees, we show that sequence at each stage. The initial queue is a sequence of one tree, the original tree:



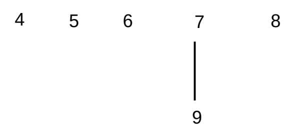
The root 1 is not equal to 7. The queue becomes the sequence of two trees:



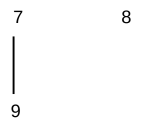
The first tree in the queue is the one with root 2, which is not equal to 7, so its sub-trees are appended to the end of the queue, resulting in:



The first tree in the queue is now the one with root 3, which is not equal to 7. So its sub-trees are added to the end of the queue, resulting in the queue:



The next three trees to be examined have roots not equal to 7 and the sub-trees are empty, so the queue becomes, in three steps:

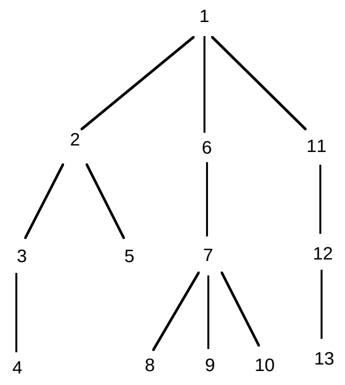


At this point, the root of the first sub-tree is equal to 7, so we stop with success.

As with depth-first search, breadth-first search also induces an ordering of the nodes, called the **breadth-first ordering**. This ordering is exactly what we would see if we read off the tree level by level, left-to-right. In the present example, this ordering is: [1, 2, 3, 4, 5, 6, 7, 8, 9].

Exercises

1 •• Consider the following tree. What are the depth-first and breadth-first numberings?



2 •• In the preceding tree, suppose `P(n)` is the property “n is a prime number greater than 5”. What node would be found in a depth-first search? in a breadth-first search?

3 ••• Modify the depth-first search algorithm so that, rather than merely returning success, the algorithm returns a list representing the path from the root to the node found. For example, if we were searching the tree above for node 10, the list returned would be `[1, 6, 7, 10]`.

4 ••• Repeat the preceding problem for the breadth-first search algorithm.

5 ••• A *tree address* is a sequence of numbers indicating a path from the root of the tree to one of its nodes:

> The root of the tree has address `[ ]`.
>
> If the node occurs in the i th subtree, the tree address of the node is i followed by the tree address of the node relative to the subtree.

We'll use the convention of numbering the subtrees starting at 0. For example, in the diagram above, the tree address of node 10 is

```
[1, 0, 2]
```

since the node occurs as root of subtree 2 of subtree 0 of subtree 1 of the overall tree.

Modify the depth-first search algorithm so that it returns the tree address of the node it finds, if any.

6 ••• Repeat the preceding problem for the breadth-first search algorithm.

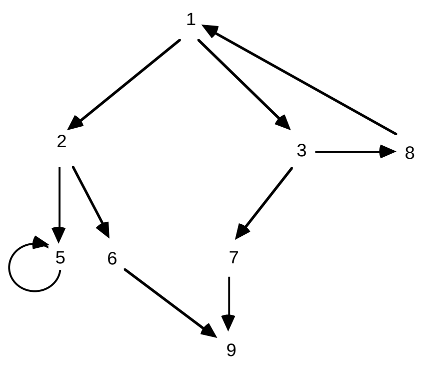
7 ••• Define in rex a function that will return the node in a tree given its tree address. It is possible that there is no node corresponding to a given address. Handle this possibility by returning a list of the node in the case the node is found, and the empty list in case of failure.

4.15 Searching Graphs

Searching a directed graph depth-first is similar to searching a tree, except that there are additional complications arising from the possibility the graph is not necessarily a tree. These are:

> A given node may be the target of more than one other node (this is sometimes called “fan-in”). We do not want to search from this node more than once. Thus we need some way of remembering that we've seen it before.
>
> A node may be in a cycle. Even though the node might be a target of just one node, unless we remember whether we've seen the node before, we could cycle forever and the search would not terminate.

The following graph is obtained by a slight modification of the previous tree. We see that fan-in occurs at nodes 5 and 9. A cycle occurs among nodes 1, 3, and 8, and also node 5 by itself.



We see that both non-tree phenomena can be handled by a common technique: refuse to search from a node from that we've already searched.

For the present, we will modify the search algorithms to keep a list of nodes that have been encountered. Loosely speaking, we check that list before searching the node a second time. There are other ways of doing this that do not involve scanning the list, but we will save them for an appropriate time later.

Another issue to be dealt with is that a general graph does not admit the representation we have been using for trees. Thus we have to use a different representation for general graphs. The one we will use now, for sake of concreteness, was introduced in *Information Structures*:

> A graph is a list. Each element is a list consisting of the name of a node followed by the targets of that node.

The list representation for the preceding graph would thus be:

```
[ [1, 2, 3],
  [2, 5, 6],
  [3, 7, 8],
  [5, 5],
  [6, 9],
  [7, 9],
  [8, 1],
  [9] ]
```

Despite this assumption, we shall try to cast the search algorithms to be relatively free of the assumption itself. We will do this by creating a function

```
get_targets(Node, Graph)
```

that, given a node and the graph, will return the list (possibly empty) of targets of the node. Only this function needs to know how the graph is represented. So if we change the representation, this is all we will need to change, not the search algorithm itself. The algorithm consists of the following rules:

Consider defining a depth-first search. The first rule is an interface function: `find_dfg(P, Node, Graph)` tries to find a node satisfying `P` in the graph, that is reachable from node `Node`.

```
find_dfg(P, Node, Graph) => find_dfg(P, [Node], Graph, [ ]);
```

The interface function calls the auxiliary function, with the set of “seen” nodes empty. If the set of nodes remaining to be searched is empty, then failure is reported.

```
find_dfg(P, [ ], Graph, Seen) => 0;
```

If there is at least one remaining node and the first node satisfies P, then success is reported.

```
find_dfg(P, [Node | Nodes], Graph, Seen) => P(Node) ? "1";
```

If the first node does not satisfy `P`, then we get the targets of the node. From those targets, we remove any that have been seen already. We add the remainder to the front of the list of nodes and continue the search, with the first node now noted as having been seen.

```
find_dfg(P, [Node | Nodes], Graph, Seen) =>
  Targets = get_targets(Node, Graph),
  New = difference(Targets, Seen),
  find_dfg(P, append(New, Nodes), Graph, [Node | Seen]);
```

For the particular graph representation described, function `get_targets` can be expressed using the built-in rex function `assoc`. Recall that this function searches a list of lists for a designated first component. If it finds one, in returns the list having that component. Otherwise it returns the empty list.

```
get_targets(Node, Graph) =>
  Found = assoc(Node, Graph),
  Found == [ ] ? [ ] : rest(Found);
```

A simple version of `difference` is as follows:

```
difference([ ], B) => [ ];

difference([A | As], B) =>
  member(A, B) ?
    difference(As, B)
  : [A | difference(As, B)];
```

For breadth-first search of a graph, we only need modify the `find` rule by changing the order of arguments to `append`:

```
find_bfg(P, [ ], Graph, Seen) => 0;

find_bfg(P, [Node | Nodes], Graph, Seen) => P(Node) ? "1";

find_bfg(P, [Node | Nodes], Graph, Seen) =>
```

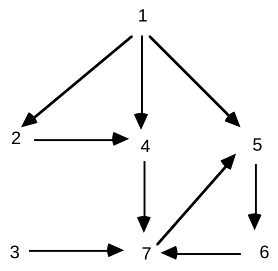
```
        Targets = get_targets(Node, Graph),
        New = difference(Targets, Seen),
        find_bfg(P, append(Nodes, New), Graph, [Node | Seen]);
```

The effect is to put new targets behind the current queue of targets.

Breadth-first searching a graph finds the node *nearest* to the starting node, in terms of the number of arrows that need traversing. A variant of it is used in shortest-path or least-cost path problems. We shall discuss this further in a later section. The concepts of breadth-first numbering and depth-first numbering apply to graphs as well as trees, except that a specific starting node must be specified. Also, the numbering depends on the order in which the targets of each node are listed.

Exercises

1 •• Consider the following graph. What are valid depth-first and breadth-first numberings relative to node 1 as a starting node?



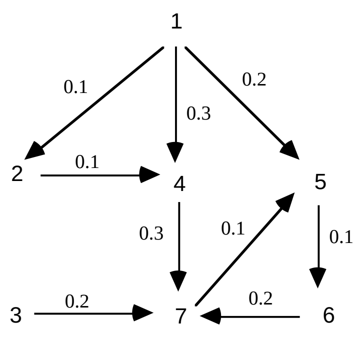
2 •• In the preceding graph, suppose that P(n) is the property "n is a prime number greater than 3". What node would be found in a breadth-first search?

3 ••• Modify the depth-first search algorithm so that, rather than merely returning success, the algorithm returns a list representing the path from the root to the node found. For example, if we were searching the graph above for node 6, the list returned might be `[1, 4, 7, 5, 6]`.

4 ••• Repeat the preceding problem for the breadth-first search algorithm.

5 ••• A third form of search is known as *iterative deepening*. As with breadth-first search, it finds nodes in order of increasing distance from the root, but it does not require storage for a queue. It is effectively a series of depth-first searches with increasing depth bounds. Construct a function for performing this form of search.

6 •••• Consider the following modification of breadth-first search: The arcs on a directed graph each have a positive numeric cost (representing, say, distance or travel time) associated with them. Devise an algorithm that, given a node, called the *source* node, computes the least-cost path between this node and all nodes. The cost of a path is defined to be the sum of the costs on the arcs in the path.



The graph in this case can be represented as follows:

```
[ [1, [0.1, 2], [0.2, 5], [0.3, 4]],
  [2, [0.1, 4]],
  [3, [0.2, 7]],
  [4, [0.3, 7]],
  [5, [0.1, 6]],
  [6, [0.2, 7]],
  [7, [0.1, 5]] ]
```

The result of the algorithm would be a list of [Cost, Node] pairs. For source node 1 this would be:

```
[[0, 1], [0.1, 2], [0.2, 4], [0.2, 5], [0.3, 6], [0.5, 7],
[Infinity, 3]]
```

4.16 Argument Evaluation Disciplines

It is often the case that there is more than one sub-expression to which rules can be applied to a term. For example, consider a rule set for `add`:

```
add(0, M) => M;
add(N+1, M) => add(N, M)+1;
```

Suppose we want to evaluate the term

```
add(0, add(0, 5))
```

Here we could apply the rule for `add(0, M)` to the outer term to get

```
add(0, 5)
```

or to the inner term `add(0, 5)`, to get the same thing. However, we will not *always* rewrite to the same thing immediately. To see this, consider a term of the form

```
add(N+1, add(K+1, M))
```

Applying a rule to the outer term gives us

```
add(N, add(K+1, M)) + 1
```

while applying to the inner term gives

```
add(N+1, add(K, M) + 1)
```

These two are obviously different, although by another rule application, we could convert both of them to

```
add(N, add(K, M) + 1) +1
```

Applicative Order

Most programming languages adopt a specific discipline about where a rule will be applied when there is a choice. The most common discipline is known as

applicative-order argument evaluation:

Give priority to applying a rule to an *argument* of a term before applying a rule to the entire term.

Example: In `f(g(0), h(1))`, apply a rule for `g` or `h` before applying any rule for `f`.

Even this is not without ambiguity, however, since there could be several arguments to which rules apply. By **leftmost applicative-order,** we give priority to the leftmost argument term first, and similarly for *rightmost applicative-order*.

Examples

In `add(N+1, add(K+1, M))`, we have an argument `add(K+1, M)` to which a rule is applicable. Moreover, this is the leftmost such argument, so the rewritten term under leftmost applicative order is `add(N+1, add(K, M)+1)`.

In `add(N+1, add(K+1, add(M+1, 2)))`, under leftmost applicative order, a rule is applicable to the second argument, `add(K+1, add(M+1, 2))`. However, this argument also has an argument to which a rule is applicable, so we must give priority to that argument rather than the outer argument. Thus, the rewritten term would be `add(N+1, add(K+1, add(M, 2)+1))`.

Normal Order

Another evaluation-order discipline with important uses is known as

normal order argument evaluation:

Give priority of applying a rule to the *entire* term over applying a rule to an argument of the term.

Example: In `f(g(0), h(1))`, apply a rule for *f* before applying any rule for *g* or *h*.

Examples

Under normal order we would have the following series of rewrites:

```
add(N+1, add(K+1, add(M+1, 2))) ==>
add(N, add(K+1, add(M+1, 2)))+1 ==>
add(N, add(K, add(M+1, 2))+1)+1 ==>
add(N, add(K, add(M, 2)+1)+1)+1
```

Contrast this with applicative order, where the rewrite series would be:

```
add(N+1, add(K+1, add(M+1, 2))) ==>
add(N+1, add(K+1, add(M, 2)+1)) ==>
add(N+1, add(K, add(M, 2)+1)+1) ==>
add(N, add(K, add(M, 2)+1)+1)+1
```

The end results are the same, but the intermediate details differ.

Even though applicative order is the most common, normal order has the advantage of terminating in some cases where applicative order does not. As an example, consider the following rules:

```
if(1, A, B) => A;
if(0, A, B) => B;

foo(N) => foo(N+1);
```

Consider the term

```
if(0, foo(0), 1)
```

Applicative order would require that `foo(0)` be evaluated before using the definition of `if`. However, `foo(0)` will never converge. Therefore applicative order will never use the definition of *if* and the entire computation *diverges*. On the other hand, with normal order, the second rule for if will be used immediately and there will be no call for the evaluation of `foo(0)`.

It can be shown that normal order is strictly more general, in the sense that there will never be a case where applicative order gives an answer but normal order fails to give one. Unfortunately, applicative order is the norm in most programming languages, not normal order. One might remember this by the following quip:

Normal order isn't.

The reason that normal order is not "normal" has to do with normal order being more complicated to implement, not that it is undesirable for mathematical reasons.

Normal Order in rex and Delayed Evaluation

As with most languages, the rex evaluator uses applicative order for all functions, except for a few "special forms" such as the conditional form __ `?` __ `:` __ , logical conjunction `&&`, and logical disjunction `||`, that evaluate arguments left to right as they need them.

It is possible to achieve a custom normal order effect through what we will call the defer operator. Any expression, including an argument to a function, can be "wrapped" as the argument to an operator `$` (read "defer"). The expression is not evaluated until it is absolutely necessary. Thus, if we have an actual argument wrapped in $:

```
    h($f(X, Y), Z)
```

this argument will effectively be treated as if a normal-order argument, while others will be treated as applicative order. Only when, if ever, it becomes necessary for `h` to know the value of `f(X, Y)` will the latter be evaluated. For example, in a conditional expression

```
    p(X) ? $f(X, Y) : g(Y, Z)
```

even if `p(X)` evaluates to 1, we do not need to know the value of `f(X, Y)`. The value of this expression is just `$f(X, Y)`. If, on the other hand, we used `f(X, Y)` in a numeric expression, such as

```
    Z + $f(X, Y)
```

it becomes necessary to know what the value of `$f(X, Y)` is. At this point the expression would be evaluated.

One of the key uses of `$` in rex will be explained in the section *Infinite Lists*.

Using Function Arguments to Achieve Delay

A traditional device for achieving the effect of delaying the evaluation of an argument expression (i.e. the *defer* operator, as discussed with normal order evaluation) is to embed the expression in question into the body of an additional function with no arguments. Then, when we want to evaluate this expression, we apply the function (to no arguments). For example, suppose that the expression we want to delay is

```
X + g(X, Y)
```

To pass this expression unevaluated, we actually pass the 0-argument function

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

Suppose that this function is bound to a variable `D`. Then when we want the evaluation of the original expression to take place, we would compute

```
D()
```

(`D` applied to no arguments). This scheme differs slightly from the *defer* scheme in rex. In the scheme being discussed, the program must know to expect the 0-argument function and arrange for its application (to no arguments). Thus a function cannot be written that will take *either* a delayed argument or an ordinary argument, unless some sort of *tag* is also passed along to indicate which.

4.17 Infinite Lists (Advanced)

This topic describes a programming paradigm that is available in very few languages (rex is one, of course!). It can be "engineered" in others, but sometimes with great difficulty. However, due to the substantial power that this approach provides, it will likely be in many high-level languages at some point in the future (how distant we hesitate to speculate).

The rewriting approach provides an ideal way to describe and implement an unusual feature that is available in some languages: the ability to manipulate lists as if they were infinite. This requires delaying the computation of the tail of the list, as in `[ A | $ L ]`, until the tail is needed. Otherwise an attempt would be made to evaluate the tail, resulting in divergence.

The List of All Natural Numbers

The simplest non-trivial example of an infinite list is the list of all natural numbers, conceptually shown as

```
[0, 1, 2, 3, …]
```

We want *from* to be a function that, with argument `N`, will generate the infinite list of numbers from `N` on. The list above is the special case `from(0)`. The definition of `from` must use a normal-order list constructor. As discussed earlier, this can be achieved by the delay wrapper `$`, as in:

```
from(N) => [ N | $ from(N+1)];
```

The idea here is that the recursive call to `from(N+1)` is not evaluated until needed. So if we are showing the result of `from(0)`, we would do these evaluations as it comes time to print each successive element. Let us check this by giving a few rewrites of `from(0)`:

```
from(0) ==>
[ 0 | $ from(1) ] ==>
[ 0 | [ 1 | $ from(2) ] ] ==>
[ 0 | [ 1 | [ 2 | $ from(3) ] ] ] ==>
[ 0 | [ 1 | [ 2 | [ 3 | $ from(4) ] ] ] ]
```

which is the same as

```
[0, 1, 2, 3 | $ from(4) ]
```

When applying a rule for a function that has such a list as an argument, the usual rules apply: a formal argument `[A | L]` matches the actual argument so that `A` is the first element of the infinite list and `L` is the rest of the infinite list. For example, define functions *first* and *rest* by

```
first( [A | L] ) => A;

rest( [A | L] ) => L;
```

Then `rest` would force the evaluation of the delayed expression as necessary:

```
rest(from(0)) ==> rest( [0 | $ from(1)] ) ==> $ from(1)
```

If the result of *rest* were used, e.g. in evaluating

```
5 + first(rest(from(0)))
```

then `$from(1)` would be further expanded to get `[1 | $ from(2)]` and `first` would extract the 1, rewriting to 5 +1, then to 6.

Using this idea, we can construct functions that have infinite lists as arguments and results. For example, the function partial_sums produces a list of the sum of the first, first two, first three, and so on, elements of its argument:

```
partial_sums( [1, 3, 5, 7, …] ) ==>
                                        [1, 4, 9, 16, …]
```

The rules are, using an auxiliary function `partial_sums2`:

```
partial_sums(X) => partial_sums2(0, X);   // 0 is initial accumulator

partial_sums2(Acc, [ ]) => Acc;

partial_sums2(Acc, [A | X]) => [(Acc + A) | $ partial_sums2(Acc+A, X)];
```

Unzipping an Infinite List

The following function "unzips" a finite or infinite list into two lists.

```
unzip(X) = [evens(X), evens(rest(X))];

evens([ ]) => [ ];
evens([A]) => [A];
evens([A, _ | X]) => [A |$ evens(X)];
```

Unzipping a list

Pipe Composition

A very attractive aspect of functions on infinite lists is that they can be composed as with pipe composition discussed earlier. An example of pipe composition for infinite lists occurs in the next example, and is previewed here.

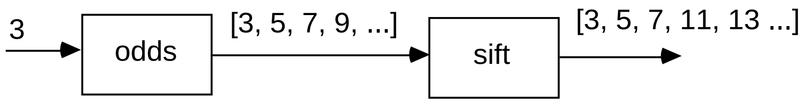


Figure 36: Piping an infinite stream through a function

This type of composition gives infinite lists value for certain computing applications, such as digital signal processing, where the application is typically structured as a set of interconnected stream-processing functions: integrators, filters, scalars, and the like.

Prime Number Sieve

The function `primes` below produces the infinite list of prime numbers beginning with 3. It does this using the technique of "sieving". Consider the infinite list of odd numbers:

```
3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, …
```

From this list, drop all those, other than the first (i.e. 3), that are multiples of the first:

```
3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, …
```

Now do the same for the *next* number that is left (5), i.e. drop all multiples of it:

```
3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …
```

Continue this way, dropping multiples of 7, then 11, … The numbers that survive the drops are the primes. At every major step, the first number survives, so this insures that every prime will eventually be produced.

The program is:

```
primes() = sift(odds(3));

odds(N) = [N | $ odds(N+2)];

sift([A | X]) => [A | $ drop((X) => divides(A, X), sift(X))];
```

Function `primes` *generates the infinite list of primes*.

To gain maximum utility from this paradigm, it is helpful to be able to compose programs with loops, as will be discussed in the next section.

Functional Programs with Loops

Another technique that can be used to generate infinite lists is to have "loops" in the defining equations. For example, the following equation:

```
Ones = [ 1 |$ Ones ];
```

defines `Ones` to be the infinite list [1, 1, 1, 1, …]. The figure below shows how this works, by piping the output back into the `|` (followed-by) function.

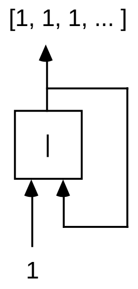


Figure 37: A simple functional program with a loop.

Here | represents the "followed-by" function used to construct lists.

Example - Another way to get the partial sums of an infinite sequence X is to use:

```
Psums = map(+, X, [0 | $ Psums]);
```

Here + is applied to two sequences, so this is the `map` form of +, rather than the simple arithmetic form. The definition of `Psums` has a "loop" in the sense that the definition itself uses the quantity `Psums`. The two programs with loops for `Ones` and `Psums` can be shown as follows:

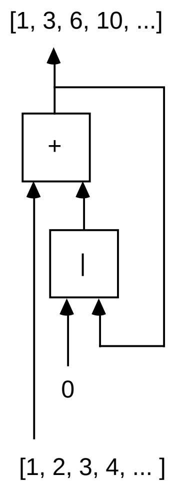


Figure 38 A functional program with a loop showing result for an example input.

Here + represents `map(+, . , .)`

Here is an example of how this works in rex:

```
rex > X = from(1);
1

rex > X;
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, …

rex > Psums = map(+, X, [0 | $ Psums]);
1

rex > Psums;
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, …
```

Fibonacci sequence using a recursive group

The examples above both defined fixed infinite sequences using loops. If we want to define *functions* using loops, we need something like an equational guard, yet slightly different. Consider the following attempt to define the function fib that generates the Fibonacci sequence:

```
fib() = Result = [1, 1 | $ map(+, Result, rest(Result)) ],
        Result;
```

This definition first defines the quantity represented by variable `Result` using an equational guard, then gives that value as the result of the function. Syntactically this definition is well-formed. However, the value of `Result` used on the right-hand side of the equation is *not* the same as the one on the left; the value is, by definition, the value of `Result` in the ambient environment (it may or may not be defined). What we want is an environment where both uses of `Result` mean the same thing. We had this in the global environment in earlier examples. But how do we get it inside the function `fib`? The answer is that we need a special construct called a *recursive group* that creates a recursive environment. In rex this is shown by giving a series of equations inside braces {…}. Each variable defined in that environment has the same meaning on the left- and right-hand sides of the equations. The last thing inside the braces is an expression, the value of which is the value of the group. The correct version of `fib()` is as follows:

```
fib() = { Result = [1, 1 | $ map(+, Result, rest(Result)) ];
          Result};
```

Here the first equation defines the variable `Result` to be a list starting with [1, 1, …]. The rest of the list is the pairwise sum of the list itself with the rest of the `Result` , [1, …]. Thus the first element in this sum is 2, the next element is therefore 1+2 ==> 3, the next 2+3 ==> 5, and so on. A rex dialog show this:

```
rex > fib();
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …
```

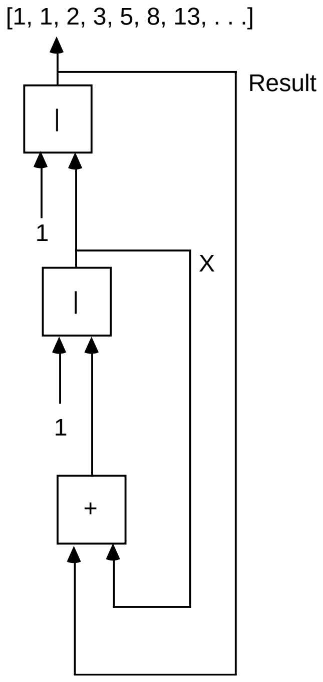


Figure 39: A functional program generating the Fibonacci sequence.

The use of `rest` in the program was eliminated by using the fact that `rest([A | X]) == X`

A definition that conforms directly to the diagram, using an additional variable `X`, is:

```
fib() = { Result = [1 | $ X];
          X = [1 | $ map(+,Result, X)];
          Result};
```

Simulating Differential Equations

Differential equations are equations that are to be solved for functions, rather than numbers, as unknowns. These equations are constructed using differential, as well as algebraic, operators. A typical type of differential equation involves real-valued functions of one variable. Often that variable is identified as a time parameter. We can simulate such equations by using a discrete approximation to time. In this case, a function of time can be represented by the sequence of values sampled at discrete time instants. With this in mind, it is possible to use our infinite lists as these sequences, i.e. to represent real-valued functions of time.

As an example, the derivative operator can be simulated by taking differences of adjacent argument values. The definition is:

```
deriv([A, B | X]) = [(B - A) | $ deriv([B | X]) ];
```

As it turns out, however, we do not use this operator directly in the solution method to be presented.

The usual algebraic operators +, -, *, etc. have to be mapped as pairwise operators on infinite lists. Thus to add the values of two "functions" `F` and `G` represented as sequences, we would use `map(+, F, G).`

First-Order Equation

Suppose that we wish to solve a first-order (meaning that the highest-order derivative is 1) linear homogenous (meaning that the *rhs* is 0) equation:

$$\frac{dX}{dt} + a*X(t) = 0$$

subject to an initial value X(0) = X0. A solution entails finding a function X that satisfies this equation. We will represent the function X by a sequence. The sequence corresponds to the values of the true function X at points 0, 0 + dt, 0 + 2dt, …, treating dt as if it were an actual interval. This interval is known as the “step size”. It will become implicit in our solution method. Solving the equation for dX:

$$dX = -a*X(t)*dt$$

But also

dX = X(t+dt) - X(t)

Combining these two and solving for X(t+dt):

X(t+dt) = X(t) - a*X(t)*dt

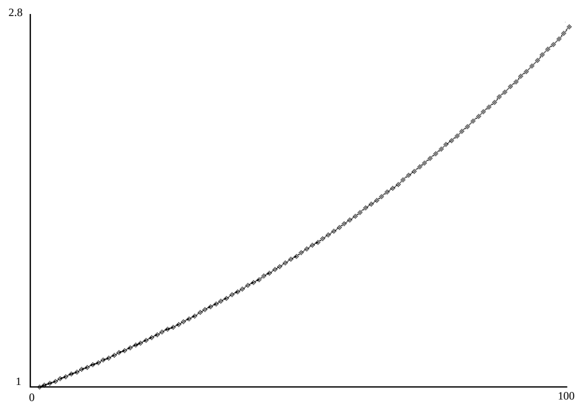
Taking dt to be 1, we have

X(t+1) = X(t) - a*X(t)

Now we have the approximation X(t+1) expressed in terms of X(t). Combining that with the known initial value `X0`, we can write in rex:

```
X = [X0 |$ map(-, X, scale(a, X))];
```

(Note that this equation has a “loop”.) As before, we are using the map version of operator - that works on two sequences pairwise. For a given values of `X0` and `a`, the sequence `X` is thus determined. For example, the following figure shows the points in sequence `X` when `X0`== 1 and `a` == -0.01.



Graph of the solution to a first-order differential equation.

Analytically, we know that the solution to the equation is X(t) = $e^{0.01t}$, which jibes with the numerical solution obtained; at t = 100, we have X(100) == 2.70481, which is approximately equal to e == 2.71828.

The solution method represented above is effectively what is called **Euler’s method**. It is not the most accurate method, but it is believed that the same solution technique using infinite lists can also be applied to more refined methods, such as the Runge-Kutta method.

Second-Order Equation

To show that the method presented above is general, we apply it to a second-order equation, of general form:

$$\frac{d^2X}{dt^2} + a*\frac{dX}{dt} + b*X(t) = 0$$

Where initial values are given for both X and $\frac{dX}{dt}$. It is common to introduce a second variable Y to represent $\frac{dX}{dt}$, transforming the original single equation to a system of equations:

$$\frac{dY}{dt} + a*Y(t)+ b*X(t) = 0$$

$$Y(t) = \frac{dX}{dt}$$

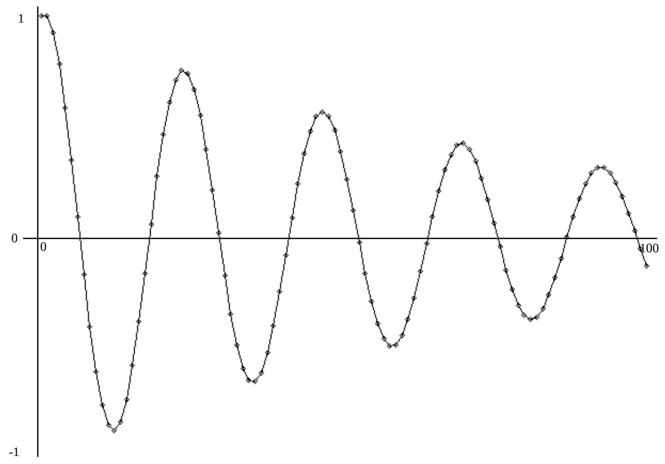
As before, we treat dt as if it were a discrete interval. As before, we solve for dX and dY, and equate these to X(t+1) - X(t) and Y(t+1) - Y(t) respectively. This gives:

```
X(t+1) = X(t) + Y(t)*dt

Y(t+1) = (1 - a)*Y(t) - b*X(t)
```

Translating into rex, using infinite lists:

```
X = [X0 | $ map(+, X, Y)];

Y = [Y0 | $ map(-, scale((1-a), Y), scale(b, X))];
```

Here when a scalar is multiplied by a sequence, the result is that of multiplying each element of the sequence by the scalar. The diagram below shows the first 100 values of `X` when `a` == 0.1, `b` == 0.075, `X0` == 1, and `Y0` == 0.



Graph of the solution to a second-order differential equation.

Exercises

1 • Trace the first few rewrites of `partial_sums(from(0))` to verify that the partial sums of the integers are [0, 1, 3, 6, 10, 15, …]

2 • Give rewrite rules for a function `odds` such that

```
odds(1) ==> [1, 3, 5, 7, 9, …]
```

3 •• Certain sets of rules on lists also make sense on infinite lists. An example is `map`, as introduced earlier. For example,

```
map(square, odds(1)) == [1, 9, 25, 49, 81, …]
```

Review the previous examples we have presented to determine which do and which don't make sense for infinite lists. Indicate where $ needs to be introduced to make the definitions effective.

4 •• Give rules for a function that takes a function, say *f*, as an argument, and produces the infinite sequence of values

```
[f(0), f(1), f(2), f(3), …]
```

5 ••• Give rules for a function that take a function, say *f*, and an argument to f, say x, as arguments, and produces the sequence of values

```
[f0(x), f1(x), f2(x), f3(x), …]
```

where $f^i(x)$ means f(f(…f(x)…)) (i times).

6 •• Suppose we use infinite lists to represent the coefficients of Taylor's series. That is, $a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots$ is represented by the infinite list $[a_0, a_1, a_2, a_3, \dots]$. Present rex functions that add two series and that multiply them by a constant.

7 ••• Continuing the above representation of series, construct a rex function that multiplies two series. The corresponding operation on infinite lists is called the *convolution* of the lists.

8 •••• Continuing the above thread, construct a rex function that derives the coefficients of

$$\frac{1}{1 - s}$$

where *s* is a series.

9 •••• Derive rex functions that generate the series of coefficients for your favorite analytic functions (*exp*, *sin*, *cos*, *sinh*, etc.).

10 ••• ["Hamming's problem"] Develop a function that generates, in order, the infinite list of numbers of the form $2^i3^j5^k$, where i, j, and k are natural numbers, i.e.

```
[2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, …]
```

11 •••• Referring to the earlier problem regarding transposing a matrix, construct a function that will transpose an *infinite* matrix, represented as an infinite list of infinite lists. For example, if the function's argument is:

```
[[0, 1, 3, 6, 10, …], [2, 4, 7, 11, …], [5, 8, 12, …],
 [9, 13, …], [14, …], …]
```

the transpose is

```
[[0, 2, 5, 9, 14, …], [1, 4, 8, 13, …], [3, 7, 12, …],
 [6, 11, …], [10, …], …]
```

12 ••• Referring to the previous problem, construct a function that will linearize an *infinite* matrix by "zig-zagging" through it. For example, zig-zagging through the first matrix above would give us:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, …]
```

13 ••• Construct a function that is the inverse of the zig-zag function in the previous problem.

14 ••• Define a version of 'pairs' that will work in the case that either or both argument lists are infinite, e.g.

```
pairs(from(0), from(0)) ==>

    [[0, 0], [1, 0], [0, 1], [2, 0], [0, 2], [1, 1], [0, 3], …
    ]
```

Thus, such a definition demonstrates the mathematical result that the Cartesian product of a countable set is countable.

15 ••• Derive solutions for cases in which the right-hand side of the above equations are replaced by “forcing functions” of t, which in turn are represented as sequences.

16 ••• Derive solutions for cases in which the coefficients of the equation are functions of t rather than constants..

17 •••• Explore the adaptation of more refined solution methods, such as *Runge-Kutta* (if you know this method) to the above approach.

4.18 Perspective: Drawbacks of Functional Programming

Functional programming is important for a number of reasons:

- It is one of the fundamental models of computability.
- It provides succinct and elegant means of manipulating potentially very large information structures without deleterious side-effects on data used by some models.
- Consequently, it is a useful model for parallel computation, which can be prone to anomalous behavior if side-effects are not managed carefully.

Functional programming can also fit well with other models, such as object-oriented and logic programming, as will be seen. Despite these desirable traits, we hesitate to recommend it as the only model one consider for software development. Instead we would prefer to see its use where it fits best.

An example of where functional seems less than ideal is computations that need to repeatedly re-assign to large arrays destructively. Here "need" is used subjectively; there is no widely-accepted theoretical definition of what it means to *require* destructive modification. Intuitively however, the following sort of computation is a canonical example: Consider the problem of maintaining a *histogram* of a set of integer data. In other words, we have an incoming stream of integers in some range, say 0 to N-1, in no particular order. We want to know: for each integer in the range, how many times does it appear in the stream. The natural way to solve this problem is to use linear addressing: for each data item in the stream, use the item to index an array of counts, adding 1 each time that integer is encountered. This method is straightforward to implement using destructive assignment to the array elements. However, a functional computation on arrays would create a new array for every element in the stream, which will obviously be costly in comparison to using destructive modification. Some functional programming languages are able to get around this problem by using clever compilation techniques that only *apparently* create a new array at each step but that actually re-use the same array at each step. However, it does not appear that such techniques generalize to all possible problems.

A place where functional programming seems to yield to object-oriented programming techniques is in programming with structures that seem to inherently require modification because there is only one of them. An example is in graphical user interface programming. Here there is only one of each widget showing on the screen, which contains the state of that widget. It does not make sense to speak of creating a new widget each time a modification of the state is made.

4.19 Chapter Review

Define the following concepts or terms:

- accumulator argument
- append
- applicative order
- auxiliary function
- beta reduction
- breadth-first search
- copy rule
- delayed evaluation
- depth-first search
- equational guard
- Euclid's algorithm
- Euler's method
- Horner's rule
- insertion sorting
- interface function
- guarded rule
- inductive definition
- radix representation
- merge sorting
- mutual recursion
- normal order
- radix principle
- recursion
- selection sorting
- sieve
- tail recursion

4.20 Further Reading

L.C. Paulson, *ML for the working programmer*, Cambridge University Press, Cambridge, MA, 1991.

Simon Thompson, *Haskell - The craft of functional programming,* Addison-Wesley, Reading, MA,1999.