
III Abstraction

Many of our data definitions and function definitions look alike. For example, the definition for a list of [Strings](#) differs from that of a list of [Numbers](#) in only two places: the names of the classes of data and the words “String” and “Number.” Similarly, a function that looks for a specific string in a list of [Strings](#) is nearly indistinguishable from one that looks for a specific number in a list of [Numbers](#).

Experience shows that these kinds of similarities are problematic. The similarities come about because programmers—physically or mentally—copy code. When programmers are confronted with a problem that is roughly like another one, they copy the solution and modify the new copy to solve the new problem. You will find this behavior both in “real” programming contexts as well as in the world of spreadsheets and mathematical modeling. Copying code, however, means that programmers copy mistakes and the same fix may have to be applied to many copies. It also means that when the underlying data definition is revised or extended, all copies of code must be found and modified in a corresponding way. This process is both expensive and error-prone, imposing unnecessary costs on programming teams.

Good programmers try to eliminate similarities as much as the programming language allows. “Eliminate” implies that programmers write down their first drafts of programs, spot similarities (and other problems), and get rid of them. For the last step, they either *abstract* or use existing (abstract) functions. It often takes several iterations of this process to get the program into satisfactory shape.

The first half of this part shows how to abstract over similarities in functions and data definitions. Programmers also refer to the result of this process as an *abstraction*, conflating the name of the process and its result. The second half is about the use of existing abstractions and new language elements to facilitate this process. While the examples in this part are taken from the realm of lists, the ideas are universally applicable.

A program is like an essay. The first version is a draft, and drafts demand editing.

16 Similarities Everywhere

If you have solved only a fraction of the exercises in [Arbitrarily Large Data](#), you know that the solutions look alike. As a matter of fact, the similarities may tempt you to copy the solution of one problem to create the solution for another one. But thou shall not steal code, not even your own. Instead, you must *abstract* over similar pieces of code and this chapter teaches you how to abstract.

Our means of avoiding similarities are specific to “Intermediate Student Language” or ISL for short. Almost all other programming languages provide similar means; in object-oriented languages you may find additional abstraction mechanisms. Regardless, these mechanisms share the basic characteristics spelled out in this chapter, and thus the design ideas explained here apply in other contexts, too.

In DrRacket, choose “Intermediate Student Language” from the

16.1 Similarities in Functions

The design recipe determines a function’s basic organization because the template is

created from the data definition without regard to the purpose of the function. Not surprisingly then, functions that consume the same kind of data look alike.

```

; Los -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (cond
    [(empty? l) #false]
    [else
     (or
      (string=? (first l) "dog")
      (contains-dog?
       (rest l)))])))

; Los -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (cond
    [(empty? l) #false]
    [else
     (or
      (string=? (first l) "cat")
      (contains-cat?
       (rest l)))])))

```

Figure 50: Two similar functions

Consider the two functions in figure 50, which consume lists of strings and look for specific strings. The function on the left looks for "dog", the one on the right for "cat". The two functions are nearly indistinguishable. Each consumes lists of strings; each function body consists of a `cond` expressions with two clauses. Each produces `#false` if the input is '(); each uses an `or` expressions to determine whether the first item is the desired item and, if not, uses recursion to look in the rest of the list. The only difference is the string that is used in the comparison of the nested `cond` expressions: `contains-dog?` uses "dog" and `contains-cat?` uses "cat". To highlight the differences, the two strings are shaded.

Good programmers are too lazy to define several closely related functions. Instead they define a single function that can look for both a "dog" and a "cat" in a list of strings. This general function consumes an additional piece of data—the string to look for—and is otherwise just like the two original functions:

```

; String Los -> Boolean
; determines whether l contains the string s
(define (contains? s l)
  (cond
    [(empty? l) #false]
    [else (or (string=? (first l) s)
              (contains? s (rest l)))]))

```

If you really needed a function such as `contains-dog?` now, you could define it as a one-line function, and the same is true for the `contains-cat?` function. Figure 51 does just that, and you should briefly compare it with figure 50 to make sure you understand how we get from there to here. Best of all, though, with `contains?` it is now trivial to look for any string in a list of strings and there is no need to ever define a specialized function such as `contains-dog?` again.

```

; Los -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (contains? "dog" l))

; Los -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (contains? "cat" l))

```

Figure 51: Two similar functions, revisited

What you have just seen is called *functional abstraction*. Abstracting different versions of

"How
to
Design
Programs"
submenu
in the
"Language"
menu.

Computing

functions is one way to eliminate similarities from programs, and as you can see with this one simple example, doing so simplifies programs.

Exercise 200. Use `contains?` to define functions that search for "atom", "basic", and "zoo", respectively. ■

Exercise 201. Create test suites for the following two functions:

```

; Lon -> Lon
; add 1 to each number on l
(define (add1* l)
  (cond
    [(empty? l) '()]
    [else
     (cons (add1 (first l))
           (add1* (rest l)))]))

; Lon -> Lon
; adds 5 to each number on l
(define (plus5 l)
  (cond
    [(empty? l) '()]
    [else
     (cons (+ (first l) 5)
           (plus5 (rest l)))]))

```

Then abstract over them. Define the above two functions in terms of the abstraction as one-liners and use the existing test suites to confirm that the revised definitions work properly. Finally, design a function that subtracts 2 from each number on a given list. ■

16.2 More Similarities in Functions

Abstraction looks easy in the case of `contains-dog?` and `contains-cat?`. It takes only a comparison of two function definitions, a replacement of a literal string with a function parameter, and a quick check that it is easy to define the old functions with the abstract function. This kind of abstraction is so natural that it showed up in the preceding two parts of the book without much ado.

This section illustrates how the very same principle yields a powerful form of abstraction. Take a look at [figure 52](#). Both functions consume a list of numbers and a threshold. The left one produces a list of all those numbers that are below the threshold, while the one on the right produces all those that are above the threshold.

```

; Lon Number -> Lon
; constructs the list of numbers from
; those on l that are below t
(define (small l t)
  (cond
    [(empty? l) '()]
    [else
     (cond
       [(< (first l) t)
        (cons (first l)
              (small (rest l) t))]
       [else
        (small (rest l) t)])]))

; Lon Number -> Lon
; constructs the list of numbers from
; those on l that are above t
(define (large l t)
  (cond
    [(empty? l) '()]
    [else
     (cond
       [(> (first l) t)
        (cons (first l)
              (large (rest l) t))]
       [else
        (large (rest l) t)])]))

```

Figure 52: Two more similar functions

The two functions differ in only one place: the comparison operator that determines whether a number from the given list should be a part of the result or not. The function

borrow
the
term
“abstract”
from
mathematics.
A
mathematician
refers
to “6”
as an
abstract
number
because
it
only
represents
all
different
ways
of
naming
six
things.
In
contrast,
“6
inches”
or “6
eggs”
are
concrete
instances
of “6”
because
they
express
a
measurement
and a
count.

on the left uses `<`, the right one `>`. Other than that, the two functions look identical, not counting the function name.

Let us follow the first example and abstract over the two functions with an additional parameter. This time the additional parameter represents a comparison operator rather than a string:

```
(define (extract R l t)
  (cond
    [(empty? l) '()]
    [else (cond
              [(R (first l) t)
               (cons (first l) (extract R (rest l) t))]
              [else
               (extract R (rest l) t)])]))
```

To apply this new function, we must supply three arguments: a function `R` that compares two numbers; a list `l` of numbers, and a threshold `t`. The function then extracts all those items `i` from `l` for which `(R i t)` evaluates to `#true`.

Stop! At this point you should ask whether this definition makes any sense. Without further ado, we have created a function that consumes a function—something that you probably have not seen before. It turns out, however, that your simple little teaching language ISL supports these kinds of functions, and that defining such functions is one of the most powerful tools of good programmers—even in languages in which function-consuming functions do not seem to be available.

Now that you have recovered from this surprise, let us see how `extract` actually works. Clearly, as long as the input list is `'()` the result is `'()`, too, no matter what the other arguments are:

```
(extract < '() 5)
=
'()
```

So next we look at a one-item list:

```
(extract < (cons 4 '()) 5)
```

The result should be `(cons 4 '())` because the only item of this list is 4 and `(< 4 5)` is true. Here is the first step of the evaluation:

```
(extract < (cons 4 '()) 5)
=
(cond
  [(empty? (cons 4 '())) '()]
  [else (cond
            [(< (first (cons 4 '())) 5)
             (cons (first (cons 4 '()))
                   (extract < (rest (cons 4 '())) 5))]
            [else (extract < (rest (cons 4 '())) 5)])]))
```

It generalizes the rule of application; the application is replaced with the body of the `extract` function and all occurrences of `R` replaced by `<`, `l` by `(cons 4 '())`, and `t` by 5. The rest is straightforward:

```
(cond
  [(empty? (cons 4 '())) '()]
```

If you have taken a calculus course, you encountered the differential operator and the indefinite integral, both of which are functions that consume and produce a function. But we do not assume that you

```

[else (cond
  [(< (first (cons 4 '())) 5)
   (cons (first (cons 4 '()))
         (extract < (rest (cons 4 '())) 5))]]
[else (extract < (rest (cons 4 '())) 5)]]])
=
(cond
  [#false '()]
  [else (cond
    [(< (first (cons 4 '())) 5)
     (cons (first (cons 4 '()))
           (extract < (rest (cons 4 '())) 5))]]
    [else (extract < (rest (cons 4 '())) 5)]]])
=
(cond
  [(< (first (cons 4 '())) 5)
   (cons (first (cons 4 '()))
         (extract < (rest (cons 4 '())) 5))]]
  [else (extract < (rest (cons 4 '())) 5)]]
=
(cond
  [(< 4 5)
   (cons (first (cons 4 '()))
         (extract < (rest (cons 4 '())) 5))]]
  [else (extract < (rest (cons 4 '())) 5)]]
=
(cond
  [#true
   (cons (first (cons 4 '()))
         (extract < (rest (cons 4 '())) 5))]]
  [else (extract < (rest (cons 4 '())) 5)]]
=
(cons 4 (extract < (rest (cons 4 '())) 5))
=
(cons 4 (extract < '() 5))
=
(cons 4 '())

```

have
taken
such
a
course
and
show
you
these
wonderful
tricks
anyway.

The last step is the equation discussed above, meaning there is no need to spell out the reasoning again.

Our final example is an application of `extract` to a list of two items:

```

(extract < (cons 6 (cons 4 '())) 5)
= (extract < (cons 4 '()) 5)
= (cons 4 (extract < '() 5))
= (cons 4 '())

```

Step 1 is new and says that `extract` determines that the first item on the list is not less than the threshold and that it therefore is not added to the result of the natural recursion.

Exercise 202. Check step 1 of the last calculation

```

(extract < (cons 6 (cons 4 '())) 5)
= (extract < (cons 4 '()) 5)

```

by hand. Show every step. ■

Exercise 203. Evaluate the expression

```
(extract < (cons 8 (cons 6 (cons 4 '()))) 5)
```

by hand. Show the new steps, rely on prior calculations where possible. ■

The calculations show that `(extract < l t)` computes the same result as `(small l t)`. Indeed, they suggest that the resulting expressions are nearly identical. Similarly, `(extract > l t)` produces the same output as `(large l t)`, which means that you can define the two original functions like this:

```
; Lon Number -> Lon
(define (small-1 l t)
  (extract < l t))

; Lon Number -> Lon
(define (large-1 l t)
  (extract > l t))
```

The important insight is **not** that `small-1` and `large-1` are one-line definitions. Once you have an abstract function such as `extract`, you can put it to good uses elsewhere:

1. `(extract = l t)`: This expression extracts all those numbers in `l` that are equal to `t`.
2. `(extract <= l t)`: This one produces the list of numbers in `l` that are less than or equal to `t`.
3. `(extract >= l t)`: This last expression computes the list of numbers that are greater than or equal to the threshold.

As a matter of fact, the first argument for `extract` need not be one of ISL's predefined operations. Instead, you can use any function that consumes two arguments and produces a **Boolean**. Consider this example:

```
; Number Number -> Boolean
; is the area of a square with side x larger than c
(define (squared>? x c)
  (> (* x x) c))
```

That is, the function checks whether the claim $x^2 > c$ holds, and it is usable with `extract`:

```
(extract squared>? (list 3 4 5) 10)
```

This application extracts those numbers in `(list 3 4 5)` whose square is larger than 10.

Exercise 204. Evaluate `(squared>? 3 10)`, `(squared>? 4 10)`, and `(squared>? 5 10)` by hand. Then show that

```
(extract squared>? (list 3 4 5) 10)
```

evaluates to `(list 4 5)`. ■

So far you have seen that abstracted function definitions can be more useful than the functions you start from. For example, `contains?` is more useful than `contains-dog?` and `contains-cat?`, and `extract` is more useful than `small` and `large`. Another important aspect of abstraction is that you now have a single point of control over all these functions. If it turns out that the abstract function contains a mistake, fixing its definition suffices to fix all other definitions. Similarly, if you figure out how to accelerate the computations of the abstract function or how to reduce its energy consumption, then all functions defined in terms of this function are improved without further ado. The following exercises indicate how these single-point-of-control improvements work.

These effects of abstraction are crucial for

Exercise 205. Abstract the following two functions into a single function:

```
; Nelon -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l))
     (first l)]
    [else
     (cond
      [(< (first l) (inf (rest l)))
       (first l)]
      [else
       (inf (rest l))]]]))
```

```
; Nelon -> Number
; determines the largest
; number on l
(define (sup l)
  (cond
    [(empty? (rest l))
     (first l)]
    [else
     (cond
      [(> (first l) (sup (rest l)))
       (first l)]
      [else
       (sup (rest l))]]]))
```

Both consume non-empty lists of numbers (*Nelon*) and produce a single number. The left one produces the smallest number in the list, the right one the largest.

Define *inf-1* and *sup-1* in terms of the abstract function. Test each of them with these two lists:

```
(list 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25)
```

Why are these functions slow on some of the long lists?

Modify the original functions with the use of *max*, which picks the larger of two numbers, and *min*, which picks the smaller one. Then abstract again, define *inf-2* and *sup-2*, and test them with the same inputs again. Why are these versions so much faster?

For a complete answer to the two questions on performance, see [Local Function Definitions](#). ■

16.3 Similarities in Data Definitions

Now take a close look at the following two data definitions:

```
; List-of-numbers
; A Lon is one of:
; - '()
; - (cons Number Lon)
```

```
; List-of-String
; A Los is one of:
; - '()
; - (cons String Los)
```

The one on the left introduces lists of numbers; the one on the right describes lists of strings. And the two data definitions are similar. Like similar functions, the two data definitions use two different names, but this is irrelevant because any name would do. The only real difference concerns the first position inside of *cons* in the second clause, which specifies what kind of items the list contains.

In order to abstract over this one difference, we proceed as if a data definition were a function. We introduce a parameter, which makes the data definition look like a function, and where there used to be different references, we use this parameter:

large, industrial programming projects. For that reason, programming language and software engineering research has focused on how to create single points of control in large projects. Of course, the same idea applies to all kinds of computer designs (word documents, spread sheets) and organizations in general.

```

; A [List-of ITEM] is one of:
; - '()
; - (cons ITEM [List-of ITEM])

```

We call such abstract data definitions *parametric data definitions* because of the parameter. Roughly speaking, a parametric data definition abstracts from a reference to a particular collection of data in the same manner as a function abstracts from a particular value.

The question is, of course, what these parameters range over. For a function, they stand for an unknown value; when the function is applied, the value becomes known. For a parametric data definition, a parameter stands for an entire class of values. The process of supplying the name of a data collection to a parametric data definition is called *instantiation*; here are some sample instantiations of the [List-of](#) abstraction:

- [\[List-of Number\]](#) says that `ITEM` represents all numbers so the notation is just another name for [List-of-numbers](#);
- [\[List-of String\]](#) defines the same class of data as [List-of-String](#); and
- if we had identified a class of inventory records, like this:

```

(define-struct ir [name price])
; An IR is
;   (make-ir String Number)

```

then [\[List-of IR\]](#) would be a name for the class of lists of inventory records.

By convention, we use names with all capital letters for parameters of data definitions, while the arguments are (usually) the names of existing data collections.

Our way to validate that these shorthands really mean what we say they mean is to substitute the actual name of a data definition, e.g., [Number](#), for the parameter `ITEM` of the data definition and to use a plain name for the data definition:

```

; A List-of-numbers-again is one of:
; - '()
; - (cons Number List-of-numbers-again)

```

Since the data definition is self-referential, we copied the entire data definition. The resulting definition looks exactly like the conventional one for lists of numbers and truly identifies the same class of data.

Let us take a brief look at a second example, starting with a structure type definition:

```

(define-struct point [hori veri])

```

Here are two different data definitions that rely on this structure type definition:

<pre> ; A Pair-boolean-string is a ; (make-point Boolean String) </pre>	<pre> ; A Pair-number-image is a ; (make-point Number Image) </pre>
---	---

In this case, the data definitions differ in two places—both marked by highlighting. The differences in the `hori` fields correspond to each other, and so do the differences in the `veri` fields. It is thus necessary to introduce two parameters to create an abstract data definition:

```

; A [CP H V] is a structure:

```



```
| ; (make-point H V)
```

Here `H` is the parameter for data collections for the `hor-i` field, and `V` stands for data collections that can show up in the `ver-i` field.

To instantiate a data definition with two parameters, you need two names of data collections. Using `Number` and `Image` for the parameters of `CP`, you get `[CP Number Image]`, which describes the collections of `points` that combine a number with an image. In contrast `[CP Boolean String]` combines Boolean values with strings in a `point` structure.

Once you have parametric data definitions, you can even mix and match them to great effect. Consider this one:

```
| ; [List-of [CP Boolean Image]]
```

The outermost notation is `[List-of ...]`, which means that you are dealing with a list. Question is what kind of data the list contains, and to answer that question, you need to study the inside of the `List-of` expression:

```
| ; [CP Boolean Image]
```

The inner part combines `Boolean` and `Image` in a `point`, naming yet another class of data that pairs up two different classes in structure instances. In short,

```
| ; [List-of [CP Boolean Image]]
```

is a list of `CPs` that combine `Booleans` and `Images`. Similarly,

```
| ; [CP Number [List-of Image]]
```

is an instantiation of `CP` that combines one `Number` with a list of `Images`. If this went too fast, tease apart this data expression, like above, and explain each pieces as you go.

Exercise 206. Here are two strange but similar data definitions:

<pre> ; A Nested-string is one of: ; - String ; - (make-layer Nested-string)</pre>	<pre> ; A Nested-number is one of: ; - Number ; - (make-layer Nested-number)</pre>
---	---

Both data definitions exploit this structure type definition:

```
| (define-struct layer [stuff])
```

Both define nested forms of data' one is about numbers and the other about strings. Make examples for both. Abstract over the two. Then instantiate the abstract definition to get back the originals. ■

Exercise 207. Take a look at this data definition:

```
| ; A [Bucket ITEM] is  
| ; (make-bucket N [List-of ITEM])  
| ; interpretation the n in (make-bucket n l) is the length of l  
| ; i.e., (= (length l) n) is always true
```

When you instantiate `Bucket` with `String`, `IR`, and `Posn`, you get three different data collections. Describe each of them with a sentence and with two distinct examples.

Now consider this instantiations:

```
; [Bucket [List-of [List-of String]]]
```

Construct three distinct pieces of data that belong to this collection. ■

Exercise 208. Here is one more parametric data definition:

```
; A [Maybe X] is one of:  
; - #false  
; - X
```

Interpret the following data definitions:

- [Maybe String]
- [Maybe [List-of String]]
- [List-of [Maybe String]]

What does the following function signature mean:

```
; String [List-of String] -> [Maybe [List-of String]]  
; returns the remainder of the list los if it contains s  
; #false otherwise  
(check-expect (occurs "a" (list "b" "a" "d")) (list "d"))  
(check-expect (occurs "a" (list "b" "c" "d")) #f)  
(define (occurs s los)  
  los)
```

Design the function. ■

16.4 Functions Are Values

The functions of this section stretch our understanding of program evaluation. It is easy to understand how functions consume more than numbers, say strings, images, and Boolean values. Structures and lists are a bit of a stretch, but they are finite “things” in the end. Function-consuming functions, however, are strange. Indeed, these kind of functions violate the BSL grammar of the first intermezzo in two ways. First, the names of primitive operations and functions are used as arguments in applications. Second, parameters are used as if they were functions, that is, the first position of applications.

Spelling out the problem tells you how the ISL grammar differs from BSL’s. First, our expression language should include the names of functions and primitive operations in the definition. Second, the first position in an application should allow things other than function names and primitive operations; at a minimum, it must allow variables and function parameters. In anticipation of other uses of functions, we agree on allowing arbitrary expressions in that position.

The changes to the grammar seem to demand changes to the evaluation rules, but they don’t change at all. All that changes is the set of values. To accommodate functions as arguments of functions, the simplest change is to say that **functions are values**. Thus, we start using the names of functions and primitive operations as values; later we introduce another way to deal with functions as values.

Exercise 209. Assume the definitions area in DrRacket contains `(define (f x) x)`.

Identify the values among the following expressions:

1. `(cons f '())`
2. `(f f)`
3. `(cons f (cons 10 (cons (f 10) '())))`

Explain why they are values and why the remaining expressions are not values. ■

Exercise 210. Argue why the following sentences are now legal definitions:

1. `(define (f x) (x 10))`
2. `(define (f x) (x f))`
3. `(define (f x y) (x 'a y 'b))`

Explain your reasoning. ■

Exercise 211. Develop `function=at-1.2-3-and-5.775?`. The function determines whether two functions from numbers to numbers produce the same results for `1.2`, `3`, and `-5.775`.

Mathematicians say that two functions are equal if they compute the same result when given the same input—for all possible inputs.

Can we hope to define `function=?`, which determines whether two functions from numbers to numbers are equal? If so, define the function. If not, explain why and consider the implication that you have encountered the first easily definable idea for which you cannot define a function. ■

17 Designing Abstractions

In essence, to abstract is to turn something concrete into a parameter. We have this several times in the preceding section. To abstract similar function definitions, you add parameters that replace concrete values in the definition. To abstract similar data definitions, you create parametric data definitions. When you will encounter other programming languages, you will see that their abstraction mechanisms also require the introduction of parameters, though they may not be function parameters.

17.1 Abstractions from Examples

When you first learned to add, you worked with concrete examples. Your parents probably taught you to use your fingers to add two small numbers. Later on, you studied how to add two arbitrary numbers; you were introduced to your first kind of abstraction. Much later still, you learned to formulate expressions that convert temperatures from Celsius to Fahrenheit or calculate the distance that a car travels at a certain speed in a given amount of time. In short, you went from very concrete examples to abstract relations.

This section introduces a design recipe for creating abstractions from examples. As the preceding section shows, creating abstractions is easy. We leave the difficult part to the next section where we show you how to find and use existing abstractions.

Recall the essence of [Similarities Everywhere](#). We start from two concrete function definitions or two concrete data definitions; we compare them; we mark the differences; and then we abstract. And that is mostly all there is to creating abstractions:

1. Step 1 is to **compare** two items for similarities.

When you find two function definitions that are almost the same except for their names and some *values* at a few *analogous* places, compare them, mark the differences. If the two definitions differ in more than one place, connect the corresponding differences with a line or a comment.

Here is a pair of similar function definitions:

```
; List-of-numbers -> List-of-numbers
; converts a list of Celsius
; temperatures to Fahrenheit
(define (cf* l)
  (cond
    [(empty? l) '()]
    [else
     (cons (C2F (first l))
           (cf* (rest l))))]))
```

```
; Inventory -> List-of-strings
; extracts the names of toys
; from an inventory
(define (names i)
  (cond
    [(empty? i) '()]
    [else
     (cons (IR-name (first i))
           (names (rest i))))]))
```

```
;
; Number -> Number
; converts one Celsius
; temperature to Fahrenheit
(define (C2F c)
  (+ (* 9/5 c) 32))
```

```
;
(define-struct IR [name price])
; An IR is (make-IR String Number)
; An Inventory is one of:
; - '()
; - (cons IR Inventory)
```

The recipe requires a substantial modification for abstracting over non-values.

The two functions apply a function to each item in a list. They differ only as to which function they apply to each item. The two highlights emphasize this essential difference. They also differ in two inessential ways: the names of the function and the names of the parameters.

2. Next we abstract. **To abstract** means to replace the contents of corresponding code highlights with new names and add these names to the parameter list. For our running example, we obtain the following pair of functions after replacing the differences with *g*:

```
(define (cf* l g)
  (cond
    [(empty? l) '()]
    [else
     (cons (g (first l))
           (cf* (rest l) g))]))
```

```
(define (names i g)
  (cond
    [(empty? i) '()]
    [else
     (cons (g (first i))
           (names (rest i) g))]))
```

This first change eliminates the essential difference. Now each function traverses a list and applies some given function to each item.

The inessential differences—the names of the functions and occasionally the names of some parameters—are easy to eliminate. Indeed, if you have explored DrRacket, you know that check syntax allows you to do this systematically and easily:

```

(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
     (cons (g (first k))
           (map1 (rest k) g))]))

(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
     (cons (g (first k))
           (map1 (rest k) g))]))

```

We choose to use `map1` for the name of the function and `k` for the name of the list parameter. No matter which names you choose, the result is two identical function definitions.

Our example is simple. In many cases, you will find that there is more than just one pair of differences. The key is to find pairs of differences. When you mark up the differences on paper and pencil, connect related boxes with a line. Then introduce one additional parameter per line. And don't forget to change all recursive uses of the function so that the additional parameters go along for the ride.

3. Now we must validate that the new function is a correct abstraction of the original pair of functions. To validate means **to test**, which here means to define the two original functions in terms of the abstraction.

Thus suppose that one original function is called `f-original` and consumes one argument and that the abstract function is called `abstract`. If `f-original` differs from the other concrete function in the use of one value, say, `val`, the following function definition

```

(define (f-from-abstract x)
  (abstract x val))

```

introduces the function `f-from-abstract`, which should be equivalent to `f-original`. That is, for every proper value `V`, `(f-from-abstract V)` should produce the same answer as `(f-original V)`. This is particularly true for all values that your tests for `f-original` use. So re-formulate and re-run those tests for `f-from-abstract` and make sure they succeed.

Let us return to our running example:

```

; List-of-numbers -> List-of-numbers
(define (cf*-from-map1 l)
  (map1 l C2F))

; Inventory -> List-of-strings
(define (names-from-map1 i)
  (map1 i IR-name))

```

A complete example would include some tests, and thus we can assume that both `cf*` and `names` come with some tests:

```

(check-expect
 (cf*
  (list 100 0 -40))
 (list 212 32 -40))

(check-expect
 (names
  (list
   (make-IR "doll" 21.0)
   (make-IR "bear" 13.0)))
 (list "doll" "bear"))

```

To ensure that the functions defined in terms of `map1` work properly, you can copy the tests and change the function names appropriately:

```
(check-expect
  (cf*-from-map1
    (list 100 0 -40))
  (list 212 32 -40))

(check-expect
  (names-from-map1
    (list
      (make-IR "doll" 21.0)
      (make-IR "bear" 13.0)))
  (list "doll" "bear"))
```

4. To make a new abstraction useful, it needs a **signature**. As [Using Abstractions](#) explains, reuse of abstract functions start with their signatures. Finding useful signatures is, however, a serious problem. For now we just use the running example to illustrate the problem. [Similarities in Signatures](#) below resolves the issue.

So consider the problem of finding a signature for `map1`. On the one hand, if you view `map1` as an abstraction of `cf*`, you might think the signature is

```
; List-of-numbers [Number -> Number] -> List-of-numbers
```

That is, the original signature extended with one signature for functions:

```
; [Number -> Number]
```

Since the additional parameter for `map1` is a function, the use of a function signature shouldn't surprise you. This function signature is also quite simple; it is a "name" for all the functions from numbers to numbers. Here `c2F` is such a function, and so are `add1`, `sin`, and `imag-part`.

On the other hand, if you view `map1` as an abstraction of `names`, the signature is quite different:

```
; Inventory [IR -> String] -> List-of-strings
```

This time the additional parameter is `IR-name`, which is a selector function that consumes `IRs` and produces `Strings`. But clearly this second signature would be useless in the first case, and vice versa. To accommodate both cases, the signature for `map1` must express that `Number`, `IR`, and `String` are coincidental.

Also concerning signatures, you are probably eager to use `List-of` by now. It is clearly easier to write `[List-of IR]` than spelling out a data definition for `Inventory`. So yes, as of now, we use `List-of` when it is all about lists and you should too.

Once you have abstracted two functions, you should check whether there are other uses for the abstract function. If so, the abstraction is truly useful. Consider `map1` for example. It is easy to see how to use it to add 1 to each number on a list of numbers:

```
; List-of-numbers -> List-of-numbers
(define (add1-to-each l)
  (map1 add1 l))
```

Similarly, `map1` can also be used to extract the price of each item in an inventory. When you can imagine many such uses for a new abstraction, add it to a library of useful functions to have around. Of course, it is quite likely that someone else has thought of it and the function is already a part of the language. For a function like `map1`, see [Using Abstractions](#).

Exercise 212. Design `tabulate`, which is the abstraction of the following two functions:

```
; Number -> [List-of Number]           ; Number -> [List-of Number]
```

```

; tabulates sin between n
; and 0 (inclusive) in a list
(define (tab-sin n)
  (cond
    [(= n 0) (list (sin 0))]
    [else
     (cons (sin n)
           (tab-sin (sub1 n))))]))

```

```

; tabulates sqrt between n
; and 0 (inclusive) in a list
(define (tab-sqrt n)
  (cond
    [(= n 0) (list (sqrt 0))]
    [else
     (cons (sqrt n)
           (tab-sqrt (sub1 n))))]))

```

When `tabulate` is properly designed, use it to define a tabulation function for `sqr` and `tan`.¹

Exercise 213. Design `fold1`, which is the abstraction of the following two functions:

```

; [List-of Number] -> Number
; computes the sum of
; the numbers on l
(define (sum l)
  (cond
    [(empty? l) 0]
    [else
     (+ (first l)
        (sum (rest l)))]))

```

```

; [List-of Number] -> Number
; computes the product of
; the numbers on l
(define (product l)
  (cond
    [(empty? l) 1]
    [else
     (* (first l)
        (product (rest l)))]))

```

Exercise 214. Design `fold2`, which is the abstraction of the following two functions:

```

; [List-of Number] -> Number
(define (product l)
  (cond
    [(empty? l) 1]
    [else
     (* (first l)
        (product (rest l)))]))

```

```

; [List-of Posn] -> Image
(define (image* l)
  (cond
    [(empty? l) empty]
    [else
     (place-dot (first l)
                 (image* (rest l)))]))

```

```

; Posn Image -> Image
(define (place-dot p img)
  (place-image dot
               (posn-x p) (posn-y p)
               img))

; graphical constants:
(define empty (empty-scene 100 100))
(define dot (circle 3 "solid" "red"))

```

Compare this exercise with [exercise 213](#). Even though both involve the `product` function, this exercise poses an additional challenge because the second function, `image*`, consumes a list of `Posns` and produces an `Image`. Still, the solution is within reach of the material in this section, and it is especially worth comparing the solution with the one to the preceding exercise. The comparison yields interesting insights into abstract signatures.¹

Last but not least, when you are dealing with data definitions, the abstraction process proceed in an analogous manner. The extra parameters to data definitions stands for collections of values, and testing means spelling out a data definition for some concrete

examples. All in all, abstracting over data definitions tends to be easier than abstracting over functions, and so we leave it to you to adapt the design recipe appropriately.

17.2 Similarities in Signatures

As it turns out, a function's signature is key to its reuse. Thus, to increase the usefulness of an abstract function, you must learn to formulate signatures that describes abstracts in their most general terms possible. To understand how this works, we start with a second look at signatures and from the simple—though possibly startling—insight that signatures are basically data definitions.

Both signatures and data definitions specify a class of data; the difference is that data definitions also name the class of data while signatures don't. Nevertheless, when you write down

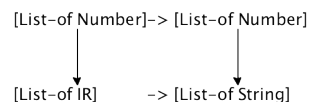
```
| ; Number Boolean -> String
| (define (f n b) "hello world")
```

your first line describes an entire class of data, and your second line states that `f` belongs to that class. To be precise, this signature describes the class of **all functions** that consume a `Number` and a `Boolean` and that produce a `String`.

In general, the arrow notation of signatures is like the `List-of` notation from [Similarities in Data Definitions](#). The latter consumes (the name of) one class of data, say `X`, and describes all lists of `X` items—without assigning it a name. The arrow notation consumes an arbitrary number of classes of data and describes collections of functions.

What this means is that the abstraction design recipe applies to signatures, too. You compare similar signatures; you highlight the differences; and then you replace those with parameters. But the process of abstracting signatures feels more complicated than the one for functions, partly because signature are already abstract pieces of the design recipe and partly because the arrow-based notation is much more complex than anything else we have encountered.

Let us start with the signatures of `cf*` and `names`:



The diagram is the result of the compare-and-contrast step. Comparing the two signatures shows that they differ in two places: to the left of the arrow, we see `Number` versus `IR` and to its right, it is `Number` versus `String`.

If we replace the two differences with some kind of parameters, say `X` and `Y`, we get the same signature:

```
| ; [X Y] [List-of X] -> [List-of Y]
```

The new signature starts with a sequence of variables, drawing an analogy to function definitions and the data definitions above. Roughly speaking, these variables are the parameters of the signature, like those of functions and data definitions. To make the latter concrete, the variable sequence is like `ITEM` in the definition of `List-of` or the `X` and `Y` in the definition of `CP` from [Similarities in Data Definitions](#). And just like those, `X` and

Y range over classes of values.

An instantiation of this parameter list is the rest of the signature with the parameters replaced by the data collections: either their names or other parameters or abbreviations such as `List-of` from above. Thus, if you replace both X and Y with `Number`, you get back the signature for `cf*`:

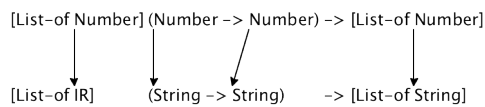
```
| ; [List-of Number] -> [List-of Number]
```

If you choose `IR` and `String`, respectively, you get back the signature for `names`:

```
| ; [List-of IR] -> [List-of String]
```

And that explains why we may consider this parametrized signature as an abstraction of the original signatures for `cf*` and `names`.

Once we add the extra function parameter to these two functions we get `map1` and the signatures are as follows:



Again, the signatures are in pictorial form and with arrows connecting the corresponding differences. These mark-ups suggest that the differences in the second argument—a function—are related to the differences in the original signatures. Specifically, `Number` and `IR` on the left of the new arrow refer to items on the first argument—a list—and the `Number` and `String` on the right refer to the items on the result—also a list.

Since listing the parameters of a signature is extra work for our purposes, we simply say that from now on all variables in signatures as parameters. Other programming languages, however, insist on explicitly listing the parameters of signatures, but in return you can articulate additional constraints in such signatures and the signatures are checked before you run the program.

Now let's apply the same trick to get a signature for `map1`:

```
| ; [List-of X] [X -> Y] -> [List-of Y]
```

Concretely, `map1` consumes a list of items, all of which belong to some (yet to be determined) collection of data called X. It also consumes a function that consumes elements of X and produces elements of a second unknown collection, called Y. The result of `map1` are lists that contain items from Y.

As you may guess from our first example, abstracting over signatures requires practice. So here is a second pair of signatures:

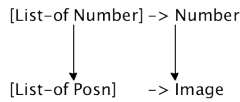
```
| ; [List-of Number] -> Number
```

```
| ; [List-of Posn] -> Image
```

They are the signatures for `product` and `image*` in [exercise 214](#). While the two signatures have some common organization, the differences are distinct. Let us first spell out the common organization in detail:

- both signatures describe one-argument functions;
- both argument descriptions employ the [List-of](#) construction;

The difference is that the first signature refers to [Number](#) twice, and the second one refers to [Posns](#) and [Images](#). Putting similarities and differences together, the first occurrence of [Number](#) corresponds to [Posn](#) and the second one to [Image](#):



To make more progress on a signature for the abstraction of the two functions in [exercise 214](#), we need to take the first two steps of the design recipe:

```
(define (pr* l bs jn)
  (cond
    [(empty? l) bs]
    [else
     (jn (first l)
          (pr* (rest l) bs jn))]))

(define (im* l bs jn)
  (cond
    [(empty? l) bs]
    [else
     (jn (first l)
          (im* (rest l) bs jn))]))
```

Since the two functions differ in two pairs of values, the revised versions consume two additional values: one is an atomic value, to be used in the base case, and the other one is a function that joins together the result of the natural recursion with the first item on the given list.

The key is to translate this insight into two signatures for the two new functions. When you do so for `pr*`, you get

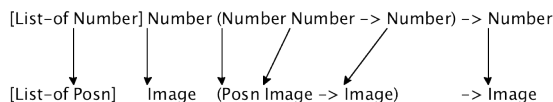
```
; [List-of Number] Number [Number Number -> Number] -> Number
```

because the result in the base case is a number and the function that combines the first item and the natural recursion is `+` in the original function. Similarly, for `im*` the signature is

```
; [List-of Posn] Image [Posn Image -> Number] -> Number
```

As you can see from the function definition for `im*`, the base case returns an image and the combination function is `place-dot`, which combines a [Posn](#) and an [Image](#) into an [Image](#).

Now we take the diagram from above and extend it to the signatures with the additional inputs:



From this diagram, you can easily see that the two revised signatures share even more organization than the original two. Furthermore, the pieces that describe the base cases correspond to each other and so do the pieces of the sub-signature that describe the combination function. All in all there are six pairs of differences but they boil down to just two:

- some occurrences of [Number](#) correspond to [Posn](#)

2. and other occurrences of `Number` correspond to `Image`.

So to abstract we need two variables, one per kind of correspondence.

Here, then, is the signature for `fold2`, the abstraction that [exercise 214](#) requests:

```
... ; [List-of X] Y [X Y -> Y] -> Y
```

Stop! Make sure that replacing both parameters of the signature, `X` and `Y`, with `Number` yields the signature for `pr*` and that replacing the same variables with `Posn` and `Image`, respectively, yields the signature for `im*`.

The two examples illustrate how to find general signatures. In principle the process is just like the one for abstracting functions. Due to the informal nature of signatures, however, it cannot be checked with running examples the way code is checked. Here is step-by-step formulation:

1. Given two similar function definitions, `f` and `g`, compare their signatures for similarities and differences. The goal is to discover the organization of the signature and to mark the places where one signature differs from the other. Connect the differences as pairs just like you do when you analyze function bodies.
2. Abstract `f` and `g` into `f-abs` and `g-abs`. That is, add parameters that eliminate the differences between `f` and `g`. Create signatures for `f-abs` and `g-abs`. Keep in mind what the new parameters originally stood for; this helps you figure out the new pieces of the signatures.
3. Check whether the analysis of step 1 extends to the signatures of `f-abs` and `g-abs`. If so, replace the differences with variables that range over classes of data. Once the two signatures are the same you have a signature for the abstracted function.
4. Test the abstract signature in two ways. First, ensure that suitable substitutions of the variables in the abstract signature yield the signatures of `f-abs` and `g-abs`. Second, check that the generalized signature is in sync with the code. For example, if `p` is a new parameter and its signature is

```
... ; ... [A B -> C] ....
```

then `p` should always be applied to two arguments, the first one from `A` and the second one from `B`. And the result of an application of `p` is going to be a `C` and should be used where elements of `C` are expected.

As with abstracting functions, the key is to compare the concrete signatures of the examples and to determine the similarities and differences. With enough practice and intuition, you will soon be able to abstract signatures without much guidance.

Exercise 215. Each of the following signatures describes a collection of functions:

```
... ; [Number -> Boolean]
... ; [Boolean String -> Boolean]
... ; [Number Number Number -> Number]
... ; [Number -> [List-of Number]]
... ; [[List-of Number] -> Boolean]
```

Describe these collections with at least one example from ISL. ■

Exercise 216. Formulate signatures for the following functions:

- `sort-n`, which consumes a list of numbers and a function that consumes two numbers (from the list) and produces a [Boolean](#); `sort-n` produces a sorted list of numbers.
- `sort-s`, which consumes a list of strings and a function that consumes two strings (from the list) and produces a [Boolean](#); `sort-s` produces a sorted list of strings.

Then abstract over the two signatures, following the above steps. Also show that the generalized signature can be instantiated to describe the signature of a sort function for lists of [IRs](#). ■

Exercise 217. Formulate signatures for the following functions:

- `map-n`, which consumes a list of numbers and a function from numbers to numbers to produce a list of numbers.
- `map-s`, which consumes a list of strings and a function from strings to strings and produces a list of strings.

Then abstract over the two signatures, following the above steps. Also show that the generalized signature can be instantiated to describe the signature of the `map-IR` function above. ■

17.3 Single Point of Control

In general, programs are like drafts of papers. Editing drafts is important to correct typos, to fix grammatical mistakes, to make the document consistent, and to eliminate repetitions. Nobody wants to read papers that repeat themselves a lot, and nobody wants to read such programs either.

The elimination of similarities in favor of abstractions has many advantages. Creating an abstraction simplifies definitions. It may also uncover problems with existing functions, especially when similarities aren't quite right. But, the single most important advantage is the creation of *single points of control* for some common functionality.

Putting the definition for some functionality in one place makes it easy to maintain a program. When you discover a mistake, you have to go to just one place to fix it. When you discover that the code should deal with another form of data, you can add the code to just one place. When you figure out an improvement, one change improves all uses of the functionality. If you had made copies of the functions or code in general, you would have to find all copies and fix them; otherwise the mistake might live on or the only one of the functions would run faster.

We therefore formulate this guideline:

Creating Abstractions: Form an abstraction instead of copying and modifying any piece of a program.

Our design recipe for abstracting functions is the most basic tool to create abstractions. To use it requires practice. As you practice, you expand your capabilities to read, organize, and maintain programs. The best programmers are those who actively edit their programs to build new abstractions so that they collect things related to a task at a single point. Here we use functional abstraction to study this practice; in other courses on programming, you will encounter other forms of abstraction, most importantly *inheritance* in class-based object-oriented languages.

17.4 Abstractions from Templates

Over the course of the first two chapters, we have designed many functions using the same template. After all, the design recipe says to organize functions around the organization of the (major) input data definition. It is therefore not surprising that many function definitions look similar to each other.

Indeed, you should abstract from the templates directly, you should do so automatically, and some experimental programming languages do so. Even though this topic is still a subject of research, you are now in a position to understand the basic idea. Consider the template for lists:

```
(define (fun-for-l l)
  (cond
    [(empty? l) ...]
    [else ... (first l) ... (fun-for-l (rest l)) ...]))
```

It contains two gaps, one in each clause. When you use this template to define a list-processing function, you usually fill these gaps with a value in the first `cond` clause and with a function `combine` in the second clause. The `combine` function consumes the first item of the list and the result of the natural recursion and creates the result from these two pieces of data.

Now that you know how to create abstractions, you can complete the definition of the abstraction from this informal description:

```
; [List-of X] Y [X Y -> Y] -> Y
(define (reduce l base combine)
  (cond
    [(empty? l) base]
    [else (combine (first l)
                    (reduce (rest l) base combine))]))
```

It consumes two extra arguments: `base`, which is the value for the base case, and `combine`, which is the function that performs the value combination for the second clause.

Using `reduce` you can define many plain list-processing functions as “one liners.” Here are definitions for `sum` and `product`, two functions used several times in the first few sections of this chapter:

<pre>; [List-of Number] -> Number (define (sum lon) (reduce lon 0 +))</pre>	<pre>; [List-of Number] -> Number (define (product lon) (reduce lon 1 *))</pre>
--	--

For `sum`, the base case always produces `0`; adding the first item and the result of the natural recursion combines the values of the second clause. Analogous reasoning explains `product`. Other list-processing functions can be defined in a similar manner using `reduce`.

18 Using Abstractions

Many programming languages provide a number of looping constructs, or *loop* for short. A loop processes a compound piece of data, one piece at a time. In our terminology a

loop abstracts over the traversal of data and applies some given function to each of its pieces. You have encountered several such *loops* in the first two sections of this chapter: `extract`, `fold1`, `map1`, etc. These functions consume a function and apply it to each item on some list.

Once you have such loop abstractions, you should use them when possible. They create single points of control data, and they are a work-saving device. To make this precise, the use of an abstraction helps **the reader** of your code to understand your intentions, in particular if the abstraction is well-known and built into the language or comes with its standard libraries.

This chapter is all about the reuse of existing ISL abstractions. It starts with a section on existing ISL abstractions, some of which you have seen under false names. The remaining sections are about re-using such abstractions. One key ingredient is a new syntactic construct, `local`, for defining functions and variables (and even structure types) locally within a function. An auxiliary ingredient, introduced in the last section, is the `lambda` construct for creating nameless functions; `lambda` is a convenience but inessential to the idea of re-using abstract functions.

```
; N [N -> X] -> [List-of X]
; constructs a list by applying f to 0, 1, ..., (sub1 n)
; (build-list n f) = (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

; [X -> Boolean] [List-of X] -> [List-of X]
; produces a list from all those items on alox for which p holds
(define (filter p alox) ...)

; [List-of X] [X X -> Boolean] -> [List-of X]
; produces a version of alox that is sorted according to cmp
(define (sort alox cmp) ...)

; [X -> Y] [List-of X] -> [List-of Y]
; constructs a list by applying f to each item on alox
; (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f alox) ...)

; [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for every item on alox
; (andmap p (list x-1 ... x-n)) = (and (p x-1) ... (p x-n))
(define (andmap p alox) ...)

; [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for at least one item on alox
; (ormap p (list x-1 ... x-n)) = (or (p x-1) ... (p x-n))
(define (ormap p alox) ...)

; [X Y -> Y] Y [List-of X] -> Y
; applies f from right to left to each item in alox and base
; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

; [X Y -> Y] Y [List-of X] -> Y
; applies f from left to right to each item in alox and base
; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)
```

```

; [X -> Number] [List-of X] -> X
; finds the (first) item in alox that maximizes f, that is:
; if (argmax f (list x-1 ... x-n)) = x-i,
; then (>= (f x-i) (f x-1)), (>= (f x-i) (f x-2)), and so on
(define (argmax f alox) ...)

; [X -> Number] [List-of X] -> X
; finds the (first) item in alox that minimizes f, that is:
; if (argmin f (list x-1 ... x-n)) = x-i,
; then (<= (f x-i) (f x-1)), (<= (f x-i) (f x-2)), and so on
(define (argmin f alox) ...)

```

Figure 53: ISL's abstract functions for list-processing

18.1 Existing Abstractions

ISL provides a number of abstract functions for processing natural numbers and lists. Figure 53 collects the header material for the most important ones. The first one processes natural numbers and builds lists:

```

> (build-list 3 add1)
(list 1 2 3)

```

The next three process lists and produce lists:

```

> (filter odd? (list 1 2 3 4 5))
(list 1 3 5)
> (sort (list 3 2 1 4 5) >)
(list 5 4 3 2 1)
> (map add1 (list 1 2 2 3 3 3))
(list 2 3 3 4 4 4)

```

In contrast, `andmap` and `ormap` process lists and produce a `Boolean`:

```

> (andmap odd? (list 1 2 3 4 5))
#false
> (ormap odd? (list 1 2 3 4 5))
#true

```

This kind of computation is called a *reduction* because a list of values is reduced to a single value.

Of all the functions in figure 53, `foldr` and `foldl` are the most powerful ones. Both reduce lists to values. The following two computations explain how to use the abstract examples in the headers of `foldr` and `foldl` to explain an application to `+`, `0`, and `(list 1 2 3 4 5)`:

<pre> (foldr + 0 '(1 2 3 4 5)) == (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0))))) == (+ 1 (+ 2 (+ 3 (+ 4 5)))) == (+ 1 (+ 2 (+ 3 9))) == (+ 1 (+ 2 12)) == (+ 1 14) == 15 </pre>	<pre> (foldl + 0 '(1 2 3 4 5)) == (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))) == (+ 5 (+ 4 (+ 3 (+ 2 1)))) == (+ 5 (+ 4 (+ 3 3))) == (+ 5 (+ 4 6)) == (+ 5 10) == 15 </pre>
--	---

As you can see from these calculations, `foldr` processes the list values from right to left

and `foldl` from left to right. While for functions such as `+` the direction seems to make no difference, this isn't true in general as you can see soon.

```
(define-struct address [first-name last-name street])
; Addr is (make-address String String String)
; interpretation associates a street address with a person's name

; [List-of Addr] -> String
; creates a string of first names, sorted in alphabetical order,
; separated and surrounded by blank spaces
(define (listing l)
  (foldr string-append-with-space
    " "
    (sort (map address-first-name l)
      string<?)))

; String String -> String
; concatenates two strings and prefixes with space
(define (string-append-with-space s t)
  (string-append " " s t))

(define ex0
  (list (make-address "Matthias" "Fellson" "Sunburst")
    (make-address "Robert" "Findler" "South")
    (make-address "Matthew" "Flatt" "Canyon")
    (make-address "Shriram" "Krishna" "Yellow")))

(check-expect (listing ex0) " Matthew Matthias Robert Shriram ")
```

Figure 54: Creating a list of names with abstractions

Figure 54 illustrates the power of composing the functions from figure 53. Its main function is `listing`. The purpose is to create a string from a list of addresses. More precisely, the expected result is a string that represents a sorted list of first names, separated and surrounded by blank spaces.

A moment's reflection suggests a straightforward design plan for this problem:

1. design a function that extracts the first names from the given list of `Addr`;
2. design a function that sorts these names in alphabetical order;
3. design a function that concatenates the strings from step 2.

Before you read on, you may wish to execute this plan. That is, design all three functions and then compose them in the sense of [Composing Functions](#) to obtain your own version of `listing`.

In the new world of abstractions, it becomes unnecessary to design three separate functions. Take a close look at the innermost expression of `listing` in figure 54:

```
(map address-first-name l)
```

By the purpose statement of `map`, it applies `address-first-name` to every single instance of `address` producing a list of first names as strings. Here is the immediately surrounding expression:

In mathematics such functions are called *associative*. And indeed, `+` is associative on integers and rational numbers in ISL. For inexact numbers, however, this is not true. See below.


```
(sort .. string<?))
```

The dots represent the result of the `map` expression. Since the latter supplies a list of strings, the `sort` expression produces a sorted list of first names. And that leaves us with the outermost expression:

```
(foldr string-append-with-space " " ..)
```

This one reduces the sorted list of first names to a single string, using a function named `string-append-with-space`. With such a suggestive name, you can easily imagine now that this reduction concatenates all the strings in the desired way—even if you do not quite understand the details of how the combination of `foldr` and `string-append-with-space` works.

Exercise 218. You can design `build-list` and `foldl` with the design recipes that you know, but they are not going to be like the ones that ISL provides. For example, the design of your own `foldl` function requires a use of the list `reverse` function:

```
; [X Y -> Y] Y [List-of X] -> Y
; my-foldl works just like foldl
(check-expect (my-foldl cons '() '(a b c)) (foldl cons '() '(a b c)))
(check-expect (my-foldl / 1 '(6 3 2)) (foldl / 1 '(6 3 2)))
(define (my-foldl f e l)
  (foldr f e (reverse l)))
```

Design `my-build-list`, which works just like `build-list`. **Hint** Recall the `add-at-end` function from [exercise 175](#). **Note** [Accumulators](#) covers the concepts that you need to design these functions properly. ■

18.2 Local Function Definitions

Take a second look at [figure 54](#). The `string-append-with-space` function clearly plays a subordinate role and has no use outside of this narrow context. Almost all programming languages support some way for stating this relationship as a part of the program. The idea is called a *local definition*, sometimes also a *private definition*. In ISL, `local` expressions introduce locally defined functions, variables, and structure types, and this section introduces the mechanics of `local`.

Here is a revision of [figure 54](#) using `local`:

```
; [List-of Addr] -> String
; creates a string of first names, sorted in alphabetical order,
; separated and surrounded by blank spaces
(define (listing.v2 l)
  (local (; String String -> String
        ; concatenates two strings and prefixes with space
        (define (string-append-with-space s t)
          (string-append " " s t)))
    ; - IN -
    (foldr string-append-with-space
      " "
      (sort (map address-first-name l)
            string<?)))))
```

The body of the `listing.v2` function is now a `local` expression, which consists of two pieces: a sequence of definitions and a body expression. All local definitions are visible everywhere within the opening parenthesis of `local` and the closing one and only there.

In this example, the sequence of definitions consists of a single function definition, the one for `string-append-with-space`. The body of `local` is the body of the original `listing` function. Its reference to `string-append-with-space` is now resolved locally, that is, there is no need to look in the global sequence of definitions. Conversely, outside of the `local` expression, it is impossible to refer to `string-append-with-space`. Since there is no such reference in the original program, it remains intact and you can confirm this with the test suite.

In general, a `local` expression has this shape:

```
(local ( ... sequence of definitions ... ) body-expression)
```

The evaluation of such an expression proceeds like the evaluation of a complete program. First, the definitions are set up, which may involve the evaluation of the right-hand side of a constant definition. Just as with the top-level definitions that you know and love, the definitions in a `local` expression may refer to each other. They may also refer to parameters of the surrounding function. Second, the `body-expression` is evaluated. The result of `body-expression` is the result of the `local` expression. It is often helpful to separate the `local` definitions from the body-expression with a comment; we use `- IN -` because the word suggests that the definitions are available in a certain expression.

```
; [List-of Number] [Number Number -> Boolean] -> [List-of Number]
(define (sort-cmp alon0 cmp)
  (local (; [List-of Number] -> [List-of Number]
        ; produces a variant of alon sorted by cmp
        (define (isort alon)
          (cond
            [(empty? alon) '()]
            [else (insert (first alon) (isort (rest alon)))]))

        ; Number [List-of Number] -> [List-of Number]
        ; inserts n into the sorted list of numbers alon
        (define (insert n alon)
          (cond
            [(empty? alon) (cons n '())]
            [else (if (cmp n (first alon))
                      (cons n alon)
                      (cons (first alon)
                            (insert n (rest alon)))]))

        (isort alon0)))
```

Figure 55: Local, interconnected function definitions

Let's take a second look at the above example. Now that we know that `local` may introduce several local definitions, we can further improve the readability of the `listing` function:

```
(define (listing.v3 l)
  (local (; String String -> String
        ; concatenates two strings and prefixes with space
        (define (string-append-with-space s t)
          (string-append " " s t))
        (define first-names (map address-first-name l))
        (define sorted-names (sort first-names string<?)))
    ; - IN -
```

```
(foldr string-append-with-space " " sorted-names)))
```

The most visually appealing difference concerns the body of the `local` expression. It is no longer a three-line expression but a single line and its purpose is clearly expressed, especially because it uses an aptly named variable to refer to the sorted list of names. An inspection of the definition of `sorted-names` shows another such uses of a `local` definition: `first-names`. In general, when you see a multi-line, deeply nested expression you may wish to formulate `local` definitions with properly named variables to bring across what the expressions compute. Future readers will appreciate it because they can often comprehend the code just by looking at the variable names and the body of the `local` expression.

For a second example, consider [figure 55](#). The organization of this definition tells the reader that `sort-cmp` needs two auxiliary functions: `isort` and `insert`. Note the reference to `cmp` in the latter, which tells the reader that the comparison function remains the same for the entire sorting process.

By making `insert` local, it also becomes impossible to abuse `insert`. Re-read its purpose statement. The adjective “sorted” means that a program should use `insert` only if the second argument is already sorted. As long as `insert` is defined at the top-level, nothing guarantees that `insert` is always used properly. Once its definition is local to the `sort-cmp` function, the proper use is guaranteed—because `isort` applies `insert` to a sorted version of `(rest alon)`.

Exercise 219. Use a `local` expression to organize the functions for drawing a polygon in [figure 46](#). If a globally defined functions is widely useful, do not make it local. ■

Exercise 220. Use a `local` expression to organize the functions for rearranging words from [Rearranging Words](#). ■

For a second example, let us take a look at the `inf` function from [exercise 205](#):

```
; Nelon -> Number
; determines the smallest number on l
(define (inf l)
  (cond
    [(empty? (rest l)) (first l)]
    [else
     (local ((define smallest-in-rest (inf (rest l))))
       (cond
         [(< (first l) smallest-in-rest) (first l)]
         [else smallest-in-rest])]))])
```

Here the `local` expression shows up in the middle of a `cond` expression. It also doesn’t define a function but a variable whose initial value is the result of a natural recursion. Now recall that the evaluation of a `local` expression evaluates the definitions once and for all before the body is evaluated. What this means here is that `(inf (rest l))` is evaluated once, but the body of the `local` expression refers to the result twice. Put differently, the use of `local` saves the re-evaluation of `(inf (rest l))` at each stage in the computation. To confirm this insight, re-evaluate the above version of `inf` on the inputs suggested in [exercise 205](#).

Exercise 221. Consider the following function definition:

```
; Inventory -> Inventory
; creates an Inventory from an-inv for all
; those items that cost less than $1
```

```
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) '()]
    [else (cond
              [(<= (ir-price (first an-inv)) 1.0)
               (cons (first an-inv) (extract1 (rest an-inv)))]
              [else (extract1 (rest an-inv))])]))
```

Both clauses in the nested `cond` expression extract the first item from `an-inv` and both compute `(extract1 (rest an-inv))`. Use a `local` expression to name the repeated expressions. Does this help increase the speed at which the function computes its result? Significantly? A little bit? Not at all? ■

Exercise 222. The `sort>` function consumes a list of numbers and produces a sorted version:

```
; Lon -> Lon
; constructs a list from the items in l in descending order
(define (sort> l0)
  (local (; Lon -> Lon
          (define (sort l)
            (cond
              [(empty? l) '()]
              [else (insert (first l) (sort (rest l)))])))

    ; Number Lon -> Lon
    (define (insert an l)
      (cond
        [(empty? l) (list an)]
        [else
         (cond
          [(> an (first l)) (cons an l)]
          [else (cons (first l) (insert an (rest l))])]))))
  (sort l0)))
```

Create a test suite for `sort>`.

Design the function `sort-<`, which sorts lists of numbers in **ascending** order.

Create `sort-a`, which abstracts `sort>` and `sort-<`. It consumes the comparison operation in addition to the list of numbers. Define versions `sort>` and `sort-<` in terms of `sort-a`.

Use `sort-a` to define a function that sorts a list of strings by their lengths, both in descending and ascending order.

Later we will introduce several different ways to sort lists of numbers, all of them faster than `sort-a`. If you then change `sort-a`, all uses of `sort-a` will benefit. ■

18.3 ... Add Expressive Power

The third and last example illustrates how `local` adds expressive power to BSL and BSL+. **Finite State Machines** presents the design of a world program that simulates how a finite state machine recognizes sequences of key strokes. While the data analysis leads in a natural manner to the data definitions in [figure 49](#), an attempt to design the main function of the world program fails. Specifically, even though the given finite state machine remains the same over the course of the simulation, the state of the world must include it so that the program can transition from one state to the next when the player

presses a key.

```

; FSM FSM-State -> FSM-State
; match the keys pressed by a player with the given FSM

(define (simulate fsm s0)
  (local (; State of the World: FSM-State

          ; FSM-State KeyEvent -> FSM-State
          ; finds the next state in the transition table of fsm0
          (define (find-next-state s key-event)
            (find fsm s)))

    ; NOW LAUNCH THE WORLD
    (big-bang s0
      [to-draw state-as-colored-square]
      [on-key find-next-state])))

; FSM-State -> Image
; renders current state as colored square
(define (state-as-colored-square s)
  (square 100 "solid" s))

; FSM FSM-State -> FSM-State
; finds the state matching current in the given transition table (fsm)
(define (find transitions current)
  (cond
    [(empty? transitions) (error "not found")]
    [else
     (local ((define s (first transitions)))
       (if (state=? (transition-current s) current)
           (transition-next s)
           (find (rest transitions) current)))]))

```

Figure 56: Power from local function definitions

Figure 56 shows an ISL solution to the problem. It uses `local` function definitions and can thus equate the state of the world with the current state of the finite state machine. Specifically, `simulate` locally defines the key event handler, which consumes only the current state of the world and the `KeyEvent` that represents the player's key stroke. Because this locally defined function can refer to the given finite state machine `fsm`, it is possible to find the next state in the transition table—even though the transition table—finite state machine representation—is **not** an argument to this function.

As the figure also shows, all other functions are defined in parallel to the main function. This includes the function `find`, which performs the actual search in the transition table. The key improvement over BSL is that a locally defined function can reference **both** parameters to the function **and** globally defined auxiliary functions.

In short, this program organization signals to a future reader exactly the insights that the data analysis stage of the design recipe for world programs finds. First, the given representation of the finite state machine remains unchanged. Second, what changes over the course of the simulation is the current state of the finite machine.

The lesson is that the chosen programming language affects a programmer's ability to express solutions, and a future reader's ability to recognize the design insight of the original creator.

18.4 Using Abstractions, by Examples

Now that you understand `local`, you are ready to use the abstractions from [figure 53](#). Let us look at some examples, starting with this one:

Sample Problem: Design the function `add-3-to-all`. The function consumes a list of `Posns` and adds 3 to the x coordinates of each of them.

If we follow the design recipe and take the problem statement as a purpose statement, we can quickly step through the first three steps:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
              (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop) '())
```

While you can run the program, doing so signals a failure in the one test case because the function returns the default value `'()`.

At this point, we stop and ask what kind of function we are dealing with. Clearly, `add-3-to-all` is clearly a list-processing function. The question is whether it is like any of the functions in [figure 53](#). The signature tells us that `add-3-to-all` is a list-processing function that consumes and produces a list. In [figure 53](#), we have several functions with similar signatures: `map`, `filter`, and `sort`.

The purpose statement and example also tell you that `add-3-to-all` deals with each `Posn` separately and assembles the results into a single list. Some reflection says that also confirms that the resulting list contains as many items as the given list. All this thinking points to one function—`map`—because the point of `filter` is to drop items from the list and `sort` has an extremely specific purpose.

Here is `map`'s signatures again:

```
; [X -> Y] [List-of X] -> [List-of Y]
```

It tells us that `map` consumes a function from X to Y and a list of Xs. Given that `add-3-to-all` consumes a list of `Posns`, we know that X stands for `Posn`. Similarly, `add-3-to-all` is to produce a list of `Posns`, and this means we replace Y with `Posn`.

From the analysis of the signature we conclude that `map` could do the job of `add-3-to-all` if we can find an appropriate function from `Posns` to `Posns`. With `local`, we can express this idea as a template for `add-3-to-all`:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
              (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop)
  (local (; Posn -> Posn
          ; ...
          (define (a-fun-from-posn-to-posn p)
            ... p ...))
    (map a-fun-from-posn-to-posn lop)))
```

Doing so reduces the problem to defining a function on `Posns`.

Given the example for `add-3-to-all` and the abstract example for `map`, you can actually imagine how the evaluation proceeds:

```
(add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
=
(map a-fun (list (make-posn 30 10) (make-posn 0 0)))
=
(list (a-fun (make-posn 30 10)) (a-fun (make-posn 0 0)))
```

And that shows how `a-fun` is applied to every single `Posn` on the given list, meaning it is its job to add 3 to the `x` coordinate.

From here, it is straightforward to wrap up the definition:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
              (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop)
  (local (; Posn -> Posn
          ; adds 3 to the x coordinate of the given Posn
          (define (add-3-to-one p)
            (make-posn (+ (posn-x p) 3) (posn-y p))))
    (map add-3-to-one lop)))
```

We chose `add-3-to-one` as the name for the local function because the name tells you what it computes. It adds 3 to (the `x` coordinate of) one `Posn`—as opposed to `add-3-to-all`, which adds 3 to **all** given `Posns`. It is `map`'s task to apply `add-3-to-one` to each of the `Posns` on `lop` and to assemble the results into one list.

You may think now that using abstractions is hard. Keep in mind, though, that this first example spells out every single detail and that it does so because we wish to teach you how to pick the proper abstraction. Let us take a look at a second example a bit more quickly:

Sample Problem: Design a function that eliminates all `Posns` from a list that have a `y` coordinate of larger than 100.

The first two steps of the design recipe yield this:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y coordinate is larger than 100

(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
              (list (make-posn 0 60)))

(define (keep-good lop) '())
```

By now you may have guessed that this function is like `filter` whose purpose is to separate “good” items from ones.

With `local` thrown in, the next step is also straightforward:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y coordinate is larger than 100
```

```
(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
               (list (make-posn 0 60)))

(define (keep-good lop)
  (local (; Posn -> Boolean
          ; should this Posn stay on the list
          (define (good? p) #true))
    (filter good? lop)))
```

The `local` function definition introduces the helper function needed for `filter` and the body of the `local` expression applies `filter` to this local function and the given list. The local function is called `good?` because `filter` retains all those items of `lop` for which `good?` produces `#true`.

Before you read on, analyze the signature of `filter` and `keep-good` and determine why the helper function consumes individual `Posns` and produces `Booleans`.

Putting all of our ideas together yields this definition:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y coordinate is larger than 100

(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
               (list (make-posn 0 60)))

(define (keep-good lop)
  (local (; Posn -> Posn
          ; should this Posn stay on the list
          (define (good? p)
            (not (> (posn-y p) 100))))
    (filter good? lop)))
```

Explain the definition of `good?` and simplify it.

Before we spell out a design recipe, let us deal with one more example:

Sample Problem: Design a function that determines whether any of a list of `Posns` is close to some given position `pt` where “close” means a distance of at most 5 pixels.

This problem clearly consists of two distinct parts: one concerns processing the list and another one calls for a function that determines whether the distance between a point and `pt` is “close.” Since this second part is unrelated to the reuse of abstractions for list traversals, we assume the existence of an appropriate function:

```
; Posn Posn Number -> Boolean
; is the distance between p and q less than d
(define (close-to p q d) ...)
```

You should complete this definition on your own.

As required by the problem statement, the function consumes two arguments—the list of `Posns` and the “given” point `pt`—and produces a `Boolean`:

```
; [List-of Posn] Posn -> Boolean
; is any Posn on lop close to pt

(check-expect (close? (list (make-posn 47 54) (make-posn 0 60))
```



```

                (make-posn 50 50))
      #true)

(define (close? lop pt) #false)

```

The latter point immediately differentiates this example from the preceding ones.

The **Boolean** range also gives away a clue with respect to [figure 53](#). Only two functions in this list produce **Boolean** values—**andmap** and **ormap**—and they must be primary candidates for defining `close?`'s body. While the explanation of **andmap** says that some property must hold for every item on the given list, the purpose statement for **ormap** tells us that it looks for only **one** such item. Given that `close?` just checks whether one of the **Posns** is close to `pt`, we should try **ormap** first.

Let us apply our standard “trick” of adding a **local** whose body uses the chosen abstraction on some locally defined function and the given list argument:

```

; [List-of Posn] Posn -> Boolean
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; ...
          (define (is-one-close? p)
            ...))
    (ormap close-to? lop)))

```

Following the description of **ormap**, the local function consumes one item of the list at a time. This accounts for the **Posn** part of its signature. Also, the local function is expected to produce **#true** or **#false**, and **ormap** checks these results until it finds **#true**.

Here is a comparison of the signature of **ormap** and `close?`, starting with the former:

```

; [X -> Boolean] [List-of X] -> Boolean

```

In our case, the list argument is a list of **Posns**. Hence **X** stands for **Posn**, which explains what `is-one-close?` consumes. Furthermore, it determines that the result of the local function must be **Boolean** so that it can work as the first argument to **ormap**.

The rest of the work requires just a bit more thinking. While `is-one-close?` consumes one argument—a **Posn**—the `close-to` function consumes three: two **Posns** and a “tolerance” value. While the argument of `is-one-close?` is one of the two **Posns**, it is also obvious that the other one is `pt`, the argument of `close?` itself. Naturally the “tolerance” argument is `5`, as stated in the problem:

```

; [List-of Posn] -> Boolean
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; is one shot close to pt
          (define (is-one-close? p)
            (close-to p pt CLOSENESS)))
    (ormap is-one-close? lop)))

(define CLOSENESS 5)

```

Note two properties of this definition. First, we stick to the rule that constants deserve definitions. Second, the reference to `pt` in `is-one-close?` signals that this **Posn** stays the same for the entire traversal of `lop`.

18.5 Designing with Abstractions

Three examples suffice for formulating a design recipe for reusing abstractions:

1. Step 1 is **to follow the design recipe for functions** for three steps. Specifically, you should distill the problem statement into a signature, a purpose statement, an example, and a stub definition.

Consider the problem of defining a function that places small red circles on a 200 by 200 canvas for a given list of `Posns`. The first three steps design recipe yield this much:

```
; [List-of Posn] -> Image
; adds the Posns on lop to the empty scene

(check-expect (dots (list (make-posn 12 31)))
              (place-image DOT 12 32 BACKGROUND))

(define (dots lop)
  BACKGROUND)

(define BACKGROUND (empty-scene 200 200))
(define DOT (circle 5 "solid" "red"))
```

2. Next we exploit the signature and purpose statement to find a matching abstraction. **To match** means to pick an abstraction whose purpose is more general than the one for the function to be designed; it also means that the signatures are related. It is often best to start with the desired output and to find an abstraction that has the same or a more general output.

For our running example, the desired output is an `Image`. While none of the available abstractions produces an image, two of them have a variable to the right of `->`:

```
; foldr : [X Y -> Y] Y [List-of X] -> Y
; foldl : [X Y -> Y] Y [List-of X] -> Y
```

meaning we can plug in any data collection we want. If we do use `Image`, the signature on the left of `->` demands a helper function that consumes an `X` and an `Image` and it produces an `Image`. Furthermore, since the given list contains `Posns`, `X` does stand for the `Posn` collection.

3. Write down **a template**. For the reuse of abstractions a template uses `local` for two different purposes. The first one is to note which abstraction to use and how in the body of the `local` expression. The second one is to write down a stub for the helper function: its signature, its purpose (optionally), and its header. Keep in mind that the signature comparison in the preceding step suggests most of the signature for the auxiliary function.

Here is what this template looks like for our running example if we choose `foldr`:

```
(define (dots lop)
  (local (; Posn Image -> Image
        (define (add-one-dot p scene) ...))
    (foldr add-one-dot BACKGROUND lop)))
```

The `foldr` description calls for a “base” `Image` value, to be used if or when the list is empty. In our case, we clearly want the empty canvas for this case. Otherwise, `foldr`

uses a helper function and traverses the list of `Posns`.

4. Finally, it is time to define the helper function inside `local`. In most cases, this auxiliary function consumes relatively simple kinds of data, like those encountered in `Fixed-Size Data`. You know how to design those in principle. The only difference is that now you may not only use the function's arguments and global constants but also the arguments of the surrounding function.

In our running example, the purpose of the helper function is to add one dot to the given scene, which you can (1) guess or (2) derive from the example:

```
(define (dots lop)
  (local (; Posn Image -> Image
          ; adds a DOT at p to scene
          (define (add-one-dot p scene)
            (place-image DOT (posn-x p) (posn-y p) scene)))
    (foldr add-one-dot BACKGROUND lop)))
```

5. The last step is **to test** the definition in the usual manner.

For abstract functions, it is occasionally possible to use the abstract example of their purpose statement to confirm their workings at a more general level. You may wish to use the abstract example for `foldr` to confirm that `dots` does add one dot after another to the background scene.

In the third step, we picked `foldr` without further ado. Experiment with `foldl` to see how it would help complete this function. Functions like `foldl` and `foldr` are well-known and are spreading in usage in various forms. Becoming familiar with them is a good idea and the next section will help.

18.6 Exercises and Projects

Each of the following set of exercises suggests small practice problems for specific abstractions in ISL.

Exercise 223. Use `map` to define the function `convert-euro`, which converts a list of U.S. dollar amounts into a list of euro amounts based on an exchange rate of 1.22 euro for each dollar.

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hands at `translate`, a function that translates a list of `Posns` into a list of list of pairs of numbers, i.e., `[List-of [list Number Number]]`. ■

Exercise 224. An inventory record specifies the name of an item, a description, the acquisition price, and the recommended sales price.

Define a function that sorts a list of inventory records by the difference between the two prices. ■

Exercise 225. Define `eliminate-exp`, which consumes a number, `ua` and a list of inventory records, and it produces a list of all those structures whose sales price is below `ua`.

Then use `filter` to define `recall`, which consumes the name of an inventory item,

called `ty`, and a list of inventory records and which produces a list of inventory records that do not use the name `ty`.

In addition, define `selection`, which consumes two lists of names and selects all those from the second one that are also on the first. ■

Exercise 226. Use `build-list` to define functions that

1. creates the list $0 \dots (n - 1)$ for any natural number n ;
2. creates the list $1 \dots n$ for any natural number n ;
3. creates the list `(list 1 1/10 1/100 ...)` of n numbers for any natural number n ;
4. creates the list of the first n even numbers;
5. creates a list of lists of `0` and `1` in a diagonal arrangement, e.g.,

```
(equal? (diagonal 3)
  (list
    (list 1 0 0)
    (list 0 1 0)
    (list 0 0 1)))
```

Finally, define `tabulate` from [exercise 212](#) using `build-list`. ■

Exercise 227. Use `ormap` to define `find-name`. The function consumes a name and a list of names. It determines whether any of the names on the latter are equal to or an extension of the former.

With `andmap` you can define a function that checks all names on a list of names start with the letter "a".

Should you use `ormap` or `andmap` to define a function that ensures that no name on some list exceeds some given width? ■

Exercise 228. Recall that the `append` function in ISL concatenates the items of two lists or, equivalently, replaces '()' at the end of the first list with the second list:

```
(equal? (append (list 1 2 3) (list 4 5 6 7 8))
  (list 1 2 3 4 5 6 7 8))
```

Use `foldr` to define `append-from-fold`. What happens if you replace `foldr` with `foldl`?

Now use one of the fold functions to define functions that compute the sum and the product, respectively, of a list of numbers.

With one of the fold functions, you can define a function that horizontally composes a list of `Images`. **Hints** (1) Look up `beside` and `empty-image`. Can you use the other fold function? Also define a function that stacks a list of images vertically. (2) Check for `above` in the libraries. ■

Exercise 229. The fold functions are so powerful that you can define almost any list-processing functions with them. Use `fold` to define `map`. ■

Exercise 230. Use existing abstractions to define the `prefixes` and `suffixes` functions from [exercise 172](#). Ensure that they pass the same tests as the original function. ■

Now that you have some experience with the existing list-processing abstractions in ISL,

it is time to tackle some real projects. Specifically, we are looking for two kinds of improvements. First, inspect the game programs for functions that traverse lists. For these functions, you already have signatures, purpose statements, tests, and working definitions that pass the tests. Change the definitions to use abstractions from [figure 53](#). Second, also inspect the game programs for opportunities to abstract. Indeed, you might be able to abstract across games and provide a framework of functions that helps you write additional game programs. The following exercises supply specific suggestions for the projects mentioned in this book. If you are using this book for a course, you may have had to deal with other non-trivial exercises; adapt the exercises to those projects.

Exercise 231. [Full Space War](#) spells out a game of space war. In the basic version, a UFO descends and a player defends with a tank. One additional suggestion is to equip the UFO with charges that it can drop at the tank; the tank is destroyed if a charge comes close enough.

Inspect the code of your project for places where it can benefit from existing abstraction, e.g., processing lists of shots or charges.

Once you have simplified the code with the use of existing abstractions look for opportunities to create abstractions. Consider moving lists of objects as one example where abstraction may pay off. ■

Exercise 232. [Feeding Worms](#) explains how another one of the oldest computer games work. The game features a worm that moves at a constant speed in a player-controlled direction. When it encounters food, it eats the food and grows. When it runs into the wall or into itself, the game is over.

This project can also benefit from the abstract list-processing in ISL. Look for places to use them and replace existing code one piece at a time, relying on the tests to ensure the program works. ■

19 Nameless Functions

The use of abstract functions employs functions as arguments. On some occasions, these functions are existing primitive functions, library functions, or functions that you defined. For example,

- `(build-list n add1)` creates `(list 1 ... n)`;
- `(foldr cons another-list one-list)` concatenates the items on `one-list` and `another-list` into a single list; and
- the expression `(foldr above a-list-of-images)` stacks the images on the given list.

On other occasions, the uses of abstract functions require the definition of simple helper functions, whose definition requires often a single line. Consider the following use of `filter`:

```
; [List-of IR] String -> Boolean
(define (find aloir threshold)
  (local (; IR -> Boolean
          (define (acceptable? ir)
            (<= (ir-price ir) threshold)))
    (filter acceptable? aloir)))
```

It finds all items on the inventory list whose price is below `threshold`. The auxiliary function is nearly trivial yet its definition takes up three lines and does not even have a purpose statement.

This situation calls for an improvement to the language. Programmers should be able to create such small and insignificant functions without much effort. This section presents `lambda`, the new feature, and introduces “Intermediate Student Language with `lambda`.” This language, short: ISL+, is an extension of ISL with `lambda`. The history of `lambda` is interesting and intimately involved with the early history of programming and programming language design. When you have time, you should chase down the origins of `lambda`.

The first two sections focus on the design of abstractions using `lambda`. The remaining sections use `lambda` and `local` to introduce some basic programming language concepts, namely, scope and abbreviations. In addition, they indicate with one use of `lambda` that the idea of functions as values has additional ramifications for program design.

19.1 Functions from `lambda`

A `lambda` expression creates a nameless function. Its syntax is straightforward:

```
((lambda (variable-1 ... variable-N) expression))
```

Its distinguishing characteristic is the keyword `lambda`. The keyword is followed by a sequence of variables, enclosed in a pair of parentheses. The last piece is an arbitrary expression, and it computes the result of the function when it is given values for its parameters.

Here are three simple examples, all of which consume one argument:

1. `(lambda (x) (expt 10 x))`, which assumes that the argument is a number and computes the exponent of 10 to the number;
2. `(lambda (name) (string-append "hello, " name ", how are you?"))`, which consumes a string and creates a greeting with `string-append`; and
3. `(lambda (ir) (<= (ir-price ir) threshold))`, which is a function on an IR struct that extracts the price field from the structure and compares it with some threshold value.

The last example should recall the `filter` example from above.

To use a function created from `lambda` expression, you can apply it to the correct number of arguments. Abstractly, you evaluate such expression according to the following law,

```
((lambda (x-1 ... x-n) exp)
 val-1 ... val-n)
= exp ; with free occurrences of x-1 replaced by val-1, etc.
```

That is, the application of a `lambda` expression proceeds just like that of an ordinary function. We replace the parameters of the function with the actual argument values and compute the value of the function body. Concretely, it works as expected:

```
> ((lambda (x) (expt 10 x)) 2)
100
> ((lambda (name rst) (string-append name ", " rst)) "Robby" "etc.")
```

In DrRacket, choose “Intermediate Student Language with `lambda`” from the “How to Design Programs” submenu in the “Language” menu.

Stated like this, the law of function application is known as

```
"Robby, etc."
> ((lambda (ir) (<= (ir-price ir) threshold)) (make-ir "bear" 10))
#true
```

Note how the second sample function requires two arguments and that the last example assumes a definition for `threshold` in the definitions window such as this one:

```
(define threshold 20)
```

The result of the last example is `#true` because the price field of the inventory record contains `10` and `10` is less than `20`.

Now the important point is that these nameless functions can be used wherever a function is required, including with the abstractions from figure 53:

```
> (map (lambda (x) (expt 10 x))
      '(1 2 3))
(list 10 100 1000)
> (foldl (lambda (name rst) (string-append name ", " rst)) "etc."
      '("Matthew" "Robby"))
"Robby, Matthew, etc."
> (filter (lambda (ir) (<= (ir-price ir) threshold))
      (list (make-ir "bear" 10) (make-ir "doll" 33)))
(list (ir ...))
```

Once again, the last example assumes a definition for `threshold`.

Exercise 233. Decide which of the following phrases are legal `lambda` expressions:

1. `(lambda (x y) (x y y))`
2. `(lambda () 10)`
3. `(lambda (x) x)`
4. `(lambda (x y) x)`
5. `(lambda x 10)`

Explain why they are legal or illegal. If in doubt, experiment in the interactions area. ■

Exercise 234. Calculate the result of the following expressions:

1.

```
((lambda (x y)
  (+ x (* x y)))
  1 2)
```
2.

```
((lambda (x y)
  (+ x
    (local ((define z (* y y)))
      (+ (* 3 z)
        (/ 1 x)))))
  1 2)
```
3.

```
((lambda (x y)
  (+ x
    ((lambda (z)
      (+ (* 3 z)
        (/ 1 z)))
     (* y y))))
  1 2)
```

Check your results in DrRacket. ■

Exercise 235. Write down a `lambda` expression that

1. consumes a number and decides whether it is less than 10;
2. consumes two numbers, multiplies them, and turns the result into a string;
3. consumes an `Posn` `p` and a rectangular `Image` and adds a red 3-pixel dot to the image at `p`;
4. consumes an inventory record and compares them by price; and
5. consumes a natural number and produces 0 if it is even and 1 if it is odd.

Demonstrate how to use these functions in the interactions area. ■

19.2 Abstracting with `lambda` I

The two interactive examples for `filter` suggest a simplification for our `find` function:

```
; [List-of IR] String -> Boolean
; VERSION 2
(define (find aloir threshold)
  (filter (lambda (ir) (<= (ir-price ir) threshold)) aloir))
```

Instead of introducing a named, local function, this second version uses `lambda` to create the auxiliary function and immediately hands it to `filter` to extract all “good” inventory records from the list. Assuming `find` came with a test suite, you can immediately ensure that this revision passes the same tests as the original version.

Although it may take you a bit to get used to `lambda` notation, you will soon notice that `lambda` makes such short functions much more readable than `local` definitions. Indeed, you will find that you can adapt step 4 of the design recipe from [Designing with Abstractions](#) to use `lambda` instead of `local`. Consider the running example from that section. Its template based on `local` is this:

```
(define (dots lop)
  (local (; Posn Image -> Image
        (define (add-one-dot p scene) ...))
    (foldr add-one-dot BACKGROUND lop)))
```

If you spell out the parameters so that their names include signatures, you can easily pack all the information from `local` into a single `lambda`:

```
(define (dots lop)
  (foldr (lambda (one-posn scene) ...) BACKGROUND lop))
```

From here, you should be able to complete the definition as well as from the original template:

```
(define (dots lop)
  (foldr (lambda (one-posn scene)
    (place-image DOT (posn-x one-posn) (posn-y one-posn) scene))
    BACKGROUND lop))
```

Let us illustrate `lambda` some more via examples from [Using Abstractions, by Examples](#):

- the purpose of the first function is to add 3 to each x coordinate on a given list of [Posns](#):

```
; [List-of Posn] -> [List-of Posn]
; (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
; should deliver
; (list (make-posn 33 10) (make-posn 3 0))

(define (add-3-to-all lop)
  (map (lambda (p) (make-posn (+ (posn-x p) 3) (posn-y p))) lop))
```

Because `map` expects a function of one argument, we use `(lambda (p) ...)`. The function then de-structures the given [Posn](#), adds 3 to the x coordinate, and repackages the data into a [Posn](#).

- the second one eliminates [Posns](#) whose y coordinate is larger than 100:

```
; [List-of Posn] -> [List-of Posn]
; (keep-good (list (make-posn 0 110) (make-posn 0 60)))
; should deliver
; (list (make-posn 0 60))

(define (keep-good lop)
  (filter (lambda (p) (<= (posn-y p) 100)) lop))
```

Here we know that `filter` needs a function of one argument that produces a [Boolean](#). First, the `lambda` function extracts the y coordinate from the [Posn](#) to which `filter` applies the function. Second, it checks whether it is less than or equal to 100, the desired limit.

- and the third one determines whether any [Posn](#) on `lop` is close to some given point:

```
; [List-of Posn] -> Boolean
; (close? (list (make-posn 3 4) (make-posn 9 9)) (make-posn 0 0))
; should be
; #true

(define (close? lop pt)
  (ormap (lambda (p) (close-to p pt CLOSENESS)) lop))

(define CLOSENESS 5)
```

Like the preceding two examples, `ormap` is a function that expects a function of one argument and applies this functional argument to every item on the given list. If any result is `#true`, `ormap` returns `#true`, too; if all results are `#false`, `ormap` produces `#false`.

It is probably best to compare the definitions from [Using Abstractions, by Examples](#) and the definitions above side by side. When you do so, you should notice how easy the transition from `local` to `lambda` is and how concise the `lambda` version is in comparison to the `local` version. Thus, if you are ever in doubt, design with `local` first and then convert this tested version into one that uses `lambda`. Keep in mind, however, that `lambda` is not a cure-all. The locally defined function comes with a name that explains its purpose and, if it is long, the use of an abstraction with a named function is much easier to understand than one with a large `lambda`.

The following exercises request that you solve the problems from [Exercises and Projects](#)

with `lambda` in ISL+ .

Exercise 236. Use `map` to define the function `convert-euro`, which converts a list of U.S. dollar amounts into a list of euro amounts based on an exchange rate of 1.22 euro for each dollar.

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hands at `translate`, a function that translates a list of `Posns` into a list of list of pairs of numbers, i.e., `[List-of [list Number Number]]`. ■

Exercise 237. An inventory record specifies the name of an inventory item, a description, the acquisition price, and the recommended sales price.

Define a function that sorts a list of inventory records by the difference between the two prices. ■

Exercise 238. Use `filter` to define `eliminate-exp`. The function consumes a number, `ua` and a list of inventory records (containing name and price), and it produces a list of all those structures whose acquisition price is below `ua`.

Then use `filter` to define `recall`, which consumes the name of an inventory item, called `ty`, and a list of inventory records and which produces a list of inventory records that do not use the name `ty`.

In addition, define `selection`, which consumes two lists of names and selects all those from the second one that are also on the first. ■

Exercise 239. Use `build-list` to define functions that

1. creates the list `0 ... (n - 1)` for any natural number `n`;
2. creates the list `1 ... n` for any natural number `n`;
3. creates the list `(list 1 1/10 1/100 ...)` of `n` numbers for any natural number `n`;
4. creates the list of the first `n` even numbers;
5. creates a list of lists of `0` and `1` in a diagonal arrangement, e.g.,

```
(equal? (diagonal 3)
  (list
    (list 1 0 0)
    (list 0 1 0)
    (list 0 0 1)))
```

■

Exercise 240. Use `ormap` to define `find-name`. The function consumes a name and a list of names. It determines whether any of the names on the latter are equal to or an extension of the former.

With `andmap` you can define a function that checks all names on a list of names start with the letter `"a"`.

Should you use `ormap` or `andmap` to define a function that ensures that no name on some list exceeds some given width? ■

Exercise 241. Recall that the `append` function in ISL concatenates the items of two lists or, equivalently, replaces `()` at the end of the first list with the second list:

```
(equal? (append (list 1 2 3) (list 4 5 6 7 8))
        (list 1 2 3 4 5 6 7 8))
```

Use `foldr` to define `append-from-fold`. What happens if you replace `foldr` with `foldl`?

Now use one of the fold functions to define functions that compute the sum and the product, respectively, of a list of numbers.

With one of the fold functions, you can define a function that horizontally composes a list of `Images`. **Hints** (1) Look up `beside` and `empty-image`. Can you use the other fold function? Also define a function that stacks a list of images vertically. (2) Check for `above` in the libraries. ■

Exercise 242. The fold functions are so powerful that you can define almost any list-processing functions with them. Use `fold` to define `map-from-fold`, which simulates `map`. ■

19.3 `lambda`, Technically

One way to understand `lambda` is to view it as an abbreviation for a `local` expression. For example,

```
(lambda (x)
  (+ (sin x) x (* 3 x)))
```

is short for

```
(local ((define some-random-name
          (lambda (x)
            (+ (sin x) x (* 3 x)))))
  some-random-name)
```

This “trick” works in general as long as `some-random-name` does not appear in the body of the function.

What this means is that `lambda` creates a function without a name, an anonymous function. This function is a value like any other value. The name of the function is irrelevant. The only interesting parts are the function parameters—the sequence of names in parentheses—and the function body—the expression in the third position.

This insight is also the missing piece that connects constant definitions and function definitions. Instead of viewing function definitions as given, we can take `lambdas` as the primitive concept and say that a function definition abbreviates a plain constant definition with a `lambda` expression on the right-hand side.

Here is a concrete example:

```
(define (f x) ; the plain version
  (+ (sin x) x (* 3 x)))
```

This function definition is just short for

```
(define f ; the lambda version
  (lambda (x)
    (+ (sin x) x (* 3 x))))
```

The `lambda` on the right-hand side creates a function of one argument. It is `define` that actually gives the `lambda` expression a name, and that is mostly relevant for recursive functions.

You may wish to test this explanation with experiments in DrRacket. Try the above example. Also add

```
; Number -> Boolean
(define (compare x)
  (= (f-plain x) (f-lambda x)))
```

to the definitions area after renaming the first `f` to `f-plain` and the second one to `f-lambda`. Run

```
(compare (random 100000))
```

a few times to make sure the two functions agree on all kinds of random inputs.

You actually do not need `define` to create recursive functions; `lambda` is enough. If you are curious, read up on Church's Y combinator.

19.4 Abstracting with `lambda` II

Because functions are first-class values in ISL+, we may think of them as another form of data and use them for data representation. This section provides a taste of this idea; the next few chapters do not rely on it. Its title uses “abstracting” because people consider data representations that use functions as abstract.

As always, we start from a representative problem:

Sample Problem: Navy strategists represent fleets of ships as rectangles (the ships themselves) and circles (their weapons' reach). The coverage of a fleet of ships is the combination of all these shapes. Design a data representation for rectangles, circles, and combinations of shapes. Then design a function that determines whether some point is within a shape.

The problem comes with all kinds of concrete interpretations, which we leave out here. A slightly more complex version was the subject of a programming competition in the mid-1990s run by Yale University on behalf of the US Department of Defense.

One mathematical approach considers shapes as predicates on points. That is, a shape is a function that maps a Cartesian point to a Boolean value. Let's translate these English words into a data definition:

```
; Shape is a function:
;   [Posn -> Boolean]
; interpretation if s is a shape and p a Posn, (s p) produces
; #true if the given Posn is inside of s, #false otherwise
```

Its interpretation part is extensive because this data representation is so unusual. Such an unusual representation calls for an immediate exploration with examples. We delay this step for a moment, however, and instead define a function that checks whether a point is inside some shape:

```
; Shape Posn -> Boolean
(define (inside? s p)
  (s p))
```

Doing so is straightforward because of the given interpretation. It also turns out that it

This problem is also solvable with a self-referential data representation that says a shape is either a circle, a rectangle, or a combination of two shapes. See the next part

simpler than creating examples and, surprisingly, the function is helpful for formulating data examples.

Stop! Explain how and why `inside?` works.

Now let us return to the problem of elements of `Shape`. Here is a simplistic element of the class:

```
; Posn -> Boolean
(lambda (p) (and (= (posn-x p) 3) (= (posn-y p) 4)))
```

As required, it consumes a `Posn` `p`, and its body compares the coordinates of `p` to those of the point (3,4), meaning this function represents a single point. While the data representation of a point as a `Shape` might seem silly, it suggests how we can define functions that create elements of `Shape`:

```
; Number Number -> Shape
(define (make-point x y)
  (lambda (p)
    (and (= (posn-x p) x) (= (posn-y p) y))))

(define a-sample-shape (make-point 3 4))
```

Stop again! Convince yourself—perhaps by using DrRacket’s stepper—that the last line creates a data representation of (3,4).

If we were to **design** such a function, we would formulate a purpose statement and provide some illustrative examples. For the purpose we could go with the obvious:

```
; creates a data representation for a point at (x,y)
```

or, more concisely,

```
; represents a point at (x,y)
```

For the examples we want to go with the interpretation of `Shape`. To illustrate, `(make-point 3 4)` is supposed to evaluate to a function that returns `#true` if, and only if, it is given `(make-posn 3 4)`. Using `inside?`, we can express this statement via tests:

```
(check-expect (inside? (make-point 3 4) (make-posn 3 4)) #true)
(check-expect (inside? (make-point 3 4) (make-posn 3 -4)) #false)
```

In short, to make a point representations we define a constructor-like function that consumes the point’s two coordinates. Instead of a record, this function uses `lambda` to construct another function. The function that it creates consumes a `Posn` and determines whether its `x` and `y` fields are equal to the originally given coordinates.

Next we generalize this idea from simple points to shapes such as circles and rectangles. In your geometry courses, you learn that a circle is a collection of points that all have the same distance to the center of the circle—the radius. For points inside the circle, the distance is smaller or equal to the radius. Hence, a function that creates a `Shape` representation of a circle must consume three pieces: the two coordinates for its center and the radius:

```
; Number Number Number -> Shape
; creates a data representation for a circle of radius r
; located at (center-x, center-y)

(define (make-circle center-x center-y r)
```

for
this
design
choice.

```
...)
```

Like `make-point`, it produces a function via a `lambda`. The function that is returned determines whether some given `Posn` is inside the circle. Here are some examples, again formulated as tests:

```
(check-expect (inside? (make-circle 3 4 5) (make-posn 0 0)) #true)
(check-expect (inside? (make-circle 3 4 5) (make-posn 0 -1)) #false)
(check-expect (inside? (make-circle 3 4 5) (make-posn -1 3)) #true)
```

The origin, `(make-posn 0 0)`, is exactly five steps away from (3,4) the center of the circle; see [Structures](#). Stop! Explain the second and third example.

Exercise 243. Use a compass to draw the shapes on grid paper. Check that the origin belongs to both. ■

Mathematically, we say that a `Posn` `p` is inside a circle if the distance between `p` and the circle's center is smaller than the radius `r`. Let's wish for the right kind of helper function and write down what we have.

```
(define (make-circle center-x center-y r)
  ; [Posn -> Boolean]
  (lambda (p)
    (<= (distance-between center-x center-y p) r)))
```

The `distance-between` function is a straightforward exercise.

Exercise 244. Design the function `distance-between`. It consumes two numbers and a `Posn`: `x`, `y`, and `p`. The function computes the distance between the points `(x, y)` and `p`.

Domain Knowledge The distance between (x_0, y_0) and (x_1, y_1) is

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

i.e., the distance of $(x_0 - y_0, x_1 - y_1)$ to the origin. ■

The data representation of a rectangle is expressed in a similar manner:

```
; Number Number Number Number -> Shape
; represent a width by height rectangle whose upper-left
; corner is located at (upper-left-x, upper-left-y)
(check-expect (inside? (make-rectangle 0 0 10 3) (make-posn 0 0)) #true)
(check-expect (inside? (make-rectangle 2 3 10 3) (make-posn 4 5)) #true)
(check-expect (inside? (make-rectangle 0 3 10 3) (make-posn -1 3)) #false)
(define (make-rectangle upper-left-x upper-left-y width height)
  (lambda (p)
    (and (<= upper-left-x (posn-x p) (+ upper-left-x width))
         (<= upper-left-y (posn-y p) (+ upper-left-y height)))))
```

Its constructor receives four numbers: the position of the upper, left corner, its width, and its height. The result is again a `lambda` expression, i.e., a function. As for circles, this function consumes a `Posn` and produces a `Boolean`, checking whether the `x` and `y` fields of the `Posn` are in the proper intervals.

At this points, we have only one task left, namely, the design of function that maps two `Shape` representations to their combination. The signature and the header are easy:

```
; Shape Shape -> Shape
; combines two shapes into one
```

```
(define (make-combination s1 s2)
  ; Posn -> Boolean
  (lambda (p)
    #false))
```

Indeed, even the default value is straightforward. We know that a shape is represented as a function from `Posn` to `Boolean`, so we write down a `lambda` that consumes some `Posn` and produces `#false`, meaning it says no point is in the combination.

So suppose we wish to combine the circle and the rectangle from above:

```
(define circle1 (make-circle 3 4 5))
(define rectangle1 (make-rectangle 0 3 10 3))
(define union1 (make-combination circle1 rectangle1))
```

We know that some points are inside and others are outside of this combination:

```
(check-expect (inside? union1 (make-posn 0 0)) #true)
(check-expect (inside? union1 (make-posn 0 -1)) #false)
(check-expect (inside? union1 (make-posn -1 3)) #true)
```

Since `(make-posn 0 0)` is inside both, there is no question that is inside the combination of the two. In a similar vein, `(make-posn 0 -1)` is in neither shape, and so it isn't in the combination. Finally, `(make-posn -1 3)` is in `circle1` but not in `rectangle1`. But the point must be in the combination of the two shapes because every point that is in one or the other shape is in their combination.

This analysis of examples implies the obvious revision of `make-combination`:

```
; Shape Shape -> Shape
(define (make-combination s1 s2)
  ; Posn -> Boolean
  (lambda (p)
    (or (inside? s1 p) (inside? s2 p))))
```


The `or` expression says that the result is `#true` if one of two expression produces `#true`: `(inside? s1 p)` or `(inside? s2 p)`. The first expression determines whether `p` is in `s1` and the second one whether `p` is in `s2`. And that is precisely a translation of our above explanation into ISL+.

Exercise 245. Design `my-animate`. Recall that the `animate` function consumes the representation of a *stream* of images, one per natural number. Since streams are infinitely long, ordinary compound data cannot represent them. Instead, we use functions:

```
; An ImageStream is a function:
; [N -> Image]
; interpretation a stream s represents a time-series of images
```

Here is a data example:

```
; ImageStream
(define (create-rocket-scene height)

  (place-image  50 height (empty-scene 100 100)))
```

You may recognize this as one of the first pieces of code in [Prologue: How to Program](#).

The task of `(my-animate s n)` is to display the images `(s 0)`, `(s 1)`, and so on at a rate

of 30 images per second up to n images total. Its result is the number of clock ticks passed since launched.

Note This case is an example where it is possible to write down examples/test cases easily but these examples/tests per se do not inform the design process of this **big-bang** function. Using functions as data representations calls for more design ideas than this book supplies. ■

Exercise 246. Design a data representation for finite and infinite sets so that you can represent the sets of all odd numbers, all even numbers, all numbers divisible by 10, etc.

Hint Mathematicians sometimes interpret sets as functions that consume a potential element e and produce `#true` if the e belongs to the set and `#false` if it doesn't.

Design the functions

1. `add-element`, which adds an element to a set;
2. `union`, which combines the elements of two sets; and
3. `intersect`, which collects all elements common to two sets;

Keep in mind the analogy between sets and shapes. ■

20 Summary

This third part of the book is about the role of abstraction in program design. Abstraction has two sides: creation and use. It is therefore natural if we summarize the chapter as two lessons:

1. **Repeated code patterns call for abstraction.** To abstract means to factor out the repeated pieces of code—the abstraction—and to parameterize over the differences. With the design of proper abstractions, programmers save themselves future work and headaches because mistakes, inefficiencies, and other problems are all in one place. One fix to the abstraction thus eliminates any specific problem once and for all. In contrast, the duplication of code means that a programmer must find all copies and fix all of them when a problem is found.
2. Most languages come with a large collection of abstractions. Some are contributions by the language design and implementation team; others are added by programmers who use the language. To enable **effective reuse of these abstractions**, their creators must supply the appropriate pieces of documentation—a **purpose statement**, a **signature**, and **good examples**—and programmers use them to apply abstractions.

All programming languages come with the means to build abstractions though some means are better than others. All programmers must get to know the means of abstractions and the abstractions that a language provides. A discerning programmer will learn to distinguish programming languages along these axes.

In addition to abstraction, this third part of the book also introduces the idea that

functions are values, and they can represent information as data.

While the idea is ancient for the Lisp family of programming languages (such as ISL+) and for specialists in programming language research, it has only recently gained acceptance in most modern mainstream languages—C#, C++, Java, JavaScript, Perl,

Python.