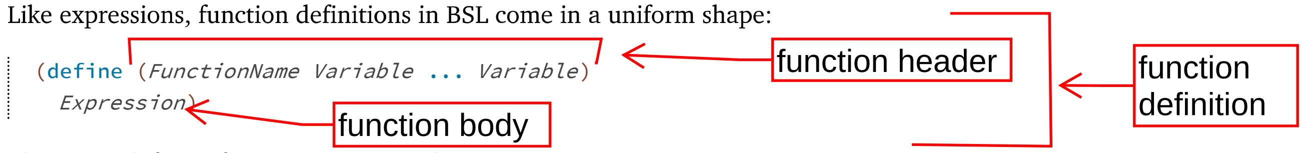
Programs are functions. Like functions, programs consume inputs and produce outputs. Unlike the functions you may know, programs work with a variety of data: numbers, strings, images, mixtures of all these, and so on. Furthermore, programs are triggered by events in the real world, and the outputs of programs affect the real world. For example, a spreadsheet program may react to an accountant's key presses by filling some cells with numbers, or the calendar program on a computer may launch a monthly payroll program on the last day of every month. Lastly, a program may not consume all of its input data at once; instead it may decide to process data in an incremental manner.

Definitions While many programming languages obscure the relationship between programs and functions, BSL brings it to the fore. Every BSL program consists of several definitions, usually followed by an expression that involves those definitions. There are two kinds of definitions:

- *constant definitions*, of the shape `(define Variable Expression)`, which we encountered in the preceding chapter; and
- *function definitions*, which come in many flavors, one of which we used in the Prologue.

Like expressions, function definitions in BSL come in a uniform shape:



```
(define (FunctionName Variable ... Variable)
  Expression)
```

That is, to define a function, we write down

- "`(define` (",
- the name of the function,
- followed by several variables, separated space and ending in ")",
- and an expression followed by ")".

And that is all there is to it. Here are some small examples:

- `(define (f x) 1)`
- `(define (g x y) (+ 1 1))`
- `(define (h x y z) (+ (* 2 2) 3))`

Before we explain why these examples are silly, we need to explain what function definitions mean. Roughly speaking, a function definition introduces a new operation on data; put differently, it adds an operation to our vocabulary if we think of the primitive operations as the ones that are always available. Like a primitive function, a defined function consumes inputs. The number of variables determines how many inputs—also called *arguments* or *parameters*—a function consumes. Thus, `f` is a one-argument function, sometimes called a *unary* function. In contrast, `g` is a two-argument function, also dubbed *binary*, and `h` is a *ternary* or three-argument function. The expression—often referred to as the *function body*—determines the output.

The examples are silly because the expressions inside the functions do not involve the variables. Since variables are about inputs, not mentioning them in the expressions means that the function's output is independent of its input and therefore always the same. We don't need to write functions or programs if the output is always the same.

Variables aren't data; they represent data. For example, a constant definition such as

```
(define x 3)
```

says that x always stands for 3. The variables in a *function header*, i.e., the variables that follow the function name, are placeholders for **unknown** pieces of data, the inputs of the function. Mentioning a variable in the function body is the way to use these pieces of data when the function is applied and the values of the variables become known.

Consider the following fragment of a definition:

```
(define (ff a) ...)     function header
```

Its function header is (ff a), meaning ff consumes one piece of input, and the variable a is a placeholder for this input. Of course, at the time we define a function, we don't know what its input(s) will be. Indeed, the whole point of defining a function is that we can use the function many times on many different inputs.

Useful function bodies refer to the function parameters. A reference to a function parameter is really a reference to the piece of data that is the input to the function. If we complete the definition of ff like this

```
(define (ff a)
  (* 10 a))     function body
```

we are saying that the output of a function is ten times its input. Presumably this function is going to be supplied with numbers as inputs, because it makes no sense to multiply images or Boolean values or strings by 10.

For now, the only remaining question is how a function obtains its inputs. And to this end, we turn to the notion of applying a function.

Applications A *function application* puts defined functions to work and it looks just like the applications of a pre-defined operation:

- write "(",
- write down the name of a defined function f,
- write down as many arguments as f consumes, separated by space,
- and add ")" at the end.

With this bit of explanation, you can now experiment with functions in the interactions area just as we suggested you experiment with primitives to find out what they compute. The following three experiments, for example, confirm that f from above produces the same value no matter what input it is applied to:

```
> (f 1)
1
> (f "hello world")
1
> (f #true)
1
```

What does (f (circle 3 "solid" "red")) yield?

See, even images as inputs don't change f's behavior. But here is what

Remember to add (require 2htdp/image) to the definitions area.

In this book, we use two pre-installed teachpacks to demonstrate the separation of data processing from parsing: `2htdp/batch-io` and and `2htdp/universe`. Starting with this chapter, we develop design recipes for **batch** and **interactive** programs to give you an idea of how complete programs are designed. Do keep in mind that the libraries of full-fledged programming languages offer many more contexts for complete programs, and that you will need to adapt the design recipes appropriately

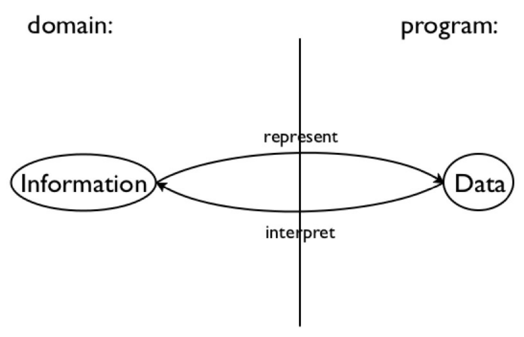


Figure 14: From information to data, and back

Given the central role of information and data, program design must start with the connection between them. Specifically, we, the programmers, -must decide how to use our chosen programming language to *represent* the relevant pieces of information as data and how we should *interpret* data as information. Figure 14 explains this idea with an abstract diagram.

To make this idea concrete, let’s work through some examples. Suppose you are designing a program that consumes and produces information in the form of numbers. While choosing a representation is easy, an interpretation requires explaining what a number such as `42` denotes in the domain:

- `42` may refer to the number of pixels from the top margin in the domain of images;
- `42` may denote the number of pixels per clock tick that a simulation or game object moves;
- `42` may mean a temperature, on the Fahrenheit, Celsius, or Kelvin scale for the domain of physics;
- `42` may specify the size of some table if the domain of the program is a furniture catalog; or
- `42` could just count the number of characters in a string.

The key is to know how to go from numbers as information to numbers as data and vice versa.

Computing scientists use “class” to mean something like a “mathematical set.”

Since this knowledge is so important for everyone who reads the program, we often write it down in the form of comments, which we call *data definitions*. A data definition serves two purposes. First, it names a collection of data—a *class*—using a meaningful word. Second, it informs readers how to create elements of this class and how to decide whether some arbitrary piece of data belongs to the collection.

Here is a data definition for one of the above examples:

```
; A Temperature is a Number.
; interpretation represents Celsius degrees
```

data definition

The first line introduces the name of the data collection, Temperature, and tells us that the class consists of all Numbers. So, for example, if we ask whether `102` is a temperature, you can respond with “yes” because `102` is a number and all numbers are temperatures. Similarly, if we ask whether `"cold"` is a Temperature, you will say “no” because no string belongs to Temperature. And, if we asked you to make up a sample Temperature, you might come up with something like `-400`.

If you happen to know that the lowest possible temperature is approximately -274C, you may wonder whether it is possible to express this knowledge in a data definition. Since our data definitions are really just English descriptions of classes, you may indeed define the class of temperatures in a much more accurate manner than shown here. In this book, we use a stylized form of English for such data definitions, and the next chapter introduces the style for imposing constraints such as “larger than `-274`.”

So far, you have encountered the names of four classes of data: Number, String, Image, and Boolean. With that, formulating a new data definition means nothing more than introducing a new name for an existing form of data, say, “temperature” for numbers. Even this limited knowledge, though, suffices to explain the outline of our design process.

The Process Once you understand how to represent input information as data and to interpret output data as information, the design of an individual function proceeds according to a straightforward process:

1. Express how you wish to represent information as data. A one-line comment suffices:

Representation comment

```
; We use numbers to represent centimeters.
```

Formulate data definitions, like the one for Temperature above for the classes of data you consider critical for the success of your program.

Data definition

2. Write down a signature, a purpose statement, and a function header.

A *function signature* is a comment that tells the readers of your design how many inputs your function consumes, from what classes they are drawn, and what kind of data it produces. Here are three examples for functions that respectively

Function signature

- consume one String and produce a Number:

```
; String -> Number
```

- consume a Temperature and produce a String:

```
; Temperature -> String
```

As this signature points out, introducing a data definition as an alias for an existing form of data makes it easy to read the intention behind signatures.

Nevertheless, we recommend to stay away from aliasing data definitions for now. A proliferation of such names can cause quite some confusion. It takes practice to balance the need for new names and the readability of programs, and there are more important ideas to understand for now.

- consume a Number, a String, and an Image:

```
; Number String Image -> Image
```

Stop! What does this function produce?

A *purpose statement* is a BSL comment that summarizes the purpose of the function in a single line. If you are ever in doubt about a purpose statement, write down the shortest possible answer to the question

Purpose statement

> *what does the function compute?*

Every reader of your program should understand what your functions compute **without** having to read the function itself.

A multi-function program should also come with a purpose statement. Indeed, good programmers write two purpose statements: one for the reader who may have to modify the code and another one for the person who wishes to use the program but not read it.

Finally, a *header* is a simplistic function definition, also called a *stub*. Pick one parameter for each input data class in the signature; the body of the function can be any piece of data from the output class. The following three function headers match the above three signatures:

Header

- `(define (f a-string) 0)`
- `(define (g n) "a")`
- `(define (h num str img) (empty-scene 100 100))`

Our parameter names reflect what kind of data the parameter represents. Sometimes, you may wish to use names that suggest the purpose of the parameter.

When you formulate a purpose statement, it is often useful to employ the parameter names to clarify what is computed. For example,

```
; Number String Image -> Image
; adds s to img,
; y pixels from the top and 10 from the left
(define (add-image y s img)
  (empty-scene 100 100))
```

signature

purpose statement

header

At this point, you can click the *RUN* button and experiment with the function. Of course, the result is always the same value, which makes these experiments quite boring.

3. Illustrate the signature and the purpose statement with some functional examples. To construct a *functional example*, pick one piece of data from each input class from the signature and determine what you expect back.

Functional examples

Suppose you are designing a function that computes the area of a square. Clearly this function consumes the length of the square's side, and that is best represented with a (positive) number. Assuming you have done the first process step according to the recipe, you add the examples between the purpose statement and the header and get this:

```
; Number -> Number
; computes the area of a square with side len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len) 0)
```

signature

purpose statement

header

4. The next step is to take *inventory*, to understand what are the givens and what we need to compute. For the simple functions we are considering right now, we

We owe the term "inventory" to Stephen Bloch.

Body template

functional examples

know that they are given data via parameters. While parameters are placeholders for values that we don't know yet, we do know that it is from this unknown data that the function must compute its result. To remind ourselves of this fact, we replace the function's body with a *template*.

For now, the template contains just the parameters, so that the preceding example looks like this:

```
(define (area-of-square len)
  (... len ...))
```

template

The dots remind you that this isn't a complete function, but a template, a suggestion for an organization.

The templates of this section look boring. As soon as we introduce new forms of data, templates become interesting, too.

5. It is now time to *code*. In general, to code means to program, though often in the narrowest possible way, namely, to write executable expressions and function definitions.

To us, coding means to replace the body of the function with an expression that attempts to compute from the pieces in the template what the purpose statement promises. Here is the complete definition for `area-of-square`:

```
; Number -> Number
; computes the area of a square with side len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len)
  (sqr len))
```

signature

Representation comment and data definition are missing

template replaced by completed code

```
; Number String Image -> Image
; adds s to img, y pixels from top, 10 pixels to the left
; given:
;    5 for y,
;    "hello" for s, and
;    (empty-scene 100 100) for img
; expected:
;    (place-image (text "hello" 10 "red") 10 5 ...)
;    where ... is (empty-scene 100 100)
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

signature

purpose statement

functional examples

Figure 15: The completion of design step 5

To complete the `add-image` function takes a bit more work than that: see figure 15. In particular, the function needs to turn the given string `s` into an image, which is then pla into the given scene.

Representation comment and data definition are missing

6. The last step of a proper design is to test the function on the examples that you worked out before. For now, testing works like this. Click the *RUN* button and enter function applications that match the examples in the interactions area:

```
> (area-of-square 2)
4
> (area-of-square 7)
49
```

```
; when needed, big-bang obtains the image of the current
; state of the world by evaluating (render cw)
```

purpose statement

```
; clock-tick-handler: WorldState -> WorldState
```

function signature

```
; for each tick of the clock, big-bang obtains the next
; state of the world from (clock-tick-handler cw)
```

purpose statement

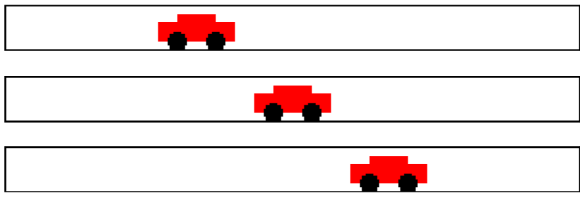
```
; key-stroke-handler: WorldState String -> WorldState
; for each key stroke, big-bang obtains the next state
; from (key-stroke-handler cw ke) where ke is the key
; stroke to obtain the new world state

; mouse-event-handler:
;   WorldState Number Number String -> WorldState
; for each mouse gesture, big-bang obtains the next state
; from (mouse-event-handler cw x y me) where x and y are
; the coordinates of the event and me is its description

; end?: WorldState -> Boolean
; when needed, big-bang evaluates (end? cw) to determine
; whether the program should stop
```

Figure 17: Signatures for interaction functions

Assuming that you have a rudimentary understanding of the workings of `big-bang`, you can focus on the truly important problem of designing world programs. Let's construct a concrete example for the following design recipe:

> **Sample Problem** Design a program that moves a car from left to right on the world canvas, three pixels per clock tick.

For this problem statement, it is easy to imagine scenes for the domain:



In this book, we often refer to the domain of an interactive `big-bang` program as a "world," and we speak of designing "world programs."

The design recipe for world programs, like the one for functions, is a tool for systematically moving from a problem statement to a working program. It consists of three big steps and one small one:

1. For all those properties of the world that remain the same over time and are needed to render it as an Image, introduce constants. In BSL, we specify such constants via definitions. For the purpose of world programs, we distinguish between two kinds of constants:

   a. "Physical" constants describe general attributes of objects in the world, such as the speed or velocity of an object, its color, its height, its width, its radius, and so forth. Of course these constants don't really refer to physical facts, but many are analogous to physical aspects of the real world.

In the context of our sample problem, the radius of the car's wheels and the distance between the wheels are such "physical" constants:

```
(define WIDTH-OF-WORLD 200)

(define WHEEL-RADIUS 5)
(define WHEEL-DISTANCE (* WHEEL-RADIUS 5))
```

Note how the second constant is computed from the first.

b. Graphical constants are images of objects in the world. The program composes them into images that represent the complete state of the world.

   Here are graphical constants for wheel images of our sample car:

```
(define WHEEL
  (circle WHEEL-RADIUS "solid" "black"))
```

> We suggest you experiment in DrRacket's interaction area to develop such graphical constants.

```
(define SPACE
  (rectangle ... WHEEL-RADIUS ... "white"))
(define BOTH-WHEELS
  (beside WHEEL SPACE WHEEL))
```

   Graphical constants are usually computed, and the computations tend to involve physical constants and other images.

It is good practice to annotate constant definitions with a comment that explains what they mean.

2. Those properties that change over time—in reaction to clock ticks, key strokes, or mouse actions—give rise to the current state of the world. Your task is to develop a data representation for all possible states of the world. The development results in a data definition, which comes with a comment that tells readers how to represent world information as data and how to interpret data as information about the world.

   Choose simple forms of data to represent the state of the world.

   For the running example, it is the car's distance to the left margin that changes over time. While the distance to the right margin changes, too, it is obvious that we need only one or the other to create an image. A distance is measured in numbers, so the following is an adequate data definition:

```
; A WorldState is a Number.
; interpretation the number of pixels between
; the left border of the scene and the car
```

data definition

   An alternative is to count the number of clock ticks that have passed and to use this number as the state of the world. We leave this design variant as an exercise.

3. Once you have a data representation for the state of the world, you need to design a number of functions so that you can form a valid big-bang expression.

```
                9 15
                (rectangle 2 20 "solid" "brown")))
```

to create a tree-like shape. Also add a clause to the `big-bang` expression that stops the animation when the car has disappeared on the right side. ▮

After settling on a first data representation for world states, a careful programmer may have to revisit this fundamental design decision during the rest of the design process. For example, the data definition for the sample problem represents the car as a point. But (the image of) the car isn't just a mathematical point without width and height. Hence, the interpretation statement—the number of pixels from the left margin—is an ambiguous statement. Does this statement measure the distance between the left margin and the left end of the car? Its center point? Or even its right end? We ignored this issue here and leave it to BSL's image primitives to make the decision for us. If you don't like the result, revisit the data definition above and modify it or its interpretation statement to adjust to your taste.

Exercise 42. Modify the interpretation of the sample data definition so that a state denotes the x-coordinate of the right-most edge of the car. ▮

Exercise 43. Let's work through the same problem statement with a time-based data definition:

```
; An AnimationState is a Number.
; interpretation the number of clock ticks
; since the animation started
```

data definition

Like the original data definition, this one also equates the states of the world with the class of numbers. Its interpretation, however, explains that the number means something entirely different.

Design functions `tock` and `render` and develop a `big-bang` expression so that you get once again an animation of a car traveling from left to right across the world's canvas.

How do you think this program relates to `animate` from Prologue: How to Program?

Use the data definition to design a program that moves the car according to a sine wave. Don't try to drive like that. ▮

We end the section with an illustration of mouse event handling, which also illustrates the advantages that a separation of view and model provide.

> Dealing with mouse movements is occasionally tricky because it isn't exactly what it seems to be. For a first idea of why that is, read On Mice and Keys.

Suppose we wish to allow people to move the car through "hyperspace:"

> **Sample Problem** Design a program that moves a car across the world canvas, from left to right, at the rate of three pixels per clock tick. **If the mouse is clicked anywhere on the canvas, the car is placed at the x-coordinate of that point.**

The bold part is the addition to the sample problem from above.

When we are confronted with a modified problem, we use the design process to guide us to the necessary changes. If used properly, this process naturally determines what we need to add to our existing program to cope with the addition to the problem statement. So here we go: