Рецепты проектирования

- [Design Process Once (For Function)](#functiondesignrecipe)

- [The Design Recipe (For Big-Bang (World) Program)](#programbigbangrecipe)

- [design recipe (4.6 Designing with Itemizations)](#design_with_itemization)

- [The Design Recipe of formulating data definitions](#Recipe_of_formulating_data_definitions)

- [Designing with Structures (repeat design recipe for functions if u must create structure for this)](#Designing_with_Structures)

- [Programming with Lists](#programming_with_lists)

- [Designing with Self-Referential Data Definitions](#desigining_with_self_ref_data)

[(and with non-empty lists additional)](#desigining_with_self_ref_data_non_empty_)

Other important points

- [What is “Number” word (when we mean the range)](#Number_word)

**Note on Numbers** The word “[Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29)” refers to a wide variety of numbers, including counting numbers, integers, rational numbers, real numbers, and even complex numbers. For most uses, you can safely equate [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29) with the number line from elementary school, though on occasion this translation is too imprecise. If we wish to be precise, we use appropriate words: Integer, Rational, and so on. We may even refine these notions using such standard terms as PositiveInteger, NonnegativeNumber, NegativeNumber, and so on. End

### 3 How to Design Programs

The first few chapters of this book show that learning to program requires some mastery of many concepts.

On the one hand, programming needs a language, a notation for communicating what we wish to compute.

**On the other hand, it is critical to learn how to get from a problem statement to a program**. We need to determine what is relevant in the problem statement and what can be ignored. We need to tease out what the program consumes, what it produces, and how it relates inputs to outputs. We have to know, or find out, whether the chosen language and its teachpacks provide certain basic operations for the data that our program is to process. If not, we might have to develop auxiliary functions that implement these operations. Finally, once we have a program, we must check whether it actually performs the intended computation. And this might reveal all kinds of errors, which we need to be able to understand and fix.

The Design Process Once you understand how to represent input information

#### 3.1 Designing Functions

#### The purpose of a program is to describe a computational process that consumes some information and produces new information. All this information comes from a part of the real world—often called the program’s domain—and the results of a program’s computation represent more information in this domain. Information plays a central role in our description. Think of information as facts about the program’s domain. For a program that deals with a furniture catalog, a “table with five legs” or a “square table of two by two meters” are pieces of information. A game program deals with a different kind of domain, where “five” might refer to the number of pixels per clock tick that some object travels on its way from one part of the canvas to another. For a program to process information, it must turn it into some form of data in the programming language; then it processes the data; and once it is finished, it turns the resulting data into information again.

#### Software engineers use the slogan model-view-controller (MVC) for the way separate data processing from parsing information into data and turning data into information.

#### Do keep in mind that the teachpacks of full-fledged programming languages offer many more contexts for complete programs, and that you will need to adapt the design recipes appropriately.

**Here we present a design recipe that integrates a step-by-step process with a way of organizing programs around problem data.** For the readers who don’t like to stare at blank screens for a long time, this design recipe offers a way to make progress in a systematic manner. For those of you who teach others to design programs, the recipe is a device for diagnosing a novice’s difficulties. For others, our recipe might be something that they can apply to other areas—say, medicine, journalism, or engineering. For those who wish to become real programmers, the design recipe also offers a way to understand and work on existing programs—though not all programmers use a method like this design recipe to come up with programs. The rest of this chapter is dedicated to the first baby steps into the world of the design recipe; the following chapters and parts refine and expand the recipe in one way or another.

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.

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](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29). |
| ; **interpretation** represents Celsius degrees |

The first line introduces the name of the data collection, Temperature, and tells us that the class consists of all Numbers.

The **Design Process Once (For Function)** 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:

; We use numbers to represent centimeters.

Formulate data definitions, or the classes of data you consider critical for the success of your program.

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

**2.1. A function signature is a** comment that tells the readers of your design how many inputs your function consumes, from which classes they are drawn, and what kind of data it produces.

consume one [String](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._string%29) and produce a [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29):

; String -> Number

consume a [Temperature](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._temperature%29) and produce a [String](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._string%29):

; Temperature -> String

consume a [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29), a [String](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._string%29), and an [Image](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._image%29):

; Number String Image -> Image

**2.2 A purpose statement is a** BSL comment that summarizes the purpose of the function in a single line.

write down the shortest possible answer to the question

*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.

**2.3.** **A Header is a** simplistic function definition, also called a stub. Pick one variable name for each class of input in the signature; the body of the function can be any piece of data from the output class.

* (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)) |

**3. Functional example**, pick one piece of data from each input class from the signature and determine what you expect back.

|  |
| --- |
| ; 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) |

**4. Take inventory, to understand what are the givens and what we need to compute.** We replace the function’s body with **a template**.

For now, the template contains just the parameters

|  |
| --- |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (area-of-square len) |
| ([...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) len [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29))) |

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

**5. Time to code.** To program, though often in the narrowest possible way, namely, to write executable expressions and function definitions.

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.

|  |
| --- |
| ; [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29) -> [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29) |
| ; computes the area of a square with side len |
| ; given: 2, expect: 4 |
| ; given: 7, expect: 49 |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (area-of-square len) |
| ([sqr](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._sqr%29%29) len)) |

**6. Test the function on the examples** that you worked out before.

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

If the result doesn’t match the expected output, consider the following three possibilities:

- You miscalculated and determined the wrong expected output for some of the examples.

- Alternatively, the function definition computes the wrong result. When this is the case, you have a logical error in your program, also known as a bug.

- Both the examples and the function definition are wrong.

|  |
| --- |
| ; [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29) [String](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._string%29) [Image](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._image%29) -> [Image](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._image%29) |
| ; adds s to img, y pixels from top, 10 pixels to the left |
| ; given: |
| ;    5 for y, |
| ;    "hello" for s, and |
| ;    ([empty-scene](http://docs.racket-lang.org/teachpack/2htdpimage.html" \l "%28def._%28%28lib._2htdp%2Fimage..rkt%29._empty-scene%29%29) 100 100) for img |
| ; expected: |
| ;    ([place-image](http://docs.racket-lang.org/teachpack/2htdpimage.html#%28def._%28%28lib._2htdp%2Fimage..rkt%29._place-image%29%29) ([text](http://docs.racket-lang.org/teachpack/2htdpimage.html#%28def._%28%28lib._2htdp%2Fimage..rkt%29._text%29%29) "hello" 10 "red") 10 5 [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)) |
| ;    where ... is ([empty-scene](http://docs.racket-lang.org/teachpack/2htdpimage.html" \l "%28def._%28%28lib._2htdp%2Fimage..rkt%29._empty-scene%29%29) 100 100) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (add-image y s img) |
| ([place-image](http://docs.racket-lang.org/teachpack/2htdpimage.html#%28def._%28%28lib._2htdp%2Fimage..rkt%29._place-image%29%29) ([text](http://docs.racket-lang.org/teachpack/2htdpimage.html#%28def._%28%28lib._2htdp%2Fimage..rkt%29._text%29%29) s 10 "red") 10 y img)) |

Figure 16: The completion of design step 5

#### 3.3 Domain Knowledge

Knowledge it takes to code up the body of a function is appropriate of the domain of the program.

There are two forms of such **domain knowledge**:

- **knowledge from external domains**, such as mathematics, music, biology, civil engineering, art, and so on. programmers must be prepared to understand the language of a variety of application areas so that they can discuss problems with domain experts.

- **knowledge about the teachpack functions** in the chosen programming language.

**Since you can never predict the area you will be working in**, or which programming language you will have to use, it is imperative that **you have a solid understanding of the full possibilities of whatever computer languages are around and suitable.**

#### 3.4 From Functions to Programs

Not all programs consist of a single function definition. Some require several functions; many also use constant definitions. When you have defined global constants, your functions may use them to compute results. To remind yourself of their existence, you may wish to add these constants to your templates.

Multi-function programs come about because:

- interactive programs automatically need functions that handle key and mouse events, functions that render the state as music, and possibly more

- batch programs may require several different functions because they perform several separate tasks

- Sometimes the problem statement itself suggests these tasks

- you will discover the need for auxiliary functions as you are in the middle of designing some function

**We recommend keeping around a list of needed functions or a wish list.**

Each entry on a wish list should consist of three things:

- a meaningful name for the function

- signature

- purpose statement

As long as the list isn’t empty, pick a wish and design the function. If you discover during the design that you need another function, put it on the list. When the list is empty, you are done.

#### 3.5 On Testing

Like many programming languages, BSL includes a testing facility, and DrRacket is aware of this facility.

|  |
| --- |
| ; [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29) -> [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29) |
| ; converts Fahrenheit temperatures to Celsius |
| ; given 32, expect 0 |
| ; given 212, expect 100 |
| ; given -40, expect -40 |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (f2c f) |
| ([\*](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._%2A%29%29) 5/9 ([-](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._-%29%29) f 32))) |

You can formulate these tests and add them to the definitions area in DrRacket:

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (f2c -40) -40)

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (f2c 32) 0)

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (f2c 212) 100)

When you now click the RUN button, you see a report from BSL that the program passed all three tests—and you have nothing else to do. In addition to getting tests to run automatically, the [check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) forms show another advantage when tests fail.

Instead of writing down the examples as comments, you can translate them directly into tests.

Check-expect also works for images.

This helps you understand how to express the function body. One way to develop such expressions is to experiment in the interactions area.

This form of testing (**check-expect**) is dubbed **unit testing**

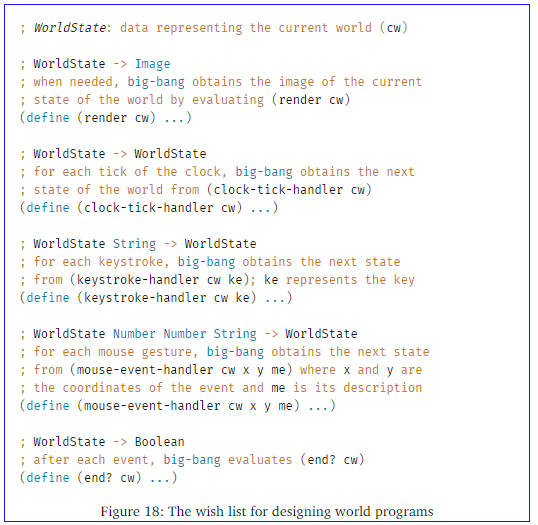
One day you will **switch to some other programming language** one of your **first tasks** will be to **find its unit-testing framework.**

#### 3.5 Designing World Programs

This section demonstrates how the design recipe also helps you create world programs systematically.

The **teachpack expects** that a **programmer develops** a **data definition** that represents **the state of the world** and **a function render** that knows how **to create an image for every possible state of the world**.

Depending on the needs of the program, the programmer must then design functions that respond to clock ticks, keystrokes, and mouse events. Finally, an interactive program may need to stop when its current world belongs to a sub-class of states; end? recognizes these final states.



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

The **Design Recipe (For Big-Bang (World) Program)**

**1. Introduce constants** (in BSLwe specify such constants via definitions)

For all properties of the world

- that remain the same over time

- that needed to render it as an Image

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.

|  |
| --- |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) WIDTH-OF-WORLD 200) |
|  |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) WHEEL-RADIUS 5) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) WHEEL-DISTANCE ([\*](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._%2A%29%29) WHEEL-RADIUS 5)) |

**b. Graphical constants**

are images of objects in the world

|  |
| --- |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) WHEEL |
| ([circle](http://docs.racket-lang.org/teachpack/2htdpimage.html#%28def._%28%28lib._2htdp%2Fimage..rkt%29._circle%29%29) WHEEL-RADIUS "solid" "black")) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) SPACE |
| ([rectangle](http://docs.racket-lang.org/teachpack/2htdpimage.html" \l "%28def._%28%28lib._2htdp%2Fimage..rkt%29._rectangle%29%29) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) WHEEL-RADIUS [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) "white")) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) BOTH-WHEELS |
| ([beside](http://docs.racket-lang.org/teachpack/2htdpimage.html" \l "%28def._%28%28lib._2htdp%2Fimage..rkt%29._beside%29%29) WHEEL SPACE WHEEL)) |

**2. Develop a data representation for all possible states of the world.**

Those properties that change over time—in reaction to clock ticks, keystrokes, or mouse actions—give rise to the current state 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.

|  |
| --- |
| ; A *WorldState* is a [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29). |
| ; **interpretation** the number of pixels between |
| ; the left border of the scene and the car |

**3. Design a functions**

for big-bang expression

To start, you need a function that maps any given state into an image

; render

Next you need to decide which kind of events should change which aspects of the world state. Depending on your decisions, you need to design some or all of the following three functions:

; clock-tick-handler;

 keystroke-handler;

 mouse-event-handler

Finally , you must design

; end?

Write generic signatures and purpose statements to the particular problems you solve so that readers of your programm know what they compute.

|  |
| --- |
| ; [WorldState](https://htdp.org/2023-5-12/Book/part_one.html" \l "%28tech._.D.K._worldstate%29) -> [Image](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._image%29) |
| ; places the image of the car x pixels from |
| ; the left margin of the BACKGROUND image |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (render x) |
| BACKGROUND) |
|  |
| ; [WorldState](https://htdp.org/2023-5-12/Book/part_one.html" \l "%28tech._.D.K._worldstate%29) -> [WorldState](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._.D.K._worldstate%29) |
| ; adds 3 to x to move the car right |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (tock x) |
| x) |

**4. Main function**

Unlike all other functions, a main function for world programs **doesn’t demand design or testing**. Its **sole reason** for existing is that you can **launch your world program** conveniently **from DrRacket’s interactions area**.

You must make decision about concerns main’s arguments.

|  |
| --- |
| ; [WorldState](https://htdp.org/2023-5-12/Book/part_one.html" \l "%28tech._.D.K._worldstate%29) -> [WorldState](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._.D.K._worldstate%29) |
| ; launches the program from some initial state |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (main ws) |
| ([big-bang](http://docs.racket-lang.org/teachpack/2htdpuniverse.html" \l "%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._big-bang%29%29) ws |
| [[on-tick](http://docs.racket-lang.org/teachpack/2htdpuniverse.html" \l "%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._on-tick%29%29) tock] |
| [[to-draw](http://docs.racket-lang.org/teachpack/2htdpuniverse.html" \l "%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._to-draw%29%29) render])) |

Naturally, **you don’t have to use the name “WorldState” for the class of data that represents the states of the world.**

Also, **you don’t have to use the names tock, end?, or render**. You may **name these functions whatever you like.**

**Single point of control**

This kind of program organization is dubbed single point of control, and good design employs this idea as much as possible.

**Modified problem**

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 expansion of the problem statement.

**4.6 Designing with Itemizations**

In this section, we refine the **design recipe** of From Functions to Programs so that you can proceed in a systematic manner when you encounter problems **concerning functions that consume itemizations**, **including enumerations and intervals**.

**Sample Problem**

The state of Tax Land has created a three-stage sales tax to cope with its budget deficit. Inexpensive items, those costing less than $1,000, are not taxed. Luxury items, with a price of more than $10,000, are taxed at the rate of eight percent (8.00%). Everything in between comes with a five percent (5.00%) markup.

**The steps of the design recipe:**

1. **carefully formulated data definitions**

A data definition must use distinct clauses for each sub-class of data or in some cases just individual pieces of data. Each clause specifies a data representation for a particular sub-class of information. The key is that each sub-class of data is distinct from every other class, so that our function can proceed by analyzing disjoint cases.

Our sample problem deals with prices and taxes, which are usually positive numbers. It also clearly distinguishes three ranges:

|  |
| --- |
| ; A *Price* falls into one of three intervals: |
| ; — 0 through 1000 |
| ; — 1000 through 10000 |
| ; — 10000 and above. |
| ; **interpretation** the price of an item |

2. **signature, purpose statement, and function header**

|  |
| --- |
| ; [Price](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._price%29) -> [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29) |
| ; computes the amount of tax charged for p |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (sales-tax p) 0) |

3. **pick at least one example from each sub-class in the data definition**

if a **sub-class is a finite range**, be sure to **pick examples from the boundaries of the range** **and** from its **interior**

4. **the template mirrors the organization of sub-classes with a cond**

1) the function’s body must be a conditional expression with as many clauses as there are distinct sub-classes in the data definition

if the data definition mentions three distinct sub-classes of input data, you need three cond clauses; if it has seventeen sub-classes, the cond expression contains seventeen clauses.

2) you must formulate one condition expression per cond clause

each expression involves the function parameter and identifies one of the sub-classes of data in the data definition

|  |
| --- |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (sales-tax p) |
| ([cond](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cond%29%29) |
| [([and](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._and%29%29) ([<=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c~3d%29%29) 0 p) ([<](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c%29%29) p 1000)) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)] |
| [([and](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._and%29%29) ([<=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c~3d%29%29) 1000 p) ([<](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c%29%29) p 10000)) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)] |
| [([>=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3e~3d%29%29) p 10000) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)])) |

5. **define the function**

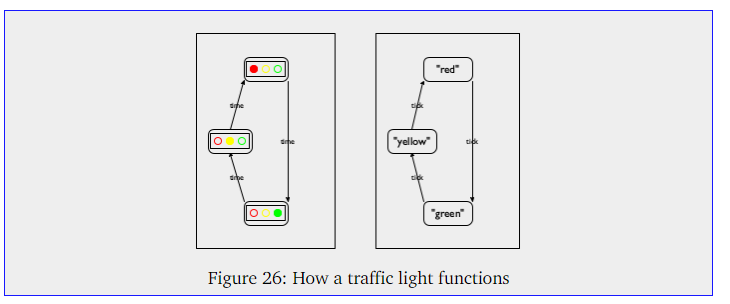
For each cond line, you may assume that the input parameter meets the condition and so you exploit the corresponding test cases. To formulate the corresponding result expression, you write down the computation for this example as an expression that involves the function parameter. (p. 3)

**Ignore all other possible kinds of input data when you work on one line; the other cond clauses take care of those.**

|  |
| --- |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (sales-tax p) |
| ([cond](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cond%29%29) |
| [([and](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._and%29%29) ([<=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c~3d%29%29) 0 p) ([<](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c%29%29) p 1000)) 0] |
| [([and](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._and%29%29) ([<=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c~3d%29%29) 1000 p) ([<](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3c%29%29) p 10000)) ([\*](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._%2A%29%29) 0.05 p)] |
| [([>=](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._~3e~3d%29%29) p 10000) ([\*](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._%2A%29%29) 0.08 p)])) |

**6. run the tests and ensure that they cover all cond clauses**

**4.7 Finite State Worlds**



Such a diagram consists of states and arrows that connect these states. **Each state depicts a traffic light in one particular configuration: red, yellow, or green.**

Each **arrow shows** how the world can change, from **which state it can transition to another state**. Our sample diagram contains three arrows, because **there are three possible ways in which the traffic light can change**. **Labels on the arrows indicate the reason for changes**; a traffic light transitions from one state to another **as time passes**.

**finite state machines (FSM), finite state automata (FSA)** - state transition diagrams have only a finite number of states and arrows

**To create a world program for an FSA** we must first **pick a data representation for the possible “states of the world,”** which **represents** those **aspects of the world** that **may change in some ways** as opposed to those that remain the same. In the **case** of our **traffic light**, what **changes is the color of the light**, that is, which bulb is turned on.

|  |
| --- |
| ; A *TrafficLight* is one of the following [String](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._string%29)s: |
| ; – "red" |
| ; – "green" |
| ; – "yellow" |
| ; **interpretation** the three strings represent the three |
| ; possible states that a traffic light may assume |

The right-hand side of figure is a diagrammatic interpretation of the TrafficLight data definition.

Also, the **arrows** are now **labeled with tick** to suggest that our world **program uses** the passing of **time** **as** the **trigger** that changes the state of the traffic light. **If we wanted** to **simulate a manually** **operated light**, we might **choose transitions based on keystroke**s.

We can write down the signature, a purpose statement, and a stub for the two functions we must design

|  |
| --- |
| ; [TrafficLight](https://htdp.org/2023-8-14/Book/part_one.html" \l "%28tech._trafficlight%29) -> [TrafficLight](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._trafficlight%29) |
| ; yields the next state, given current state cs |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (tl-next cs) cs) |
|  |
| ; [TrafficLight](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._trafficlight%29) -> [Image](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._image%29) |
| ; renders the current state cs as an image |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (tl-render current-state) |
| ([empty-scene](http://docs.racket-lang.org/teachpack/2htdpimage.html" \l "%28def._%28%28lib._2htdp%2Fimage..rkt%29._empty-scene%29%29) 90 30)) |

p. 53

Умение программировать – это гораздо больше, чем простое знание языка

Проще говоря, умение программировать – это системный подход к решению задач и передача этой системы в коде. Самое замечательное, что такой подход делает программирование доступным для всех – он служит сразу двум хозяевам.

…

Операции с данными в BSL – префиксная форма записи – операция записывается перед данными.

Изображение – это прямоугольный фрагмент визуальных данных, например фотография или геометрическая фигура и ее рамка.

Ваши программы могут манипулировать изображениями с помощью элементарных операций трех видов:

1. Операции первого вида создают элементарные изображения
2. Операции второго вида возвращают свойства изображений
3. функции для объединения изображений

..

Существует только два вида логических значений: #true и #false. Программы используют логические значения для представления решений или состояния переключателей. Вычисления с логическими значениями тоже очень просты. В частности, программы на BSL используют в основном три операции: or, and и not.

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.

Evaluating a function application proceeds in three steps: DrRacket determines the values of the argument expressions; it checks that the number of arguments and the number of function parameters are the same; if so, DrRacket computes the value of the body of the function, with all parameters replaced by the corresponding argument values. This last value is the value of the function application.

Again, we state an imperative slogan:

For every constant mentioned in a problem statement, introduce one constant definition.

This long list of steps might look overwhelming, the average function composition in a pre-algebra course involves two functions, possibly three. Keep in mind, though, that programs accomplish a real-world purpose while exercises in algebra merely illustrate the idea of function composition.

**design function process recipe**

1) data represent information – formulate and take one-line comment;

; We use numbers to represent centimeters.

2) 2.1 signature, a statement of purpose, and a function header

signature func - is a comment that tells how many inputs consumes, what kind of data produces

; String -> Number

2.2 statement of purpose – comment, shortest possible answer to the question ,,what does the function compute?,, (add to ch.1 comment maybe,,no)

; computes the area of a square with side len

2.3 function header - simplistic function definition, also called a stub

(define (f a-string) 0)

3) functional example – comment, show how pick one piece of data from each input class from the signature and determine what you expect back (instead of writing down the examples as comments, you can translate them directly into tests (check-expect))

; given: 2, expect: 4

; given: 7, expect: 49

4) inventory, the function’s body with a template – pre source code of func (no comment)

(define (area-of-square len)

(... len ...)

5) code - replace the body of the function with an expression in the template. write executable expressions and function definitions.

6) test the function – with examples from ch.3 . call of func (not a comment)

> (area-of-square 2)

4

> (area-of-square 7)

49

p.106-110

we recommend keeping around a list of needed functions or a wish list. Each entry on a wish list should consist of three things: a meaningful name for the function, a signature, and a purpose statement. As long as the list isn’t empty, pick a wish and design the function.

testing

check-expect - this form of testing is dubbed unit testing (framework for novice programmers). when you will switch to some other programming language; one of your first tasks will be to figure out its unit-testing framework.

p.111 – 117

**design recipe for world programs**

1) for properties of the world that remain the same over time introduce constants.

there are two kinds of constants

- “physical” constants describe general attributes of objects in the world

(define WIDTH-OF-WORLD 200)

(define WHEEL-RADIUS 5)

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

- graphical constants are images of objects in the world

(define WHEEL

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

(define SPACE

(rectangle ... WHEEL-RADIUS ... "white"))

(define BOTH-WHEELS

(beside WHEEL SPACE WHEEL))

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

2) for properties that change over time and give rise to the current state of the world – u must to develop a data representation for all possible states of the world (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)

; A WorldState is a Number.

; interpretation the number of pixels between

; the left border of the scene and the car

3) design a number of functions for big-bang expression

; render

; WorldState -> Image

; places the image of the car x pixels from

; the left margin of the BACKGROUND image

(define (render x)

BACKGROUND)

; clock-tick-handler

; WorldState -> WorldState

; adds 3 to x to move the car right

(define (tock x)

x)

; keystroke-handler

; mouse-event-handler

; end?

4) main func (for launch a program) //with bb expression I mean

**conditional expression**

**if** expressions are much less suited for multi-situation contexts; they are best used when all we wish to say is “one or the other.”

use **cond** for situations when we wish to remind the reader of our code that some distinct situations come directly from data definitions

**“else” in cond exp**

It is therefore better to formulate the function definition with “else” in cond exp, if you know that you want the exact opposite—called the complement—of all previous conditions in a cond

**intervals**An interval is a description of a class of numbers via boundaries.  
Visualizing the data definition in this manner helps with the design of functions in two ways. First, it immediately suggests how to pick examples. Clearly we want the function to work inside of all the intervals, and we want the function to work properly at the ends of each interval. Second, the image tells us that we need to formulate a condition that determines whether or not some “point” is within one of the intervals.

Putting the two together also raises a question, namely, how exactly the function deals with the end points. In the context of our example, two points on the number line belong to two intervals: Such overlaps usually cause problems for programs, and they ought to be avoided.

BSL functions avoid them naturally due to the way cond expressions are evaluated.

// To avoid multiple changes for a single element, programmers try to avoid copies. You have two choices to fix this problem. The first one is to use constant definitions, which you might recall from early chapters. The second one is to think of the cond expression as an expression that may appear anywhere in a function, including in the middle of some other expression//

In general, intervals deserve special attention when you make up examples, that is, they deserve at least three kinds of examples: one from each end and another one from inside.

**itemizations**some data definitions need to include elements from intervals and enumerations. They use **itemizations**, which generalize intervals and enumerations

example of data definition from enumerations

//; An LR (short for launching rocket) is one of:

; – "resting"

; – NonnegativeNumber

; interpretation "resting" represents a grounded rocket

; a number denotes the height of a rocket in flight//

**design function working with itemization process recipe**

1. carefully formulated data definitions, data definition must use distinct clauses for each sub-class of data or in some cases just individual pieces of data

; A Price falls into one of three intervals:

; — 0 through 1000

; — 1000 through 10000

; — 10000 and above.

; interpretation the price of an item

2. signature, purpose statement, and function header

; Price -> Number

; computes the amount of tax charged for p

(define (sales-tax p) 0)

3. functional examples

pick at least one example from each sub-class in the data definition. Also, if a sub-class is a finite range, be sure to pick examples from the boundaries of the range and from its interior

(check-expect (sales-tax -100) "enter a real cost")

(check-expect (sales-tax 0) 0)

(check-expect (sales-tax 500) 0)

(check-expect (sales-tax 1000) (\* 0.05 1000))

(check-expect (sales-tax 5000) (\* 0.05 5000))

(check-expect (sales-tax 10000) (\* 0.08 10000))

(check-expect (sales-tax 15000) (\* 0.08 15000)

4. the template mirrors the organization of sub-classes with a cond.

- the function’s body must be a conditional expression with as many clauses as there are distinct sub-classes in the data definition. **if the data definition mentions three distinct sub-classes of input data, you need three cond clauses; if it has seventeen sub-classes, the cond expression contains seventeen clauses**

- formulate one condition expression per cond clause

(define (sales-tax p)

(cond

[(and (<= 0 p) (< p 1000)) ...]

[(and (<= 1000 p) (< p 10000)) ...]

[(>= p 10000) ...]))

5. define the function

for each cond line, you may assume that the input parameter meets the condition and so you exploit the corresponding test cases. **ignore all other possible kinds of input data when you work on one line; the other cond clauses take care of those.**

(define (sales-tax p)

(cond

[(and (<= 0 p) (< p 1000)) 0]

[(and (<= 1000 p) (< p 10000)) (\* 0.05 p)]

[(>= p 10000) (\* 0.08 p)]))

6. run the tests and ensure that they cover all cond clauses

when you do encounter a mismatch between expected results and actual values, we recommend that you first reassure yourself that the expected results are correct. if so, assume that the mistake is in the function definition. otherwise, fix the example and then run the tests again. if you are still encountering problems, you may have encountered the third, somewhat rare, situation.

**structure**

this chapter introduces BSL’s mechanics, so-called structure type definitions, and how to design programs that work on compound data.

Every structure type definition introduces a new kind of structure, distinct from all others. Programmers want this kind of expressive power because they wish to convey **an intention** with the structure name. wherever a structure is created, selected, or tested, the text of the program explicitly reminds the reader of this intention.

nesting information is natural. The best way to represent such information with data is to mirror the nesting with nested structure instances.

(define-struct vel [deltax deltay])

(define ball1 (make-ball (make-posn 30 40) (make-vel -10 5)))

vs

(define-struct ballf [x y deltax deltay]) // programmers call this a flat representation

In sum, nesting information is natural. The **best way to represent** such **information** with data is to mirror **the nesting with nested structure** instances. Doing so makes it easy to interpret the data in the application domain of the program, and it is also straightforward to go from examples of information to data.

The selector expressions raise the question whether we need to refine this sketch even more. After all, the two expressions extract instances of Posn and Vel, respectively. These two are also structure instances, and we could extract values from them in turn. Doing so obviously makes the sketch look quite complex, however. For truly realistic programs, following this idea to its logical end would create incredibly complex program outlines. More generally,

***If a function deals with nested structures, develop one function per level of nesting.***

Определение данных (объявляем структуру) -> открывает дорогу к новой коллекции данных (ко всем вариантам экземпляров этой структуры)

Поскольку определения данных играют такую важную роль в процессе проектирования, часто желательно сопровождать определения данных примерами, как мы сопровождаем функции примерами, иллюстрирующими их поведение. Создавать примеры данных из определений довольно просто:

- для встроенной коллекции данных (число, строка, логическое значение, изображение) используйте любые примеры, какие вам нравятся;

ПРИМЕЧАНИЕ. *Иногда для обозначения встроенных коллекций данных люди используют описательные имена, например NegativeNumber или OneLetterString. Но они не являются заменой хорошо написанного определения данных.* КОНЕЦ.

- для перечисления используйте несколько элементов перечисления;

- для интервалов используйте конечные точки (если они есть) и хотя бы одну точку внутри;

- для детализаций проиллюстрируйте каждую часть отдельно;

- для структур следуйте описанию на естественном языке, то есть используйте конструктор и выберите примеры из коллекций данных, соответствующих каждому полю.

Это все, что потребуется для создания примеров на основе определений данных в большей части этой книги, правда, сами определения данных будут становиться все сложнее с каждой последующей главой.

5.10 A Graphical Editor

6 Itemizations and Structures

The preceding two chapters introduce two ways of formulating data definitions. Those that employ itemization (enumeration and intervals) are used to create small collections from large ones. Those that use structures combine multiple collections.

ex. As far as our domain isNo worries, the next part of the book is about firing as many missiles as you want, without reloading. concerned—the actual game—the first kind of state represents the time before the tank has launched its sole missile and the second one the time after the missile has been fired.

This chapter introduces the basic idea of itemizing data definitions that involve structures. Because we have all the other ingredients we need, we start straight with itemizing structures.

6.1 Designing with Itemizations, Again

The first step - development of data definitions

data definition - describe the construction of data that represents the state of the world and describe all possible pieces of data that the event-handling functions of the world program may consume

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Designing with Structures

The introduction of structure types reinforces the need for all six steps in the design recipe.

This section adds a design recipe, illustrating it with the following:

**Sample Problem** Design a function that computes the distance of objects in a 3-dimensional space to the origin.

1. **When a problem calls for the representation of pieces of information that belong together or describe a natural whole, you need a structure type definition. It requires as many fields as there are relevant properties.** An instance of this structure type corresponds to the whole, and the values in the fields correspond to its attributes.

A data definition for a structure type introduces a name for the collection of instances that are legitimate. Furthermore, it must describe which kind of data goes with which field. Use only names of built-in data collections or previously defined data definitions.

In the end, we (and others) must be able to use the data definition to create sample structure instances. Otherwise, something is wrong with our data definition. To ensure that we can create instances, **our data definitions should come with data examples**.

|  |
| --- |
| ([define-struct](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define-struct%29%29) r3 [x y z]) |
| ; An *R3* is a structure: |
| ;   (make-r3 [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29) [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29) [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29)) |
|  |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) ex1 (make-r3 1 2 13)) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) ex2 (make-r3 -1 0 3)) |

The structure type definition introduces a new kind of structure, r3, and the data definition introduces R3 as the name for all instances of r3 that contain only numbers.

2. You still need a signature, a purpose statement, and a function header

|  |
| --- |
| ; [R3](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._r3%29) -> [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29) |
| ; determines the distance of p to the origin |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (r3-distance-to-0 p) |
|  |

3. Use the examples from the first step to **create functional examples**. For each field associated with intervals or enumerations, make sure to pick end points and intermediate points to create functional examples.

4. A function that consumes structures usually—though not always—extracts the values from the various fields in the structure. To remind yourself of this possibility, **add a selector for each field to the templates for such functions**.

([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (r3-distance-to-0 p)  ([...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) (r3-x p) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) (r3-y p) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) (r3-z p) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)))

5. Use the selector expressions from the template when you define the function. Keep in mind that you may not need some of them.

6. Test. Test as soon as the function header is written. Test until all expressions have been covered. Test again when you make changes.

**The Design Recipe of formulating data definitions**

В этой главе исследуется возможность объединения двух и более средств описания данных, и **уточненный рецепт проектирования** отражает это обстоятельство, особенно первый его шаг:

**1)** the need for itemizations is due to distinctions among different classes of information in the problem statement. Similarly, the need for structure-based data definitions is due to the demand to group several different pieces of information. **An itemization of different forms of data—including collections of structures—is required when your problem statement distinguishes different kinds of information and when at least some of these pieces of information consist of several different pieces**. One thing to keep in mind is that data definitions may refer to other data definitions. Hence, **if a particular clause in a data definition looks overly complex, it is acceptable to write down a separate data definition for this clause and refer to this auxiliary definition**. And, as always, formulate data examples using the data definitions.

Вы уже знаете, что **необходимость в детализации обусловлена различиями между разными классами информации в постановке задачи**. Точно так же потребность в определении данных на основе структуры возникает из-за необходимости сгруппировать несколько разных элементов информации.

**Детализация разных форм данных, включая коллекции структур, требуется, когда в постановке задачи присутствуют разные виды информации и когда хотя бы некоторые из них отличаются разным составом элементов.** Следует иметь в виду, что определения данных могут ссылаться на другие определения данных. Поэтому **если конкретное предложение в определении данных выглядит слишком сложным, допускается добавить отдельное определение данных для этого предложения и сослаться на это вспомогательное определение**. И как всегда, нужно сформулировать примеры данных, используя их определения;

example:

**1.1. Определения данных**

(define-struct aim [ufo tank])

(define-struct fired [ufo tank missile])

UFO -- это Posn.

; интерпретация: (make-posn x y) -- местоположение НЛО

; (используется соглашение о системе координат с началом в левом верхнем углу)

(define-struct tank [loc vel])

; Tank -- это структура:

; (make-tank Число Число).

; интерпретация: (make-tank x dx) определяет позицию:

; (x, HEIGHT) и скорость перемещения танка: dx пикселей за такт часов

; Missile -- это Posn.

; интерпретация: (make-posn x y) -- местоположение ракеты

; SIGS -- это одно из следующих значений:

; -- (make-aim UFO Tank)

; -- (make-fired UFO Tank Missile)

; интерпретация: представляет полное состояние игры

; с космическими захватчиками

**1.2. примеры**

(make-aim (make-posn 20 10) (make-tank 28 -3))

следующий пример соответствует состоянию игры после пуска ракеты:

(make-fired (make-posn 20 10)

(make-tank 28 -3)

(make-posn 28 (- HEIGHT TANK-HEIGHT)))

и последний пример соответствует состоянию, когда ракета попала в НЛО:

(make-fired (make-posn 20 100)

(make-tank 100 3)

(make-posn 22 103))

**2) Formulate a function signature** that mentions only the names of defined or built-in data collections, add a purpose statement, and create a function header.

Сформулируйте сигнатуру функции, в которой упоминаются только имена вновь добавленных или встроенных коллекций данных, добавьте описание назначения и определите заголовок функции

example:

SIGS -> Изображение

; добавляет TANK, UFO и, возможно, MISSILE в

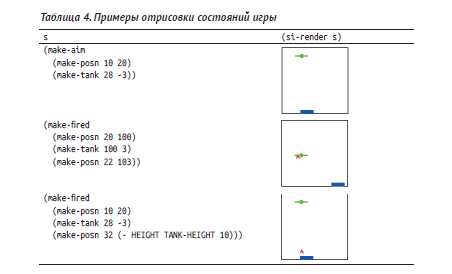
; сцену BACKGROUND

(define (si-render s) BACKGROUND)

**3) You still need to formulate functional examples** that illustrate the purpose statement from the second step, and you still need **one example per item in the itemization**.

сформулировать примеры применения функций, иллюстрирующие описание назначения из второго шага, и все так же должны записать **хотя бы по одному примеру для каждого элемента в детализации**;

example:



**4) development of the template func (разработка макета функции)**

exploits two different dimensions:

4.1. the itemization itself and

the body of the **template consists of a cond expression that has as many cond clauses as the itemizations has items**. Furthermore, you must add a condition to each cond clause that identifies the sub-class of data in the corresponding item.

4.2 the use of structures in its clauses

**if an item deals with a structure**, the **template contains the selector expressions—in the cond clause** that deals with the sub-class of data described in the item.

When you choose to describe the data with a separate data definition, however, you do not add selector expressions. Instead, you create a template for the separate data definition to the task at hand and refer to that template with a function call. The latter indicates that this sub-class of data is being processed separately.

**разработка макета – самый важный этап процесса проектирования**

два разных аспекта: саму детализацию и структуры в ее предложениях

4.1. Для учета первого аспекта **тело макета должно состоять из выражения cond, включающего столько условий, сколько имеется элементов в детализации**. Также **в каждое условие нужно добавить выражение cond**, **идентифицирующее подклассы данных в соответствующем элементе**.

4.2. **если элемент представлен структурой, макет должен содержать выражения применения селекторов в каждом условии cond**, идентифицирующем подкласс данных, описанный в элементе.

**Прежде чем приступить к разработке макета, поразмышляйте о характере функции**. **Если формулировка задачи предполагает необходимость решения нескольких задач**, вполне вероятно, что **вместо одной функции потребуется спроектировать и написать несколько функций**. В таком случае пропустите этот шаг;

example:

мы знаем, что тело si-render должно содержать выражение cond с двумя условиями. Согласно рецепту проектирования, этими условиями должны быть (aim? s) и (fired? s), они различают два возможных вида данных, которые может получить si-render

(define (si-render s)

(cond

[(aim? s) ...]

[(fired? s) ...]))

...

Во-вторых, добавим селекторы в каждое условие cond, обрабатывающее структуры. В данном случае оба условия обрабатывают структуры: aim и fired. Первая имеет два поля и, следовательно, требует добавить в первое условие cond два селектора, а вторая имеет три поля и, соответственно, требует добавить три селектора:

(define (si-render s)

(cond

[(aim? s) (... (aim-tank s) ... (aim-ufo s) ...)]

[(fired? s) (... (fired-tank s) ... (fired-ufo s)

... (fired-missile s) ...)]))

**5) Fill the gaps in the template. (заполните пробелы в макете)**. The more complex you make your data definitions, the more complex this step becomes. The good news is that this design recipe can help in many situations.

ill the gaps in the template. The more complex you make your data definitions, the more complex this step becomes. The good news is that this design recipe can help in many situations.

If you are stuck, fill the easy cases first and use default values for the others. While this makes some of the test cases fail, you are making progress and you can visualize this progress.

If you are stuck on some cases of the itemization, analyze the examples that correspond to those cases. Determine what the pieces of the template compute from the given inputs. Then consider how to combine these pieces (plus some constants) to compute the desired output. Keep in mind that you might need an auxiliary function.

Also, if your template “calls” another template because the data definitions refer to each other, assume that the other function delivers what its purpose statement and its examples promise—even if this other function’s definition isn’t finished yet.

заполните пробелы в макете. Чем сложнее определение данных, тем сложнее этот шаг. Но не отчаивайтесь, потому что данный рецепт дизайна может помочь даже в самых сложных

ситуациях.

Если вы застопорились, то сначала заполните простые случаи, а для остальных используйте значения по умолчанию. Некоторые тестовые примеры могут показывать неверный результат,

тем не менее это будет вполне видимый шаг вперед.

Если вы застопорились на некоторых случаях детализации, проанализируйте примеры, соответствующие им. Определите, что вычисляют части макета, соответствующие заданным входным данным. Затем подумайте, как объединить эти части (и некоторые константы) для получения желаемого результата. Имейте в виду, что вам может понадобиться вспомогательная функция. Кроме того, если ваш макет «вызывает» другой макет из-за ссылок друг на друга в определениях данных, то исходите из предположения, что та другая функция делает именно то, что обещают ее описание назначения и примеры, даже если определение той другой функции еще не завершено;

example:

Макет содержит почти все, что нужно для завершения поставленной задачи. Чтобы завершить определение, нужно выяснить для каждого условия в cond, как объединить имеющиеся значения, дабы получить желаемый результат. Кроме входных данных, мы можем использовать глобальные константы, например BACKGROUND, что явно не будет лишним; элементарные или встроенные операции; и, если потребуется, функции из списка желаний, то есть мы дополнительно должны описать функции, которые хотели бы иметь. Рассмотрим первое условие cond. Здесь у нас есть представление данных о танке (aim-tank s) и представление данных об НЛО (aim ufo s). Из первого примера в табл. 4 мы знаем, что функция должна добавить в сцену два объекта. Кроме того, рецепт проектирования предполагает, что если эти элементы данных имеют собственные определения данных, то необходимо рассмотреть возможность определения дополнительных (вспомогательных) функций и использовать их для вычисления результата:

... (tank-render (aim-tank s)

(ufo-render (aim-ufo s) BACKGROUND))

Здесь tank-render и ufo-render – это функции, добавленные в список желаний:

; Tank Изображение -> Изображение

; добавляет t в заданное изображение im

(define (tank-render t im) im)

; UFO Изображение -> Изображение

; добавляет u в заданное изображение im

(define (ufo-render u im) im)

Точно так же можно поступить со вторым условием в выражении cond. В листинге 19 показано законченное определение функции отображения.

Самое замечательное, что мы можем сразу же использовать наши функции из списка желаний, tank-render и ufo-render, и нам остается только добавить функцию, отображающую ракету в сцене. Вот соответствующая запись в списке желаний:

; Missile Изображение -> Изображение

; добавляет m в заданное изображение im

(define (missile-render m im) im)

Как и прежде, комментарии достаточно подробно описывают желаемое поведение функции.

Листинг 19. Законченное определение функции отображения

; SIGS -> Изображение

; отображает состояние игры поверх BACKGROUND

; примеры см. в табл. 4

(define (si-render s)

(cond

[(aim? s)

(tank-render (aim-tank s)

(ufo-render (aim-ufo s) BACKGROUND))]

[(fired? s)

(tank-render

(fired-tank s)

(ufo-render (fired-ufo s)

(missile-render (fired-missile s)

BACKGROUND)))]))

**6) Test.** If tests fail, determine what’s wrong: the function, the tests, or both. Go back to the appropriate step.

Если тесты не проходят, определите, где находится причина: в функциях, в тестах или там и там. Вернитесь к соответствующему шагу. (тесты по примерам из п. 3 functional examples)

6.3 Input Errors

One **central point** of this chapter concerns the role of **predicates**.

This section therefore presents **one way** of **protecting programs** from **inappropriate inputs**.

Hence the rough definition of checked-area-of-disk looks like this:

(define MESSAGE "area-of-disk: number expected")

(define (checked-area-of-disk v)

(cond

[(number? v) (area-of-disk v)]

[(boolean? v) (error MESSAGE)]

[(string? v) (error MESSAGE)]

[(image? v) (error MESSAGE)]

[(posn? v) (error MESSAGE)]

...

[(tank? v) (error MESSAGE)]

...))

The use of else helps us finish this definition in the natural way:

; Any -> Number

; computes the area of a disk with radius v,

; if v is a number

(define (checked-area-of-disk v)

(cond

[(number? v) (area-of-disk v)]

[else (error "area-of-disk: number expected")]))

And just to make sure we get what we want, let’s experiment:

> (checked-area-of-disk "my-disk")

area-of-disk:number expected

Writing checked functions is important if we distribute our programs for others to use. Designing programs that work properly, however, is far more important. **This book focuses on the design process for proper program design, and, to do this without distraction, we agree that we always adhere to data definitions and signatures.**

You might wonder how you can **design your own predicates.**

; A MissileOrNot is one of:

; – #false

; – Posn

; interpretation#false means the missile is in the tank;

; Posn says the missile is at that location

; Any -> Boolean

; is a an element of the MissileOrNot collection

(define (missile-or-not? v)

(cond

[(false? v) #true]

[(posn? v) #true]

[else #false]))

It is a **good practice** to **use questions as purpose statements for predicates**, because applying a predicate is like asking a question about a value. The question mark “?” **at the end of the name also reinforces this idea**; some people may tack on “huh” to pronounce the name of such functions.

Вообщем есть предложение если создаешь собственное определение денных (перечиление, интервалы и т.д, но н е структуру, к ней предикат же создается автоматом), хороший инструмент если действительно нужна проверка самому написать предикат как отдельную самописную функцию со знаком вопроса в названии на конце, как будто это предикат к встроенным данным. пример выше).

6.4 Checking the World

**In a world program, many things can go wrong**. For example, **one of** our **functions may produce a result that isn’t quite an element of your data representation for world states**. At the same time, big-bang accepts this piece of data and holds on to it, until the next event takes place. It is only when the following event handler receives this inappropriate piece of data that the program may fail. But it may get worse because even the second and third and fourth event-handling step may actually cope with inappropriate state values, and it all blows up much later in the process.

[**big-bang**](http://docs.racket-lang.org/teachpack/2htdpuniverse.html#%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._big-bang%29%29)**comes with an optional**[**check-with**](http://docs.racket-lang.org/teachpack/2htdpuniverse.html#%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._check-with%29%29)**clause** that accepts a predicate for world states. If, for example, we chose to represent all world states with [Number](https://htdp.org/2023-5-12/Book/part_one.html#%28tech._number%29), we could express this fact easily like this:

([define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (main s0)

([big-bang](http://docs.racket-lang.org/teachpack/2htdpuniverse.html#%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._big-bang%29%29) s0 [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) [[check-with](http://docs.racket-lang.org/teachpack/2htdpuniverse.html#%28form._world._%28%28lib._2htdp%2Funiverse..rkt%29._check-with%29%29) [number?](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._number~3f%29%29)] [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29)))

6.5 Equality Predicates

An equality predicate is a function that compares two elements of the same collection of data.

(string=?)

your programs will ought to use key=? and mouse=?, two equality predicates. Naturally, key=? is an operation for comparing two KeyEvents; similarly, mouse=? compares two MouseEvts.

**We recommend using key=? in key-event handlers and mouse=? in mouse-event handlers from now on.**

The use of key=? in a key-event handler ensures that the function really compares strings that represent key events and not arbitrary strings.

**7 Summary**

In this first part of the book, you learned a simple but important lessons

1. **good programmer designs programs** (not try before program seems to work)

2. design recipe has two dimensions

- sequence of steps

- how the chosen data representation influences the design process

3. well-designed **program consists of**

- constant definitions

- structure type definitions

- data definitions

- function definitions

**Batch programs** – consists of “main” function and several other functions to perform its computation

**Interactive programs** - the big-bang function plays the role of the main function and it’s specifies

- the initial state of the program

- an image-producing output function

- and at most three event handlers: one for clock ticks, one for mouse clicks, and one for key events

In both kinds of programs, function definitions are presented “top down,” starting with the main function, followed by those functions mentioned in the main function, and so on.

4. Programmers must be able to determine the meaning of each sentence in a programming language (learn vocabulary and a grammar of language)

5. A programmer should become comfortable with the functions that contained in teachpacks that supplied with Programming languages. Learn their signatures and purpose statements. Doing so simplifies life.

6. A programmer must get to know the “tools” that a chosen programming language offers (such as cond or max).

**Intermezzo 1: Beginning Student Language**

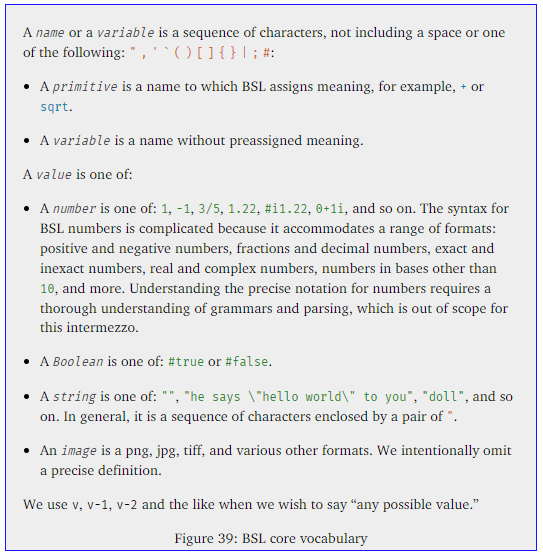
A programming language does have **a vocabulary and a grammar**, though programmers use the word **syntax** for these elements.

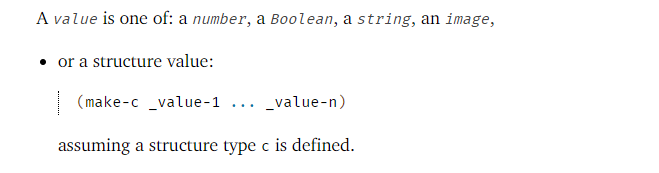
To determine whether a sentence is meaningful, we must know **the meaning of a language**; programmers call this **semantics**.

**This intermezzo presents BSL as if it were an extension of the familiar language** **of arithmetic and algebra** in middle school. After all, computation starts with this form of simple mathematics, and we should understand the **connection between this mathematics and computing.**

**The first three sections present the syntax and semantics of a good portion of BSL**. The fourth resumes our discussion of errors. The remaining sections expand this understanding to the complete language; the last one expands the tools for expressing tests.

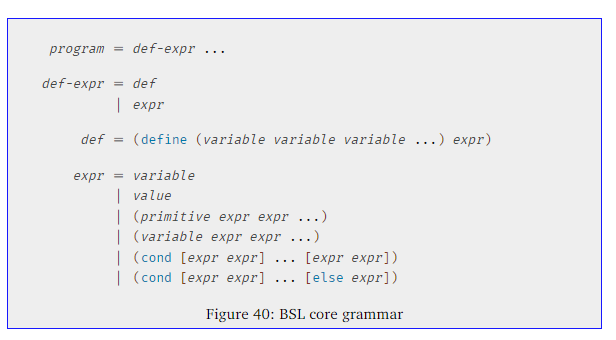
**BSL Vocabulary**





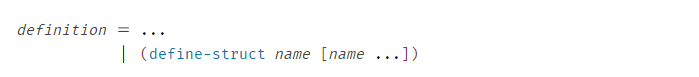
**BSL Grammar**

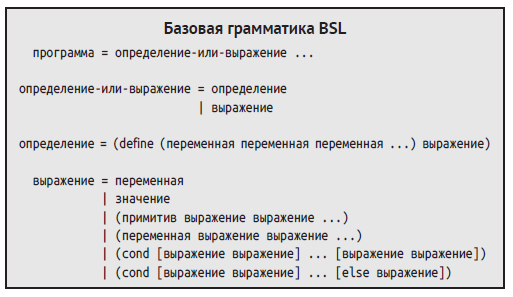












define, cond, and else. - called keywords.

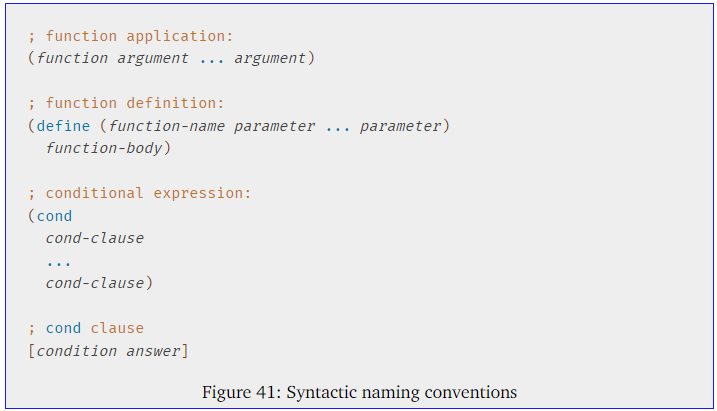
- Первая синтаксическая категория говорит, что программа – это определение-выражения. выражения могут ссылаться на определения;

- Вторая синтаксическая категория говорит, что определение-выражения является либо определением, либо выражением; программа в DrRacket в действительности состоит из двух отдельных частей: области определений и выражений в области взаимодействий.

- Последнее определение перечисляет все способы формирования выражения, и второе из них – это значение.

**keyword define** distinguishes definitions from expressions.

The **components of compound sentences have names**. We have introduced some of these names on an **informal basis**.



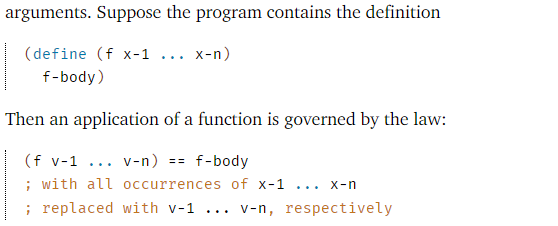
if a function definition

- вторая часть определения (вторые скобки) – **заголовок функции** (по математически – левая часть)

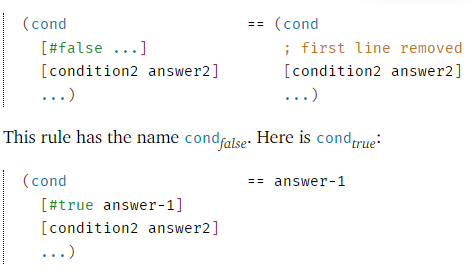
- expression после вторых скобок- **тело функции** (по математически – правая часть)

**BSL Meaning**

**beta rule (beta-value rule)**

****

**rules that determine the value of cond expressions**

****

**Meaning and Computing**

A scientist calls the stepper a model of DrRacket’s evaluation mechanism.

You can, and you ought to, use the stepper when you don’t understand how a new language construct works.

**You may also wish to use the stepper when you are surprised by the result that a program computes**. Using the stepper effectively in this way requires practice. For example, it often means copying the program and pruning unnecessary pieces. But once you understand how to use the stepper well this way, you will find that this procedure clearly explains run-time errors and logical mistakes in your programs.

**BSL Errors**

Вообщем если в выражении или функции заложена ошибка (например привходящей нулевой переменной получается где-то в выражении функции деление на ноль, вычислять когда дойдет дело до этой ошибки в выражениях cond, а также по сути можно с помощью правила

**Always choose the outermost (самое внешнее) and left-most nested expression that is ready for evaluation.**

С поомщью этоо правила можно понять как Степпер Dr.Racket считает арифметические выражения и cond (в какой последовательности ведет вычисления).

**Boolean Expressions**

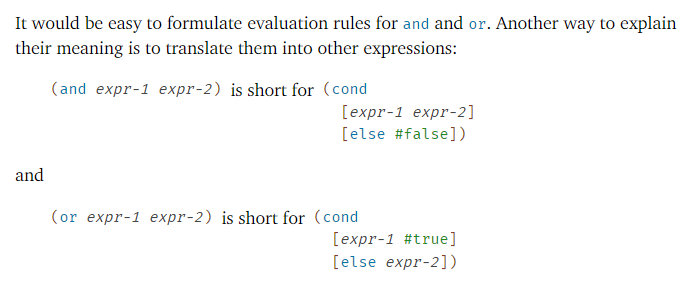
or and and expressions

understand their syntax and then their semantics

expr = (and expr expr)|(or expr expr)

and and or are keywords

They are not function applications



So if you are ever in doubt about how to evaluate an and or or expression, use the above equivalences to calculate

**Constant Definitions**

definition = (define name expr)

While the keyword [define](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) distinguishes constant definitions from expressions, it does not differentiate from function definitions. For that, a human reader must look at the second component of the definition.

Можно задавать опредления констант с помощью выражений

(define RADIUS 5)

(define DIAMETER (\* 2 RADIUS))

И с помощью функций (то что ниже легально)

(define RADIUS 10)

(define DIAMETER (\* 2 RADIUS))

(define (area r) (\* 3.14 (\* r r)))

(define AREA-OF-RADIUS (area RADIUS))

Только следить чтобы функция была определена до использования.

**Structure Type Definitions**

define-struct is the most complex BSL construct

definition= (define-struct name [name ...])

the use of define-struct extends the universe of values

For example, the definition of point adds values of this shape:

(make-point 1 2 -1)

(make-point "one" "hello" "world")

(make-point 1 "one" (make-point 1 2 -1))

the evaluation of

(define-struct c [s-1 ... s-n])

introduces the following functions into the program:

make-c: a constructor;

c-s-1... c-s-n: a series of selectors; and

c?: a predicate.

The relationship between the new data constructor and the selectors is best characterized with n equations added to BSL’s rules:

(c-s-1 (make-c V-1 ... V-n)) == V-1

(c-s-n (make-c V-1 ... V-n)) == V-n

The predicate c? can be applied to any value. It returns #true if the value is of kind c and #false otherwise. We can translate both parts into two equations:

(c? (make-c V-1 ... V-n)) == #true

(c? V) == #false if V is a value not constructed with make-c

**BSL Tests**

When you click the RUN button, DrRacket collects all testing expressions and moves them to the end of the program, retaining the order in which they appear. It then evaluates the content of the definitions area. Each test evaluates its pieces and then compares them with the expected outcome via some predicate.

**BSL Error Messages**

in this chapter actual list of errors that occur when writing programs

Иногда заглядывать сюда, когда допускаешь ошибки

**- When an error shows up and you need help, find the appropriate figure, search the entries for a match, and then study the complete entry**

**- An open parenthesis must always be followed by a keyword or the name of a function**

**II Arbitrarily Large Data**

- [self-referential definition first example list](#self_referential_definition_first)

Every data definition in Fixed-Size Data describes data of a fixed size. Many programming problems, however, deal with an undetermined number of pieces of information that must be processed as one piece of data. For example, one program may have to compute the average of a bunch of numbers and another may have to keep track of an arbitrary number of objects in an interactive game. Regardless, it is impossible with your knowledge to formulate a data definition that can represent this kind of information as data.

**This part revises the language of data definitions so that it becomes possible to describe data of (finite but) arbitrary size.**

**8 Lists**

While a **definition may in general contain several references to itself**, this chapter presents useful examples that need just one, starting with the one for lists.

The introduction of lists also spices up the kind of applications we can study.

**8.1 Creating Lists**

BSL comes with built-in support for creating and manipulating lists, similar to the support for Cartesian points (posn). In contrast to points, the data definition for lists is always left to you.

When we **form a** list, we always start out with the **empty list**.

'()

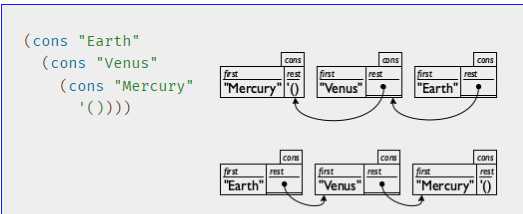
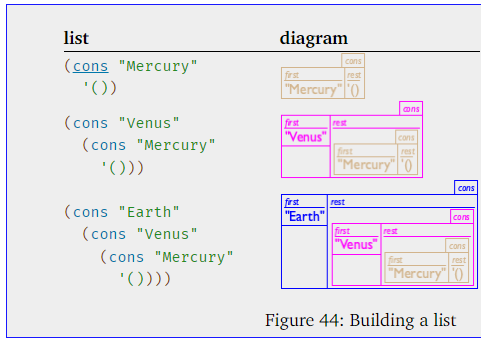
Like #true or 5, '() is just a constant.

When we add something to a list, we construct another list; in BSL, the **cons operation** serves this purpose.

(cons x y) → list? \\ Constructs a list

(cons "Mercury" '())

constructs a list from the '() list and the string "Mercury".



Hence the second diagram immediately tells you what first would have produced when applied to the list, no matter how long the list is. For this reason, programmers prefer the second arrangement.

In general a list does not have to contain values of one kind, but may contain arbitrary values:

(cons "Robbie Round"

(cons 3

(cons #true

'())))

Data definition uses cons:

; A List-of-names is one of:

; – '()

; – (cons String List-of-names)

; interpretation a list of invitees, by last name

This is **self-referential definition**

At a minimum, we can generate '() as one example, using the first clause in the itemization.

Given '() as an element of List-of-names, it is easy to make a second one:

(cons "Findler" '())

Here we are using a String and the only list from List-of-names to generate a piece of data according to the second clause in the itemization.

And while these lists all contain one name (represented as a String), it is actually possible to use the second line of the data definition to create lists with more names in them

(cons "Felleisen" (cons "Findler" '()))

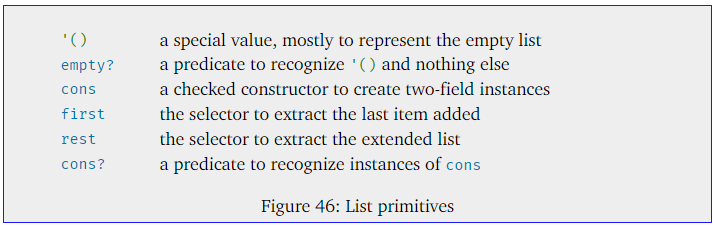
This piece of data belongs to List-of-names because "Felleisen" is a String and (cons "Findler" '()) is a confirmed List-of-names.

**8.2 What Is '(), What Is cons**

'() is a new kind of atomic value, distinct from any other kind: numbers, Booleans, strings, and so on. It also isn’t a compound value, like Posns.

'() is so unique it belongs in a class of values all by itself

cons appears to be the constructor for a two-field structure: the first one for any kind of value and the second one for any list-like value



What this chapter teaches, then, is not a new way of creating data but a **new way of formulating data definitions.**

**8.3** **Programming with Lists**

It doesn’t look too different from the definitions in the first chapter of the book.

It consists of a signature, a purpose statement, two examples, and a definition.

The only way in which this function definition differs from anything, that is, the self-reference (in the function). This shouldn’t be too surprising, because the data definition is self-referential, too.

**Важно.** При написании функции по обработке таких списков ссылающихся сами на себя лучше для обработки последнего ‘rest’ компонента использовать if или or. но не cond. если cond то обязательно использование else. прямо написать его. с последним может быть путаница (самый вложенный конд в дает тру а внешние все еще фалсе и ответ в итоге непрfвильный. **вообщем лучше or или if. cond только с прописанным else.**

Example:

Sample Problem

You are working on the contact list for some new cell phone. The phone’s owner updates and consults this list on various occasions. For now, you are assigned the task of designing a function that consumes this list of contacts and determines whether it contains the name “Flatt.”

0. The data definition

|  |
| --- |
| ; A *List-of-names* is one of: |
| ; – '() |
| ; – ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) [String](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._string%29) [List-of-names](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._list._of._name%29)) |
| ; **interpretation** a list of invitees, by last name |

1,2. turn to the header material (a signature, a purpose statement)

; List-of-names -> Boolean

; determines whether "Flatt" is on a-list-of-names

(define (contains-flatt? a-list-of-names)

#false)

3. we next make up some examples that illustrate the purpose of the function

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (contains-flatt? '()) #false)

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (contains-flatt? ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "Find" '())) #false)

([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) (contains-flatt? ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "A" ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "Flatt" ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "C" '()))))#true)

4. design a function template that matches the data definition

|  |
| --- |
| ; [List-of-names](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._list._of._name%29) -> [Boolean](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._boolean%29) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (contains-flatt? alon) |
| ([cond](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cond%29%29) |
| [([empty?](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._empty~3f%29%29) alon) #false] |
| [([cons?](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons~3f%29%29) alon) |
| ([...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) ([string=?](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._string~3d~3f%29%29) ([first](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._first%29%29) alon) "Flatt") [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) |
| [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29) (contains-flatt? ([rest](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._rest%29%29) alon)) [...](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._......%29%29))])) |

5. Programming a the complete definition

|  |
| --- |
| ; [List-of-names](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._list._of._name%29) -> [Boolean](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._boolean%29) |
| ; determines whether "Flatt" occurs on alon |
| ([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) |
| (contains-flatt? ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "X" ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "Y"  ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "Z" '())))) |
| #false) |
| ([check-expect](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._check-expect%29%29) |
| (contains-flatt? ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "A" ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "Flatt" ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) "C" '())))) |
| #true) |
| ([define](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._define%29%29) (contains-flatt? alon) |
| ([cond](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cond%29%29) |
| [([empty?](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._empty~3f%29%29) alon) #false] |
| [([cons?](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons~3f%29%29) alon) |
| ([or](http://docs.racket-lang.org/htdp-langs/beginner.html#%28form._%28%28lib._lang%2Fhtdp-beginner..rkt%29._or%29%29) ([string=?](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._string~3d~3f%29%29) ([first](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._first%29%29) alon) "Flatt") |
| (contains-flatt? ([rest](http://docs.racket-lang.org/htdp-langs/beginner.html" \l "%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._rest%29%29) alon)))])) |

#### 8.4 Computing with Lists

Programmers must have an intuitive understanding of how this kind of calculation works (if the function for processing self-referential data is defined in a step-wise fashion)

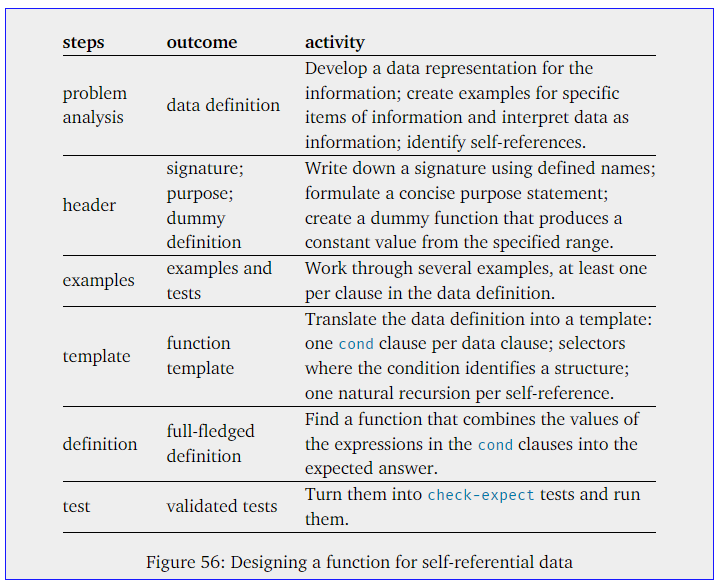
#### - 9 Designing with Self-Referential Data Definitions

In this section we generalize the design recipe so that it works even better for self-referential data definitions.

**The new parts concern the process of discovering when a self-referential data definition is needed**, **deriving a template, and defining the function body.**

figure 56 summarizes the design recipe of this section in a tabular format

practice helps you master the process



**1.** **define self-referential data definition**

If a problem statement is about information of arbitrary size, you need a self-referential data definition to represent it.

valid self-referential data (2 conditions):

- it must contain at least two clauses

- at least one of the clauses must not refer back to the class of data that is being defined

A List of strings is one of:

-- ‘()

-- (cons String (List of strings))

check the validity of self-referential data definitions with the creation of data examples:

-- Start with the clause that does not refer to the data definition

'()

-- continue with the other one, using the first example where the clause refers to the definition itself

(cons "a" '())

(cons "b" (cons "a"'()))

**If it is impossible to generate examples from the data definition, it is invalid. If you can generate examples but you can’t see how to generate increasingly larger examples, the definition may not live up to its interpretation.**

**2. the signature, the purpose statement, and the dummy definition**

focus on what the function computes not how it goes about it

example of concrete header material:

; List-of-strings -> Number

; counts how many strings alos contains

(define (how-many alos)

0)

there is no need to think ahead about how you might formulate this idea as a BSL function

**3. functional examples**

the purpose of a template is to express the data definition as a function layout.

work through inputs that use the self-referential clause of the data definition several times

given wanted

'() 0

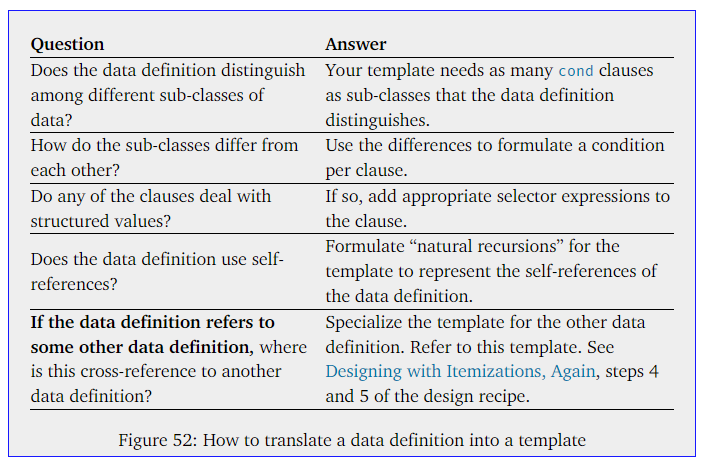
(cons "a" '()) 1

(cons "b" (cons "a" '())) 2

**4. the development of the template**

Specifically, **we formulate a cond expression with as many cond clauses** **as there are clauses in the data definition**, match each recognizing condition to the corresponding clause in the data definition, and write down appropriate selector expressions in all cond lines that process compound values.

Хорошая табличка, чек-ин по созданию шаблона функции



In the left column it states questions about the data definition for the argument, and in the right column it explains what the answer means for the construction of the template.

Apply the first three questions from the table to create *any* function that consume List-of-strings, we arrive to this template:

(define (fun-for-los alos)

(cond

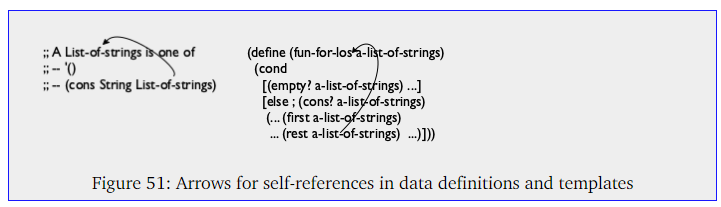
[(empty? alos) ...]

[else

(... (first alos) ... (rest alos) ...)]))

when a data definition is self-referential in the ith clause and the kth field of the structure mentioned there, the template should be self-referential in the ith cond clause and the selector expression for the kth field.

all important pieces of the data definition must find a counterpart in the template, and this guideline should also hold when a data definition is self-referential—contains an arrow from inside the definition to the term being defined. for each such selector expression, add an arrow back to the function parameter. At the end, your template must have as many arrows as we have in the data definition. (like at the figure 51)



use an alternative to the arrow, namely, a self-application of the function to the appropriate selector expression

completed function template for our example:

(define (fun-for-los alos)

(cond

[(empty? alos) ...]

[else

(... (first alos) ...

... (fun-for-los (rest alos)) ...)]))

self-use of a function – ***recursion***(in the first four parts of the book - natural recursion)

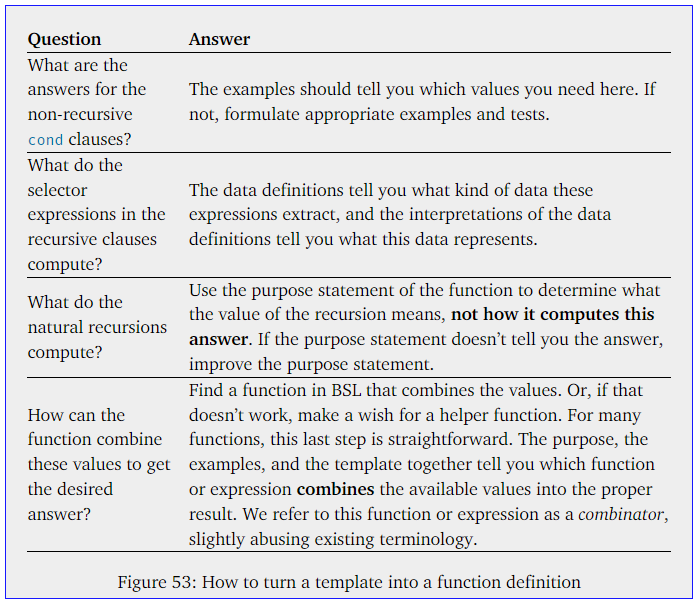
**5. deal with the self-referential cases**

For the function body we start with those cond lines without recursive function calls, known as base cases. The corresponding answers are typically easy to formulate or already given as examples. This is simple.

Then we deal with the self-referential cases.

We start by reminding ourselves what each of the expressions in the template line computes. For the natural recursion we assume that the function already works as specified in our purpose statement.

Formulates the first four questions and answers for this step. Let’s use this table to complete the definition clause with self-referential case.



For our example after answering of a question (we rename template name from from ‘fun-for-ios’ to name of specific function ‘how-many’)

; List-of-strings -> Number

; determines how many strings are on alos

(define (how-many alos)

(cond

[(empty? alos) ...]

[else

(... (first alos) ...

... (how-many (rest alos)) ...)]))

Thinking, answering a question and fill out the template

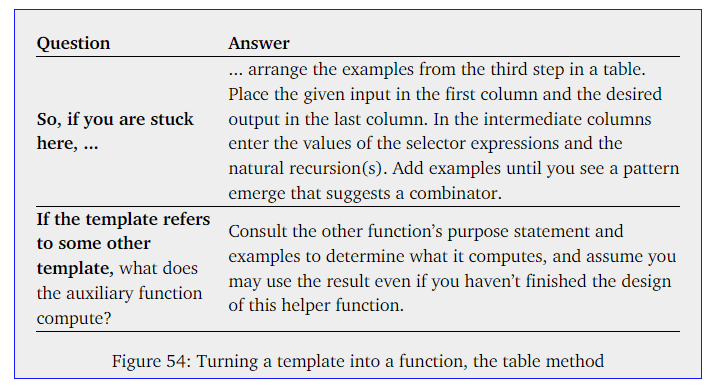
(define (how-many alos)

(cond

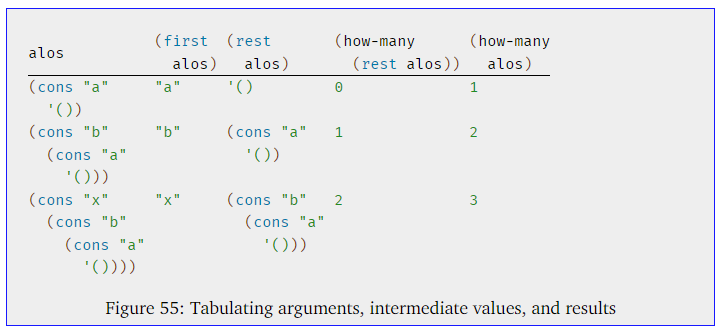
[(empty? alos) 0]

[else (+ (how-many (rest alos)) 1)]))

Если затруднения с созданием кода для рекурсии-само ссылающейся функции. Ответить для себя на вопросы из этой таблички



Табличка советует сделать другую табличку , в которой соотносятся тестируеммые входные значения, количество вызовов рекурсии и ожидаемый итог. Она должна помочь с написанием части функции с рекурсией.



Из этой таблички для нашего примера мы поняли, что ожидаемое количество на выходе (подсчет строк в списке) всегда на одно больше, чем вызовов рекурсии. Из этого родилось такое строка для кода функции

(+ (how-many (rest alos)) 1)

Из нее же мы поняли, что для подсчета количества строк нам не нужно (first alos) значение первого элемента списка вообще, мы заменяем его на 1 просто потому что он есть. (вообще надо бы проверять что в первом элементе именно строка, на автор на это забивает)

add more example rows to the table until you have a different idea.

Этот плюсик в финальной формуле для рекурсии очень важен с его помощью мы как бы комбинируем количество результатов вызовов рекурсии. Из этого получается итоговое значение работы основной функции. Иногда вместо комбинирования оперетивами ( +/-) надо написать отдельную функцию для такого комбинирования.

; List-of-strings -> Number

; determines how many strings are on alos

(define (how-many alos)

(cond

[(empty? alos) 0]

[else (+ (how-many (rest alos)) 1)]))

**6. turn examples into tests**

that these tests pass, and that running them covers all the pieces of the function

(check-expect (how-many '()) 0)

(check-expect (how-many (cons "a" '())) 1)

(check-expect (how-many (cons "b" (cons "a" '()))) 2)

Doing so also helps if you need to resort to the table-based guessing approach of the preceding step.

#### 9.2 Non-empty Lists

For inform future readers through the signature doesn’t work for empty lists (there are such cases), we need a data representation for lists that excludes '().

something like this:

(define ABSOLUTE0 -272)

; A CTemperature is a Number greater than ABSOLUTE0.

; An NEList-of-temperatures is one of:

; – (cons CTemperature '())

; – (cons CTemperature NEList-of-temperatures)

; interpretation non-empty lists of Celsius temperatures

(instead

; A List-of-temperatures is one of:

; – '()

; – (cons CTemperature List-of-temperatures)

)

While this definition differs from the preceding list definitions, it shares the critical elements: a self-reference and a clause that does not use a self-reference.

some examples of new data representation

base clause example:

(cons c '())

where c stands for a CTemperature, like thus: (cons ABSOLUTE0 '())

all non-empty elements of List-of-temperatures are also elements of the new class of data

(cons 1 (cons 2 (cons 3 '()))) fits the bill if (cons 2 (cons 3 '())) does, and (cons 2 (cons 3 '())) belongs to NEList-of-temperatures because (cons 3 '()) is an element of NEList-of-temperatures, as confirmed before.

Теперь в функции мы можем не проверять на наличие пустого листа во входных данных и не придумывать что с ним делать, если сама задача и определение данных в мире наличие пустого листа не предполагает. (снизу функция без конд с отдельным пунктом для пустого листа, а сразу обработка значений в листе, рекурсия конечно предполагается, весь само-ссыльные данные у нас же остаются)

; NEList-of-temperatures -> Number

; computes the average temperature

(check-expect (average (cons 1 (cons 2 (cons 3 '()))))

2)

(define (average ne-l)

(/ (sum ne-l)

(how-many ne-l)))

Вместо проверки пустого листа (в первом пункте cond) как до этого в empty list

; List-of-strings -> Number

; determines how many strings are on alos

(define (how-many alos)

(cond

**[(empty? alos) ...]**

[else

(... (first alos) ...

... (how-many (rest alos)) ...)]))

При обработке non-empty list мы проверяем в первом пункте cond

; NEList-of-temperatures -> Number

(define (sum ne-l)

(cond

**[(empty? (rest ne-l)) ...]**

[else ...]))

Напомню в первом пункте мы проверяем первое из двух типов данных указанных в перечне данных для листа. В empty list проверяли пустой потому что он указан в

; A List-of-temperatures is one of:

**; – '()**

; – (cons CTemperature List-of-temperatures)

)

В non-empty проверяем rest на пустоту чтобы обработать первый из двух элементов non-empty

; An NEList-of-temperatures is one of:

**; – (cons CTemperature '())**

; – (cons CTemperature NEList-of-temperatures)

; interpretation non-empty lists of Celsius temperatures

Т.е. при построении функции отталкиваемся от определения данных

Дале внизу тоже самое каждому first and rest свой предикат в функции

#### 9.3. Natural Numbers

one of several functions that produce lists

(make-list 2 "hello")

(cons "hello" (cons "hello" '()))

this function consumes atomic data, it produces arbitrarily large pieces of data.

In kindergarten you called these numbers **“counting numbers”, that is, these numbers are used to count objects**. **In computer science**, these numbers **are dubbed natural numbers**.

natural numbers come with a data definition:

; An N is one of:

; – 0

; – (add1 N)

; interpretation represents the counting numbers

to work with natural numbers, it is necessary to use not simple operators, but constructors specifically for working with them

add1

sub1 (Given any natural number m not equal to 0, you can use sub1 to find out the number that went into the construction of m.)

add1 is like cons and sub1 is like first and rest

distinguish 0 from those natural numbers that are not 0:

zero? which determines whether some given number is 0

positive? which determines whether some number is larger than 0

#### 9.5 Lists and World

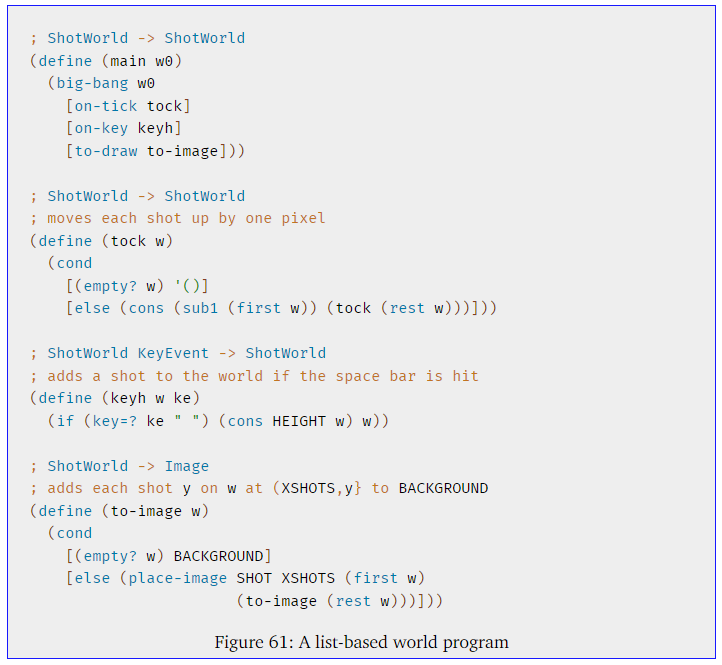
**Sample Problem** Design a world program that simulates firing shots. Every time the “player” hits the space bar, the program adds a shot to the bottom of the canvas. These shots rise vertically at the rate of one pixel per tick.

; A *Shot* is a [Number](https://htdp.org/2023-8-14/Book/part_one.html#%28tech._number%29).

; **interpretation** represents the shot's y-coordinate

|  |
| --- |
| ; A *List-of-shots* is one of: |
| ; – '() |
| ; – ([cons](http://docs.racket-lang.org/htdp-langs/beginner.html#%28def._htdp-beginner._%28%28lib._lang%2Fhtdp-beginner..rkt%29._cons%29%29) [Shot](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._shot%29) [List-of-shots](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._list._of._shot%29)) |
| ; **interpretation** the collection of shots fired |

|  |
| --- |
| ; A *ShotWorld* is [List-of-numbers](https://htdp.org/2023-8-14/Book/part_two.html#%28tech._list._of._number%29). |
| ; **interpretation** each number on such a list |
| ; represents the y-coordinate of a shot |



#### 9.6 A Note on Lists and Sets

**sets** - as collections of BSL values

There is one question that this book consistently asks about sets, and it is whether some element is in some given set. (4 is in Number, while "four" is not.)

difference between sets and lists

**sets - is a concept** we use to discuss steps in the design of code

**lists** - is one of many **forms of data** in BSL

*(in a straightforward sets are basically lists)*

lists and sets are related in that both are about collections of values, but they also differ strongly:

|  |  |  |
| --- | --- | --- |
| property | lists | sets |
| Membership (если задано условие что число должно присутствовать в множестве оно должно там присутствовать, если задано условие, что число может присутствовать в множестве только n-раз, оно должно там приуставать максимум n раз) | one among many | critical |
| ordering | critical | irrelevant |
| # of occurrences (если не задано условие для множества) | sensible | irrelevant |
| size | finite but arbitrary | finite or infinite |

Many of the sets mentioned in this book are infinitely large, for example, Number, String, and also List-of-strings. In contrast, a list is always finite though it may contain an arbitrarily large number of items.

**how sets are represented inside of BSL as data?**

while lists have a special status in BSL, sets don’t but at the same time sets somewhat resemble (similar) lists.

The key difference - is the kind of functions a program normally uses with either form of data.

**in a straightforward** and possibly naive manner says **sets are basically lists**.

BSL is not expressive enough to represent infinite sets, exercise 299 introduces a completely different representation of sets, a representation that can cope with infinite sets, too. The question of how actual programming languages represent sets is beyond the scope of this book, however.

Example for this:

let’s focus on lists of numbers:

we can use lists in two different ways to represent sets

|  |  |
| --- | --- |
| ; A Son.L is one of:  ; – empty  ; – (cons Number Son.L)  ;  ; Son is used when it  ; applies to Son.L and Son.R | ; A Son.R is one of:  ; – empty  ; – (cons Number Son.R)  ;  ; Constraint If s is a Son.R,  ; no number occurs twice in s |

Both of two data definitions basically say that a set is represented as a list of numbers.

the right definition comes with the constraint that no number may occur more than once on the list

After all, the key question we ask about a set is whether some number is in the set or not, and whether it is in a set once, twice, or three times makes no difference. (the second definition is not set?)

; Son

(define es '())

empty set, which in both cases is represented by the empty list.

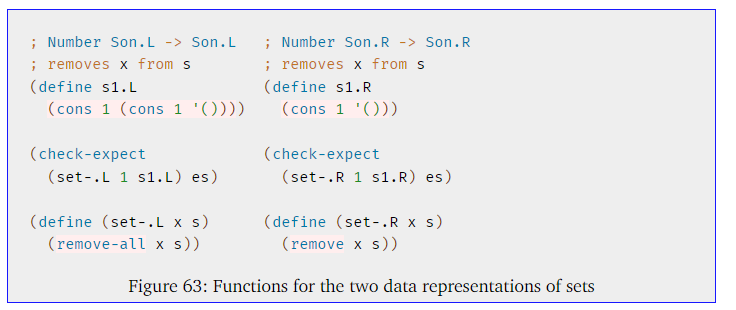
Not-empty sets:

(cons 1 (cons 2 (cons 3 '())))

(cons 2 (cons 1 (cons 3 '())))

(cons 1 (cons 2 (cons 1 (cons 3 '()))))

while **the order of cons cannot matter,** the constraint in the right-hand data definition rules out the last list as a Son.R because it contains 1 twice (т.е. **надо смотреть на условие определения множества, как именно его можно выразить через список,** здесь например правое множество нельзя выразить через список с повторяющимся значением любой цифры)



тут он показывает что смотря от того какое у нас определение множества, мы будем отталкиваясь от этого проектировать функции, работающие с ними (тут там где нельзя в множестве иметь два одинаковых числа у нас другое определение данных, и другое определение функции, нет нужды использовать remove-all, так как вхождение любого числа только одно)

(define set123-version1

(cons 1 (cons 2 (cons 3 '()))))

(define set123-version2

(cons 1 (cons 3 (cons 2 '()))))

(define set23-version1

(cons 2 (cons 3 '())))

(define set23-version2

(cons 3 (cons 2 '())))

(check-member-of (set-.v1 1 set123.v1)

set23-version1

set23-version2)

**чтобы грамотно спроектировать check-expect для функций, обрабатывающих множества надо использовать check-satisfied** (чтобы получить ответ для любого вида множеств, с любым количеством элементов, потому что выше, если у нас будет больше чем два, например 3 элемента в выходной функции, нам надо определять 6 возможный вариантов ответа и так далее, надо что то универсальное, с помощью check-satisfied)

**Правильно:**

1. проектируем проверочную функцию для check-satisfied (это легче чем бесконечное количество итоговых множеств для проверки при испозовании check-expect, check-member-of)

; Son -> Boolean

; #true if 1 is not a member of s; #false otherwise

(define (not-member-1? s)

(not (in? 1 s)))

2. Using not-member-1?, we can formulate the test case as follows

(check-satisfied (set- 1 set123) not-member-1?)

#### 10 More on Lists

**Lists are a versatile form of data that come with almost all languages now.** Programmers have used them to build large applications, artificial intelligences, distributed systems, and more. This chapter illustrates some ideas from this world, including functions that create lists, data representations that call for structures inside of lists, and representing text files as lists.

#### 10 Functions that Produce Lists

In case you want to practice the development of templates, use the questions from figure 52.( 9 Designing with Self-Referential Data Definitions)