BOOTSTRAP
www.bootstrapworld.org

Grading Rubric for Student Programs

Obviously, programmers care that their programs produce the right answers. Good programmers, however, care about much more! They also care that programs are easy to read, understand, and modify. Sometimes, a programmer has to change their code a few weeks or months later to make it do something different. Sometimes, a programmer has to change someone else's code to make it do something different. A high-quality program is designed to be maintained over time, potentially by people other than the original programmer.

The guidelines in the rest of this document target different aspects of creating readable and maintainable code.

Grading a Design Recipe

The design recipe is not just a method for developing a program: it also provides useful guidelines for grading student work. When grading a program, consider the following questions:

Contract and Purpose

Every part of a student's Contract and Purpose statement comes directly from the original word problem.

- Does the contract include all three parts: a `name`, `domain` and `range`?

- Does the `name` reflect what the function computes?

> *Write a function that takes in a topping and returns the value of a pizza with that topping*
>
> - Confusing: `pizza: string -> number`
> - Bad: `c: string -> number`
> - Good: `cost: number -> number`

Even though math textbooks often use generic function names like *f* and *g*, when writing programs we prefer descriptive names because they will help us remember what the function does when we refer back to it later.

- Are the domain(s) and the range `types`, rather than descriptive terms?

> *Write a function 'red-star' that takes in a radius and draws a solid red star of that size.*
>
> - Bad: `red-star: size -> image`
> - Better: `red-star: number -> image`

Good contracts are clear about what kind of data can be provided to the function. Types (e.g. `number`, `string`, `image` and `boolean`) are unambiguous, whereas descriptive types could be interpreted differently by different people. *The descriptive content belongs in the Purpose Statement*.

- Does the Purpose Statement contain all of the relevant parts of the original word problem, and nothing else?

> *A car is moving at 55mph down a country road. Write a function that describes its distance as a function of time*
>
> - Copying: `a car goes 55mph. write a function that describes distance as a function of time.`
> - Bad: `how long has the car been moving?`
> - Bad: `how far did the car go?`
> - Weak: `given a` **`time`**`, how far did the car go on a country road?`
> - Weak: `given a` **`time`**`, how far did a car moving` **`55mph`** `go?`
> - Weak: `how far did the car go in a given # of` **`hours`**`?`
> - Good: `given a #` **`hours`**`, how far did a car moving` **`55mph`** `go?`

Good purpose statements should be a thoughtful restatement of the original problem, in the student's own words.They should include *all* of the relevant information from the problem.

Examples

Every part of a student's Examples should trace back to the Contract and Purpose Statement. If an example cannot be completely written based on the contract and purpose, check back to the previous section to see what the student is missing.

- Do the function's name and types of input and output match the contract?

> *Write a function 'red-star' that takes in a radius and draws a solid red star of that size.*
>
> ```
> ; red-star : Number -> Image
> ; given a radius, draw a solid red star of that size
> ```
>
> - Bad: `(EXAMPLE (rs 50 "red") 50))`
> - Bad: `(EXAMPLE (rs 126) (star 126 "solid" "red"))`
> - Good: `(EXAMPLE (red-star 65) (star 65 "solid" "red"))`

- Is at least one example easy enough that the student can be sure they have the right answer? It's always good to have a baseline to start from!

A car is moving at 55mph. Write a function that describes it's distance as a function of time.

```
; distance : Number -> Number
; given # hours, how far did a car moving 55mph go?
```

- Good:
```
(EXAMPLE (distance 0)
         (* 0 55))
```

- Do the examples use *different* values for each input?

Write a function 'red-star' that takes in a radius and draws a solid red star of that size.

```
; red-star : Number -> Image
; given a radius, draw a solid red star of that size
```

- Bad:
```
(EXAMPLE (red-star 50)
         (star 50 "solid" "red"))
(EXAMPLE (red-star 50)
         (star 50 "solid" "red"))
```
- Good:
```
(EXAMPLE (red-star 19)
         (star 19 "solid" "red"))
(EXAMPLE (red-star 22)
         (star 22 "solid" "red"))
```

Varying the inputs in the examples helps make sure that the function actually *uses* all of the inputs. In the example above, the student might accidentally use 50 as the size in their function definition. Since all of the examples also used 50 as the size, the student wouldn't discover the problem.

- Is there an example for each *interesting* input?

Write a function 'safe-left?' that takes in an x-coordinate and checks whether if it is greater than 50

```
; safe-left? : Number -> Boolean
; is the given x-coordinate > -50?
```

- Good:
```
(EXAMPLE (safe-left?  500)  (> 500 -50))
(EXAMPLE (safe-left?  -50)  (> -50 -50))
```

As your students progress in the curriculum, picking good examples is one of the most important skills they can develop. Think about a function like `safe-left?`, that detects whether a character is within the screen on the left side. A good set of examples would cover three situations: one in the middle of the screen, one at the left edge, and one off-screen just beyond the left edge. Whenever a function should produce different answers around some boundary, a good set of examples will check the inputs that define the boundary, as well inputs on either side of the boundary.

For functions that produce booleans…

- ❖ are there are examples that would produce each of true and false?

For functions that use `cond`…

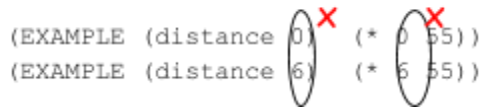
- ❖ Is there at least one example that satisfies each of the conditions?
- ❖ Is there an example that covers the 'else' condition?

- Are the variables circled, and given names that come from the purpose statement?

A car is moving at 55mph. Write a function that describes it's distance as a function of time.

```
; distance : Number -> Number
; given # hours, how far did a car moving 55mph go?
```

- Bad:



- Good:

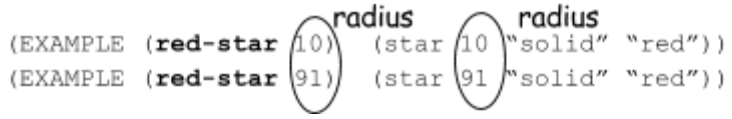
Definitions

Every part of a student's definition should be able to be traced back to the Examples and the labeled variables. If the definition cannot be completely written based on the Examples and labels, check back to the previous section to see what the student is missing.

- Does the function header match the pattern established in the examples?
  - Does the name in the header match the one in the examples?
  - Do the number and name of the variables to the function match the number and names of the variables labeled in the examples?

Write a function 'red-star' that takes in a radius and draws a solid red star of that size.

```
; red-star : Number -> Image
; given a radius, draw a solid red star of that size
```



- Bad: `(define (rs x color) (star x "solid" color))`
- Bad: `(define (rs x) (star x "solid" color))`
- Weak: `(define (red-star x) (star x "solid" "red"))`
- Good: `(define (red-star radius) (star radius "solid" "red"))`

- Does the code have line-breaks and indentation in places that make the code easy to read?

- Bad:
```
(define (distance x1 x2 y1 y2)
  (sqrt (+ (sqr (- x2 x1)) (sqr (- y2 y1)))))
```

- Overkill:
```
(define (distance x1 x2 y1 y2)
  (sqrt (+ (sqr
             (- x2
                x1))
           (sqr
             (- y2
                y1)))))
```

- Good:
```
(define (distance x1 x2 y1 y2)
  (sqrt (+ (sqr (- x2 x1))
           (sqr (- y2 y1)))))
```