

THE BASIC DESIGN RECIPE

Susan Older and Jim Royer
January 2008 revision for CIS 252

We shall use a general *design recipe*¹ when developing programs in this course. The intent of the design recipe is to provide you with three things:

1. A series of questions that help guide you through the design process,
2. A way to tie up the initial program specification with the testing of the completed (but maybe not correct) program, and
3. A scheme for explaining what the program does, together with pre-packaged tests to convince the reader the program is as advertised.

Below we discuss the *basic* design recipe. We shall revise the recipe over time to cover new ways to construct programs. Right now we are presuming only the “high-school algebra” part of Haskell.² Here is a tiny, complete example of the end result of following the basic design recipe. (Recall that anything following a “--” is a comment in Haskell.)

```
-- CONTRACT
--   areaOfRing :: number -> number -> number

-- PURPOSE
--   Takes two numbers, outer and inner, and computes the area of
--   a ring with radius outer and with a hole with radius inner.

-- EXAMPLES
--   (areaOfRing 5 3) should return 50.24.
--   (areaOfRing 5 4) should return 28.26.

-- DEFINITION
areaOfRing outer inner = (areaOfDisk outer) - (areaOfDisk inner)

-- TESTS
ans1 = areaOfRing 5 3 -- should be 50.24
ans2 = areaOfRing 5 4 -- should be 28.26
```

We shall use the above as a running example in discussing the parts of the design recipe below.

¹ This design recipe is adapted the text *How to Design Programs* by Felleisen, Findler, Flatt, and Krishnamurthi [FFFK01]. Their book is on the web [here](#), and their first discussion of the design recipe is [here](#). Note that, instead of Haskell, they are the language Scheme (a second cousin to Haskell), so the finer details of that section will not apply here. We shall also use their “area of a ring” example in explaining the ingredients of the design recipe.

²Which is just like ordinary high school algebra with an enlightened view of syntax.


Parts of the design recipe

The design recipe is essentially a programming version of Polya's *How to Solve It* scheme [Pol45]. Polya tells us that our first step is to understand the problem. With a program this usually comes down to answering:

1. What kind of inputs should the program take?
2. What kind of output should the program produce?
3. How does the output depend on the input? (i.e., what's computed?)

Question 3 is the interesting one, but first let's deal with 1 and 2.

Recipe step 1: The contract

Example. We want a program to compute the area of a ring. A *ring* consists of a disk with a circular hole in it. [Polya: Draw the picture!] Here is a picture: 

Our givens are: the radius of the outer disk and the radius of the inner disk. The area of the ring is going to be a number too.

So the answer to Question 1 is that the inputs are two numbers and the answer to Question 2 is that output is a number.

In the program text we state our answer to Questions 1 and 2 with a *contract*. For our ring-area example, the contract would be:

```
-- areaOfRing :: number -> number -> number
```

The general form of a contract for an n -input program is:

```
-- <program-name> :: <type1> -> ... -> <typen> -> <typeout>
```

(Later on, we shall generalize the stuffings out of this.) For the time being think of “:” and “->” just as Haskell punctuation. The funny things of the form $\langle name \rangle$ are “fill in the blank” variables. The name of your program goes in the $\langle program-name \rangle$ slot. The type of the i -th input goes in the $\langle type_i \rangle$ slot, and the type of the output goes in the $\langle type_{out} \rangle$ slot. *Type* is the formal name of “the kind of data.” Types can describe numbers, character strings, lists, pictures, trees, and on and on.

It turns out that Haskell cares intensely about contracts. We will later place our contracts outside of comments so the Haskell interpreter can check them for us.

What is the point of contracts?

1. If you do not understand what sort of inputs your program should take and what sort of output it produces, you most definitely do not understand the problem your program is trying to solve.
2. Programs are typically constructed by putting together smaller programs in various ways. The contracts let us (and Haskell) check whether we are trying to “place a square peg in a round hole.”³

Recipe step 2: Analysis and statement of the purpose

Now let us deal with Question 3: *How does the output depend on the input?*

The example, continued. In this example, we are working with a problem that is reasonably well stated. (Life is usually not so easy.) So we can use the problem statement as the purpose statement.

```
-- PURPOSE
-- Takes two numbers, outer and inner, and computes the
-- area of a ring with radius outer and with a hole
-- with radius inner.
```

Let’s dig a bit deeper into the “what is computed” problem. We know from high-school algebra that the area of a disk of radius r is: $\pi \cdot r^2$. So the ring’s area is:

$$(\text{the big ring's area}) - (\text{the hole's area}) = \pi \cdot r_o^2 - \pi \cdot r_i^2,$$

where r_o is the radius of the outer disk and r_i is the radius of the inner disk. So the problem we will have in the definition step is translating this math into Haskell.

The goals of this step is to understand the problem well enough so as to do the following:

1. Give a cogent English⁴ advertisement for what the program is supposed to do.
2. Be able to compute, by *ad hoc* means, what the program should produce on some particular inputs.

What is the point of the purpose analysis and statement? The intent of the first goal is pretty clear—especially if you have ever tried to read someone else’s uncommented program. The

³ When an application program crashes on you, it is often because the language in which the program is written (e.g., C or C++) permitted “a square peg in a round hole.”

⁴ Or French, Chinese, Hindi, Etruscan, ... as the case may be.

contract gives you the general shape of the program. The purpose statement fills in the contents of that shape.

The second goal may seem silly since you are shortly going to write the program anyhow. The point of doing some hand calculations is to play computer with concrete data and to try to think about possible ways of doing the computations.

Recipe step 3: Examples

The example, continued. We decide to use 3.14 as an approximation to π . So we can compute some values of what `areaOfRing` should produce. Specifically:

```
-- EXAMPLES
-- (areaOfRing 5 3) should return 50.24.
-- (areaOfRing 5 4) should return 28.26.
```

The goal here is to produce concrete examples of what the program should do. In particular, you want to identify key examples that illustrate tricky cases that the program needs to handle.

What is the point of the examples?

1. Working through some cases, particularly tricky ones, with concrete data makes you think through these cases. This helps refine your understanding from step 2.
2. Making these examples explicit helps set a standard with which you can measure the success of program in the testing step. Coming up with the examples before the program gets written helps keep the examples unbiased.⁵

Recipe step 4: The definition

The example, continued. To keep our story short, we suppose we just happen to have lying around the program

```
areaOfDisk r = 3.14 * r * r
```

that computes the area of a disk with radius `r`. So, based on step 2,

```
areaOfRing outer inner = (areaOfDisk outer) - (areaOfDisk inner)
```

will serve as our definition of `areaOfRing`.

The goal here is to produce a correct definition of our program based on the analysis of the previous three steps.

⁵ This approach to programming goes under the name *testing-first programming* and is much beloved by the extreme programming folks. For example, see [here](#) and [here](#).

What is the point of the definition? Well, we are doing computing, not (just) mathematics! Our definition is a *purported* algorithmic solution to the original problem.

N.B.⁶ This step of the design recipe will be greatly expanded in our refinements of the recipe. When dealing with “high-school algebra” programs, there isn’t much to say here.

Recipe step 5: Tests

This step corresponds to Polya’s question *Can you check your result?* in his looking-back step.

The example, concluded. Now we are ready to run some tests of our definition. All of the examples from step 3 become test examples.

```
-- TESTS
ans1 = areaOfRing 5 3 -- should be 50.24
ans2 = areaOfRing 5 4 -- should be 28.26
```

When we look at the values of `ans1` and `ans2` they agree with the target values. So, we declare success—we (more or less) trust our definition.

The goal here is to put your program through its paces on some well-chosen data.

What is the point of the tests? If there are difficulties with your definition and if you chose good test cases, then those difficulties are likely to show up here. (The goodness of your test conditions needs to be established in steps 2 and 3.) So if your program passes your test, you have good evidence for the correctness of your program.

N.B. To guarantee correctness with just testing, you have to check *all possible inputs*, and there are almost always too many inputs to do this. In the case of our pitiful little example, if we assume our inputs are 64-bit floating point numbers, then a little calculation shows us that there are

340,282,366,920,938,463,463,374,607,431,768,211,456

possible test cases.⁷

The after-dinner step: Can we do better?

Producing a well-documented, well-tested program is just one of the end results of the design process. Another less obvious one is that we’ve learned something about the algorithmic problem we’ve just solved—and you have often learned more than you have realized. So, as Polya suggests, can we look back and see, if we can solve the problem in different way or use what we’ve learned here in another place. This can be where the fun starts.

In the case of our little example, we’ll leave well enough alone.

⁶“N.B.” abbreviates the latin phrase *nota bene* which means “note well”, that is, pay special attention.

⁷ As a point of comparison, the age of the Earth in seconds is about 200,000,000,000,000,000 ($= 2 \times 10^{17}$).

A summary

The design recipe has five basic steps.

1. *The contract*

- What kind of inputs should the program take?
- What kind of output should the program produce?

2. *Analysis and statement of the purpose*

- How does the output depend on the inputs?
- Can I compute what the result should be for some sample inputs?

3. *Examples*

- What are some typical inputs and their results?
- What are some tricky inputs to handle?

4. *Definition*

- How can I apply what I learned from steps 2 and 3 in program form?
... (*More to come.*) ...
- Have I handled all the cases?

5. *Testing*

- What (if anything) went wrong?
- Suppose the program returns unexpected results.
 - Is the program wrong?
 - Are some target values wrong?
 - What is the source of the confusion?
- Am I testing all the cases I should?

References

- [FFFK01] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi, *How to design programs: An introduction to computing and programming*, MIT Press, 2001.
- [Pol45] G. Polya, *How to solve it*, Princeton University Press, 1945.