# No Magic in CS1: How to Design Programs

**Matthias Felleisen** Northeastern University
**Robert Bruce Findler** University of Chicago
**Matthew Flatt** University of Utah
**Shriram Krishnamurthi** Brown University

## 1 Introduction

Many graduates of an introductory programming course

- cannot describe a process for going from a problem statement to its solution in the form of a program;
- cannot reliably distinguish a correct solution program from an incorrect one;
- and cannot even explain how a correct program arrives at its result for a given input.

The reason is that many such courses still convey a view of programming as tinkering and of computers as magical machines. In particular, courses hardly ever present an *explicit* model of programming, but rely on *implicit* learning via a series of examples.

Over the past few years, we have developed an alternative approach [1]. It builds on an understanding of arithmetic and pre-algebra that students have developed in elementary through high school. Based on this understanding, we teach a simple but rigorous model of computation. We then introduce a systematic process for developing programs. The process is based on reasoning about classes of data and is thus an ideal introduction to class-oriented programming.

Sections 2, 3, and 4 describe our models of computation and programming. Section 5 is a brief experience report.

## 2 Computation

Students are taught to compute from their earliest classroom days. In elementary school, students learn computational rules such as

$$\begin{array}{l} 2 + 3 \ \rightarrow\ 5 \\ 4 * 7 \ \rightarrow\ 28 \end{array}$$

They also learn how to perform computations that involve sub-calculations, such as

$$2 + (3 * 7) \ \rightarrow\ 2 + 21 \ \rightarrow\ 23$$

After several years of learning arithmetic, a student's computational skills grow radically in pre-algebra. Students learn to combine a function definition, such as

$$f(x) = x + 1$$

with a use of the function, such as

$$f(2) \ \rightarrow\ 2 + 1 \ \rightarrow\ 3$$

They find the definition of $f$, discover that its argument is named $x$ and that its right-hand side is $x + 1$, and then substitute 2 for $x$ in $x + 1$ to replace the function invocation. Even with functions, the principle of subcomputation continues to apply:

$$f(f(2)) \ \rightarrow\ f(2+1) \ \rightarrow\ f(3) \ \rightarrow\ 3+1 \ \rightarrow 4$$

Since a pre-algebra graduate has written pages of such computations (under instructions to "show all work"), there is no magic in how $f(f(2))$ evaluates to 4.

### 2.1 Adjusting Notation

Traditional algebra exploits dozens of informal notational conventions. These make it easy for (experienced) people to communicate in its notation. But, it also makes it difficult for programming language software to communicate with people. We therefore pick a more regular notation, both to simplify the interface with the programming environment, and to reinforce the notion that programming is a predictable exercise—right down to the error messages.

We choose a syntax based on Lisp and Scheme:

- Operators appear before their operands.

$$1 + 2 \implies (+\ 1\ 2)$$

- Every use of an operator is immediately preceded by "(". The last argument is followed by ")". Parentheses are not optional, and extra parentheses are not allowed.

$$1 + 2 * 4 \implies (+\ 1\ (*\ 2\ 4))$$

- Function calls obey the same syntax as uses of built-in operators. The function name is preceded by a left parenthesis, and the last argument is followed by a right parenthesis.

$$f(f(2)) \implies (f\ (f\ 2))$$

- The = in a function definition becomes **define** and moves to the beginning of the definition. The left-hand side mimics the function-call syntax. The entire definition is wrapped in parentheses.

$$f(x) = x + 1 \implies (\textbf{define}\ (f\ x)\ (+\ x\ 1))$$

In our experience, this simple translation helps most students overcome any reservations they may have about our notation.

**Note**: A vital aspect of our programming environment is that it recognizes precisely the language that we define for teaching purposes, no more; further, this language grows as the term progresses. Students therefore never encounter error messages that use terminology they have not seen before, thereby avoiding the notion that the computer is an unfathomable magic box.

### 2.2 Rich Algebra of Data

To support typical programming tasks with our computational model, we enrich the set of data and computations. For example, we add booleans—true and false—and operators like < and =. We provide rules of evaluation, which often build on arithmetic relations that the students know already.

$$\begin{array}{ll}(<\ 1\ 2) & \rightarrow \mathsf{true}\\ (=\ 1\ 2) & \rightarrow \mathsf{false}\\ \ldots\end{array}$$

We do not restrict ourselves to values described with letters and numbers. We also provide graphical values, with intuitive evaluation rules.

$$\begin{array}{ll}(\mathsf{empty\text{-}image}\ 4\ 4) & \rightarrow \square\\ (\mathsf{draw\text{-}point}\ \square\ 2\ 2) & \rightarrow \boxed{\cdot}\\ (\mathsf{draw\text{-}point}\ \boxed{\cdot}\ 2\ 3) & \rightarrow \boxed{:}\\ \ldots\end{array}$$

In this way, we can reinforce our model with examples that are more compelling than mere arithmetic.

$$\begin{array}{l}(\mathsf{draw\text{-}point}\ (\mathsf{empty\text{-}image}\ 4\ 4)\ 2\ (+\ 1\ 1))\\ \quad\rightarrow (\mathsf{draw\text{-}point}\ \square\ 2\ (+\ 1\ 1))\\ \quad\rightarrow (\mathsf{draw\text{-}point}\ \square\ 2\ 2)\\ \quad\rightarrow \boxed{\cdot}\end{array}$$

Another way to enrich the algebra is to use conditionals. The conditional form we use is **cond**, whose syntax is

$$\begin{array}{l}(\textbf{cond}\\ \ (\textit{question-expr}\ \ \textit{answer-expr})\\ \ \ldots\\ \ (\textit{question-expr}\ \ \textit{answer-expr}))\end{array}$$

It corresponds to the traditional algebraic notation

$$\begin{cases}\textit{answer-expr} & \text{if} \quad \textit{question-expr}\\ \ldots\\ \textit{answer-expr} & \text{if} \quad \textit{question-expr}\end{cases}$$

Two evaluation rules describe **cond** computations. The first rule is that when the initial question is true, the **cond** expression evaluates to the corresponding answer.

$$(\textbf{cond}\ (\mathsf{true}\ 17)\ ((<\ 5\ 13)\ 9)) \rightarrow 17$$

The second rule is that when the initial question is false, the question and its answer are discarded.

$$(\textbf{cond}\ (\mathsf{false}\ 17)\ ((<\ 5\ 13)\ 9)) \rightarrow (\textbf{cond}\ ((<\ 5\ 13)\ 9))$$

If the question expression of the first clause of **cond** is not true or false, then a sub-computation must be applied to reduce the question expression, just like computing the arguments for a primitive operation.

$$(\textbf{cond}\ ((<\ 5\ 13)\ 9)) \rightarrow (\textbf{cond}\ (\mathsf{true}\ 9)) \rightarrow 9$$

No evaluation rule exists for (**cond**), which implies that if all questions produce false, an error is reported.

## 3 Design

Our ultimate goal is to have students implement their own programs, not just evaluate our programs. To this end, we teach a class-based approach to designing programs. Students first learn that they must represent information from the real world as data in their programming language. This very first decision, and writing down the decision in the form of a data class definition, drives the rest of the program design.

The key element of our process is the notion of a *design recipe*. It consists of six steps, with well-defined outcomes, and guides the student through the implementation process. When a student is stuck, a teacher can ask for the intermediate outcomes. For each intermediate outcome, we provide suggestive questions that help students overcome a roadblock. In turn, the output of the steps forms a portfolio of the student's work on the problem, resulting to a much richer grading rubric than examining the final program alone. In the remainder of this section, we illustrate the essential elements of the design recipe through a series of examples.

### 3.1 Design Recipe

Figure 1 shows the general structure of the design recipe. We refine these steps (via question and answer games) for the variety of data definitions that programmers typically encounter. If the program uses a built-in class of data, say *nums*, the design process begins with the specification of a function's *contract*. The contract defines the kind of data that are given to a function, and the kind of data that it returns.

1. Analyze and describe the classes of problem data.
2. Formulate a contract and purpose statement.
3. Illustrate the purpose statement with examples.
4. Create a function layout based on this information.
5. Write the body of the function.
6. Turn the examples into (automatic) test cases.

Figure 1: The Design Recipe, General Structure

For example, suppose the problem is to write a *feed-it* function, which consumes (a representation of) a zoo animal and returns the animal after reeding it five pounds of food. If we only care about the weight of the animal, a number is a suitable representation.

Since *feed-it* consumes and produces a number, its contract and purpose are[1]

```
;; feed-it : num → num
;; to feed an animal 5 pounds of food
```

After the student has written these, the next step is to make up examples. Together with the contract, this step helps ensure that a student understands the task at hand. Each example shows a call of the function with a particular argument, and also shows the expected result.

```
(feed-it 20) = 25
(feed-it 40) = 45
```

For atomic forms of data, such as *num*, the design recipe offers little more advice, except that the definition must have the form

```
(define (feed-it a) ... a ...)
```

The recipe suggests the *a* in the body of the function to remind the student that this information is available for implementing the function body. The student must then arrive at the definition

```
(define (feed-it a) (+ a 5))
```

This step is the creative part of programming, and no design recipe can automate it.

Given a complete function, the recipe requires one addition step: executing the examples to test the function. Well chosen examples also serve as effective tests.

## 3.2 Design with Structure Types

Most likely, we will want to track more about our zoo animals than just the animal's weight. Suppose that our zoo contains zebras, and we want to keep track of each zebra's weight and stripes. Although each of those quantities can be represented as a number, a complete zebra representation must combine the numbers.

To define new kinds of compound data, we introduce one last piece of syntax. The **define-struct** form declares a structure type, given a name for the structure and a name for each field. For example, we declare a zebra structure with

```
(define-struct zebra (weight stripes))
```

This declaration introduces a *value constructor* called make-zebra that combines two values. Thus, (make-zebra 250 10) is a value, representing a 250-pound zebra with 10 stripes. This value has the same status as the number 7 or the boolean false, which means that it does not evaluate to anything else. Fields in a construction expression that are not yet values get evaluated to yield values. Thus,

```
(make-zebra (+ 250 5) 10)  →  (make-zebra 255 10)
```

and (make-zebra 255 10) is the result.

In addition to make-zebra, the declaration above also defines zebra-weight, zebra-stripes, and zebra?, and it establishes the following evaluation rules.

```
(zebra-weight (make-zebra X Y)) → X
(zebra-stripes (make-zebra X Y)) → Y
(zebra? (make-zebra X Y)) → true
(zebra? anything-else) → false
```

So, (zebra-stripes (make-zebra 250 10)) evaluates to 10.

Nothing about the *zebra* declaration requires the slots of a make-zebra construction to contain numbers, though that was our intent. To make this intent clear, we write down a *data definition*.

```
;; A zebra is (make-zebra num num)
```

That is, when *zebra* appears in a contract, it refers to a value constructed with make-zebra where the slots contain numbers. We use *zebra* to write a contract for *feed-zebra*:

```
;; feed-zebra : zebra → zebra
;; to feed a zebra 5 pounds of food
```

The data definition for *zebra* then guides the construction of examples for *feed-zebra*. We start with

```
(feed-zebra
```

and then consult the contract for *feed-zebra*, where we see *zebra* to the left of the arrow. Now we consult the data definition for *zebra*, which says that every *zebra* starts with (make-zebra. Hence, we get

```
(feed-zebra (make-zebra
```

The data definition further says that two numbers appear after make-zebra. So, we pick two numbers:

```
(feed-zebra (make-zebra 250 10))
```

Now we write the expected answer for this particular call. Given a (representation of a) 250-pound zebra with 10 stripes, we expect to get back a (representation of a) 255-pound zebra with 10 stripes.

```
(feed-zebra (make-zebra 250 10)) = (make-zebra 255 10)
```

After making examples, we are ready to implement the function. In the case of compound data, the design

[1]We write the contract and purpose in comments, indicated by the semicolon at the beginning of the line.

recipe offers additional guidance. It directs us to create a *template* for the implementation:

```
(define (feed-zebra z)
  ... (zebra-weight z) ... (zebra-stripes z) ...)
```

The presence of (zebra-weight *z*) and (zebra-stripes *z*) in the template body is driven by the contract and data definition. The contract says that *z* is a *zebra*, and the data definition says that we can extract two parts from a *zebra*. To complete the function, we can take the suggestions of the template and combine them to complete the function:

```
(define (feed-zebra z)
   (make-zebra (+ 5 (zebra-weight z))
               (zebra-stripes z)))
```

The template is a crucial step in the design recipe, even for functions on simple kinds of compound data. The template reminds a student which pieces of data are readily available for constructing an answer.

### 3.3 Design with Multiple Structures

To make our zoo more interesting, we could add snakes, keeping track of each snake's weight, length, and whether it is poisonous. Clearly, we need a new structure and data definition.

```
(define-struct snake (weight length poisonous))
;; A snake is (make-snake num num bool)
```

Given this data definition, we could write *feed-snake*. Eventually, we want a single function that feeds an arbitrary animal. The next step, then, is to define *animal*.

```
;; An animal is either
;;   a zebra, or
;;   a snake
```

This data definition differs from the previous definitions in that it has no associated structure, but also in that it contains a choice. Suppose we want to define a *feed-animal* function, with the contract

```
;; feed-animal : animal → animal
;; to feed an animal 5 pounds of food
```

When making examples for this function, we have a choice for arguments. Indeed, the design recipe prescribes at least one example for each choice for a data definition like *animal*. So our examples might include

```
(feed-animal (make-zebra 250 10))
  = (make-zebra 255 10)
(feed-animal (make-snake 10 12 false))
  = (make-snake 15 12 false)
```

The design recipe dictates a template where the function body is a **cond** expression. The **cond** expression must have two cases, because the data definition has two choices. The question for the first case is (zebra? *a*) because the first case in the data definition is *zebra*, and the question for the second case is (snake? *a*). Finally, the design recipe instructs us to select or write a function to process the data of each case. In the first case for *feed-animal*, we can process a *zebra* with *feed-zebra*, and in the second case, we can process a *snake* with *feed-snake*. The complete template is

```
(define (feed-animal a)
  (cond
   ((zebra? a) ... (feed-zebra a)... )
   ((snake? a) ... (feed-snake a)... )))
```

Given *feed-zebra* and *feed-snake*, the template actually provides the complete implementation.

The shape of the template matches the shape of the data definition. The data definition has two cases, so the template contains a **cond** with two cases. The data definition refers to two other data definitions, so the template refers to two other functions. This correlation between the data definition and the implementation is the heart of the design recipe. It is also the reason our approach transitions well to courses that use object-oriented languages.

## 4 Recursion without Magic

At this point, we know how to represent a single animal. We could also define a structure to represent a zoo of size 5 or 25, but to represent a realistic zoo, which has no fixed size, we need a better representation. The representation does not require new syntax, however. It requires only a data definition of the following shape:

```
(define-struct bigger (first rest))
;; A zoo is either
;;   empty, or
;;   (make-bigger animal zoo)
```

where empty is a constant, like true or false, representing an empty zoo. Meanwhile, (make-bigger *an-animal a-zoo*) represents a zoo just like *a-zoo*, except that is also has *an-animal*. Here are three example zoos:

```
empty
(make-bigger (make-zebra 250 10) empty)
(make-bigger (make-snake 10 12 false)
   (make-bigger (make-zebra 250 10) empty))
```

Suppose we need to write the *feed-zoo* function, which feeds every animal in a zoo. The contract is

```
;; feed-zoo : zoo → zoo
;; feed all animals in the given zoo
```

To make examples, we can use the example zoos above.

```
(feed-zoo empty) = empty
(feed-zoo (make-bigger (make-snake 10 12 false)
            (make-bigger (make-zebra 250 10) empty)))
   = (make-bigger (make-snake 15 12 false)
       (make-bigger (make-zebra 255 10) empty))
```

To implement the function, we start with a template. The template contains a **cond** with two cases. In the second case, we need a function that processes animals and one that processes *zoo*s. Well, we are developing a function that consumes *zoo*s and we might as well use it. After all, the data definition refers to itself, so the template should refer to itself, too.

```
(define (feed-zoo z)
 (cond
  ((empty? z) . . . )
  ((bigger? z) . . . (feed-animal (bigger-first z))
                . . . (feed-zoo (bigger-rest z)) . . . )))
```

We complete the function case-by-case. In the first case, our example tells us the right answer. In the second case, the template gives us an expression that feeds the first animal, and an expression that feeds the rest of the animals. All we have to do is combine the fed animals into a bigger zoo.

```
(define (feed-zoo z)
 (cond
  ((empty? z) empty )
  ((bigger? z) ( make-bigger (feed-animal (bigger-first z))
                              (feed-zoo (bigger-rest z))))))
```

All of the code outside the boxes comes from the template, and writing the template is essentially mechanical. Templates reduce mistakes in the uninteresting portions of the program and help the programmer focus on the parts that require an understanding of the specific problem.

Using our well-defined evaluation rules, we can even validate by hand that the function works on our example. Showing all steps takes more space than we have available, but we show the major steps.[2]

```
(feed-zoo (make-bigger (make-zebra 250 10) empty))
→ ... → (make-bigger (feed-animal (make-zebra 250 10))
                     (feed-zoo empty))
→ ... → (make-bigger (make-zebra 255 10)
                     (feed-zoo empty))
→ ... → (make-bigger (make-zebra 255 10)
                     empty)
```

## 5 Experience

The course described in this paper is the result of a decade of development, including an array of courseware such as a programming environment, on-line exercises and solutions, and training workshops. We have successfully field-tested our approach in a wide variety of contexts over many years. We exported the course to colleagues at our own institution, then at other public and private colleges, and finally to high schools across the nation. Educators at all levels who have implemented the approach have responded enthusiastically; they have also provided significant feedback that we have incorporated.

Although our approach does not use the syntax of a currently fashionable programming language, it prepares students to succeed in a wide variety of contexts. In the several universities where this course is now the main introductory course, the follow-on data structures course is taught in Java. Students who have completed our course perform well in the Java course, and they tend to outperform students who enter the second-semester course after going through a mainstream CS1 course based on C++ or Java. At the high school level, many teachers use the course as preparation for the AP test, which means they teach C++ (the current AP language) for only the second half of the AP class. Despite this shift in languages, teachers report that students have better programming skills than before, and that the students achieve high scores on the AP test.

[2]Our programming environment provides a tool to browse through every step of the evaluation.

## 6 Conclusion

We have just presented the essential elements of our novel approach to the first course on programming. The course provides an explicit model of computation and programming and thus actively discourages the idea of computing and programming as black magic. The key innovation is the notion of a design recipe, which not only provides a bridge from problem statement to solution, but, with its emphasis on documentation, data-induced code structure, and testing, also encourages good software engineering habits from the first day.

Our approach extends well beyond the processing of structures and linked lists. Tree processing is a simple generalization, as is processing instances of mutually recursive data definitions (e.g., directories and files in a filesystem), or graphs. Furthemore, our course builds on the design recipe to include coverage of higher-order functions, parametric and object polymorphism, state encapsulation, and state change.

## References

[1] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. *How to Design Programs.* The MIT Press, Cambridge, Massachusetts, 2001. `http://www.htdp.org/`.