1 Introduction

1.1 Our Philosophy

Please watch the video on YouTube. Someday there will be a textual description here instead.

1.2 The Structure of This Book

Unlike some other textbooks, this one does not follow a top-down narrative. Rather it has the flow of a conversation, with backtracking. We will often build up programs incrementally, just as a pair of programmers would. We will include mistakes, not because I don't know the answer, but because *this is the best way for you to learn*. Including mistakes makes it impossible for you to read passively: you must instead engage with the material, because you can never be sure of the veracity of what you're reading.

At the end, you'll always get to the right answer. However, this non-linear path is more frustrating in the short term (you will often be tempted to say, "Just tell me the answer, already!"), and it makes the book a poor reference guide (you can't open up to a random page and be sure what it says is correct). However, that feeling of frustration is the sensation of learning. I don't know of a way around it.

At various points you will encounter this:

Exercise

This is an exercise. Do try it.

This is a traditional textbook exercise. It's something you need to do on your own. If you're using this book as part of a course, this may very well have been assigned as homework. In contrast, you will also find exercise-like questions that look like this:

Do Now!

There's an activity here! Do you see it?

When you get to one of these, **stop**. Read, think, and formulate an answer before you proceed. You must do this because this is actually an *exercise*, but the answer is already in the book—most often in the text immediately following (i.e., in the part you're reading right now)—or is something you can determine for yourself by running a program. If you just read on, you'll see the answer without having thought about it (or not see it at all, if the instructions are to run a program), so you will get to neither (a) test your knowledge, nor (b) improve your intuitions. In other words, these are additional, explicit attempts to encourage active learning. Ultimately, however, I can only encourage it; it's up to you to practice it.

1.3 The Language of This Book

The main programming language used in this book is Racket. Like with all operating systems, however, Racket actually supports a host of programming languages, so you

must tell Racket *which* language you're programming in. You inform the Unix shell by writing a line like
`#!/bin/sh`
at the top of a script; you inform the browser by writing, say,
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" ...>`
Similarly, Racket asks that you declare which language you will be using. Racket languages can have the same parenthetical syntax as Racket but with a different semantics; the same semantics but a different syntax; or different syntax and semantics. Thus every Racket program begins with `#lang` followed by the name of some language: by default, it's Racket (written as `racket`). In this book we'll almost always use the language

```
plai-typed
```

. When we deviate we'll say so explicitly, so unless indicated otherwise, put
`#lang plai-typed`
at the top of every file (and assume I've done the same).

In DrRacket v. 5.3, go to Language, then Choose Language, and select "Use the language declared in the source".

The *Typed PLAI* language differs from traditional Racket most importantly by being statically typed. It also gives you some useful new constructs: `define-type`, `type-case`, and `test`. Here's an example of each in use. We can introduce new datatypes:

```
(define-type MisspelledAnimal
  [caml (humps : number)]
  [yacc (height : number)])
```

There are additional commands for controlling the output of testing, for instance. Be sure to read the documentation for the language In DrRacket v. 5.3, go to Help, then Help Desk, and in the Help Desk search bar, type "plai-typed".

You can roughly think of this as analogous to the following in Java: an abstract class `MisspelledAnimal` and two concrete sub-classes `caml` and `yacc`, each of which has one numeric constructor argument named `humps` and `height`, respectively.

In this language, we construct instances as follows:

```
(caml 2)
(yacc 1.9)
```

As the name suggests, `define-type` creates a type of the given name. We can use this when, for instance, binding the above instances to names:

```
(define ma1 : MisspelledAnimal (caml 2))
(define ma2 : MisspelledAnimal (yacc 1.9))
```

In fact you don't need these particular type declarations, because Typed PLAI will infer types for you here and in many other cases. Thus you could just as well have written

```
(define ma1 (caml 2))
(define ma2 (yacc 1.9))
```

but we prefer to write explicit type declarations as a matter of both discipline and comprehensibility when we return to programs later.

The type names can even be used recursively, as we will see repeatedly in this book (for instance, section 2.4).

The language provides a pattern-matcher for use when writing expressions, such as a function's body:

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
    [caml (humps) (>= humps 2)]
    [yacc (height) (> height 2.1)]))
```

In the expression `(>= humps 2)`, for instance, `humps` is the name given to whatever value was given as the argument to the constructor `caml`.

Finally, you should write test cases, ideally before you've defined your function, but also afterwards to protect against accidental changes:

```
(test (good? ma1) #t)
(test (good? ma2) #f)
```

When you run the above program, the language will give you verbose output telling you both tests passed. Read the documentation to learn how to suppress most of these messages.

Here's something important that is obscured above. We've used the same name, `humps` (and `height`), in *both* the datatype definition and in the fields of the pattern-match. This is absolutely unnecessary because the two are related by *position*, not name. Thus, we could have as well written the function as

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
    [caml (h) (>= h 2)]
    [yacc (h) (> h 2.1)]))
```

Because each `h` is only visible in the case branch in which it is introduced, the two `h`s do not in fact clash. You can therefore use convention and readability to dictate your choices. In general, it makes sense to provide a long and descriptive name when defining the datatype (because you probably won't use that name again), but shorter names in the `type-case` because you're likely to use use those names one or more times.

I did just say you're unlikely to use the field descriptors introduced in the datatype definition, but you can. The language provides selectors to extract fields without the need for pattern-matching: e.g., `caml-humps`. Sometimes, it's much easier to use the selector directly rather than go through the pattern-matcher. It often isn't, as when defining `good?` above, but just to be clear, let's write it without pattern-matching:

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (cond
    [(caml? ma) (>= (caml-humps ma) 2)]
    [(yacc? ma) (> (yacc-height ma) 2.1)]))
```