THE Transparent Web

Functional, Reactive, Isomorphic

Christopher J. Wilson

MEAP

MANNING

MEAP Edition
Manning Early Access Program

The Transparent Web

Functional, Reactive, Isomorphic
Version 1

Copyright 2016 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/the-transparent-web

Welcome

Thank you for buying the MEAP for *The Transparent Web: Functional, Reactive, Isomorphic*. If the title isn't enough of a hint, this book is a little bit unlike many other tech books. It is a broad survey of what's on the horizon of web development.

This book requires a bit of programming experience. Whether you have created a handful of sites or you have many more under your belt, this is for you. You'll also need a dash of curiosity to get the most out of it. More concretely, I assume a little bit of object-oriented programming experience as well.

This MEAP starts with chapters 1, 2, and 5. Chapter 1 is a broad overview of all the concepts in the book. Chapter 2 dives into writing an application using an isomorphic web framework, Opa. And chapter 5 looks into what modern static typing brings to the table.

Over the course of the rest of the book, I'll dig into three big themes: functional, reactive, and isomorphic. Functional programming serves as the backdrop for the rest of the concepts in the book. Functional programming is less widespread in web application development, but it brings with it a rich toolbox of techniques for building complex apps. As it applies to web development, *isomorphic*, means being able to reuse the same code on both the client and the server. Looked at another way, I think of it as treating the client and server as one *unified platform.* More holistically, this also includes things like compiling native code to JavaScript, and creating applications which include their own operating system as a library. Lastly, we'll look at the concept of reactive programming. This flips the normal control flow of user-facing applications on its head. Instead of writing programs which expect to be in control, reactive programming handles interaction with the user in terms of what the user is doing. Surprisingly, the result isn't chaos, but clean, elegant apps. I look at reactivity in both JavaScript and Elm, a whole language built around these ideas.

This book has been a big undertaking for me. Now, I hope you'll join me in the MEAP process to make the book better. Let me know what you think, what could be changed, or what you'd like to see in later chapters. I look forward to seeing your feedback in the Author Online forum.

I can't thank you enough for taking time to pick up this book. I hope you'll find it useful, and thought-provoking.

— Chris Wilson

brief contents

1 Advancing The Web

Web development can often feel like writing the same thing twice. We first write database schemas and application logic for the server. Then on the client we have to implement much of the same logic in order to validate inputs and provide realtime feedback. We are able to share data but not the code that implements application logic.

Even the way that we share that data seems awkward. We create routes structured around manipulating resources via a few distinct verbs: *CREATE*, *READ*, *UPDATE*, and *DELETE*. This leaves us with a one size fits all API that has much more to do with how the web works than how our application is structured.

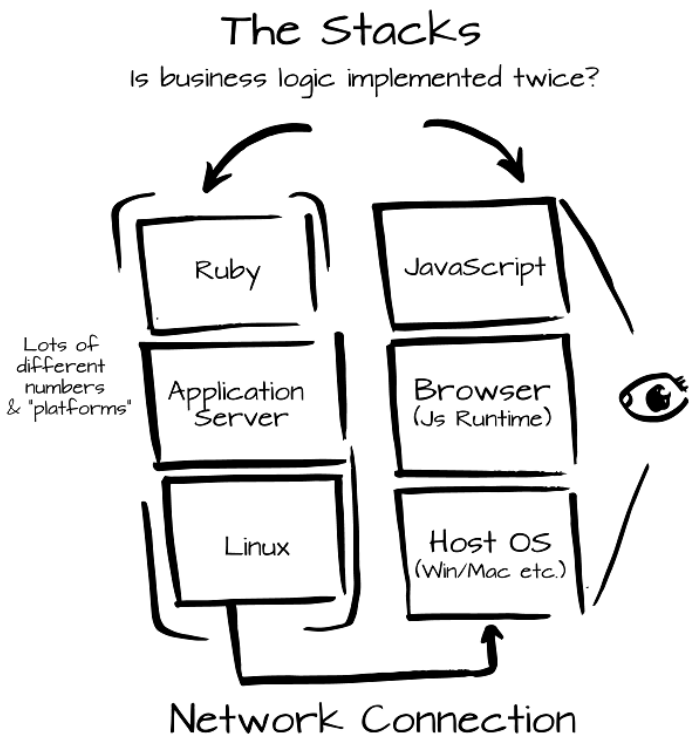
When we write user interfaces on the web, we do so by first downloading a document interspersed with formatting and structuring commands. We then write JavaScript to imperatively modify this document, swapping in new chunks here, or altering the display of existing parts there. Again, this whole process marches to the drummer of the way that the web works. HTML, derived from SGML, is a document markup language, not a user-interface system.

And when it comes to interactivity, JavaScript is what you get. Much in the way that Henry Ford quipped about the Model T, *Any customer can have a car painted any color that he wants as long as it is black*, JavaScript is the only language supported by all modern browsers. The cause for JavaScript's immense popularity eventually circles back around to the fact that it is massively popular. Whatever the merits of JavaScript, and it does have many, the fact that there's no choice is a drawback. These issues are present in current web development:

©Manning Publications Co. We welcome reader comments about anything in t

- Having to write similar code for multiple platforms (browser and server)
- Awkward or tedious client-server communication
- An ill-fitting UI language,
- An imperative and limited scripting language

Figure 1.1. Applications can end up being written twice



This book, *The Functional Web*, is ultimately about making the web a better place for developers to develop and, by extension, a better web for all of us to use.

The first I call *unified stacks*, an example of this is combining server-side and client-side code into one code base. Then there's *functional programming*, a particular style of writing code. Functional programming is a big topic. My interest with it in this book is how it informs writing clear, succinct code that expresses programmer intent. Static typing is also a topic that I will lump in with functional programming throughout this book. Though it is neither a necessary nor sufficient condition for functional programming, static typing nonetheless fits well there. I feel that static typing is complementary to both functional programming and web development. It provides the structure while functional programming brings the dynamism. Lastly, there is *reactive programming*. Like functional programming this is a big topic! As it applies to this book, it describes methods for orienting

©Manning Publications Co. We welcome reader comments about anything in t

applications to be responsive to outside input. In a user interface, this means reacting to clicks and key presses. It is conceptualizing applications not as a big run loop, but as small functions to be run in response to outside events.

This book's mode is comparative and exploratory rather than prescriptive — I want to unearth options rather than try and find the "one true way". And because of that, it may come across as an odd tech book. Rather than an exhaustive tutorial of some technology, I'm going to introduce you to many things. We'll learn enough about each new language or technology to see how it could fit as a future direction for programming.

Many of the ideas herein draw from or touch on functional programming. But I don't consider this to be a book *about* functional programming, rather it is a book about coming to grips with the complexity of modern web application development. It just so happens that functional programming has a lot to say about cutting that knot of complexity.

1.1 Major Themes of The Functional Web

When I first noticed Opa and Elm, two languages that we'll talk about in later chapters, there seemed to be some thread connecting them. Though they were developed by different people, they clearly shared some characteristics. They both allow the developer to program in just one language. They both include syntax for dealing with common web-related tasks (HTML, SQL, etc.). There were *themes* underlying these separate languages.

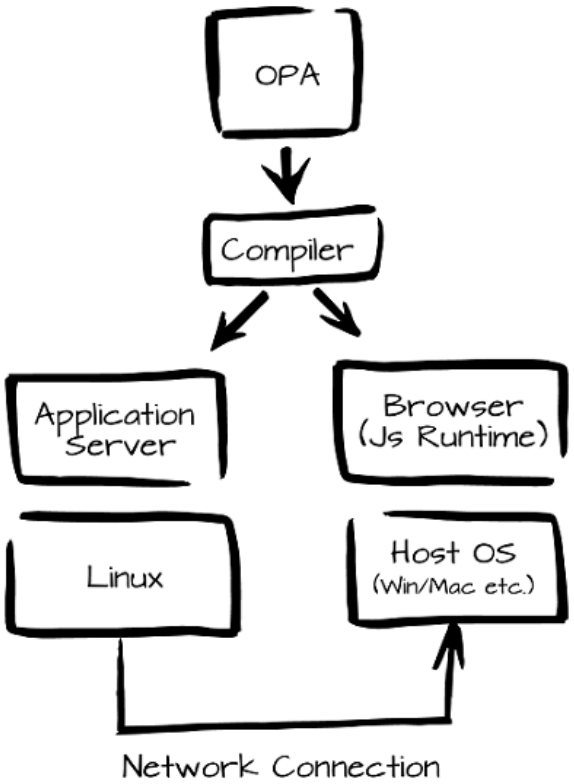
Through this lens, I saw that there were many more technologies that seemed to fit. I realized that what Opa and Elm were really up to was simplifying, collapsing, or*unifying* parts of the web stack. JavaScript, HTML, CSS and even, in Opa's case, the server-side could be programmed together.

In the following sections, I'll explain how each of these themes guide our exploration through the Functional Web.

©Manning Publications Co. We welcome reader comments about anything in t

1.1.1 Unified Stacks

Figure 1.2. Server and Client-side applications can be written in the same codebase



Web application programming has progressed to a much richer model. The big shift is that we now write an API server that consults a database and pair it with a client-side application that makes requests of that API. This scheme might be called a "rich client-side application" or just "client-side application". In this model, the client-side application is often where most of the complexity resides. And there is complexity in the *interplay* between the two sides as well! But this way of doing things has the benefit of being more flexible. The same backing-API may be used by many different front-ends be they web-based, iOS app, Android, or etc.

The *unifying platforms* approach to this architecture assumes the above as a given. We're going to have an app "split in half" and communicating via an API. The communication between the halves will be baked into the framework. In fact, in Opa, this detail is somewhat hidden by the compiler. We can write code without regard for where it runs. There are still some necessary divisions, say for hiding sensitive business logic from the client-side, but these are divisions that are

©Manning Publications Co. We welcome reader comments about anything in t

meaningful in the context of the application and are not just technological limitations.

Let's look at this client/server division in Opa. In most cases code will be available on both the client and the server. But there are times when it is important that code should *definitely* be running only on the client or only on the server. Opa has a declaration syntax to cover that situation.

Listing 1.1. Where code should run in Opa

```
function client_or_server(x, y) { ... }        ❶

client function client_function(x, y) { ... }  ❷

server function server_function(x, y) { ... }  ❸
```

❶ Opa decides where to compile this function
❷ This function is compiled for the client-side
❸ This function is compiled for the server-side

The compiler is able to determine that certain things, such as database access must be compiled into server-side code, whereas reading the value of a CSS selector pertains to the client.

Unifying code and writing at this abstracted level is another main theme of this book. There are other ways that we'll explore this *unifying* theme: unifying the server.

MirageOS is an example of a so-called *library OS* or a *unikernel*. The network, storage, and other drivers that would usually be a part of the operating system are instead compiled into a standalone program. This creates an executable that can run directly within a hypervisor (in this case, the Xen). Put another way, there is no Linux or Windows OS underlying the application!

1.1.2 Functional Programming

A whole book could be written about functional programming and many have been. To give you a brief overview, a functional language is one where the primary means of structuring code is around functions. Functions, in the mathematical sense, define a relationship between inputs and outputs. In programming, the traditional meaning of *function* has been expanded from what we'd recognize from mathematics. Functions as used in programming are really procedures, lists of steps to be carried out, possibly with inputs and outputs. In the mathematical world, a function without inputs or outputs wouldn't make sense. There would be no way to convey information into or out of the function.

The goal of functional programming is to create more tractable code. When the only things that can affect the outcome of a function call are its arguments, it's

©Manning Publications Co. We welcome reader comments about anything in t

much easier to find problems.

Reducing statefulness

Functional programming demands a fundamental change in how you write software. *Functions* become the primary means of structuring your code, and, rather than mutating variables, state is carried in function parameters. From the point of view of the function, values flow in through input parameters and a new value is returned from the function — those variables never change.

This means that each time a function is called it behaves identically when given the same arguments. We can't tell the difference between the first time that we call a function and the 100th. The function does not carry any state. There's no hidden record that the function is keeping, no history that it maintains anywhere. That "history" or "hidden record" is also referred to as *state* and functional programming is partly defined by not having it.

Languages that work in this way are much easier to deal with, even if it doesn't seem so at first. Most tests can be written as simple input/output pairs. We save on a lot of set up and tear down.

Composition and modularity

Software, if it's to have any hope of growing to a large size and still be practicable, must be written in a modular way out of composable pieces. In a well-designed system say you have two operations you'd like to perform, one after the other. It should be possible to combine those operations together to yield a third operation that is the combination of the two. Think of how arithmetic works for the basic operations of addition, subtraction and multiplication.

```
(X + 2) * 3 = 3X + 6
```

The expression on the left is equivalent to the expression on the right. This is natural to the mathematical way of thinking. But if I wrote this as a "program" it might give us pause for a minute. Here is the same idea, but expressed as two functions, f and g.

```
f(x) = x + 2
g(x) = x * 3
```

We can still combine these functions in the same way. If I want to form "g following f", I can write this new function that combines the two.

```
gf(x) = g(f(x)) = g(x + 2) = (x + 2) * 3
```

In each step, I just expanded out what the function was doing. So then compositionality in a nutshell is about something being the *sum of its parts*. Expanding on my little math example of the idea of composition or

©Manning Publications Co. We welcome reader comments about anything in t

compositionality, here's a JavaScript function that uses a few other functions.

```
function foo1(input) {
    var x, y, z;

    x = bar(input);
    y = baz(x);
    z = quux(y);

    return z;
}
```

Now imagine that we didn't really care about what foo, bar, baz, or quux are — we want everything but those specifics. We just want the skeleton of that function.

```
function foo2(input, f, g, h) {
    var x, y, z;

    x = f(input); y = g(x); z = h(y);

    return z;
}
```

And furthermore all the skeleton is really doing is a sort of plumbing of those functions together. We can more clearly express this:

```
function foo3(input, f, g, h) {
    return h(g(f(input)));
}
```

The true nature of foo is revealed! And given that simpler nature, we could express `foo` just as a simple result of other operations:

Listing 1.2. Final version of `foo`, written as a composition of functions

```
function compose(f, g) {
  return function(x) { return f(g(x)); }
}

function chain(funcs) {
  if (funcs.length === 1) {
    return funcs[0];
  }
  return compose(chain(funcs.slice(1)), funcs[0]); ❶
}

var foo4 = chain([bar, baz, quux]);
```

❶ `funcs.slice(1)` returns all elements of the `funcs` array after the first one. This operation is sometimes called "tail" or "rest".

©Manning Publications Co. We welcome reader comments about anything in t

I've showed we can take simpler parts, in this case the bar, baz, and quux functions and combine them to create more complex behavior.

Now I know that the idea that this is simpler may seem really far-fetched but there's a way in which it absolutely is. Once we've written compose and chain, which are handy in their own right, they allow us to see that foo is really just gluing other things together. The function composition example expresses the essence of compositionality; we can understand code by what its pieces do and how they're combined. This idea, flipped around, is a powerful way to write code. If you combine small pieces that you understand individually, you can build large programs that are easier to understand.

Functional programming seeks, as much as possible, to build programs by putting together such simple pieces like this.

Suffice it to say that I think functional programming has lots of ideas worth borrowing. If you're new to the idea of functional programming but familiar with the web, then I'm happy to say that this is a great place to be. I'll be pointing out ideas that are coming from functional programming as we go along. There will be sections devoted to the ideas behind the technology that I'm discussing and not just the technology itself.

If you're familiar with some of the ideas that I listed above, DSLs, type systems, functional reactive programming, well then, I'm hoping that you'll still find lots to learn as you see how these ideas can be applied to the web.

1.1.3 Reactive Programming

Reactive programming means structing our applications around how we'd like to respond to events from the outside world. We'll see a lot more about how this works when we look at Elm in WritingReactiveGUIsWithElm. In particular, we'll zero in on the variety of reactive programming known as *functional reactive programming*. Like functional programming itself, reactive programming is a broad style, and encompasses many other ideas.

Functional reactive programming, or *FRP*, is a technique that aims to directly incorporate time into programming. It has seen applications in GUIs, robots, music, and elsewhere. In this book I'll be focusing on how it relates to GUIs. FRP gives us an another way of dealing with user interaction.

This book will use the Elm language to look at FRP. The core abstraction in the Elm language, and some other FRP implementations, is that of the Signal. A Signal is like a variable in a programming language but with the added dimension that it varies over time. In this example, `Mouse.position` is a Signal; specifically it is a Signal of an x-y position on the screen and it has type `(Int, Int)`.

©Manning Publications Co. We welcome reader comments about anything in t

Tuples

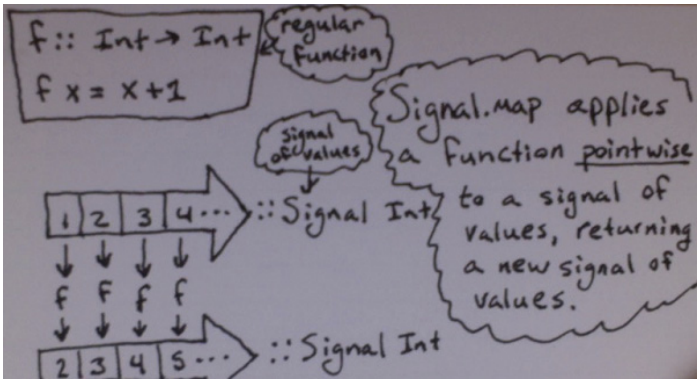
The parentheses notation is called a *tuple* and means that these values are paired together.

```
main : Signal Element
main = map showMouse Mouse.position

showMouse : (Int, Int) -> Element
showMouse (x, y) = join ", " [toString x, toString y] |> show
```

This produces outputs like, `22, 345`, in the uppermost left corner of the window. `Mouse.position` is a `Signal` that contains a tuple of integers, or as Elm would write it:`Signal (Int, Int)`. The `showMouse` function takes a pair of Ints and converts them into a displayable form to be shown on the page, an `Element` in Elm.

Since `Mouse.position` is a signal, and `showMouse` is a normal function, we have to adapt it before we can use it. We can think of this as applying the `showMouse` function at all times. "The function that we use to do this is called `map` and it is in Elm's `Signal` module. `map` is a *higher-order function*, which means it is a function which accepts other functions as arguments. Specifically, `map` takes a function as its first argument and returns a new function as a result. The expression `map showMouse` is a new function which converts a value with type `Signal (Int, Int)` into a value of type `Signal Element`. It acts as that "at all times" adapter that we need. It converts from functions on simple values (like `(Int, Int) → Element`) to functions on Signals (like `Signal (Int, Int) → Signal Element`). Here's a diagram of what's going on with map.

Figure 1.3. Map applies a function *pointwise* to a signal of values



You can see in the figure how map modifies a whole function, returning a new one. It is this new function that's applied to Mouse.position. Function calls associate to the left and if we put in parentheses they would look like this.

```
main = (map showMouse) Mouse.postion
```

©Manning Publications Co. We welcome reader comments about anything in t

By modifying normal functions to work on `Signal` values, `map` lets us work with values that vary over time. Lots of interactive things can be naturally expressed as this way. A Signal of a static image is an animation (i.e. a time-varying image)!

1.2 Summary

Web development is in a transitional period. Since starting modestly in the mid '90s, rich web applications have become a key part of modern computing. We're seeing a Cambrian explosion in web application development. Web development has a lot to borrow from the world of functional programming and I believe ideas like those explored in this book, collectively or in part, represent the future of programming.

In this chapter, we saw that:

- I believe the web is in a transitional stage. Parts of the web stack will coalesce while others will absorb ideas from functional programming. This will make programming for the web more coherent.
- "The Functional Web" consists of the themes of Unified stacks, functional programming, and reactive programming.
- We delved into each of these themes to get a sense for how they fit.

This book proceeds through a series of examples and discussions that will demonstrate what these concepts are and how they can be applied. Where possible, I'll also point out analogous features between the different frameworks and techniques. This will make the new ideas clearer by showing many of them in a few different ways.

If all that sounds good then let's get started!

©Manning Publications Co. We welcome reader comments about anything in t

2 Transparent Client-Server Programming With Opa

In this chapter we'll turn our attention to Opa. Opa sits right in the intersection of all of the themes of this book. In fact, Opa itself was what first spurred my thinking on toward what later became this book. Opa is described on its website as being a "framework for JavaScript", but this is a slightly misleading statement. Opa falls in among the growing ranks of languages that compile to JavaScript and so Opa-the-language is something other than plain-old-JavaScript. But far from being a drawback, this frees Opa to add many great features that would be hard or impossible to implement just as a JavaScript library. Opa features many of the *Transparent Web* characteristics: a single language for both client and server, a static type checker, a database domain-specific language, and a healthy dose of functional programming.

We've encountered some of these ideas a bit earlier during our discussion of Meteor. Meteor implements some of the same features, but does so in JavaScript. The gains in Opa then come from shedding JavaScript for a new functional language that's statically typed. Meteor is a great place to see how these ideas work in the context of a more familiar language.

2.1 Tutorial Overview

Before diving right into reading an Opa app, let's cover the basics of the language. The good news is that there are lots of syntactic similarities to JavaScript. So at least on the surface, Opa code will resemble what you may already be used to seeing. In many ways, you can almost pretend that Opa is a souped-up, super-JavaScript. The differences between Opa and JavaScript stem from Opa going some extra lengths to catch many errors at compile-time.

©Manning Publications Co. We welcome reader comments about anything in t

2.1.1 Variables

Variables have the common set of allowed characters. They can contain letters, numbers, and underscores. Variables are defined by using an equals sign like this:

```
x = 1
y = "cat"
```

Underscores have a special meaning when used for variable names. If an underscore is used to *start* a variable name, then the compiler will not issue a warning for an unused variable.

Finally, variables may be defined by enclosing them in backticks:

```
`is my middle name` = "danger"
```

This syntax allows us to use characters which would otherwise not be allowed in variable names.

Precisely, variable names in Opa adhere to the following regular expression:

```
([a-zA-Z_][a-zA-Z0-9_]*|`[^`\n\r]*`)
```

Summarizing the above, a variable name starts with a letter or an underscore followed by any number of alphanumeric characters and underscores. Alternatively, it is any number of characters excluding backticks, newlines, and carriage returns, all enclosed in backticks.

2.1.2 Functions

As a functional language, Opa's building blocks are (you guessed it) functions. Functions in Opa have the familiar declaration syntax that is common in JavaScript and among C-family languages. Functions are treated just like other values and so the same variable naming rules as above apply.

```
function my_function(a, b, c) { ... }
```

Functions can be defined in a curried way:

```
function make_adder(a)(b) { a + b }
```

The meaning of this is that the arguments to `curried_function` can be supplied one after another rather than all at once.

```
add_5 = make_adder(5)
add_5(3) // returns 8
```

We can also convert a function with *multiple arity*, (e.g. `function make_adder(a, b) { a + b }` has arity two) into a function with lower arity by *partially applying* it:

©Manning Publications Co. We welcome reader comments about anything in t

```
function make_adder_2(a, b) { a + b }
add_5 = make_adder_2(5, _)  // note the underscore
add_5(3) // returns 8
```

Currying and partial application really come into their own when used in one-off situations such as mapping over a collection. We can often "re-tool" a very general function for the specific purpose at hand. This is one of the ways that functional programming engenders code reuse.

Opa also includes special function syntax related to web programming. Normally, the Opa compiler will decide where the generated code will be run. But the Opa language includes `client` and `sever` statements to tell the compiler to generate code for the client or for the server.

```
client function my_client_function(x, y) { ... }
server function my_server_function(u, v) { ... }
```

A particular use case is security-sensitive code which can be constrained explicitly to the server.

2.1.3 Special syntax

Opa affords special syntax for working in the *web environment*. For working with HTML, Opa lets us simply include the HTML that we want directly as a value.

```
my_html =
  <p>
    And now for something <i>completely</i> different
  </p>
```

Notice the complete lack of quotes. The above `xhtml` fragment is a value within the language, the same as an `int` or a `float`, and it isn't a string. Values can be interpolated inside `xhtml` fragments:

```
str = "inside"
html = <p>Values <i>{str}</i> html</p>
```

This technique can be used to build up more complex xhtml values.

Opa also provides syntax for CSS selectors. These are used to get a hold of parts of the DOM. In its most basic form, we can replace, prepend, or append content to a DOM element with id `sunglasses`.

```
#sunglasses  = "with"  // replace
#sunglasses += "deal"  // prepend
#sunglasses =+ "it"    // append
```

Opa also has syntax for natively working with databases. We declare them using the following syntax:

```
database memes {
```

©Manning Publications Co. We welcome reader comments about anything in t

```
  map(string, string) /name
}
```

This declares a database named "memes". It stores string values indexed by string keys, it is a map from string types to string types (i.e. `map(string, string)`). And lastly, the subpath within the `memes` database is `/name`. Putting that all together, that means we can read and write to the database using the following syntax:

```
/memes/name["yo dawg"]        // read: "I heard you like..."
/memes/name["doge"] <- "Wow." // write
```

If the value stored in the database is an `int`, we can also increment and decrement it:

```
database memes {
  int /num
}

/memes/num += 1  // increment
/memes/num -= 1  // decrement
```

2.1.4 Records

Records are a major category of data in Opa. These are akin to objects in JavaScript or database rows. Records are key-value pairings where the type and number of pairs can vary. As opposed to lists, the types in a record can differ and so they are the primary way to organize heterogeneous data.

Here's a record for storing information about a person:

```
import stdlib.web.mail

type person = { string name, int age, Email.address address }
bob = { name: "Bob", age: 40,
        address: { local: "bob", domain: "example.com" } }
```

We declare the type of the record on line 3 by writing down the names and types for each field in the record. On line 4, we create a value, `bob` of type `person`. Notice that in creating the `bob` value, we've also embedded a second record inside, for the email address. The definition for that record comes from the standard library, imported on line 1.

Records have many associated operations and special syntax for working with them. This makes sense given their central role in the language. An example is that we can update and extend records easily. Given the above definition of `bob`, we can extend the record:

```
bob = { ... } // as before
robert = {bob with name: "Robert", more_formality: true}
```

©Manning Publications Co. We welcome reader comments about anything in t

At this point, the `robert` record contains all the fields that the record `bob` did, but with two fields added: `name` and `more_formality`.

This `with` syntax allows us to both update and replace fields within the existing record. There's also syntax for the the common case where we'd like to create a record using the names and values of in-scope variables. The following records, `bob1`, `bob2`, and `bob3`, are all equivalent:

```
name = "bob"
age = 40

bob1 = { name: name, age: age }
bob2 = { ~name, ~age }
bob3 = ~{ name, age }
```

In the first example, `bob1`, we're explicitly setting the field called `name` with the value of the variable `name`; same with `age`. Since it is a bit redundant to assign a field with the same name as a variable (`{ email_address: email_address }`, etc.), Opa lets us use the syntax in the second example, `bob2`. That example means that I have a variable called `name` and I want to set the field in the record, `name`, to that value. In the last example, `bob3`, the record only contains the fields `name` and `age` and I want to set them both. I also have variables with matching names. I can then move the tilde (~) to the front of the record meaning that I'm setting both fields at once and to their corresponding values. Each example achieves the same effect but with increasing shorthand as we go.

Lastly, we can access the fields of a record using dot notation. This should look very similar to object access in JavaScript.

```
bob.name // equals: "bob"
bob.age  // equals: 40
// etc.
```

At this point, if you're familiar with JavaScript, you may be asking yourself: "what's the big deal? These are just like JavaScript objects." And it's true, records look and act much like JS objects. The difference though is that these are type checked. If we make a mistake about the name or type of a field, the compiler will give us an error and point to exactly where it happened.

This has nice implications for writing functions which use records! The following function accepts a record called `r`, which must posses `int` fields named `a` and `b`:

```
function add_a_b(r) {
  r.a + r.b
}
```

This declares `add_a_b` as taking a type, `{int a, int b, …}`. Note that there are ellipses after the two known fields `a` and `b`. This notation means that there could be any other fields in the record and the `add_a_b` function won't care. Let's try calling

©Manning Publications Co. We welcome reader comments about anything in t

our function with an extra field.

```
add_a_b({a:1, b:2, c:"cat"}) // equals: 3
```

As expected, the extra field, `c`, didn't interfere with the function. Let's try to call our function with a missing field.

```
add_a_b({a:1, c:2}) // no 'b' field

Error: File "foo.opa", line 18, characters 11-20,
(18:11-18:20 | 290-299)
Type Conflict
  (12:5-12:16)        { a: int; c: int } / 'c.a
  (15:3-15:5)         { a: int; b: int; r.a }

  The argument of function add_a_b should be of type
    { a: int; b: int; r.a }
  instead of
    { a: int; c: int } / c.a
```

Opa notices and flags the line where the offending variable is defined and used. The compiler error also shows that we were trying to use a record with fields `a` and `c`, rather than the correct one with fields `a` and `b`. The extra term at the end of the error message, `'r.a`, is Opa telling us that "you must *at least* provide fields a and b (of type int), but you may also provide any additional fields as well."

In JavaScript, you get much the same behavior except that the logic is flipped around. It is as if the function were saying "I'm going to use *these* properties of that object so*watch out!*" But imagine if your JavaScript code could tell you, upfront, that the argument passed to a given function wasn't going to be adequate.

This feature of dynamic languages that I'm describing, like using `r.a` or `r.b` above, is sometimes called "duck typing". It means that if I can use properties of some thing: `thing.quack()`, `thing.swim()`, and `thing.bill`, than that thing can be treated as a `duck`. It doesn't matter what the actual *type* of `thing` is as long as it behaves sufficiently duck-like. Bringing it back to programming, that means that an object which responds to all the same methods, "acts like" any other such object.

Opa, by being able to check that a record supports all the fields that we need, is giving us something that works like "static duck typing."

2.2 Re-visiting the Biking App: Opa Example

Let's revisit the small biking application that we talked about earlier in WritingABikingApplication.

We'll re-implement this application using features that Opa provides. Though this application is simple, it will still manage to show off a lot of the core ideas of this book. In particular, we'll see how we are able to hook up an `onclick` handler so

©Manning Publications Co. We welcome reader comments about anything in t

that it calls directly into code that saves input into the database.

As a brief overview and reminder of the *Biking* application, let's talk about what it's supposed to do. The main page of the application should be a list of recent bike rides. Each ride should describe the rider, the date, and the distance ridden. There's also a form where users can enter their own rides. To make the application be more manageable, I've omitted many features that that would be all but required in a real application. We'll totally ignore issues like, user management, pagination, and import/export. What follows is just to show how Opa's features fit in with the Transparent Web but, if you're interested, the Opa website is a great place to learn more about developing with Opa.

All lengthy code examples from this book can be found online. I've created a git repo of all of the source in the book, it can be found here:https://github.com/twopoint718/tw.

We're now ready to start our application. We can generate the skeleton of a new site automatically.

```
opa create biking
```

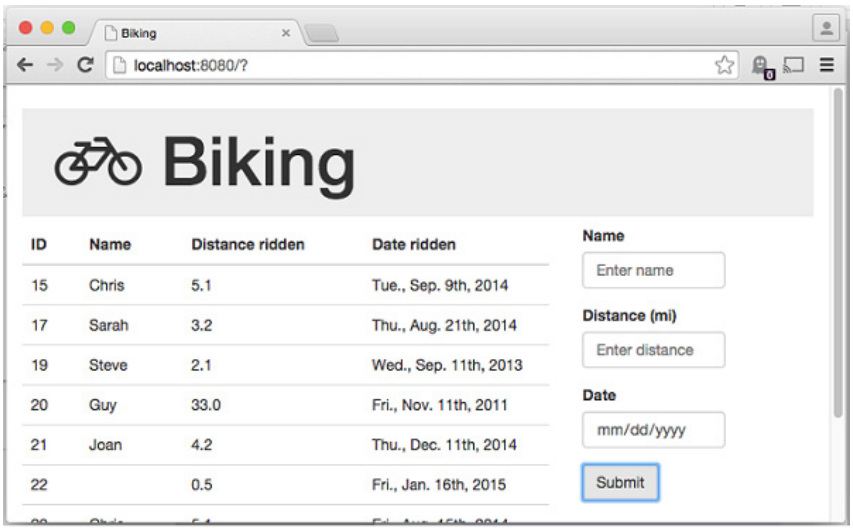
You should see output similar to the following.

```
OpaCreate: Generating biking/Makefile...
OpaCreate: Generating biking/Makefile.common...
OpaCreate: Generating biking/opa.conf...
OpaCreate: Generating biking/resources/css/style.css...
OpaCreate: Generating biking/src/controller.opa...
OpaCreate: Generating biking/src/model.opa...
OpaCreate: Generating biking/src/view.opa...

Now you can type:
$ cd biking
$ make run
```

This has created a new directory named `biking` that contains everything we need to get started. Now we're all set! Opa created the skeleton of a site for us. All we have to do is write the entire app! When we're done we should end up with this.

©Manning Publications Co. We welcome reader comments about anything in t

Figure 2.1. The finished Biking application



2.2.1 Model code

I find it most useful to start any application by thinking about what kinds of *domain objects* there are. By this, I don't literally mean *objects* as in object oriented programming, but simply what sort of "logical entities" are we going to talk about while building this application. In the biking application we're chiefly concerned with the concept of a *bike ride* — I'm just going to call this a `ride`.

Let's start by importing some standard library code. We'll import the `date` library for converting between different date formats and parsing dates from strings.

```
import stdlib.core.date
```

Next, we get right down to the meat of the application. We define the type of a `ride`. A `ride` will consist of an ID, a user's name, the distance that they rode, and the date on which they rode. Putting this into a type definition looks like this.

Listing 2.1. Types

```
type ride_id = int                    ❶

type ride = {                         ❷
  ride_id id,
  string user_name,
  float distance,
  Date.date date
}
```

❶ We wrap an int into more descriptive type

©Manning Publications Co. We welcome reader comments about anything in t

❷ We define a record to hold the details of a bike ride

To store our application's data, we use the built-in database capabilities of Opa to store rides directly. Notice that we're using the `ride` type that we just defined to store a set of rides indexed by their `id`. Our application will also need an index counter to point to the last-inserted ride. This is akin to an auto-incrementing key in a relational database management system. Here, this guarantees that even if two rides share the same details (`user_name` & etc.) their `id`s will differ.

Listing 2.2. Database

```
database biking {
  ride /rides[{id}]                    ❶
  int /index = 0                       ❷
}
```

❶ This declares that we'll be storing rides in the database under the path "/biking/rides". We'll use the "ride_id" to index into this collection of rides, so accessing a specific ride would look like: "/biking/rides[4]".

❷ We store an int in the database to keep track of the next "ride_id" to insert into the database.

Lastly, we'll create a helper function to create rides in the database. This takes the name of the rider, the distance ridden, and the date and returns a ride. Because we're fetching these values out of the DOM, they'll be arriving in our function as strings. We'll do some checking inside the function to make sure the inputs are valid. We then create a new record and store it in the database. There's new syntax here that we haven't seen before. Assigning values to records usually looks like this:

```
int var1 = 1
record = { variable_one: var1 }
```

But if the variable and the field in the record have the same name, then we can use a shorthand syntax.

```
int variable_one = 1
record = {~variable_one}
```

And furthermore, if *all* the variables being assigned share their names with their respective fields, we can shorten the code up even more. We move the ~ outside the record.

```
int variable_one = 1
int variable_two = 2
record = ~{variable_one, variable_two}
```

©Manning Publications Co. We welcome reader comments about anything in t

Listing 2.3. Module code

```
module Model {
  function create_ride(string user_name,
                       string dist,
                       string raw_date) {
    distance = Float.of_string(dist)                          ❶

    scanner = Date.generate_scanner("%Y-%m-%d")
    result  = Date.of_formatted_string(scanner, raw_date)     ❷
    date = match(result) {                                    ❸
      case {some: d} : d                                      ❹
      case {none}    : Date.now()                             ❺
    }

    int id = /biking/index
    /biking/rides[~{id}] <- ~{id, user_name, distance, date} ❻
    /biking/index <- id + 1
  }
}
```

❶ We parse the (string representation) of the distance into an actual floating point number.

❷ Using the string format from the previous line, dates like "2015-01-15", we attempt to parse the date provided by the user.

❸ Next we use *pattern matching* to check the result of our attempt at parsing.

❹ The first pattern matches if the parse succeeded. We'll get a value like `{some: [a date value]}`, from which we extract the date part.

❺ In the case that the parse failed, we'll match `{none}`. We default to choosing today's date.

❻ We use the record shorthand to assign all fields in the `ride` record and then store this in the database under the current index. On the next line we increment the index.

Note that we've started out the application by structuring it around the types involved. In this small application, there's just the `ride` type, but this extends naturally to more types.

We also place core tasks inside the model module, here that's persisting the rides; the `create_ride` function. A functional design that I've found valuable is to draw a clear *border* between the inside and the outside of an application. As soon as it is possible, safely convert untyped external values into well-understood domain values. And that's just what we're doing inside `create_ride`, we're parsing the date and distance into time and numeric values, respectively. Once we've converted them, we're free to use them within the app. Here we immediately store them in the database.

Now that we have a solid core of data, we can move on to the views that will display it.

2.2.2 View code

The view code is a little bit longer than the model was. More of the view code is concerned with creating a particular look, and this bulks it up somewhat. I'm going to try and focus only on those parts that show off interesting functionality. The entire program can be obtained from the code repository.

Listing 2.4. Default page layout

```
function page_template(title, content) {
  html =
    <div>                                                    ❶
      -->
      <div class="navbar navbar-fixed-top">
        <div class=navbar-inner>
          <div class=container>
            <a class=brand href="./index.html"></a>
          </div>
        </div>
      </div>                                                 ❷
      -->
      <div id=#main class=container-fluid>
        {content}
      </div>
    </div>
  Resource.page(title, html)                                 ❸
}

function default_page() {
  content =
    <div>
      <div class=hero-unit>
        <div class=jumbotron>
          <h1><i class="fa fa-bicycle"></i> Biking</h1>
        </div>
      </div>
      <div class=row>
        <div class=col-md-8>{main_table()}</div>             ❹
        -->
        <div class=col-md-3>{input_form()}</div>
      </div>
    </div>

  page_template("Biking", content)                           ❺
}
```

❶ Here we create a snippet of HTML, assigned to the "html" variable, that defines the outermost part of the page we're creating.

❷ We interpolate the eventual content of the page into the template here.

❸ Lastly, we create a Resource from this HTML and a title. A Resource is what Opa eventually renders as a page.

©Manning Publications Co. We welcome reader comments about anything in t

❹ Inside the "default_page" function, we lay out some more formatting. We create, left-hand and right-hand columns. The "main_table" function will be rendered on the left-hand side of the page — we'll show this next. On the right-hand side we render the "input_form" function.

❺ Lastly, we use the `page_template` function with a title of "Biking" and the content that we just created.

We've now written the overall layout of the page. Let me call your attention to the fact that since we're using functions to define HTML and we can easily interpolate other HTML, structuring pages modularly is simple. We don't have to lay a page out in the order that it is rendered, as is normal with regular HTML. As a small syntactic point, notice that HTML attributes unquoted when they don't contain whitespace. See that `class=jumbotron` is not quoted (it's a single word) but `class="fa fa-bicycle"`.

Next we turn our attention to creating the `main_table` function. This will display a table of the details of a group of rides. The `main_table` function creates the structure of the table and it will then call out to a function that renders all the rows of data in the table, `table_rows`.

The `table_rows` function queries the database for a set of rides. This value is a `dbset`, an abstract collection of `ride`. We'll use some library functions to get an *iterator* over this collection. We will then use the iterator to perform a *fold* over the data returned.

A short interlude on folding

A *fold* is a common operation in functional programming. In Ruby this function is called *inject*. In other languages it may be called *reduce*. The idea is that we want to take a collection of some sort and compute a final value from it. Fold steps through the collection one element at a time, combining the current element with the further result of folding the rest of the collection. To provide a base case, fold requires that we pass in a value to be used when the collection is empty. Here's an example of what that would look like.

Listing 2.5. Folding a list of numbers using addition

```
function add(n, total) {
  return n + total;
}

function fold(f, end, list) {
  if(list.length == 0) {
    return end;                                          ❶
  }

  return f(list[0], fold(f, end, list.slice(1)));        ❷
}
```

©Manning Publications Co. We welcome reader comments about anything in t

```
console.log(fold(add, 0, [1, 2, 3, 4, 5])) // prints 15
```

❶ The base case (when the list is empty) we immediately return the caller-provided `end` value.

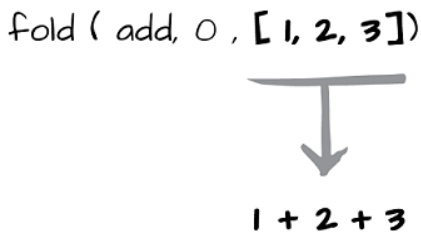
❷ Otherwise we call the provided function on the curent value, `list[0]`, and the eventual result of folding the rest of the list. We can think of that second value like a snowball that we're rolling up, each step adds a bit more snow to the ball.

When looking at folding step by step, you can see that it applies the function repeatedly. Each step of the fold nests an inner call to itself. Here's how the evaluation proceeds step by step.

```
  fold(add, 0, [1, 2, 3])
= add(1, fold(add, 0, [2, 3]))
= add(1, add(2, fold(add, 0, [3])))
= add(1, add(2, add(3, fold(add, 0, []))))
= add(1, add(2, add(3, 0)))
= add(1, add(2, 3))
= add(1, 5)
= 6
```

But expanding it out that way actually hides a simple intuition. Just imagine the list of the numbers to be added and then replace the commas with the function that we're running. I put in the parentheses because that reflects the way that the computation proceeds. In the case of addition, we know that the parentheses don't matter because addition is associative. Recall that this just means that `(1 + 2) + 3 = 1 + (2 + 3) = 1 + 2 + 3`.

Figure 2.2. Folding a list is like replacing the commas



More abstractly, any structure where we can think of a way to combine a single element with "the rest," we can create a fold for that structure. This includes things like trees and lists. It's a powerful way to abstract iteration into a succinct operation.

Using a fold to build a table

A fold plays an interesting role in how we build up the HTML for our table rows. Inside the `table_row` function, we can see how this fold business comes into play. We return an HTML value that looks like this `<>{acc}<tr>…</tr></>`. That is,

©Manning Publications Co. We welcome reader comments about anything in t

we're interpolating the HTML that we already have, `acc`, with the HTML for the current row, `r`. When repeated, this has the effect of appending row after row to our table. When we've stepped through all the rows, we're left with the body of the table.

Listing 2.6. Creating the table of rides

```
function main_table() {
  <table class="table table-hover">
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Distance ridden</th>
        <th>Date ridden</th>
      </tr>
    </thead>                                          ❶
    -->
    <tbody>
      {table_rows()}
    </tbody>
  </table>
}

function table_rows() {
  all_rides = /biking/rides                           ❷
  it = DbSet.iterator(all_rides)                      ❸
  Iter.fold(table_row, it, <></>)                     ❹
}

function table_row(row, acc) {                        ❺
  pr = Date.generate_printer("%a., %b. %E, %Y");
                                                      ❻
  <>
    {acc}
    <tr>
      <td>{row.id}</td>
      <td>{row.user_name}</td>
      <td>{row.distance}</td>
      <td>{Date.to_formatted_string(pr, row.date)}</td>
    </tr>
  </>
}
```

❶ We interpolate the table rows into the body of our rides table.

❷ Inside the table_rows function we fetch all the rides from the database. The result is returned as a dbset of rides.

❸ We derive an iterator from the result set returned. This will allow us to step through the rides that have been returned.

❹ We then fold the collection of rides using the table_row function and the iterator. This will yield a finished chunk of HTML.

©Manning Publications Co. We welcome reader comments about anything in t

❺ The table_row function combines the current ride, r, with the accumulated HTML fragment, acc.
❻ Each new ride is converted into a row of an HTML table here. We select the id, user_name, distance, and date fields of the ride record and put each in its own column. The only special thing to note is that we apply some special date formatting to the r.date field.

The last part of the view is a function to generate the form for entering new rides into the database. For this we have some more HTML markup, which by now should be starting to look pretty familiar. We take advantage of some HTML5 input types to make the form a little easier to write. I like to use standard HTML controls whenever possible, but in some older browsers these may not be recognized. We need to collect the user's name, the distance that they rode, and the date on which they rode. We're using a counter for the id numbers and so we don't ask that of the user.

Listing 2.7. New ride input form

```
function input_form() {
  <form>
    <div class=form-group>
      <label for=name>Name</label>
      <input class=form-control id=name
        placeholder="Enter name">
    </div>
    <div class=form-group>
      <label for=distance>Distance (mi)</label>
      <input type=number class=form-control
        min=0 step=0.1 id=distance
        placeholder="Enter distance">
    </div>
    <div class=form-group>
      <label for=date>Date</label>
      <input type=date class=form-control id=date
        placeholder="2015-01-13">
    </div>
    <button type=submit class="btn btn-default"
      onclick={function(_) {                          ❶
        name = Dom.get_value(#name)
        distance = Dom.get_value(#distance)
        date = Dom.get_value(#date)
        Model.create_ride(name, distance, date)
      }}>
      Submit
    </button>
  </form>
}
```

❶ The Submit button has an *onclick* handler that collects the name, distance, and date fields from the form. We'll use these to create a new ride and store it in the database.

©Manning Publications Co. We welcome reader comments about anything in t

2.2.3 Controller code

The controller serves three main functions. The first is that it examines the route and selects which view to serve. The second function is that it starts the server, setting up resources such as CSS and the default document type. Lastly, as it's done in the MVC pattern, we tie together model logic and view logic.

Listing 2.8. The dispatcher function

```
module Controller {
  dispatcher = {
    parser {
    case (.*) : View.default_page()                    ❶
    }
  }
}
```

❶ For this simple application, we match all URLs and render the `default_page` from the View module.

The last thing left to do to get our code up and running is to determine some application-wide settings and then make a call to the `Server.start` function. The actual type of `Server.start` is rather complex because it can take a variable number (and type!) of arguments. We'll with something very close to the default generated by the `opa create` command.

Listing 2.9. Starting the server

```
resources = @static_resource_directory("resources")    ❶

Server.start(Server.http, [
  { register:
    [ { doctype: { html5 } },
      { js: [ ] },
      { css: [ "/resources/css/style.css" ] }          ❷
    ]
  },
  { ~resources },                                      ❸
  { custom: Controller.dispatcher }                    ❹
])
```

❶ We set a *directive* that defines the "resources" directory to be for static assets. These will get served as-is by the server.

❷ We specify a stylesheet here.

❸ Here we tell the server to use the resources directory that we defined above.

❹ We tell the server about our dispatch function.

2.3 Summary

In this chapter we saw what working with Opa is like. But more importantly we saw how the themes of the Transparent Web play out in a web framework that's designed to enable fast development. In particular we saw these things.

- We created a simple Opa application
- We saw how to work with the Opa tools
- We looked at the functional programming concept known as a *fold*
- We talked about records and how they are like "static duck typing"

One thing that was hard to convey in this chapter was the process of writing the application. I found that by trying to compile frequently, I was able to get quick feedback about errors in my code. These errors went way beyond what I'd get while developing in something like JavaScript. On more than one occasion Opa determined that my HTML was nested incorrectly, or that I was trying to call a function with the wrong arguments. The combination of type checking and web development is quite pleasant.

5 Understanding Static Typing

Readers familiar with dynamically-typed application programming languages like Ruby may be unfamiliar with the central role that types play in many functional programming languages. Types fill a role similar to the tests used in Test-Driven Development. That is, types provide guidance in writing code, quick feedback about errors, and documentation of APIs. Types have interesting properties in their own right though, and that's what this chapter is about. Here you'll learn about what types are, how they can be used (along with other language features) to solve problems, and finally a little about how they work.

As an overview, types are a light-weight mechanism for checking for the absence of certain types of program behavior. This can be used to look for bugs, enforce constraints, or tighten up security amongst other things. Types model a simplified version of our programs and examine them for flaws, we can than take lessons learned from that simplified environment back to our full-fledged program and conclude things about it, too.

This chapter serves as a stepping stone to the rest of the book. In the chapters that follow, concepts involving types will play an increasingly important role — they won't be the focus, but they'll be present and it is important that these concepts make sense. I won't and can't tell you everything there is to know about types. But I want to tell you enough so that types add to rather than distract from other topics. This book is about the future of programming for the web and I think types are a part of that future. It'll help to understand a little bit about them.

A secondary goal of mine is to let you in on a big secret — that static typing is a useful and pragmatic feature in a programming language. It isn't obscure or weird, just really useful. This isn't a secret because some shadowy group of functional

©Manning Publications Co. We welcome reader comments about anything in t

programmers is hiding it, far from it. Whatever the cause, it seems that this info isn't widely distributed. If you're coming from a background consisting of dynamic languages (Ruby, Python, Perl, etc.) and maybe a tiny smattering of static typing in the style of Java, then this chapter is for you.

5.1 Collections of Values

Imagine that all the values used in a program are drawn from big bins. We might pull the number 5 out of one bin or the string "Hello, World!" from another. The bins each have a label on their side that tells us what sorts of values are contained within. Oh, also, we're assuming that no malefactor has put values into the wrong bins! To introduce types then, we can say that they are like the labels on the bins. Then we'd say that "5 came out of the integer bin", or as we'd really say it: "5 has type integer."

A simple example from the Opa language shows how we can view a type as a label. Here it "labels" a variable, `x`, by saying that `x` will hold values that are integers and then we assign it a particular value, `1`.

```
int x = 1
```

Where this view really starts to come into its own is when functions enter the picture. Functions can be seen as a way to associate bins with one another. A function with a type of `int → string` will expect a value from the `int` bin and yield a value from the `string` bin. This already starts to tell us about the function works. We know that we won't be able to get a 5 from this function because that's not in the `string` bin. We also know that we're only able to *use* the result of this function in places where we could use strings.

5.2 What types can be used for

Types are a tool — among many — for improving the correctness of our code. Testing is another common means of doing the same thing. Regardless of whether we're using a type system or software testing, the goal is to catch bugs as early as possible. It is easiest and quickest to fix bugs right after you introduce them. Hunting down bugs from a production stack trace that a baffled customer is reading to you over the phone is no fun.

Through the 90's, with its profusion of dynamic languages like Ruby, PHP, Python, and Perl, static types seemed to take a lesser role. At the same time, testing came to the forefront. But lately, it seems that static types are poised for a comeback. There's been a resurgence with languages like: Scala, Swift, F#, and Rust, which all have a static type system. Some of these languages are being heavily supported by large companies (among them: Apple, Microsoft, and Mozilla). This points to big practical benefits combined with enough ease-of-use that these companies are willing to bet on them.

We should keep in mind that types are not the only way to achieve confidence. Rigorous software testing *must* also play a role. Types characterize what we can know about the code "as written", while tests exercise the code "as run". Unlike types, tests can take advantage of runtime information. This enables us to catch *different kinds* of bugs — which can't be found using types alone.

The culture of Ruby espouses a strong discipline for writing comprehensive tests. A good test suite can catch bugs early on in the development cycle. Ideally, when something isn't working as expected, there'll be a unit test that points to why.

But unit testing depends on you having written enough fine-grained tests. When tests are too broad, it is hard to point to a line and say "that's the problem". But when tests are too narrow they can become a maintenance nightmare.

Types act as a sort of pervasive and always-on suite of tests that you don't have to write. Accordingly, types are good at finding and preventing similarly pervasive problems. The sorts of errors that types excel at are exactly those that are widespread throughout the codebase and involve the programmer having to meticulously and continuously keep things straight.

Two big problems that fit this particular bill are *null-pointer errors* and *injection* errors.

5.2.1 Avoiding null-pointer errors

Tony Hoare famously called the *null reference* his "billion-dollar mistake." [1] The null reference error is common and so it goes by several different names, in Java it's known as `NullPointerException` (or even `NPE`), and it goes something like this.

Listing 5.1. Anatomy of a NPE in Java

```
public class NullPointer {

  static class Greeter {
    String greet(int n) {                    ❶
      if (n > 0) {
        return "Hello";                      ❷
      } else {
        return null;                         ❸
      }
    }
  }

  static class Responder {
    String answerGreeting(Greeter klass) {
```

[1] In a talk given in 2009 Tony Hoare told the story of when he was implementing the programming language ALGOL W. While designing the language he had the option of adding null references to the language. He settled on adding null references but now feels that it was a bad idea. The *billion-dollar* part comes in because he speculates that it has cost that much in lost time and effort, "more than a tenth of a billion, probably less than 10 billion". He also won the ACM A.M. Turing Award, sort of the Nobel Prize of computing, so I'll let this one slide.

©Manning Publications Co. We welcome reader comments about anything in t

```
      String greeting = klass.greet(0);        ❹
      if (greeting.equals("Hello")) {
        return "Nice to meet you!";
      } else {
        return "Um, Hello?";
      }
    }
  }

  public static void main(String args[]) {
    Greeter greeter = new Greeter();
    Responder other = new Responder();

    String s = other.answerGreeting(greeter);  ❺

    //int x = null;  // not okay, this is a compile error
    System.out.println(s);
  }
}
```

❶ We begin okay enough. `Greeter` has a `greet` method that chooses a greeting depending on the value of an `int` that's passed in.

❷ If n is greater than zero, then `greet` returns a `String`. This matches what's declared in the method signature.

❸ Here's where problems begin. Perhaps when we were writing this method we thought "if n is not positive then no greeting is appropriate." We select `null` as an indicator that effectively means "no greeting".

❹ We call the `greet` method with the argument `0`. Based on the definition of `greet`, *greeting* will get the value `null`. This is, admittedly, easy to spot. But this exact situation, with more indirection could prove very hard to track down.

❺ This call sets off the whole chain of events. The compiler does not complain about this code, but we do get a runtime NPE.

When we run the code in listing 5.1 , we'll get the following message:

```
Exception in thread "main" java.lang.NullPointerException
        at NullPointer$Responder.answerGreeting(NullPointer.java:16)
        at NullPointer.main(NullPointer.java:28)
```

This is a simple example, but some things to note were:

1. The compiler didn't complain.
2. The error message implicated the wrong lines (depending on how you think about it). The error points out `greeting.equals ("hello")` as being the cause. That's *true*, but it doesn't get at the root of the flaw. I'd say that happens in the *definition* of `greet`, not its *call site*.

This is not unique to Java! The same sort of behavior can show up in Ruby (and many other languages). The Ruby version would look like Listing 5.2, .

©Manning Publications Co. We welcome reader comments about anything in t

Listing 5.2. Ruby's version of a NPE

```
class Greeter
  def greet(n)
    return 'Hello' if n > 0       ❶
  end
end

class Responder
  def answer_greeting(klass)
    greeting = klass.greet(0)
    if greeting.match(/[Hh]ello/) ❷
      'Nice to meet you!'
    else
      'Um, Hello?'
    end
  end
end

class NullPointer
  def self.run
    greeter = Greeter.new
    s = Responder.new.answer_greeting(greeter)
    puts s
  end
end

NullPointer.run
```

❶ Here we only return a string in the case that our test, `n > 0`, is true. Otherwise we return (by default) `nil`.

❷ The only big thing that differs from the Java code is that the == method is defined for `nil` in Ruby. I decided to give `use_greeting` a souped-up version that checks to see if either "Hello" or "hello" is contained in the greeting.

And a similar sort of runtime blowup will happen:

```
null_pointer.rb:10:in `use_greeting': undefined method `match' for
nil:NilClass (NoMethodError)
        from null_pointer.rb:21:in `run'
        from null_pointer.rb:26:in `<main>'
```

Types provide the answer for this sort of problem. When we defined `greet` in Listing 5.1 , we declared it as having a return type of `String`, but that's only sort of true. The return value is either going to be a `String` or it will be `null`. The problem is that Java (and many other languages) consider `null` to be an acceptable *value* for any reference type (e.g. `String`). That is, wherever you have a variable that's meant to hold an object, it could also hold `null`:

```
MyClass a = new MyClass(); // Okay
```

©Manning Publications Co. We welcome reader comments about anything in t

```
MyClass b = null;          // Okay
```

But in some languages, `null` is not considered to be a valid value for any type. In Java at least you can see this sort of behavior if you attempt to assign `null` to a non-reference type:

```
int x = null; // not okay, this is a compile error
...
error: incompatible types
    int x = null;
            ^
  required: int
  found:    <null>
1 error
```

So the glimmer of the solution is there. The complier can simply not allow variables to *ever* hold values of the wrong type. When this sort of rule is in place, assignments where the types don't match are forbidden.

```
MyClass a = null; // compile error
```

There are languages that have this philosophy built into the language. Haskell is an example of a language where variables are non-nullable. In such a language mechanisms like *option types* (ML-family) or *Maybe* (Haskell) are used to capture the idea of something that either has or does not have a sensible value. But to understand how an option type works we'll first have to understand something called a *sum type*.

Sum types

A sum type is a natural idea cloaked in what may be unfamiliar terms. Think of this as a type representing a bunch of non-overlapping options. A similar idea is Java enums. To understand sum types, let's bring back our fruit example from SmoothieRecipeFunction. There we were talking about how to express the idea of a fruit*generally*. One way to think about this is by thinking of sets again. A way to express the idea of "fruit" could be to just list them all: {apple, banana, cherry, grape, …}. That would look like this in Opa.

```
type fruit = {apple} or {banana} or {cherry}
```

It means that we're defining a new type, `fruit`, that has three distinct values, `{apple}`, `{banana}`, and `{cherry}`. To use the language of sets, `{apple}`, `{banana}`, and `{cherry}`are the three elements of our set of `fruit` (s). Any function which uses the `fruit` type, will handle values of either `{apple}`, `{banana}`, or `{cherry}`. If we want to pick out which value is being used we do something called *pattern matching*. This is something like a case statement in other languages.

©Manning Publications Co. We welcome reader comments about anything in t

```
function prepare(fruit f) {
  match(f) {
    case {apple}: "chop"
    case {banana}: "peel"
    case {cherry}: "remove pit from"
  }
}
```

Here, the `fruit f` will take on *exactly one* of those options. Depending upon which value `fruit f` has, we return a short preparation instruction (of type string).

This is better than just doing an `if` check to see which of the three possibilities `f` has assumed. For starters, the compiler will check that we're examining all the possibilities. If we comment out one of the cases in our `prepare` function, the compiler will notice.

```
function prepare(fruit f) {
  match(f) {
    case {apple}: "chop"
    case {banana}: "peel"
    // case {cherry}: "remove pit from"
  }
}
```

And when we try to compile this, we'll get an immediate, compile-time, error.

```
Warning pattern
File "option.opa", line 3, characters 27-137, (3:27-9:1 | 73-183)
Incomplete pattern matching: case {cherry} is missing
Error: Fatal warning: 'pattern'
```

In this example Opa is performing *exhaustiveness checking*. The compiler checks that all the cases that make up `fruit` are being dealt with in the match expression. Being able to check that no case is left un-handled is important to the solution of the billion-dollar mistake. In the next section we will want to see that a value is either of one type or another and that we handle both cases.

To summarize, sum types, are a way that we can describe things that take on just one of many possible values at runtime.

Option types

Now that we have a feel for sum types, we can introduce the idea of an *option type*. You can think of an option type as a kind of wrapper or label that surrounds another type. The option type lets us distinguish between valid examples of the value and something like "no value" or "no answer". This is useful whenever the result of performing some operation may be unknown, impossible, or could fail. In programming, this is really common.

A simple example of this is if we want to model integer division. As we learned in

©Manning Publications Co. We welcome reader comments about anything in t

school, dividing by zero is not defined when the denominator is zero. The usual approach to handling this is to signal an exception or error at runtime when we divide by zero.

```
function div(int n, int d) {
  n / d
}

div(1, 0) // Error: Exception : Division by zero
```

If we examine the type of our `div` function, we'll see that it is inferred as `int, int → int`, that is it maps two `int` values to an `int` value. But I would argue that we're omitting information from our function. As we just noted, *division by zero is undefined*. We can model this using an option type. *Most* of the time when we divide two `int` values we get another `int` value, but when the denominator is zero, then there's no answer.

```
function div(int n, int d) {
  if(d == 0) {
    {none}
  } else {
    some(n / d)
  }
}
```

If we now look at the type inferred for this function, we see that it is pretty interesting.

```
div : int, int -> {none} / {some : int}
```

The `{none} / {some : int}` notation means that this function is returning a sum type. One of the possible types is `{none}`, representing no value, the other option is `{some : int}` — which means exactly that, the answer is going to be some integer. This describes exactly what is happening in the division function.

When we look at many uses of `null` across different programming languages, we notice that they are often just shorthand for a more nuanced concept. We can often represent this in a type-safe way with options.

- When we look up something in a dictionary, hash, or list we sometimes use `null` to mean "element not found". We should instead use an option type.
- When reading values out of strings, like JavaScript's `parseInt`. We can use an option type to represent a failed parse. Or better yet, we can use a sum type that gives the reason for the failed parse and a position of the error.

To summarize, null references, have a pernicious influence on software. Because nulls can be lurking inside any reference, they call into question lots of code. Developers must carefully check to make sure that references are valid before using them.

©Manning Publications Co. We welcome reader comments about anything in t

So in order to solve this problem, we first have to make a few changes to our language. We remove null references from the language. To cover previous uses of null, we introduce *option types*. Think of all of the standard library functions that used to raise an exception or return a `null` when an element is not found. These will be replaced with Option. Option types help because they explicitly mark return values as "might be empty" and then the type system ensures that we handle all the places where this could cause trouble. Option types are a specific case of a *sum type*. Which is in turn a type used to model a series of mutually-exclusive values. For example, if we're using an `` `option int ``:

- There is no `int` value present
- There is an `int` value present and it is `1` (for example)

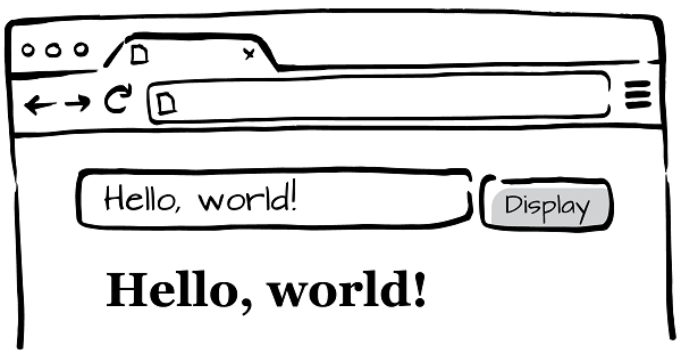
This feature is tremendously useful and used in languages like Rust, Scala, Swift, Haskell, and more.

5.2.2 Solving injections

It often comes up in web programming that you must include program fragments within other programs. Examples are JavaScript in an HTML file, or HTML in a JavaScript string. In the latter case, you're left with a JavaScript string containing literal HTML syntax. While this isn't so bad with HTML, strings containing SQL statements or JavaScript code fraught with security risks.

Let's illustrate the situation with a form included in a web page. Using the form users can fill in text that they'd like to be displayed on the page.

Figure 5.1. Filling out the form to display text



The code to accomplish this can be found in Listing 5.3 . When the user clicks the button it grabs the contents of the text field and inserts it literally in the page.

Listing 5.3. A simple input form

```
<html>
```

©Manning Publications Co. We welcome reader comments about anything in t

```
  <head><title>Display</title></head>
  <body>
    <script>
      function display() {
        var target = document.getElementById('target');
        var text = document.getElementById('text');
        target.innerHTML = text.value;
      }
    </script>

    <form action="#">
      <input id="text">
      <button onclick="display()">Display</button>
    </form>

    <h1 id="target"></h1>
  </body>
</html>
```

But there's already a problem with this. Whatever the user types in the input field is inserted directly into the page. As we just saw, when this is normal text, everything's fine. But this form also allows us to include markup like in Figure 5.2 .

Figure 5.2. Inserting HTML into the form leads to an injection

Admittedly, this is a pretty benign example of injection. But the root of the error can be quite dangerous in the wild. The problem was we failed to make a *type-based* distinction between HTML and plain text. We allowed plain text to be treated as if it were HTML. In the context of an HTML page, what was just text like `<i>world</i>`has become active when interpreted according to the rules of HTML. When plain text is interpreted as code, like JavaScript, all sorts of bad things can happen.

To sketch out a solution to this problem, I'll be using a dialect of JavaScript called TypeScript, which supports type annotations. TypeScript extends JavaScript with type annotations. These annotations look like this.

©Manning Publications Co. We welcome reader comments about anything in t

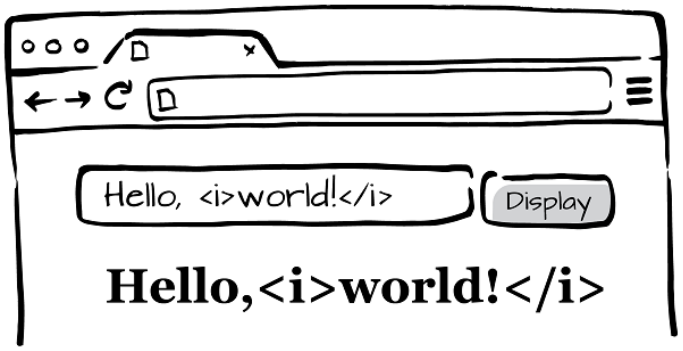
```
function escape(str : string) : HTML { ... }
```

The `escape` function will replace all characters in the string, `str`, which have special meaning in HTML with their equivalent HTML character entity. For example, `<` (open angle bracket) will be converted to `<` and `&` will be converted to `&`, etc. These character entities are ignored as far as HTML is considered, but are displayed as their human-readable equivalent. This function makes any string *safe* to use as HTML.

In the code above the parts after the colons, `string` and `HTML`, are `type signatures and they label the type of the variables. The` `` `HTML' `` *type is the return type of the function.*

So, using this slightly different code, we can protect ourselves from this sort of injection. You can see the properly escaped output in figure 5.3 . We are seeing the "markup" only because it isn't being treated as *markup!*

Figure 5.3. HTML inserted into the form is escaped before display



First, I've split out the TypeScript source into its own file so that we can run the typescript compiler, `tsc`, on it. Having removed the script to its own file leaves the HTML as it is in listing Listing 5.4 .

Listing 5.4. The HTML form alone

```
<html>
  <head>
    <title>Display</title>
    <script src="types.js"></script>
  </head>
  <body>
    <form action="#">
      <input id="text">
      <button onclick="display()">Display</button>
    </form>
```

©Manning Publications Co. We welcome reader comments about anything in t

```
    <h1 id="target"></h1>
  </body>
</html>
```

The TypeScript is very close to JavaScript but it has added type annotations. We've broken the form handling into three separate functions: `escape`, `insert`, and `display`. The division of labor is that `escape` converts values of type `string` into values of type `HTML`. The `insert` function accepts only `HTML` values and then places them in the page. Lastly, `display` combines these two functions, it first gets user input from the form and then escapes it before inserting it in the page.

Listing 5.5. Re-written code including type signatures

```
class HTML {                                                    ❶
  html: string;
  constructor(html : string) {
    this.html = html;
  }
}

function escape(str : string) : HTML {                          ❷
  var escaped = str
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&apos;")
  return new HTML(escaped);                                     ❸
}

function insert(html : HTML) {                                  ❹
  var target = document.getElementById('target');
  target.innerHTML = html.html;                                 ❺
}

function display() {
  var text = document.getElementsByTagName('input')[0];
  insert(escape(text.value));                                   ❻
}
```

❶ We first define a new class (type) for HTML values. This will act as a wrapper around a string value that indicates that it has been properly escaped. Put another way, that it is valid HTML.

❷ The `escape` function is the only way that we provide to create HTML values. Since everything is going through the same gateway, we'll know that any HTML values that we encounter will have been escaped.

❸ Here we return the value after it has been escaped.

❹ The `insert` function puts valid HTML values into the page. The type of its argument is HTML.

❺ Here we "unwrap" the HTML value and get the underlying string that it contains.

©Manning Publications Co. We welcome reader comments about anything in t

❻ Here we escape and then insert the value from the input form. If we forget to call escape on the input value, we'll get a type error.

If we alter the last line to read:

```
insert(text.value); ❶
```

❶ `escape` is omitted

We will receive the following error when we try to compile the code:

```
types.ts(25,3): error TS2082: Supplied parameters do not match any
signature of call target:
        Type String is missing property html from type HTML.
types.ts(25,3): error TS2087: Could not select overload for call
expression.
```

Which, amid some other things, is pretty clearly pointing out that we've got the call wrong. The `insert` function will only accept HTML values, but we're trying to supply it a string.

Another error is double-escaping. This wouldn't crop up in actual code as blatantly as I'll show it here. The error may be hidden in different parts of the codebase that are well separated. But through some chain of calls what we effectively end up doing is this:

```
escape(escape("Hello, <i>world!</i>"));
```

And again the compiler catches us:

```
types.ts(28,1): error TS2081: Supplied parameters do not match any
signature of call target.
types.ts(28,1): error TS2087: Could not select overload for call
expression.
```

Now this escaping is something that we could do in any language, typed or not. What types have bought us here is that they provide a way to "label" values according to what they mean in our application. Once we can tell the difference between values — even if they are both the same underlying type — we can enforce rules about how to work with them. The type-based solution to injections is doing just this. We track when values have been properly escaped and limit display functions to only work with "safe" values.

5.3 How types work

In "Collections of Values", it was perfectly clear what I was getting at with my simple classifications. The bins of values grouped all *things* that shared some property. What's ultimately important for us as programmers is what we can *do* with these values. So that property for the `Int` means "I can perform all the

©Manning Publications Co. We welcome reader comments about anything in t

arithmetic operations on this". For strings, we might consider that to be "things that I can substring, concat, upper-case, lower-case, etc."

We've thus hit upon the idea of membership, what it means to be a member of some group. The mathematical concept here is a set, and the programming language idea is a type. We can identify things that are *members* of a *set* or that are *inhabitants* of a *type*. Mostly though, we'll just talk about something like the number *3* as having a type of Int.

The rough idea of what a type system is then, is a way to classify variables and expressions in a program. These types then all have to mesh together so that no type rules are violated (e.g. addition only works with two integers). If we more carefully formulate this idea, we arrive at a nicer definition for what a type system is.

5.3.1 A definition of "type system"

The definition that I found is heavy, but I think it makes a lot of sense once you unpack it.

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. — Pierce (TAPL)

This captures an important idea about what a type system does and is for. A type system is *syntactic* which means it is just reading your source code. Everything that the type system learns, it learns without running your program. This is important in two ways.

The first is that some things we'd like to know before we ever run a program. Like, "are there any obvious flaws that I can quickly fix?" There's a real software engineering benefit to a type system. It acts as a pervasive suite of tests where failing them means that the program won't compile. This prevents problems really early in the software development cycle with only a little extra effort.

The second is the nature of the things that type systems look for. If we run a program and nothing bad happens we may not have learned very much. Had we run it a minute longer, maybe we would have found a new bug. Maybe the code path that we took skipped over has some bad behavior. Type checking always examines all the code as written. This means that it can catch bugs even in sections of the code that rarely run.

Here's an example where "checking all the code" comes into play. In Ruby I can write an expression that decides on a greeting.

```
def prepare_greeting(name)
  greeting = if true
      "Hello, "
    else
```

©Manning Publications Co. We welcome reader comments about anything in t

```
      5
    end

  greeting + name
end
```

It's silly because it is that it is clear to *us programmers* that we're never going to run the `else` branch. In real code, perhaps the `true` condition wouldn't be so clear, or perhaps it looked more like this:

```
def prepare_greeting(name)
  greeting = if false_on_leap_days_that_are_also_tuesdays
  # ...
end
```

Now that's more trouble! But if we put on our pointy philosopher hats and ask "what sort of value does `greeting` represent?" That's more interesting. If the true/false status of the if expression could go either way, that means the type of *greeting* is sometimes a string and sometimes a number. But that's big trouble because + (plus) only works with two strings [2]. The point is that we'll only hit this problem when the test in the if statement happens to be false. In some code, that could be a rare occurrence.

Let's do the same set up, but with static types.

```
prepareGreeting name =
  let greeting = if True
        then "Hello, "
        else 5
  in  greeting ++ name
```

We still have `True` for the test in the boolean expression; the `else` clause is *dead code*, it'll never run. But when we try and compile this we get an error that essentially says: "The expected type was String but the actual type was Int." This is suggestive of lots of possible fixes, but the point here is that we have to make the types agree for this to be a valid program.

In practice, I find this to be (in general) a good thing. The reason is for maintenance. Code that's inert today may not be in the future. It is easy to make innocuous changes that will then introduce bugs.

5.3.2 How a type system is implemented

If all the talk about types seems hard to ground the way that type checking is implemented is not that complex but we first have to understand a little about how

[2] In Ruby, **+** is a method that's defined on both strings and numbers. The catch is that Ruby won't *coerce* a string into a number. This means that the expression `5 + "Chris"` will fail only because Ruby can't figure out a way to make "Chris" into a number. We could more clearly see the error if I had written the code: `greeting.concat(name)`. That expression would fail with `NoMethodError: undefined method `concat 'for 5:Fixnum`

©Manning Publications Co. We welcome reader comments about anything in t

languages are compiled.

Parsers and compilers are a big juicy fascinating topic, but they are also one that goes a little beyond the scope of this book. For readers that are really interested in knowing more about parsing and languages, there are several books listed in the references that would be good to check out.

My discussion here should just give you a feel for what's going on when a program is type checked. We start with a short snippet of a made up language that supports static types and type inference.

```
var x = 5

func add(a, b) {
  return a + b
}

add(1, x)
```

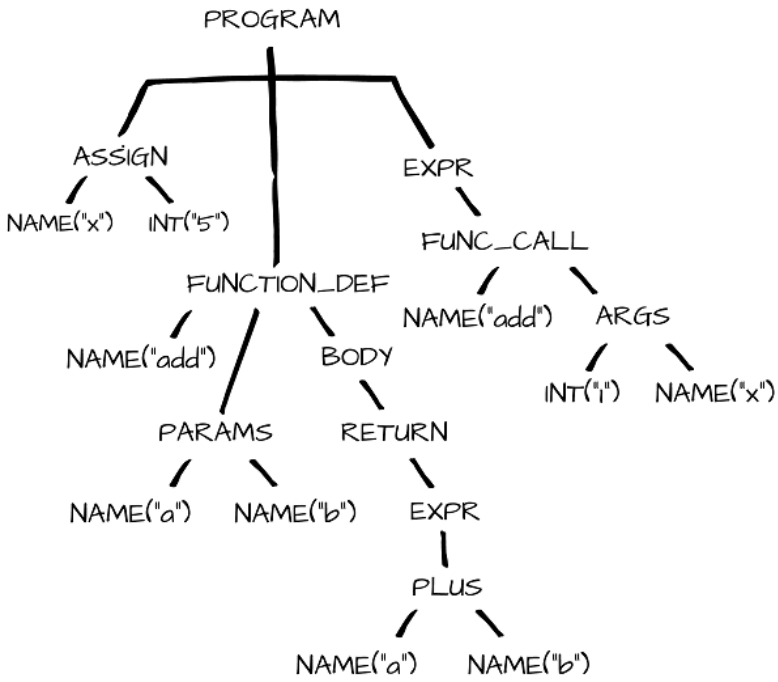
The first stop in type checking, is shared by pretty much any programming language. The text of the program is first lexed, or converted into a stream of *tokens* — which are placeholders for significant syntactical structures. This phase of compilation removes unnecessary detail like the exact amount of whitespace or, in some cases comments. When this phase is complete the program will "look" like this. I'm scare-quoting that because this phase does not result in actual program text, this would be implemented as a list of tokens internally. I'll represent this as somewhat similar to the source text, but in reality this should be thought of as a stream of tokens.

```
VAR NAME("x") ASSIGN INT("5")

FUNCTION_DEF NAME("add") LPAREN NAME("a") COMMA NAME("b") LBRACKET
RETURN NAME("a") PLUS NAME("b")
RBRACKET

NAME("add") LPAREN INT("1") COMMA NAME("x") RPAREN
```

Next, the truly interesting work begins. This is where the parser converts this unstructured input into an abstract syntax tree (AST). Depending on the exact working of the compiler, there may be other phases before or after this point. For the most part though, the above stream of tokens is converted into a tree that captures the semantic structure of the program. This looks something like this can be seen in Figure 5.4 .

©Manning Publications Co. We welcome reader comments about anything in t

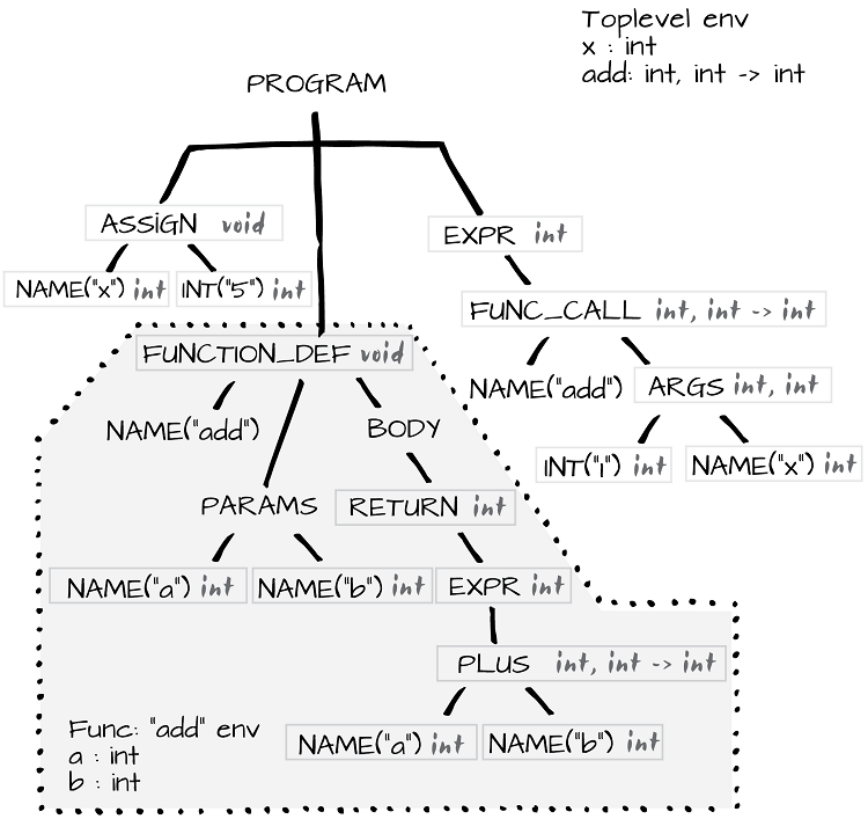
Figure 5.4. The parser constructs an Abstract Syntax Tree from the tokens



This AST is illustrative of a whole slew of subtle variations, depending on what we want to accomplish with the language. The next phase is when the type checker analyzes this AST and ornaments it with type information. In Figure 5.5 we can see how types have been added to the AST.

©Manning Publications Co. We welcome reader comments about anything in t

Figure 5.5. After the type checker analyzes the AST



The rough idea is that we walk through the tree and look for things that impose constraints on the type of values, we then "bubble up" these constraints and look for situations where there's a mismatch. There's one last wrinkle in the process of type checking that we'll need. I've introduced the idea of a "type environment" to the diagram. This is a mapping from names to types. Entries are added to the environment when we do things like define a var or function. Let's now start with the leftmost, deepest part of the tree.

We first encounter `var x` (i.e. `NAME("x")`) at the start of the program. Initially we don't know what type this variable has because this is the first we've seen of `x` and the type environment is empty. At this point we are not able to assign a type to `x` and so we continue checking, treating `x` as having an unconstrained type. Next we examine the other branch of the assignment and find the literal int 5. We know that this has a type of int (we know this from just looking at it). Assignment has the side effect of adding new types to the type environment, since assignment means that `x` is receiving the value of the right hand side, we know that their types must be the same. Thus we enter `x : int` into the top level type environment. The overall

©Manning Publications Co. We welcome reader comments about anything in t

type of this expression is `void` since assignment doesn't return a value.

Next we move into the function definition for `add`. Like variable assignment, function definitions enter a new entry into the typing environment. Here we'll initially enter an unknown type for the `add` function. As a slight twist on variable assignment, function definitions also create their own type environment, which you can see at the bottom of the diagram. Next we inspect the parameters to the function. Much like variable assignments, parameters create new entries in the type environment, here `a` and `b` initially have unknown types. We then proceed into the body of the function. The deepest part is the `+` (plus) operation. Plus requires its two operands to be `int` (again this can be taken as a *thing we just know*, like "5" having type `int`). We thus infer that `a` and `b` both have type int within the `add` type environment. This also allows us to see that the return type of `add` is int. So we can then fill in the entry for `add` in the top level type environment.

Lastly, we move on to the function call. Here we can make use of our type environment to set a constraint for the `add` function call to be `int, int → int`. We immediately see that the first argument is literally an `int`, next we encounter `x`, we look this up in the type environment and find that it too is `int`. The arguments for the call to `add` are thus `int, int`. The overall value of the expression is then also `int`. And that concludes the program.

Our analysis proceeded really nicely and every type lined up exactly. But we can hit snags along the way. We could find that types are *constrained too much*. In one place a type could be treated as an int but in another place it is treated as a bool. This is a type error. Or we could find that types are *not constrained enough*, this may or may not be an error. There could be multiple valid ways the types are being used in which case the compiler might just pick one or it could also require the programmer to specify which use they mean. Another possibility for underconstrained types is polymorphism. I didn't talk about this in this example, but you can think of things like a list of any type. We may only care about operations on the structure of the list and so we leave the type of the list elements to be purposefully vague.

I've skipped over some details here, but this gives you a feel for how type checking with inference works. We look for constraints in the program and try to find type assignments which satisfy them.

5.3.3 An example of a hard-to-spot bug

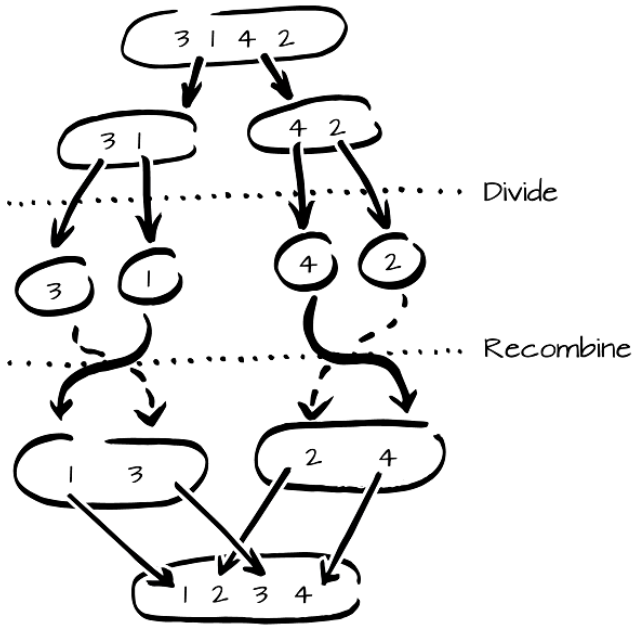
I found this example in a talk titled Strong Typing given back in 1999 for a Perl meetup. It is an excellent example of the kinds of problems that can be found with a type system. The example is written in Standard ML, a functional programming language related to Opa. Opa will be covered later in this book. I'm going to describe this sorting program as if it worked correctly. But as a fair warning to you there's a subtle bug in the `sort` function. I will fully explain it by the end. The

©Manning Publications Co. We welcome reader comments about anything in t

whole point of this section is that the bug is hard to find.

The example is a short program to do a merge sort. Merge sort is a way of sorting a list where you first recursively divide the list in half. When you have appropriately small lists, say a single element, you then recombine those lists be recursively drawing from the smaller of the two sublists. Here's a picture of how the algorithm works.

Figure 5.6. A schematic of how the merge sort algorithm works



The place to start is with the subdividing. Here's a function called `split` that recursively splits up a list into two smaller lists. If you've never seen ML syntax before, here are some things to spot. Functions are defined as a series of definitions. This should look somewhat familiar from math class back in the day. Here is a definition of the Fibonacci numbers, the pattern you get when you always add the previous to terms in the series together (1, 1, 2, 3, 5, 8,…).

```
f(1) = 1
f(2) = 1
f(n) = f(n-1) + f(n-2)
```

Likewise in the `split` function defined in ML, we write the function by giving examples of each of the cases that we're interested in.

©Manning Publications Co. We welcome reader comments about anything in t

Listing 5.6. The `split` function of the merge sort algorithm

```
fun split []        = ([], [])                    ❶
  | split [h]       = ([h], [])                   ❷
  | split (x::y::t) = let val (s1, s2) = split t  ❸
                       in (x::s1, y::s2)          ❹
                      end
```

❶ When the list that we're given is empty, there's nothing much to do. We just return a pair of empty lists.

❷ When there's a single element in the list, put that element in the first of the two sublists that we are returning. Since we don't have any other elements to use, the second sublist is empty.

❸ This is the most general case. We have a list with at least two elements in it. The syntax `x::y::t` denotes a *pattern match* which is taking apart a list. This is called the `cons` operator. We name these elements `x` and `y`, we also name `t` which is the "rest" of the list (or the *tail*). Next, we recursively call split on the rest of the list and name the two sublists that we get from that as `s1` and `s2`.

❹ We now have to assemble all these pieces together. We take the first element, `x` and put it on the front of the first sublist, `s1`. Next, we take the second element, `y` and put it on the front of the second sublist `s2`. This becomes the return value of our function.

When dealing with recursive function definitions like this, it sometimes helps to think of the base cases. In listing 5.6, , if we had exactly two elements in the list, (3) would mean that `split t` would return two empty lists, because of (1). Then in (4) that means that we'd be prepending `x` and `y` onto two empty lists, which would yield two one-element lists: `([x], [y])` and we'd be done.

Because this is a chapter all about types, it is important to point out what type this function has. In ML, the type system is able to *infer* types from how we use them in the bodies of our functions. This is a neat trick because it saves us from having to write type signatures in many places. Anyway, the type signature that `split` has is:

```
'a list -> a list * a list'
```

This says that it takes a list of any type and returns a pair of lists of any type. The `'a'` syntax means that this is a type variable. This is roughly equivalent to the idea of generics in Java, where it would be written like `List<A>`. The `*` part above, that looks like we're multiplying two generic lists together is what's called a product type. This is a pair, or tuple. The intuition is that you have two types put together and you can think of the overall type as being all the possible values of the first member of the pair times all the possible values of the second member. If that seems a little opaque, don't worry about it. Just know that the overall type means we're going from a list to a pair of lists and the contents of those lists can be any type (e.g. int).

After we split our lists, we need to merge them back together, sorting as we go. This function is actually going to be pretty similar to `split` except, well, in reverse. The idea is that we'll be given a pair of lists as input and the job of `merge` is to combine them together. We sort the lists by finding the lesser of the two elements on the front of the lists and then put that element at the front of the list. We then take the rest of the remaining lists and `merge` that together.

Listing 5.7. The `merge` function of the merge sort algorithm

```
fun merge ([], x)          = x:int list           ❶
  | merge (x, [])          = x                    ❷
  | merge (h1::t1, h2::t2) =                      ❸
    if h1 < h2    then h1::merge(    t1, h2::t2)  ❹
                  else h2::merge(h1::t1,     t2)  ❺
```

❶ Again we match on patterns to define the function. If the first list is empty then the result is the second list. Though it is uncommon, here we do have have to give the type checker a hint here about what type we expect. We ask the type checker to assume that `x` is a list of `int`.

❷ Symmetrically with the first case, if the second list is empty, then the result is the first list.

❸ In this case we have two lists. We pull the respective first elements from each list. We name the first `h1` and the second one `h2` ("head one" and "head two").

❹ Now we do our first ever comparison! We check to see which of `h1` or `h2` is smaller. If `h1` is smaller, it is going to go on the front of the merged list that we're building. We recursively call `merge` with the rest of the first list `t1` and the entire second list `h2::t2` (notice how "pulling a list apart" looks just like putting it back together? That's pattern matching).

❺ In the case that `h2` is smaller (or equal) we do the reverse. We put `h2` on the front of the recursively-merged list.

Again, to understand recursive functions like these, think of base cases and, what I think of as, *the destination*. Here the destination is that we want to build a list of `int`. In the first two cases we just are given a list and we don't really have anything we can do but leave it untouched. In the last case we have four pieces that we've pattern matched: `h1`, `t1`, `h2` and `t2`. We have to use them all because if we omitted one our `merge` function would drop elements (really bad!). Knowing that we want to put the overall list in order helps here. We reduce this problem to picking which of `h1` and `h2` is the smaller and then putting it first. By the nature of recursive functions, we rely on the fact that calling `merge` with the "rest of the pieces" will do the right thing.

To convince ourselves that `merge` works, think of a base case of `merge` where we just have two elements. One of them will be chosen as the smaller and then merge will be called with an empty list and a one-element list. This in turn will equal that one-element list. That means it has ordered our two-element list.

The type that's inferred for this function is more specific than `split` was.

```
int list * int list -> int list
```

In `merge` we actually need to inspect the contents of the list we are given. Namely, we compare two list elements with `<` (less than). The less than function works on `int` and so the type checker has inferred that the contents of the two lists must be `int`. Since we are building a list with these elements using the list concatenation function, `::`(sometimes pronounced "cons"), which has a type ’a → 'a list → `a list`, the return type of the function is `int list`.

The last function that we’ll look at is the one that puts `split` and `merge` together to actually sort a list. We’ll write a second version of this function in just a bit, so we’ll call this one `sort_1`. `sort_1` works as a kind of control structure which calls the helper functions.

Listing 5.8. The `sort_1` function of the merge sort algorithm

```
fun sort_1 []  = []                              (1)
  | sort_1 x   = let val (p, q) = split x        (2)
                  in merge (sort_1 p, sort_1 q)  (3)
                 end
```

(1) An empty list is considered to be sorted.

(2) For any other list, `x`, we first split the list into two sublists, `p` and `q`.

(3) We recursively sort the two sublists and then merge them together.

The basic form of this is correct, but as I mentioned earlier, there’s a bug in this function. The smoking gun is the type signature. We would expect the type signature of this function to be a list of `int` to a list of `int`.

Listing 5.9. We would expect this type signature

```
int list -> int list
```

But the type signature that we actually get is this.

Listing 5.10. The inferred type signature for `sort_1`

```
'a list -> int list
```

This is a bit confusing and it hints at the bug. This type signature is saying that we can give the `sort_1` function a list of anything and it can somehow give us back a list of`int`. But the input shouldn’t be totally unconstrained like that. In the `merge` function, we’re using the less than operator on the input list. The less than operator is only defined for numbers, it wouldn’t work if the input list were a list of strings. The only way for the type checker to not see that we’re going to use `<` on the elements of the list is if it never gets there. And if we never get to that

part of the program (in `merge`) that means we must go into an infinite loop.

Sure enough, when we load this program into an ML interpreter (I'm using *SMLNJ v110.77*), we can see that we go into an infinite loop whenever we try to use sort with anything other than an empty list.

```
- sort_1 [1, 2, 3];

Interrupt
```

The interrupt is when I hit Ctrl-C to halt the program. The problem with this sort function is that it doesn't cover all the cases that can come up, namely we have to say what happens when `sort_1` is given single element list. When that happens, we just return the list because it is already sorted. The actual infinite looping stems from when we recursively call `sort_1` on one of the two split sublists. One list will be the empty list, but the other will be a single element list that's then passed to sort_1 — which is what was passed to sort_1 in the first place, hence an infinite loop.

Listing 5.11. Tracing through a call to `sort_1`

```
sort_1 [x]                ❶
...
split_1 [x] = ([x], [])   ❷
...
sort_1 [x]                ❸
```

❶ We call sort with a single element list.

❷ This matches the x case in the definition of `sort`, so `split` gets called.

❸ This leads to `sort` being called again on a single element list, and thus no progress was made in this sublist. It'll loop forever like this.

The fix is straightforward. We just say that a single element list is already sorted.

Listing 5.12. The final and corrected sort function, `sort_2`, is this:

```
fun sort_2 []  = []
  | sort_2 [x] = [x]                              ❶
  | sort_2 x   = let val (p, q) = split x
                  in merge (sort_2 p, sort_2 q)
                 end
```

❶ We added this line which says that a single element list is already sorted.

So we've just seen that a type system can catch hard to spot bugs before we've even run the code! In this case we noticed that the type of the function didn't make sense and that in turn pointed to incorrect run-time behavior.

©Manning Publications Co. We welcome reader comments about anything in t

5.4 Summary

This introduction has just scratched the surface of types. The rest of this book will deal with languages and features which make heavy use of types for a variety of reasons. Besides looking for bugs, types are used to demarcate server-side and client-side code, to protect SQL and JavaScript from code injections, and to model time-varying values. We'll see more of these specific uses in later chapters. What we've seen here will help to make those uses much clearer.

In this chapter we covered a few big ideas.

- Types can be thought of as categorizing expressions within a program. We can then check that these categorizations are consistent.
- We saw that types can help in solving real-world problems like code injections and null-pointer errors.
- We saw how type checking and type inference work to analyze a program.
- We saw an example of catching an infinite loop before we even ran a program.

There is a catch to this, however. Because we are simplifying our program in order to analyze it, we'll always miss out on some details. The simplified model will not capture the full range of things that the real program will encounter. We'll always face two main problems with types:

- Types are conservative. There will always be programs which are fine but the type system can't see that fact, mistakenly reporting that the program has a type error.
- Programs that successfully type check can still have bugs. Certain properties are difficult or impossible to translate into types.

That said, types are a good bargain in software development. They catch a lot of subtle and not-so-subtle bugs with only a minimum of overhead. In most languages it is possible to dial up or down the amount of information carried by types until a good compromise is found. In my own experience, it is better to have a robust type system and strategically side-step it on occasion than to not have one at all.

©Manning Publications Co. We welcome reader comments about anything in t