CHAPTER 9

Elm

Elm is a functional language ideal for interactive applications.

First appeared	2011
Creator	Evan Czaplicki
Notable versions	0.10 (2013) • 0.17 (2016) • 0.18 (2016)
Recognized for	Interactive functional programming
Notable uses	User interfaces, Games
Tags	Functional, Statically-typed, Subscription-oriented
Six words or less	Functional programming for the web

Evan Czaplicki created Elm in 2011 as a language for building browser-based applications in the style of functional reactive programming, or FRP. Like CoffeeScript, Elm applications transpile to JavaScript. But while CoffeeScript looks like Python or Ruby—with dynamic typing and assignable variables—Elm is part of the statically-typed "ML family" of languages, and borrows a little from Haskell. Elm will be the only ML language we will feature in this book, so take the time to enjoy the many features it offers.

In this chapter we'll visit many of the ideas originating in the design and development of the ML languages, including **type variables** and extensive type inference. We'll also encounter pattern matching and destructuring. We'll even see a bit of Haskell-inspired syntax. Haskell programmers may find much of Elm quite familiar.

But Elm is not an ML or Haskell knock-off. It adds many new features of its own and fuses old and new ideas quite nicely. While Elm has moved on from its FRP roots, it now aims to supersede JavaScript for HTML applications on the client side by replacing JavaScript's callback-heavy architecture with a **subscription**-based approach to interactivity. Elm also differs from JavaScript by living very far to the "pure" side of the functional language spectrum. Data structures are persistent, like those of Clojure. You bind variables rather than assigning to them.

We'll begin our exploration of Elm with our three standard introductory programs, running in the browser, since Elm doesn't usually script standard input and standard output. We'll then look at base language elements including modules, functions, and types. We'll cover Elm's approach to type inference (inherited, of course, from its ancestors in the ML family).

We'll spend time with advanced types such as tagged unions and a particularly interesting record type with some neat syntax for its associated operations. We'll close with an overview of subscriptions and their role in interactive programs.

9.1 HELLO ELM

Let's get right to the traditional first program:

```
import Html exposing (text, ul, li)
import List exposing (map, concatMap, filterMap, repeat, range)

main =
    range 1 40 |> concatMap (\c ->
        range 1 (c-1) |> concatMap (\b ->
            range 1 (b-1) |> filterMap (\a ->
                if a*a + b*b == c*c then Just (a, b, c) else Nothing)))
        |> map (toString >> text >> repeat 1 >> li [])
        |> ul []
```

There are no for-loops in Elm (nor statements for that matter), so we generate our right-triangle triples via mapping and filtering. Rather than writing to standard output, we build an HTML unordered list to display in a web browser.

Despite its size, this little program highlights quite a few areas that benefit from some explanation:

- Function application does not require parentheses, so `f x` means $f(x)$. Unlike CoffeeScript, Elm's application is left-associative, so `f x y` means $f(x)(y)$.
- `x |> f` means $f(x)$. Similarly, `x |> f |> g |> h` means $h(g(f(x)))$.
- Anonymous functions employ a backslash and a thin arrow, so `\x -> e` denotes the function with parameter x and body e.
- `g >> f` is the function that applies g then f, i.e., `(g >> f)x` means $f(g(x))$.
- `concatMap` maps a list-producing function over a list, and flattens (concats) the result. This ensures that we pass a complete list of triples, rather than several lists of triples, to the test for inclusion in our output.
- `filterMap` maps an optional-producing function over a list and keeps only the present ("`Just`") values. Elm's optionals are called **maybes**. `If`-expressions always have both `then` and `else` parts, so we had to combine maybes with filtering in order to select the appropriate triples.
- The `ul` and `li` functions produce HTML `<ul>` and `<li>` elements. Functions in the `Html` module take a list of attributes and a list of child elements. For example, the Elm expression `div [id "notes"] [(text "hello"), (em [] [text "world"])]` yields the HTML `<div id="notes">hello<em>world</em></div>`.
- The function call `repeat n x` produces a list of n x's. So the expression `repeat 1` evaluates to a function that creates a one-element list containing its argument.
- Elm automatically renders the value bound to `main`.

Now let's generate permutations. We'll take our input from an HTML text box, so we'll write the application in two parts to separate the UI from the core computation. First we have a **module** that exposes a function returning the anagrams of a string in a list:

```
module Anagrams exposing (anagrams)

import String exposing (toList, fromList)
import List exposing (concatMap, map, foldr)

insertEverywhere : a -> List a -> List (List a)
insertEverywhere x xs =
    case xs of
      [] -> [[x]]
      (y::ys) -> (x::y::ys) :: map ((::)y) (insertEverywhere x ys)

permutations : List a -> List (List a)
permutations =
    foldr (concatMap << insertEverywhere) [[]]

anagrams : String -> List String
anagrams s =
    s |> toList |> permutations |> map fromList
```

This little module defines three functions but exports only the one named on the module declaration line. We've supplied **type annotations** for each of them. Though unnecessary, it's good practice to provide the annotations, as they add useful documentation and serve as a check against getting something wrong. Our `anagrams` function, of type `String -> List String` produces a list of anagrams from a given string. The `permutations` function produces a list of all permutations of a given list. Note the **type variables** in the function type. Elm's `List a` serves the same purpose as Java's `List<T>`, namely defining the type of lists whose elements all have the same type. And `insertEverywhere` produces a list of lists, each with a given element in a different place in a given list, such that all places are covered. Examples should help to clarify things a bit:

`anagrams "dog"` ⇒ `["dog","odg","ogd","dgo","gdo","god"]`

`permutations [1,2,3]` ⇒ `[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`

`insertEverywhere 1 [2,3]` ⇒ `[[1,2,3],[2,1,3],[2,3,1]]`

Our module illustrates a few characteristics of **functional programming**: there are no assignments, no mutable variables (we have only immutable function parameters), and no control flow other than function combination. It also introduces new items from the core library: function composition (`<<`), list prefixing (`::`, pronounced "cons" for historical reasons), and folding from the right (`foldr`). These functions behave as follows:

$(f$ `<<` $g)\ x \Rightarrow f(g(x))$

x `::` $[a_0, ..., a_n] \Rightarrow [x, a_0, ..., a_n]$

`foldr` $f\ x\ [a_1, a_2, a_3, a_4] \Rightarrow f(a_1, f(a_2, f(a_3, f(a_4, x))))$

We now turn to the main application, which runs in a web browser. It displays a textfield and responds to each change in the box's content by rendering all of the permutations directly on the page.

```
import Anagrams exposing (anagrams)
import Html exposing (Html, div, text, input, beginnerProgram)
import Html.Attributes exposing (placeholder, value, maxlength)
import Html.Events exposing (onInput)

type alias Model =
    String

type Message
    = ChangeTo String

main =
    Html.beginnerProgram { model = "", view = view, update = update }

update : Message -> Model -> Model
update (ChangeTo newModel) model =
    newModel

view : Model -> Html Message
view model =
    div [] <|
        (input
            [ placeholder "Text to anagram"
            , value model
            , maxlength 6
            , onInput ChangeTo
            ][]) :: List.map (\s -> div [] [ text s ]) (anagrams model)
```

The entire program may have been longer than you expected. Some of the length stems from having a large number of imports, but this is an indication of emphasizing modularity in the core libraries. Many languages have string and list functions built right in, but we need to explicitly import them in Elm. The HTML `<div>` and `<input>` elements, along with their associated attributes and events, have representations in Elm which we must also import. And the application itself follows the *Elm architecture*, a style of writing code made from:

- A model, representing our application's "state" (here just the string we wish to anagram),
- A view function, which takes in the current model and produces the HTML to render. The HTML may include elements that generate messages. In our case, we have an HTML `<input>` element that generates `oninput` events. We've arranged that these events send the message `ChangeTo` *s*, where *s* is the string in the textfield, to the `update` function.
- An update function, which receives a message and the current model, and produces the new model. In our app, the message tells us exactly what to do to: update the model with the string in the message.

Our traditional third program has been to count words from standard input. To keep things simple for now, we'll hardcode the source text. We'll leave it to you as an exercise to generalize the script with HTML textareas and file uploads.

```
import List exposing (map, foldl)
import Dict exposing (Dict, insert, get, toList)
import String exposing (toLower)
import Regex exposing (regex, find)
import Maybe exposing (withDefault)
import Html exposing (table, tr, td, text)
import Html.Attributes exposing (style)

message =
    "The fox. It jumped over/under...it did it did the Fox."

countWord : String -> Dict String Int -> Dict String Int
countWord word counts =
    insert word (withDefault 0 (get word counts) + 1) counts

tableCell data =
    td [] [text <| toString data]

main =
    find Regex.All (regex "[A-Za-z']+") message
    |> map (.match >> toLower)
    |> foldl countWord Dict.empty
    |> toList
    |> map (\(word, count) -> tr [] [tableCell word, tableCell count])
    |> table [style [("border", "1px solid black")]]
```

Once again, we've used quite a few of Elm's core modules. The script itself introduces few new linguistic features, but the following demand a little explanation:

- Extracting words uses Elm's regular expression matching, which returns matches in a list of Elm **records**. We'll discuss records later in the chapter.
- We're storing words and their counts in a **dictionary**, which Elm provides in the core `Dict` module. Dictionaries are persistent data structures, which we introduced in the last chapter. Invoking `get key dict` produces a `Maybe` value, and the function `withDefault` (from the `Maybe` module) operates on maybes. This is in contrast to Java, Python, Ruby and many other languages in which the dictionary itself must have special knowledge of missing keys. Java, in particular, has methods on the dictionary class named `getOrDefault` and `putIfAbsent`. Elm dictionaries need no such knowledge: they simply deliver maybes on lookup.
- Elm dictionaries require comparable keys, and are always processed in key order when operated on by functions such as `keys`, `values`, `toList`, and `foldl`.

9.2 THE BASICS

As a nearly pure functional language, Elm is quite small. So small, in fact, we've encountered most of the syntax in our opening example programs. It features five basic types: `Bool`, `Int`, `Float`, `Char`, and `String` and four mechanisms for defining new types:

- **Tuples**. The value `(7, "abc")` has type `(Float, String)`. Tuple types are Elm's product type. They can have any number of constituent types, including zero.[1]
- **Records**. The value `{ x = 3.5, y = 5.88, color = "red" }` has type `{ color : String, x : Float, y : Float }`. Note that the record type *includes* the field names; records are essentially tuples with labeled components.
- **Tagged Unions**. Values of a tagged union type (sometimes called a **discriminated sum type**) contain a tag together with a value from another type—or even just a tag. Elm's unions solve the same problem sum types solve, and can beautifully model enumerations, state machines, type hierarchies from the popular "object-oriented languages," and even lists and dictionaries. The requirement for including tags with values allows for code more readable and expressive than your basic sum type.
- **Functions**. The value `\x -> (x, x)` has type `Int -> (Int, Int)`.

We'll cover all of these mechanisms with example scripts[2] beginning with tuples. Our first example also illustrates the use of a **type alias**. An alias does not create a new type; rather, it simply provides a convenient name for a more complex type expression.

```
import SimpleAssert exposing (assertAll)

f : (Int, Int, Int) -> Int
f (a, _, c) =
    3 * c + a

type alias Vector = (Float, Float)

magnitude : Vector -> Float
magnitude v =
    let (x, y) = v in
        sqrt(x * x + y * y)

main = assertAll
    [ (6, 8) == (7 - 1, 4 * 2)      -- value semantics for ==
    , f (1, 2, 3) == 10             -- pattern match
    , magnitude (3, 4) == 5.0       -- destructuring
    , (\x -> (x, x)) 1 == (1, 1)    -- returning tuple from function
    ]
```

Tuples are accessed via pattern matching, which works both in passing the function argument and in `let` bindings. Note we said *the* function argument. Like its relatives in the ML family, function types have the form $T_1 \to T_2$ for types T_1 and T_2. Instead of thinking about "multiple parameters" and "multiple return types," think about accepting and returning tuple values.

But wait! None of the examples in the opening section of this chapter used tuples, though they appeared to take "multiple arguments." They looked quite different, like the example on the right below:

[1] The empty tuple type, `()`, is commonly called the **unit type**.

[2] As in all previous chapters, our pedagogical style is to illustrating language features with assertions. Elm doesn't have an assert expression, so we've rolled our own little module called `SimpleAssert` which we will cover momentarily.

```
plus: (Int, Int) -> Int
plus (x, y) =
    x + y
```

```
plus: Int -> Int -> Int
plus x y =
    x + y
```

The example on the left returns the sum of its two tuple components, but what of the example on the right? What is the type `Int -> Int -> Int`? The function type operator `->` is right-associative, so we actually have `Int -> (Int -> Int)`. The function accepts an integer and returns a function from integers to integers. And although it appears to take two parameters, the definition on the right simply sugars the direct "one parameter" definition that returns a function:

```
-- Sugared definition
plus: Int -> Int -> Int
plus x y =
    x + y
```

```
-- Desugared definition
plus: Int -> (Int -> Int)
plus x =
    \y -> x + y
```

The act of rewriting a function of type `(a,b)->c` into an equivalent of type `a->(b->c)` is called **currying**; the reverse process is called **uncurrying**. Curried functions allow something called **partial application**. For example, although we might normally write

```
plus 3 4
```

we can just as easily write

```
plus 3
```

to obtain the function that adds its argument to three. We are free to apply this function at some later time to obtain essentially a sum of two integers.

Elm's infix operators are actually defined as curried functions; for example, the type of the division operator (`/`) is `Float -> Float -> Float`. Although designed to appear between its operands, Elm allows an infix operator to appear in prefix position by enclosing it in parentheses; for example `(/) 5 8` evaluates to 0.625. This feature was not written solely to make Clojure programmers happy; rather it gives us a way to use the function as a value in its own right, as well as use the function resulting from applying the operator to just one operand. Study the two functions in the example below:

```
import SimpleAssert exposing (assertAll)

main = assertAll
    [ List.foldl (*) 1 [2, 3, 4, 5] == 120
    , List.map ((-) 20) [5, 12, 19] == [15, 8, 1]
    , List.map (flip (-) 5) [10, 20, 30] == [5, 15, 25]
    ]
```

Notice that `(-) 20` is the function "subtract from 20" while `flip (-) 5` is the function "subtract 5 from."

As a practical example of partial application involving infix operators, we offer our homegrown `SimpleAssert` module. Because Elm lacks an assertion expression, we built a module exposing a function that asserts the truth of every expression in a list. For failed assertions, we have admittedly abused the `crash` function from the `Debug` module. While not designed for testing (the `elm-test` package plays that role), `Debug.crash` is perhaps the simplest way to embed assertions directly inside an application. When an Elm script is translated to JavaScript, via `elm make prog.elm -output prog.js`, and run on the command line, `Debug.crash` will generate a decent error message and a non-zero process exit code:

```
module SimpleAssert exposing (assertAll)

import Html exposing (text)

assertAll assertions = text <| toString <|
    if List.all ((==) True) assertions
        then "All tests passed"
        else Debug.crash "Assertion failure"
```

Now to *define* an infix function, supply a function definition with the name in parentheses (as if it were a prefix function). Using functions as infixes introduce questions of precedence and associativity. In Elm, precedences are integers from 0 (lowest) to 9 (highest), and associativities are left, right, or none. The common infix functions from the packages `Basics` and `List` (from the Elm core library) follow.

Operator(s)	Prec	Assoc	Description
`<<`	9	R	composition: $(f \texttt{<<} g)x = f(g(x))$
`>>`	9	L	composition: $(f \texttt{>>} g)x = g(f(x))$
`^`	8	R	exponentiation
`* / // % rem`	7	L	multiply, divide, floor-divide, modulo, remainder
`+ -`	6	L	add, subtract
`:: ++`	5	L	cons, append
`< <= > >= = /=`	4	None	less, less or equal, greater, greater or equal, equal, not equal
`&&`	3	R	(short-circuit) logical and
`||`	2	R	(short-circuit) logical or
`<|`	0	R	application: $f \lhd x = f(x)$
`|>`	0	L	application: $x \rhd f = f(x)$

By default, custom operators have precedence 9 and are left-associative, but you can change these values. In the following script, we create two operators that compute $2x+y$, one with a high precedence (8) and one with low precedence (3). We use this operator in an expression with multiplication (precedence level 7) to show the effect of the different precedences.

```
import SimpleAssert exposing (assertAll)

(<-*->): Int -> Int -> Int
(<-*->) x y = 2 * x + y

(>*<): Int -> Int -> Int
(>*<) x y = 2 * x + y

infix 8 <-*->
infix 2 >*<

main = assertAll
    [ (8 * 3 <-*-> 5) == 88        -- 8*(2*3+5)
    , (8 * 3 >*< 5) == 53          -- 2*(8*2)+5
    ]
```

We've chosen to make our new operators non-associative; to set left or right associativity we would use `infixl` or `infixr`, respectively.

Note that the standard operators (addition, multiplication, less-than, etc.) are not wired in to the language but are simply defined as plain Elm functions in the standard library. Indeed, examining the source code for the Elm compiler we see:

```
(/) : Float -> Float -> Float
(/) =
  Native.Basics.floatDiv

infixl 7 /
```

The implementation of floating point division will execute fast since the Elm compiler can access lower-level code through its `Native.Basics` module,[3] but the point is that you won't see basic operators appearing in the syntax definition for the language; they are simply functions.

A final note on precedence: prefix function application *always* binds tighter than infix application. While very simple, the rule does lead to a few gotchas that the programmer coming to Elm from other languages might want to keep in mind:

```
negate 3 ^ 4  parses as  (negate 3) ^ 4
show x * y    parses as  (show x) * y + z
```

You can, of course, use parentheses as expected, or make use of the function application operator `<|` from precedence level 0:

```
negate <| 3 ^ 4  parses as  negate(3 ^ 4)
show <| x * y    parses as  show(x * y + z)
```

Surprisingly, that covers most of the basics. In the remainder of the chapter we will cover two fascinating items that are characteristic of the ML family, a powerful type inference facility and tagged unions; one characteristic of modern ML family languages, extensible records; and Elm's approach to interactivity (subscriptions and commands).

9.3 TYPE INFERENCE

Elm performs static typing: it checks the types of expressions at compile time and refuses to execute programs that may have type errors. It is the second such language in this text; Java was the first.[4] In order to typecheck prior to execution, the compiler must be given information about the intended types of expressions or be smart enough to *infer* types. Java is capable of relatively little **type inference**, requiring type annotations on nearly all variables, method parameters and return values, fields in classes and interfaces, etc. Elm

[3] Needless to say, most Elm programs will never touch `Native.Basics`, or its cousins such as `Native.Utils`, `Native.String`, and `Native.Time`.

[4] Though technically, you may recall, Java does leave a tiny amount of type checking to runtime in some cases.

can infer the type of nearly every expression. The three other statically-typed languages[5] in this book, Go, Swift, and Rust, fall somewhere in the middle of this spectrum.

Elm's type inferencing ability, like that of all ML-family languages, has its theoretical roots in the **Hindley-Milner type sytem** [135, 86, 22]. Elm's compiler computes (or deduces), the least general type of any expression not already explicitly annotated. Let's start with a very simple example. Consider the Elm function:

```
mystery x s =
    String.repeat (round (sqrt x)) s
```

None of the identifiers have type annotations, but Elm can infer them. How? Roughly, we proceed as follows:

1. Working from left-to-right, we start by assigning **type variables**: `x` gets type α, `s` gets β, and therefore `mystery` gets $\alpha \to \beta \to \gamma$.
2. We know the type of `repeat` is `Int -> String -> String`, so the type of `round(sqrt x)` must be `Int`.
3. We know the type of `round` is `Float -> Int`, so the type of `sqrt x` must be `Float`.
4. We know the type of `sqrt` is `Float -> Float`, so the type of `x` must be `Float`. Since we had previously assigned the type variable α to `x` we can now **unify** α and `Float`.[6]
5. The second argument of `repeat` (`s`) must be a string, so we unify β with `String`.
6. Because `repeat` produces a string, we unify γ with `String`.
7. Therefore, the type of `mystery` is `Float -> String -> String`.

Let's look at a more interesting case:

```
mystery (x, y) z =
    (y / 2.0, z, [x, x, x])
```

Here we begin by mapping the type variable α to `x`, β to `y`, γ to `z`, and thus $(\alpha, \beta) \to \gamma \to \delta$ to `mystery`. Because we know the type of float division (`/`) we can infer that y has type `Float` and unify β with `Float`. We can't get any new information for `z` or `x`, but we can determine the return type of the function: (`Float`, γ, `List` α). While it is customary to use Greek letters for type variables when discussing type inference, Elm sticks to lowercase letters for its type variables, and reports the type of `mystery` as `(a, Float) -> b -> (Float, b, List a)`.

Generally, type variables are allowed to be unified with any type, but Elm has three[7] special type variables that unify *only* with particular types:

[5] Colloquially, we say a *language* is statically-typed if it performs the majority of type checking prior to execution.

[6] In this case unification appears to be simply instantiation of a type variable with a concrete type; however, in more complex cases, we might discover that two distinct type variables must refer to the same type, and thus need to be "unified" into one variable.

[7] Perhaps strangely, a fourth such variable, `compappend`, arises internally when type checking expressions that must both be comparable and appendable. You may see it "leaked" into your own world when the compiler tells you that it has inferred the type of `f` in `f x y = if (x < y) then (x ++ y) else (y ++ x)` to be `compappend -> compappend -> compappend`; however, the compiler will not respect your use of this variable. You will carry out further explorations in the exercises.

- `number`, which unifies only with `Int` and `Float`, to support functions such as `(+)`, `(-)`, and `(*)`;
- `comparable`, which unifies only with `Int`, `Float`, `Char`, `String`, tuple types containing comparables, and list types containing comparables, to support `(<)`, `(<=)`, `(>)`, `(>=)`, `min`, `max`, and `compare`; and
- `appendable`, which unifies only with `String`, `Text`, and list types, to support `++`.

These special type variables can be seen in several function definitions in Elm's core package:

```
-- In the Basics module
(>) : comparable -> comparable -> Bool
abs : number -> number
(++) : appendable -> appendable -> appendable
min: comparable -> comparable -> comparable
clamp : number -> number -> number -> number

-- In the List module
sum : List number -> number
maximum : List comparable -> Maybe comparable
sort : List comparable -> List comparable
sortBy : (a -> comparable) -> List a -> List a

-- In the Set module:
insert : comparable -> Set comparable -> Set comparable
member : comparable -> Set comparable -> Bool
union : Set comparable -> Set comparable -> Set comparable
fromList : List comparable -> Set comparable

-- In the Dict module:
insert : comparable -> v -> Dict comparable v -> Dict comparable v
member : comparable -> Dict comparable v -> Bool
get : comparable -> Dict comparable v -> Maybe v
```

Whenever one of the special type variables appears more than once in a type expression, all occurrences must unify together, so `(comparable, comparable)` does *not* unify with `(Int, String)`. However, the latter does unify with `(comparable, comparable1)` since the variables `comparable` and `comparable1` are distinct.[8]

It is tempting to think of Elm's special type variables as interfaces, but this view is completely wrong. Defining your own type and overloading `(++)` does *not* make your type `appendable`, nor does overloading any particular set of functions ever make your own new type `comparable`. So don't go looking to place values of your own types in sets or use them as dictionary keys.

However, you *can* do such things in Haskell, another member of the ML family of languages. Haskell introduced the notion of **type classes**. Haskell types can belong to zero or more type classes. Over a dozen of these classes exist, including `Eq`, `Ord`, `Enum`, `Integral`,

[8] Technically, Elm provides an infinite number of special type variables made by suffixing `number`, `comparable`, or appendable. As you may have guessed, `number1` and `number2` behave as `number`.

`Real`, `Fractional`, `Floating`, `Read`, `Show`, and `Monad`. Each class specifies certain operations (equality for `Eq`, comparison for `Ord`, conversion to strings for `Show`, etc.) and you can, when defining your own types, specify which type classes they belong to, provided of course, that you provide implementations for the functions required by the type class.

Other ML-family languages take a different approach. Rather than type classes, Standard ML and OCaml feature relatively sophisticated module systems, which ultimately do allow one to carry out the same kind of generic programming supported by Haskell's type classes and Java's interfaces. As a young language, Elm may well evolve to gain type classes or other features, with academic-sounding names such as higher-kinded polymorphism [49, Ch. 18] and Rank-N types [52]. You may find Elm's choice of fixed special type variables (in lieu of type classes or SML-style modules) limiting, or you may value its simplicity and pragmatism. Either way, the use of type variables and type inference based on Hindley-Milner types give Elm a great deal of expressivity. We now turn to two of the language's powerful mechanisms for defining new types: its tagged unions and records.

9.4 TAGGED UNIONS

You've undoubtably run across situations in which values of a type looked different: responses to a query could be successful (with an answer) or erroneous (with an error code); trees can be empty or nonempty; users can be logged in (with credentials) or anonymous; shapes can be circles or rectangles; entities in a program representation could be numbers, identifiers, or operators. Java, Python, Ruby and others model these situations via subtyping. Elm uses sum types in which the variants in the sum are each labeled, or tagged. In the literature, you'll see these types called **tagged unions**, **discriminated sums**, or **disjoint sums**.

We'll explore via examples. First we see how to define a union type and note that the tags play the role of constructor functions:

```
import SimpleAssert exposing (assertAll)

type Response a
    = Ok a
    | Error String

squareRoot x =
    if x < 0 then Error "negative" else Ok (sqrt x)

main = assertAll
    [ squareRoot 9 == Ok 3
    , squareRoot 100.0 == Ok 10.0
    , squareRoot -4 == Error "negative"
    ]
```

Next, we build a tree type and show that functions processing unions use pattern matching on the variants. Our trees are either empty, or a node containing a value and a list of trees as its children.

```
import SimpleAssert exposing (assertAll)
import List exposing (map, foldr, sum, all)

type Tree a
    = Empty
    | Node a (List (Tree a))

size : Tree a -> Int
size t =
    case t of
        Empty ->
            0
        Node _ children ->
            1 + (sum <| map size children)

str : Tree a -> String
str t =
    case t of
        Empty ->
            "()"
        Node value children ->
            "(" ++ toString value
                ++ (foldr (++) "" <| map str children)
                ++ ")"

main =
    let
        t0 = Empty
        t1 = Node "dog" []
        t2 = Node 17 [Node 31 [], Node 53 []]
        t3 = Node 41 [t2, t2, Empty, Node 3 []]
    in
        assertAll
            [ size t0 == 0
            , size t1 == 1
            , size t3 == 8
            , str t0 == "()"
            , str t1 == "(\"dog\")"
            , str t2 == "(17(31)(53))"
            , str t3 == "(41(17(31)(53))(17(31)(53))()(3))"
            ]
```

Now here's a representation of a tiny "class hierarchy" of shapes:

```
module Shapes exposing (..)

type Shape
    = Circle Float
    | Rectangle Float Float
```

```
area : Shape -> Float
area s =
    case s of
        Circle r ->
            pi * r * r
        Rectangle h w ->
            h * w

perimeter : Shape -> Float
perimeter s =
    case s of
        Circle r ->
            2 * pi * r
        Rectangle h w ->
            2 * (h + w)
```

```
import SimpleAssert exposing (assertAll)
import Shapes exposing (Shape(Circle, Rectangle), area, perimeter)

main = assertAll
    [ area (Circle 10) == (100 * pi)
    , perimeter (Circle 10) == (20 * pi)
    , area (Rectangle 2 8) == 16
    , perimeter (Rectangle 2 8) == 20
    ]
```

Elm's `List` and `Maybe` types are tagged unions, too. Because a list is either empty, or an element (the head) connected to a list (its tail), we have two list constructors: `[]` for the empty list and `(::)` for lists with a head and tail. We'll illustrate pattern matching on lists with a very simple example:

```
import SimpleAssert exposing (assertAll)

double : List a -> List a
double list =
    case list of
        [] -> []
        x :: xs -> x :: x :: double xs

main = assertAll
    [ double [] == []
    , double [5, 5] == [5, 5, 5, 5]
    , double [2, 5, 3] == [2, 2, 5, 5, 3, 3]
    ]
```

A `Maybe` either wraps a value or it doesn't. `Maybe` is Elm's option type. We first presented option types in the Java chapter as alternative to the "Billion Dollar Mistake," a.k.a. null references. Elm has no null references, so you'll see maybes quite often. Set and dictionary

lookup, retrieving the head, tail, minimum, and maximum of a list all employ maybes. The `Maybe` module defines the type as a tagged union:

```
type Maybe a  = Just a  | Nothing
```

While you may need to occasionally pattern match on a maybe object, you will more likely employ one of the convenience methods from the `Maybe` module, such as:

- `withDefault`, which unwraps a maybe, yielding a default if nothing is wrapped. We used this earlier in the chapter to use a dictionary to count words: `insert word (withDefault 0 (get word counts) + 1) counts`.
- `map`, which safely operates on the wrapped value; e.g., `map abs (Just -8) == (Just 8)` while `map abs Nothing == Nothing`.
- `andThen`, which safely performs a maybe-producing-callback on the wrapped value only if present, e.g., `([1,2,3,4] |> tail) |> andThen tail == Just [3,4]`. Use `andThen` when you need to chain callbacks but a `Nothing` can be produced at any point on the chain.

9.5 RECORDS

Just like tagged unions generalize sum types by labeling variants, Elm's **records** generalize product types (tuples) by labeling components. Let's take a look:

```
import SimpleAssert exposing (assertAll)

boss = {name = "Alice", salary = 200000 }
worker = {name = "Bob", salary = 50000, supervisor = boss}
newWorker = {worker | name = "Carol" }

payrollTax : {a | salary : Float} -> Float
payrollTax {salary} =
    salary * 0.15

main = assertAll
    [ .name boss == "Alice"
    , worker.supervisor == boss
    , payrollTax worker == 7500.0
    , newWorker.supervisor == boss
    ]
```

We've created three records: the first of type `{ name : String, salary : number }`, the second with type `{ name : String, salary : number, supervisor : { name : String, salary : number } }`, and the third with the same type as the second. We defined our third record, `newWorker`, to be just like `worker` except with the name `"Carol"` (instead of `"Bob"`). Next we defined a function to compute a 15% payroll tax, illustrating pattern matching for record arguments. Read the type of the function, `{a | salary : Float} -> Float` as "a function that accepts any record that has a field named `salary` of type `Float`, and produces a `Float`." That's pretty flexible. We never introduced any type names at all, never subclassed employees into workers and managers, and yet Elm ensures the entire program is fully type-safe.

You may find Elm's **type aliases** useful in describing records. An alias is *not* a new type, but rather an abbreviation for a type:

```
type alias Person =
    { name : String, id : Int }

type alias Widget =
    { name : String, id : Int }

type PrimaryColor
    = Red
    | Blue
    | Green

type alias PixelColor =
    PrimaryColor

p : Person
p = { name = "Alice", id = 239 }

w : Widget
w = p           -- Disappointing perhaps, but legal

g : PixelColor
g = Green
```

While tagged union definitions (e.g., `PrimaryColor` in the example script), *do* introduce new, distinct, named types, we can only name record types via aliasing. In the lingo of programming languages, record types live in a world of **structural typing** rather than **nominative typing**. Only the record's structure—the names and types of its fields—is used to determine its type. We can, however, use tagged unions with a single variant to make the distinction:[9]

```
type Person
    = Person { name : String, id : Int }
type Widget
    = Widget { name : String, id : Int }

p : Person
p = Person { name = "Alice", id = 239 }
-- w : Widget; w = p -- would be a syntax error now.
```

We can use type aliasing to describe a record type that need have only certain fields:

```
import SimpleAssert exposing (assertAll)

type alias Positioned a =
    { a | x : Float, y : Float, z : Float }
```

[9]This technique also applies when you need to avoid recursive records, which you'll have an opportunity to explore in the end-of-chapter exercises.

```
move : Positioned a -> Float -> Float -> Float -> Positioned a
move object dx dy dz =
    { object | x = object.x + dx, y = object.y + dy, z = object.z + dz}

main =
    let
        robotAtStart = { name = "Mari", x = 3, y = 8, z = -2 }
        robotAtEnd = move robotAtStart 7 -2 9
        expectedEnd = { name = "Mari", x = 10, y = 6, z = 7 }
    in
        assertAll [ robotAtEnd == expectedEnd ]
```

We just moved a robot. But we can, in fact, move *anything* that has `x`, `y`, and `z` fields, without the need to include some kind of position field within the object, or derive our object from a position object or class.

9.6 EFFECTS

One of the more useful dimensions on which to characterize and compare programming languages is the extent to which the language's design, and its culture, encourage, discourage, contain, or try to eliminate **side-effects**. Roughly, a side effect occurs when a function changes a program's state in a way other than producing an explicit return value. Side-effects, and side-causes—entities manipulated by a function other than its explicit parameters [69]—are associated with mutable variables, files, and time sensitive features such as user interaction and animation. Languages we think of as *functional* go to great lengths to discourage and contain side-effects. Clojure allows controlled mutation via vars, refs, and agents, and even lets us drop into Java. Relatives of Elm, such as Standard ML and OCaml, provide references that wrap mutable values.

Some languages, like Haskell, work to contain even I/O effects. Think back to the way operations on persistent data structures never mutate a structure but rather produce a new structure. If we wrap (snapshots of) our files and variable values into a "state" object, then a function may accept a state as an argument and produce a new state as a result. In general, an input state can also contain *event histories* such as key presses, clicks or gestures on an input device, canvas resizings, or the passing of time. Older versions of Elm directly supported these history sequences, calling them **signals**. The signal `Mouse.position`, of type `Signal (Int, Int)`, for example, would have a value such as:

```
(3,4) (7,5) (11,4) (13,3) (19,6)
```

Signals, like all sequences, could be mapped (to perform an operation on each event), filtered (to ignore events of no concern), merged (to handle events from multiple sources), and folded (to accumulate state). To sketch on a canvas in the early days of Elm, we folded a line segment generation function over the mouse position signal![10] Dozens of other signals like `Keyboard.Arrows` and `Window.Dimensions` lived in the standard library. Manipulating asynchronous streams with higher-order functions is known as **functional reactive programming**, or FRP.

[10]The folding operation was called neither `foldl`, folding from the left, nor `foldr`, folding from the right, but `foldp`, folding from the past.

Elm has left its FRP heritage behind and has removed signals from its core libraries, replacing them with two kinds of **managed effects**: **commands** and **subscriptions**. A command object requests an action: "fetch geolocation data" or "send this message over a websocket." A subscription expresses interest in an event: "tell me when the mouse moves" or "tell me when an amout of time has passed."

Effects are managed within programs. A program is a record with four components:

1. `init: (model, Cmd msg)` is the initial model, or state, of the application, together with a command to launch on startup.
2. `view: model -> Html msg` produces HTML for a given model. The HTML may generate messages of the type *msg*.
3. `subscriptions: model -> Sub msg` produces, for the given model, a subscription to sources that can produce messages (of type *msg*) that will be routed to the `update` function.
4. `update: msg -> model -> (model, Cmd msg)` produces, given a model and message, the new model (next state) and a command to invoke.

Elm provides functions for creating subscriptions on windows, keyboards, mice, web sockets, geolocation changes, timers, and the browser's rendering refresh cycle. You'll also find functions for generating commands to fire random numbers and perform various asynchronous tasks.

Let's start with a simple example. We'll render an image at the current mouse position: the JavaScript logo when the mouse button is up and the Elm logo when the mouse button is held down. The model is the mouse position together with the name of the image. The view function renders the image at the given position. We subscribe to mouse movements and the up/down state of the mouse button. We won't use any commands in this example.

```
import Html exposing (Html, program, img)
import Html.Attributes exposing (style, src)
import Mouse

main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }

type alias Model =
  { x : Int
  , y : Int
  , image : String
  }

type Msg
  = Down
  | Up
  | MoveTo Int Int
```

```
init : (Model, Cmd Msg)
init =
  ({x = 0, y = 0, image = "js"}, Cmd.none)

subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.batch
    [ Mouse.moves (\{x, y} -> MoveTo x y)
    , Mouse.downs (\_ -> Down)
    , Mouse.ups (\_ -> Up)
    ]

update: Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Down ->
      ({model | image = "elm"}, Cmd.none)
    Up ->
      ({model | image = "js"}, Cmd.none)
    MoveTo x y ->
      ({model | x = x, y = y}, Cmd.none)

view: Model -> Html Msg
view model =
  img
    [ style
      [ ("position", "absolute")
      , ("left", toString (model.x+2) ++ "px")
      , ("top", toString (model.y+2) ++ "px")
      ]
    , src (model.image ++ "-logo.png")
    ]
    []
```

Following the convention outlined in the Elm Architecture, we've defined all possible messages in a tagged union type. In our subscriptions function, we turn the interesting system-generated events (mouse down, mouse up, mouse move) into message objects that Elm will route to our update function, where we generate the new model. Our view function renders the model (the correct image at the correct position) when necessary.

We'll close with an illustration of time effects, and throw in a bit of Elm's support for Scalable Vector Graphics, or SVG. Our next script counts down from 10 to 0. It subscribes to the current time via `Time.every second`, which gives us notifications every second and sends a `Tick` message, together with the current epoch time[11] to our update function. We're not interested in the real time, but we do want to decrement a counter. When the counter reaches zero, we'll "stop subscribing" by setting our subscriptions to none.

[11] The number of seconds elapsed since midnight, January 1, 1970 UTC, or 1970-01-01T00:00Z.

```
import Html exposing (Html, program)
import Svg exposing (svg, circle, text_, text)
import Svg.Attributes exposing (..)
import Time exposing (Time, second)

main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }

type alias Model = Int

type Msg
    = Tick Time

init : (Model, Cmd Msg)
init =
    (10, Cmd.none)

subscriptions : Model -> Sub Msg
subscriptions model =
    if model <= 0 then
        Sub.none
    else
        Time.every second Tick

update : Msg -> Model -> (Model, Cmd Msg)
update action model =
    case action of
        Tick _ ->
            (model - 1, Cmd.none)

view : Model -> Html Msg
view model =
    svg [ viewBox "0 0 100 100", width "400px" ]
        [ circle [ cx "50", cy "50", r "45", fill "#A88428" ] []
        , text_ [x "50", y "50", fontSize "64", textAnchor "middle",
            dominantBaseline "central"] [text <| toString model]
        ]
```

Our examples covered only the basic outline of the Elm architecture with keyboard, mouse, and time subscriptions. You will want to continue your study by looking into animation frames, web sockets, asynchronous HTTP requests, HTML local storage, browser histories, geolocation tracking, and other platform services.

9.7 ELM WRAP UP

In this chapter we briefly looked at Elm. We learned that:

- Elm is a nearly-pure functional programming language from the ML family of languages, with a Haskell-inspired syntax.
- Elm applications generally run in a web browser. Production-quality Elm apps will be organized around the *Elm Architecture*, a convention for writing programs centered around models, updates, views, and subscriptions.
- Like Clojure, Elm greatly facilitates functional programming: functions are first class values, data structures are immutable and persistent, and there is no assignment statement.
- The basic types are `Int`, `Bool`, `Float`, `Char`, and `String`. New types are created via functions, tuples, records, and tagged unions.
- Lists and options (known as maybes) are defined as tagged unions.
- Curried functions are the norm.
- You can create your own infix operators in Elm, defining precedence and associativity.
- Unary operators always bind tighter than binary operators.
- Elm type inference is based on the Hindley-Milner type system. Type annotations are practically never required.
- Elm (currently) has no type classes, but has a few special variables that approximate a type class facility.
- Tagged unions and record serve as labeled sum and product types, respectively.
- Elm was once a language that facilitated functional reactive programming. It featured signals, sequences of values generated over time in response to events, such as keyboard or mouse events, or the passage of time. Signals would be mapped, filtered, merged, and especially folded. Folding over signals was the manner in which application state could be accumulated or stepped forward.
- Modern applications subscribe to event sources. The Elm Architecture ensures that subscribed events are turned into customized messages routed to a centralized update function to step the state of the application forward.

To continue your study of Elm beyond the introductory material of this chapter, you may wish to find and research the following:

- **Language features not covered in this chapter**. Tasks, commands, JavaScript interoperation, caching of effect managers.
- **Open source projects using Elm**. Studying, and contributing to, open source projects is an excellent way to improve your proficiency in any language. You may enjoy the following projects written in Elm: Sketch-n-sketch (`https://github.com/ravichugh/sketch-n-sketch`), Gipher (`https://github.com/matthieu-beteille/gipher`), Hop (`https://github.com/sporto/hop`), and Elm WebGL (`https://github.com/elm-community/elm-webgl`).
- **Reference manuals, tutorials, and books**. Elm's home page is `http://elm-lang.org/`, from which prominent links will take you to documentation, a playground,

community resources, and the official blog. The portal *Awesome Elm*, at `https://github.com/isRuslan/awesome-elm`, contains an up-to-date curated set of links to articles, examples, videos, tutorials, and many other resources.

EXERCISES

Now it's your turn. Continue exploring Elm with the activities and research questions below, and feel free to branch out on your own.

9.1 Try out several of the examples at `http://elm-lang.org/try`.

9.2 Install Elm on your own machine. Find the REPL. Experiment with various kinds of expressions, including records. Generate a few type errors.

9.3 We've not provided type annotations for `main` in our examples. Find out what types are allowed.

9.4 What is the type of the Elm expression `\f g x y. f(g x)y`?

9.5 What are the types of the operators `<|`, `|>`, `<<` and `>>`?

9.6 Trace out the evaluation of the expression `insertEverywhere 5 [1,2,3]` on pencil and paper. The function `insertEverywhere` is defined in the anagrams module in the opening section of this chapter.

9.7 Suppose you wished to fold the `++` operator over a list of lists. Does it matter if you fold from the left or the right? Why or why not?

9.8 If you are acquainted with HTML and CSS, stylize the anagrams web application from earlier in the chapter with a modern look and feel. This exercise will give you some practice with Elm's `Html` library.

9.9 Research some of the scenarios in which the unit type `()` is used in practice.

9.10 What is the curried version of the function `\(x,(y,z)) -> z - y * x`?

9.11 See if you can redefine the precedence of the `+` operator to have higher precedence than multiplication. Were you able to do so?

9.12 Locate a description of the function known as the Y Combinator. Can this function be defined in Elm? Why or why not? Can it be defined in Clojure? Why or why not?

9.13 Read an article on the Hindley-Milner type system and give an explanation of unification.

9.14 There is a chance that by the time you read this chapter, Elm may have gained type classes. If so, create your own comparable type and demonstrate that values of this type can be sorted. If type classes have not been introduced into Elm, research the online discussion in the community and summarize the arguments for and against their inclusion in the language.

9.15 (Mini-project) Build your own set type backed by a binary search tree defined by a tagged union. Include unit tests.

9.16 Define a tagged union for a user data type, where users can be anonymous "guests" or logged-in users with names and credentials.

9.17 Do a side by side comparison of Java optionals and Elm maybes. Be as extensive as possible.

9.18 In this chapter, we saw a `Shape` type defined as a tagged union of circles and rectangles, with separate area and perimeter functions performing pattern matching to perform the correct computation. Explain how this design illustrates the expression problem. Research approaches to solving the expression problem in Elm.

9.19 Explain why the function `andThen` is needed to take the tail of the tail of a list instead of just using composition (i.e., `tail << tail`).

9.20 Given

```
supervisor: Person -> Maybe Person
supervisor p =
    case p of
        Alice -> Nothing
        Bob -> Just Alice
        Chi -> Just Alice
        Dinh -> Just Bob
        Emmy -> Just Dinh
        Faye -> Just Emmy
        Guo -> Just Chi
```

Write the function `bossOfBossOfBoss: Person -> Maybe Person` to return the supervisor of the supervisor of the supervisor of its person argument, or `Nothing` if appropriate. Your function should return `Just Alice` for argument `Emmy`, and `Nothing` for `Chi`.

9.21 Define type aliases called `Percentage` (aliasing the `Float` type), and `Employee` aliasing records containing, among other fields, a name field and a salary field. Write a function called `giveRaise` of type `Person -> Percentage -> Person`.

9.22 Are Elm's records are value types or reference types? Does this question even make sense in Elm? Why or why not?

9.23 Implement a "recursive record" in Elm. Specifically, create a representation of a type for an employee, containing a name, a date of hire, and a supervisor (who is also an employee).

9.24 Learn about the Haskell language. How does it perform I/O? Does it admit that file manipulation inherently involves side-effects, or is this language able to completely contain these effects within pure functions? To help you in your research, read Kris Jenkins's two-part series on functional programming [69] and the section on Basic Input/Output in the Haskell Language Report [96].

9.25 Our logo dragging example does not begin well. A logo is rendered in the upper-left corner of the display area and jumps to the current mouse position when the mouse is first moved. Rewrite the example so that this does not happen.

9.26 Implement a sketching program. Experiment with Elm's graphics capabilities such as changing the thickness and colors of the lines.

9.27 Create a simple game in which a player navigates an avatar through various obstacles.

9.28 How is Elm's use of subscriptions both like and unlike traditional callback-style interactive programming? How is it both like and unlike the use of promises?