Exploring ReasonML

and functional programming

Dr. Axel Rauschmayer

Exploring ReasonML

Dr. Axel Rauschmayer

2020

reasonmlhub.com

Contents

Part I

Background

Chapter 1

About this book

1.1 Questions and answers about this book

- What am I going to learn?
  - This book teaches the programming language *ReasonML* by Facebook.
  - It is also an introduction to functional programming. Especially people familiar with C-style languages (Java, JavaScript, C#, etc.) will profit from ReasonML's familiar syntax.
- Is there any required knowledge?
  - You should know how to program, e.g. in a mainstream language such as Java, JavaScript, C#, Python, C/C++, PHP, Ruby, Go, etc.
- How can I get started a quickly as possible?
  - Read the whole book in order, skip chapters and sections marked as "advanced".
- Does this book cover all of ReasonML?
  - This book explains the language and functional programming. It also gives tips for using the standard library.
  - It does not cover ReasonReact and JavaScript interop. Chapter "What to read next?" points to information on those topics.

1.2 Warning: This book is outdated

Sadly, after the initial version, I couldn't afford to keep this book updated. It describes ReasonML as of 2018.

1.3 About the cover

Image by courtesy of the National Gallery of Art:

- "A Dromedary" (1551–1572) by Georg Mattheus. Woodcut on laid paper.
- URL: https://www.nga.gov/collection/art-object-page.153950.html

Chapter 2

What is ReasonML?

This chapter gives a brief high-level explanation of Facebook's new programming language, ReasonML.

2.1 What is ReasonML?

ReasonML is a new object-functional programming language created at Facebook. In essence, it is a new C-like syntax for the programming language OCaml. The new syntax is intended to make interoperation with JavaScript and adoption by JavaScript programmers easier. Additionally, it removes idiosyncrasies of OCaml's syntax. ReasonML also supports JSX (the syntax for HTML templates inside JavaScript used by Facebook's React framework). Due to ReasonML being based on OCaml, many people use the two names interchangeably. The following diagram shows how ReasonML fits into the OCaml ecosystem.

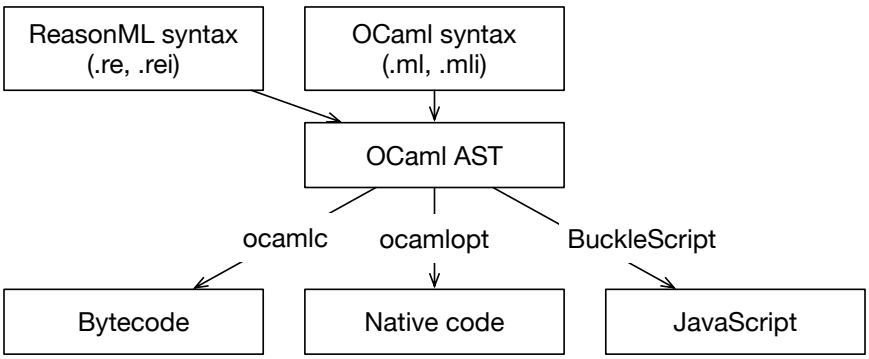

Figure 2.1: This is how ReasonML fits into the OCaml ecosystem.

At the moment, ReasonML's default compilation target is JavaScript (browsers and Node.js).

This is what ReasonML code looks like:

```
type color = Red | Green | Blue;

let stringOfColor = (c) =>
  switch (c) {
  | Red => "Red"
  | Green => "Green"
  | Blue => "Blue"
  };
```

Several things are notable:

- Many elements of the syntax are borrowed from JavaScript. For example:
  - The name `switch` (OCaml: `match`)
  - The syntax '(x) => ···' for functions
  - Semicolons
- Other elements are typical for a functional programming language. For example:
  - `color` is a *variant type*
  - `switch` performs pattern matching
- No type annotations were needed (e.g. the parameter `c` of `stringOfColor` does not have one).

2.2 The benefits of OCaml

ReasonML's foundation, OCaml, brings the following benefits:

- It is an established language (created in 1996) that has proven itself in many projects. Facebook itself is using it in several projects (e.g. Flow).
- Its core is a functional programming language with a full-featured type system. But it also supports object-orientation and mutable state.
- It can be compiled to either bytecode, fast native code or JavaScript.
- Compilation to JavaScript is fast. Quoting the blog post "Messenger.com Now 50% Converted to Reason":

  > Full rebuild of the Reason part of the codebase is ~2s (a few hundreds of files), incremental build (the norm) is <100ms on average. The BuckleScript author estimates that the build system should scale to a few hundred thousands files in the current condition.

2.3 Improving OCaml

The ReasonML team also aims to improve the OCaml ecosystem:

- Better tooling (testing, documentation, editor support, etc.).
- Better interoperation with JavaScript. The ReasonML-to-JavaScript compiler is already very fast and produces relatively readable code.
- Better standard library (there is a fair amount of competition in this space in OCaml, without a clear winner).

2.4 Conclusion

ReasonML feels much like what you'd get if you cleaned up JavaScript and turned it into a statically typed functional programming language. I'm ambivalent about JSX in ReasonML – it has pros and cons. I'm glad that ReasonML doesn't reinvent the wheel and is strictly based on the established OCaml.

OCaml's pragmatism means that you don't get some of the more fancy functional features (that, e.g., Haskell has), but it also leads to fast compilation, efficient code and decent error messages.

Chapter 3

Getting started with ReasonML

In this chapter, I give tips for getting started with the programming language ReasonML.

3.1 Installation

There are two things to install:

- `bs-platform`: Installs BuckleScript and enables you to compile ReasonML to JavaScript. The installation is described in the ReasonML docs.
- `reason-cli`: Needed to support ReasonML in editors, but also contains various tools, including the interactive ReasonML command line `rtop`. The installation is described in the ReasonML docs. Editor support is provided by two parts:
  - On one hand, a so-called language server provides services for working with ReasonML code.
  - On the other hand, editor plugins and similar extension mechanisms communicate with the server to provide the actual support.

3.2 Quickly trying out ReasonML

3.2.1 The ReasonML online playground

The ReasonML website contains an online playground that is very useful for seeing how the language works and what the corresponding JavaScript and OCaml code is. It can also convert from OCaml to ReasonML (more on that later).

The playground's examples give you a first taste of the language.

3.2.2 `rtop`, the interactive ReasonML command line

`rtop` is an interactive command line for ReasonML and started via `rtop` from a shell. Once it runs, interacting with it looks as follows.

```
Reason # 3 + 4;
- : int = 7
```

You can already see that everything has a static type in ReasonML. Don't forget the semicolon at the end – it triggers evaluation! You can quit `rtop` via Ctrl-D or via `#quit;`

3.3 Template projects

There are two template projects to get you started. They are created via `bsb` (which is part of `bs-platform`):

- Node.js code:

  ```
  bsb -init my-first-app -theme basic-reason
  ```

- Web development (React):

  ```
  bsb -init my-react-app -theme react
  ```

3.4 Important tip: converting OCaml to ReasonML

Given that most material relevant for ReasonML uses OCaml's syntax, it's very useful to be able to convert from OCaml's syntax to ReasonML's. There are two ways of doing so:

- The online playground "Try Reason".
- The tool `refmt` ("ReasonML format") that is part of `reason-cli`. Get documentation via `refmt --help`

Chapter 4

What is planned for ReasonML?

This chapter takes a brief look of a few key pieces of ReasonML that are still being worked on:

- Better support for writing asynchronous code that is compatible with JavaScript Promises.
  - One option is to provide special syntax (see issue on GitHub).
  - Another option is to add support to OCaml's concurrency library Lwt (it's one of the branches).
- Better support for polymorphism. At the moment, different types mean different function or operator names for ReasonML (for example, there is `+` for ints and `+.` for floats). Haskell has type classes to solve this problem. A similar approach is being worked on for OCaml (and therefore ReasonML): modular implicits.
- A better standard library. Many things are being explored in this area (see repository reasonml-community/belt). OCaml has a fair amount of fragmentation here, so standardization will be welcome.
- Better support for Unicode. At the moment, OCaml has no support for Unicode whatsoever and OCaml characters are 8 bit in size. You can get some Unicode support via BuckleScript's custom string literals (which compiles to JavaScript strings):

  ```
  Js.log({js|äöü|js});
  ```

  In the future, ReasonML may add more support for OCaml. They could, e.g., treat strings as UTF-8 with tool functions for accessing grapheme clusters and code points. Compilation to JavaScript will present challenges (e.g. accessing characters/units), because JavaScript is basically UTF-16.
- Long(er) term, ReasonML also has a compelling story for multicore code, via OCaml's algebraic effects.

For more information on what's planned for ReasonML, consult its Frequently Asked Questions.

I'm excited about what's in store. Better async support is especially important to me, as it will make Node.js development much more pleasant. Is there anything that you'd like to have that's not on this list?

Chapter 5

FAQ: ReasonML

5.1 Where is module `Str` in BuckleScript?

When you compile to JavaScript, you can't use module `Str`, whose functionality is too dependent on native strings. Instead, you have to use JavaScript-specific modules:

- Strings: module `Js.String`
- Regular expressions: module `Js.Re`

Part II

Core language

Chapter 6

A first look at ReasonML's syntax

In this chapter, I want to give you a first impression of what ReasonML code looks like. Therefore: Don't try to understand (yet) – proper explanations will be provided step by step in the following chapters.

This is ReasonML code:

```
/* A comment (no single-line comments, yet) */
/* You can /* nest */ comments */

/* Variable binding */
let myInt = 123;

/* Functions */
let id = x => x;
let add = (x, y) => x + y;

/* Defining a variant type */
type color = Red | Green | Blue;

/* A function that switches over a variant type */
let stringOfColor = (c: color) =>
  switch (c) {
  | Red => "Red"
  | Green => "Green"
  | Blue => "Blue"
  };

/* Calling stringOfColor() */
stringOfColor(Red); /* "Red" */
```

Again: There is no need to understand what I've just shown you. But if you want to dig deeper *right now*, you can:

- Pattern matching and `switch`
- Functions
- Variant types

6.1 Most things are expressions

For example, you can use `if-then-else` almost anywhere:

```
let myBool = true;
id(if (myBool) "yes" else "no");
```

And you can use blocks almost anywhere, too:

```
let abcabc = {
  let abc = "abc";
  abc ++ abc; /* "abcabc" */
};
```

In fact, the following two expressions are equivalent:

```
if (myBool) "yes" else "no";
if (myBool) {
  "yes";
} else {
  "no";
};
```

6.2 Semicolons matter

You may have noticed that there are many semicolons in the code in this chapter. Most of these are mandatory. Extra semicolons are allowed and ignored. Especially the interactive command line `rtop` will only evaluate expressions terminated with semicolons.

At first, it is a bit strange to even see semicolons after code blocks, but it makes sense, given that a code block is also an expression. With that knowledge, take another look at the two `if` expressions:

```
if (myBool) "yes" else "no";
if (myBool) {
  "yes";
} else {
  "no";
};
```

The semicolon at the end of the first line looks logical. But then the semicolon at the very end is logical, too, because we have only replaced the expression `"no"` with a block.

6.3 Everything is camel-cased in ReasonML

ReasonML is based on OCaml, which uses snake-casing for lowercase names (`create_resource`) and camel-casing for uppercase names (`StringUtilities`). That's why you'll occasionally see snake-cased names.

But all new ReasonML code is camel-cased (`StringUtilities`, `createResource`).

6.4 Special prefixes and suffixes for variable names

A prefixed underscore means: don't warn me about this variable not being used.

```
let f = (x, _y) => x;
  /* No warning about _y */
```

Suffixed apostrophes are legal (in math, `x'` means a modified version of `x`):

```
let x = 23;
let x' = x + 1;
```

Prefixed apostrophes are reserved for type variables (think generic types in C-style languages):

```
let len = (arr: array('a)) => Array.length(arr);
```

Type variables are explained in the chapter on variant types. They are similar to generic types in C-style languages.

Chapter 7

Basic values and types

In this chapter, we'll look at ReasonML's support for booleans, integers, floats, strings, characters and the unit type. We'll also see a few operators in action.

To explore, we'll use the interactive ReasonML command line `rtop`, which is part of the package `reason-cli` (the docs explain how to install it).

7.1 Interactions in `rtop`

Interactions in `rtop` look as follows.

```
# !true;
- : bool = false
```

Two observations:

- You must end the expression `!true;` with a semicolon in order for it to be evaluated.
- `rtop` always prints out the types of the results it computes. That is especially helpful later on, with more complicated types, e.g. the types of functions.

7.2 ReasonML is statically typed – what does that mean?

Values in ReasonML are statically typed. What does *static typing* mean?

On one hand, we have the term *type*. In this context, *type* means "set of values". For example *bool* is the name of the type of all boolean values: the (mathematical) set `{false, true}`.

On the other hand, we make the following distinction in the context of the life cycle of code:

- Static: at compile time, without running the code.
- Dynamic: at runtime, while the code is running.

Therefore *static typing* means: ReasonML knows the types of values at compile time. And types are also known while editing code, which supports intelligent editing features.

7.2.1 Get used to working with types

We have already encountered one benefit of static typing: editing support. It also helps with detecting some kinds of errors. And it often helps with documenting how code works (in a manner that is automatically checked for consistency).

In order to reap these benefits, you should get used to working with types. You get help in two ways:

- ReasonML often *infers* types (writes them for you). That is, a passive knowledge of types gets you surprisingly far.
- ReasonML gives descriptive error messages when something goes wrong that may even include tips for fixing the problem. That is, you can use trial and error to learn types.

7.2.2 No ad hoc polymorphism (yet)

Ad hoc polymorphism may sound brainy, but it has a simple definition and visible practical consequences for ReasonML. So bear with me.

ReasonML does not currently support ad hoc polymorphism where the same operation is implemented differently depending on the types of the parameters. Haskell, another functional programming language supports ad hoc polymorphism via *type classes*. ReasonML may eventually support it via the similar *modular implicits*.

ReasonML not supporting ad hoc polymorphism means that most operators such as `+` (int addition), `+.` (float addition) and `++` (string concatenation) only support a single type. Therefore, it is your responsibility to convert operands to proper types. On the plus side, ReasonML will warn you at compile time if you forget to do so. That's a benefit of static typing.

7.3 Comments

Before we get into values and types, let's learn comments.

ReasonML only has one way of writing comments:

```
/* This is a comment */
```

Conveniently, it is possible to nest this kind of comment (languages with C-style syntax are often not able to do that):

```
/* Outer /* inner comment */ comment */
```

Nesting is useful for commenting out pieces of code:

```
/*
foo(); /* foo */
bar(); /* bar */
*/
```

7.4 Booleans

Let's type in a few boolean expressions:

```
# true;
- : bool = true
# false;
- : bool = false
# !true;
- : bool = false
# true || false;
- : bool = true
# true && false;
- : bool = false
```

7.5 Numbers

These are integer expressions:

```
# 2 + 1;
- : int = 3
# 7 - 3;
- : int = 4
# 2 * 3;
- : int = 6
# 5 / 3;
- : int = 1
```

Floating-point expressions look as follows:

```
# 2.0 +. 1.0;
- : float = 3.
# 2. +. 1.;
- : float = 3.
# 2.25 +. 1.25;
- : float = 3.5

# 7. -. 3.;
- : float = 4.
# 2. *. 3.;
- : float = 6.
# 5. /. 3.;
- : float = 1.66666666666666674
```

7.6 Strings

Normal string literals are delimited by double quotes:

```
# "abc";
- : string = "abc"
# String.length("ü");
- : int = 2

# "abc" ++ "def";
- : string = "abcdef"
# "There are " ++ string_of_int(11 + 5) ++ " books";
- : string = "There are 16 books"

# {| Multi-line
string literal
\ does not escape
|};
- : string = " Multi-line\nstring literal\n\\ does not escape\n"
```

ReasonML strings are encoded as UTF-8 and not compatible with JavaScript's UTF-16 strings. ReasonML's support for Unicode is also worse than JavaScript's – already limited – one. As a short-term workaround, you can use BuckleScript's JavaScript strings in ReasonML:

```
Js.log("äöü"); /* garbage */
Js.log({js|äöü|js}); /* äöü */
```

These strings are produced via multi-line string literals annotated with `js`, which are only treated specially by BuckleScript. In native ReasonML, you get normal strings.

7.7 Characters

Characters are delimited by single quotes. Only the first 7 bits of Unicode are supported (no umlauts etc.):

```
# 'x';
- : char = 'x'
# String.get("abc", 1);
- : char = 'b'
# "abc".[1];
- : char = 'b'
```

`"x".[0]` is syntactic sugar for `String.get("x", 0)`.

7.8 The unit type

Sometimes, you need a value denoting "nothing". ReasonML has the special value `()` for this purpose. `()` has its own type, `unit` and is the only element of that type:

```
# ();
- : unit = ()
```

In contrast to `null` in C-style languages, `()` is not an element of any other type.

Among other things, the type `unit` is used for functions with side effects that don't return anything. For example:

```
# print_string;
- : (string) => unit = <fun>
```

The function `print_string` takes a string as an argument and prints that string. It has no real result.

7.9 Converting between basic types

ReasonML's standard library has functions for converting between the basic types:

```
# string_of_int(123);
- : string = "123"
# string_of_bool(true);
- : string = "true"
```

All of the conversion functions are named as follows.

```
«outputType»_of_«inputType»
```

7.10 More operators

7.10.1 Comparison operators

The following are comparison operators. They are part of the few operators that work with several types (they are *polymorphic*).

```
# 3.0 < 4.0;
- : bool = true
# 3 < 4;
- : bool = true
# 3 <= 4;
- : bool = true
```

You cannot, however, mix operand types:

```
# 3.0 < 4;
Error: Expression has type int but expected type float
```

7.10.2 Equality operators

ReasonML has two equality operators.

Double equals (equality by value) compares values and does so even for reference types such as lists.

```
# [1,2,3] == [1,2,3];
- : bool = true
```

In contrast, triple equals (equality by reference) compares references:

```
# [1,2,3] === [1,2,3];
- : bool = false
```

== is the preferred equality operator (unless you really want to compare references).

Chapter 8

`let` bindings and scopes

In this chapter, we look at how to introduce new variables and scopes in ReasonML.

8.1 Normal `let` bindings

Variables are defined as follows:

```
# let x = 123;
let x: int = 123;
# x;
- : int = 123
```

Each *binding* (variable-value pair) that is created in this manner is *immutable* – you cannot assign a different value to the variable. The norm is for the value to also be immutable, but it doesn't have to be.

Given that the binding is immutable, it is logical that you have to immediately initialize the variable. You can't leave it uninitialized.

8.2 Redefining variables

ReasonML does not prevent you from redefining variables:

```
# let x = 1;
let x: int = 1;
# x;
- : int = 1
# let x = 2;
let x: int = 2;
# x;
- : int = 2
```

This is not in conflict with the immutability of bindings: It works more like shadowing in nested scopes than like changing the value of a variable.

Being able to redefine variables is especially useful in interactive command lines.

8.3 Type annotations

You can also annotate the variable with a type:

```
# let y: string = "abc";
let y: string = "abc";
```

Declaring types is occasionally necessary with more complicated types, but redundant with simple types.

8.4 Creating new scopes via scope blocks

The *scope* of a variable is the syntactic construct in which it exists. Blocks enable you introduce new scopes. They start and end with curly braces (`{}`):

```
let x = "hello";
print_string(x); /* hello */
{ /* A */
    let x = "tmp";
    print_string(x); /* tmp */
};  /* B */
print_string(x); /* hello */
```

The block starts in line A and ends in line B.

The interior of a block has the same structure as the top level of a file: it is a sequence of expressions that are separated by semicolons.

Why is there a semicolon after the closing curly brace in line B? A block is just another expression. Its value is the value of the last expression inside it. That means you can put code blocks wherever you can put expressions:

```
let x = { print_string("hi"); 123 }; /* hi */
print_int(x); /* 123 */
```

Another example:

```
print_string({
    let s = "ma";
    s ++ s;
}); /* mama */
```

This continues a common theme in ReasonML: most things are expressions.

Chapter 9

Pattern matching: destructuring, `switch`, `if` expressions

In this chapter, we look at three features that are all related to pattern matching: destructuring, `switch`, and `if` expressions.

9.1 Digression: tuples

To illustrate patterns and pattern matching, we'll use tuples. Tuples are basically records whose parts are identified by position (and not by name). The parts of a tuple are called *components*.

Let's create a tuple in `rtop`:

```
# (true, "abc");
- : (bool, string) = (true, "abc")
```

The first component of this tuple is the boolean `true`, the second component is the string `"abc"`. Accordingly, the tuple's type is `(bool, string)`.

Let's create one more tuple:

```
# (1.8, 5, ('a', 'b'));
- : (float, int, (char, char)) = (1.8, 5, ('a', 'b'))
```

9.2 Pattern matching

Before we can examine destructuring, `switch` and `if`, we need to learn their foundation: pattern matching.

Patterns are a programming mechanism that helps with processing data. They serve two purposes:

- Check what structure data has.

- Extract parts of data.

This is done by *matching* patterns against data. Syntactically, patterns work as follows:

- ReasonML has syntax for creating data. For example: tuples are created by separating data with commas and putting the result in parentheses.
- ReasonML has syntax for processing data. The syntax of patterns mirrors the syntax for creating data.

Let's start with simple patterns that support tuples. They have the following syntax:

- A variable name is a pattern.
  - Examples: `x`, `y`, `foo`
- A literal is a pattern.
  - Examples: `123`, `"abc"`, `true`
- A tuple of patterns is a pattern.
  - Examples: `(8,x)`, `(3.2,"abc",true)`, `(1, (9, foo))`

The same variable name cannot be used in two different locations. That is, the following pattern is illegal: `(x, x)`

9.2.1 Equality checks

The simplest patterns don't have any variables. Matching such patterns is basically the same as an equality check. Let's look at a few examples:

Pattern	Data	Matches?
`3`	`3`	yes
`1`	`3`	no
`(true, 12, 'p')`	`(true, 12, 'p')`	yes
`(false, 12, 'p')`	`(true, 12, 'p')`	no

So far, we have used the pattern to ensure that the data has the expected structure. As a next step, we introduce variable names. Those make the structural checks more flexible and let us extract data.

9.2.2 Variable names in patterns

A variable name matches any data at its position and leads to the creation of a variable that is bound to that data.

Pattern	Data	Matches?	Variable bindings
`x`	`3`	yes	`x = 3`
`(x, y)`	`(1, 4)`	yes	`x = 1, y = 4`
`(1, y)`	`(1, 4)`	yes	`y = 4`
`(2, y)`	`(1, 4)`	no	

The special variable name _ does not create variable bindings and can be used multiple

times:

Pattern	Data	Matches?	Variable bindings
`(x, _)`	`(1, 4)`	yes	`x = 1`
`(1, _)`	`(1, 4)`	yes	
`(_, _)`	`(1, 4)`	yes	

9.2.3 Alternatives in patterns

Let's examine another pattern feature: Two or more subpatterns separated by vertical bars form an *alternative*. Such a pattern matches if one of the subpatterns matches. If a variable name exists in one subpattern, it must exit in all subpatterns.

Examples:

Pattern	Data	Matches?	Variable bindings
`1|2|3`	`1`	yes	
`1|2|3`	`2`	yes	
`1|2|3`	`3`	yes	
`1|2|3`	`4`	no	
`(1|2|3, 4)`	`(1, 4)`	yes	
`(1|2|3, 4)`	`(2, 4)`	yes	
`(1|2|3, 4)`	`(3, 4)`	yes	
`(x, 0) | (0, x)`	`(1, 0)`	yes	`x = 1`

9.2.4 The `as` operator: bind and match at the same time

Until now, you had to decide whether you wanted to bind a piece of data to a variable or to match it via a subpattern. The `as` operator lets you do both: it's left-hand side is a subpattern to match, its right-hand side is the name of a variable that the current data will be bound to.

Pattern	Data	Matches?	Variable bindings
`7 as x`	`7`	yes	$x = 7$
`(8, x) as y`	`(8, 5)`	yes	$x = 5, y = (8, 5)$
`((1,x) as y, 3)`	`((1,2), 3))`	yes	$x = 2, y = (1, 2)$

9.2.5 There are many more ways of creating patterns

ReasonML supports more complex data types than just tuples. For example: lists and records. Many of those data types are also supported via pattern matching. More on that in later chapters.

9.3 Pattern matching via let (destructuring)

You can do pattern matching via `let`. As an example, let's start by creating a tuple:

```
# let tuple = (7, 4);
let tuple: (int, int) = (7, 4);
```

We can use pattern matching to create the variables `x` and `y` and bind them to `7` and `4`, respectively:

```
# let (x, y) = tuple;
let x: int = 7;
let y: int = 4;
```

The variable name _ also works and does not create variables:

```
# let (_, y) = tuple;
let y: int = 4;
# let (_, _) = tuple;
```

If a pattern doesn't match, you get an exception:

```
# let (1, x) = (5, 5);
Warning: this pattern-matching is not exhaustive.
Exception: Match_failure.
```

We get two kinds of feedback from ReasonML:

- At compile time: A warning that there are `(int, int)` tuples that the pattern doesn't cover. We'll look at what that means when we learn `switch` expressions.
- At runtime: An exception that matching failed.

Single-branch pattern matching via `let` is called *destructuring*. Destructuring can also be used with function parameters (as we'll see in the chapter on functions).

9.4 switch

`let` matched a single pattern against data. With a `switch` expression, we can try multiple patterns. The first match determines the result of the expression. That looks as follows:

```
switch «value» {
| «pattern1» => «result1»
| «pattern2» => «result2»
...
}
```

`switch` goes through the branches sequentially: the first pattern that matches `value` leads to the associated expression becoming the result of the `switch` expression. Let's look at an example where pattern matching is simple:

```
let value = 1;
let result = switch value {
| 1 => "one"
| 2 => "two"
```

```
};
/* result == "one" */
```

If the `switch` value is more than a single entity (variable name, qualified variable name, literal, etc.), it needs to be in parentheses:

```
let result = switch (1 + 1) {
| 1 => "one"
| 2 => "two"
};
/* result == "two" */
```

9.4.1 Warnings about exhaustiveness

When you compile the previous example or enter it in `rtop`, you get the following compile-time warning:

```
Warning: this pattern-matching is not exhaustive.
```

That means: The operand `1` has the type `int` and the branches do not cover all elements of that type. This warning is very useful, because it tells us that there are cases that we may have missed. That is, we are warned about potential trouble ahead. If there is no warning, `switch` will always succeed.

If you don't fix this issue, ReasonML throws a runtime exception when an operand doesn't have a matching branch:

```
let result = switch 3 {
| 1 => "one"
| 2 => "two"
};
/* Exception: Match_failure */
```

One way to make this warning go away is to handle all elements of a type. I'll briefly sketch how to do that for recursively defined types. These are defined via:

- One or more (non-recursive) base cases.
- One or more recursive cases.

For example, for natural numbers, the base case is zero, the recursive case is one plus a natural number. You can cover natural numbers exhaustively with `switch` via two branches, one for each case.

For now, it's enough to know that, whenever you can, you should do exhaustive coverage. Then the compiler warns you if you miss a case, preventing a whole category of errors.

If exhaustive coverage is not an option, you can introduce a catch-all branch. The next section shows how to do that.

9.4.2 Variables as patterns

The warning about exhaustiveness goes away if you add a branch whose pattern is a variable:

```
let result = switch 3 {
| 1 => "one"
| 2 => "two"
| x => "unknown: " ++ string_of_int(x)
};
/* result == "unknown: 3" */
```

We have created the new variable `x` by matching it against the `switch` value. That new variable can be used in the expression of the branch.

This kind of branch is called "catch-all": it comes last and is evaluated if all other branches fail. It always succeeds and matches everything. In C-style languages, catch-all branches are called `default`.

If you just want to match everything and don't care what is matched, you can use an underscore:

```
let result = switch 3 {
| 1 => "one"
| 2 => "two"
| _ => "unknown"
};
/* result == "unknown" */
```

9.4.3 Patterns for tuples

Let's implement logical And (&&) via a `switch` expression:

```
let tuple = (true, true);

let result = switch tuple {
| (false, false) => false
| (false, true) => false
| (true, false) => false
| (true, true) => true
};
/* result == true */
```

This code can be simplified by using an underscore and a variable:

```
let result = switch tuple {
| (false, _) => false
| (true, x) => x
};
/* result == true */
```

9.4.4 The `as` operator

The `as` operator also works in `switch` patterns:

```
let tuple = (8, (5, 9));
let result = switch tuple {
```

```
| (0, _) => (0, (0, 0))
| (_, (x, _) as t) => (x, t)
};
/* result == (5, (5, 9)) */
```

9.4.5 Alternatives in patterns

Using alternatives in subpatterns looks as follows.

```
switch someTuple {
| (0, 1 | 2 | 3) => "first branch"
| _ => "second branch"
};
```

Alternatives can also be used at the top level:

```
switch "Monday" {
| "Monday"
| "Tuesday"
| "Wednesday"
| "Thursday"
| "Friday" => "weekday"
| "Saturday"
| "Sunday" => "weekend"
| day => "Illegal value: " ++ day
};
/* Result: "weekday" */
```

9.4.6 Guards for branches

guards (conditions) for branches are a `switch`-specific feature: they come after patterns and are preceded by the keyword `when`. Let's look at an example:

```
let tuple = (3, 4);
let max = switch tuple {
| (x, y) when x > y => x
| (_, y) => y
};
/* max == 4 */
```

The first branch is only evaluated if the guard `x > y` is `true`.

9.5 `if` expressions

ReasonML's `if` expressions look as follows (`else` can be omitted):

```
if («bool») «thenExpr» else «elseExpr»;
```

For example:

```
# let bool = true;
let bool: bool = true;
```

```
# let boolStr = if (bool) "true" else "false";
let boolStr: string = "true";
```

Given that scope blocks are also expressions, the following two `if` expressions are equivalent:

```
if (bool) "true" else "false"
if (bool) {"true"} else {"false"}
```

In fact, `refmt` pretty-prints the former expression as the latter.

The `then` expression and the `else` expression must have the same type.

```
Reason # if (true) 123 else "abc";
Error: This expression has type string
but an expression was expected of type int
```

9.5.1 Omitting the `else` branch

You can omit the `else` branch – the following two expressions are equivalent.

```
if (b) expr else ()
if (b) expr
```

Given that both branches must have the same type, `expr` must have the type `unit` (whose only element is `()`).

For example, `print_string()` evaluates to `()` and the following code works:

```
# if (true) print_string("hello\n");
hello
- : unit = ()
```

In contrast, this doesn't work:

```
# if (true) "abc";
Error: This expression has type string
but an expression was expected of type unit
```

9.6 The ternary operator (_?_:_)

ReasonML also gives you the ternary operator as an alternative to `if` expressions. The following two expressions are equivalent.

```
if (b) expr1 else expr2
b ? expr1 : expr2
```

The following two expressions are equivalent, too. `refmt` even pretty-prints the former as the latter.

```
switch (b) {
| true => expr1
| false => expr2
};
```

```
b ? expr1 : expr2;
```

I don't find the ternary operator operator very useful in ReasonML: its purpose in languages with C syntax is to have an expression version of the `if` statement. But `if` is already an expression in ReasonML.

Chapter 10

Functions

This chapter explores how functions work in ReasonML.

10.1 Defining functions

An anonymous (nameless) function looks as follows:

```
(x) => x + 1;
```

This function has a single parameter, `x`, and the body `x + 1`.

You can give that function a name by binding it to a variable:

```
let plus1 = (x) => x + 1;
```

This is how you call `plus1`:

```
# plus1(5);
- : int = 6
```

10.1.1 Functions as parameters of other functions (high-order functions)

Functions can also be parameters of other functions. To demonstrate this feature, we briefly use *lists*, which are explained in their own chapter. *Lists* are, roughly, singly linked lists and similar to immutable arrays.

The list function `List.map(func, list)` takes `list`, applies `func` to each of its elements and returns the results in a new list. For example:

```
# List.map((x) => x + 1, [12, 5, 8, 4]);
- : list(int) = [13, 6, 9, 5]
```

Functions that have functions as parameters or results are called *higher-order functions*. Functions that don't are called *first-order functions*. `List.map()` is a higher-order function. `plus1()` is a first-order function.

10.1.2 Blocks as function bodies

A function's body is an expression. Given that scope blocks are expressions, the following two definitions for `plus1` are equivalent.

```
let plus1 = (x) => x + 1;

let plus1 = (x) => {
    x + 1
};
```

10.2 Single parameters without parentheses

If a function has a single parameter and that parameter is defined via an identifier, you can omit the parentheses:

```
let plus1 = x => x + 1;
```

10.3 Recursive bindings via `let rec`

Normally, you can only refer to `let`-bound values that already exist. That means that you can't define mutually recursive and self-recursive functions.

10.3.1 Defining mutually recursive functions

Let's examine mutually recursive functions first. The following two functions `even` and `odd` are mutually recursive (this is an example, not how you'd actually implement these functions). You must use the special `let rec` to define them:

```
let rec even = (x) =>
  if (x <= 0) {
    true;
  } else {
    odd(x - 1);
  }
and odd = (x) =>
  if (x <= 0) {
    false;
  } else {
    even(x - 1);
  };
```

Notice how `and` connects multiple `let rec` entries that all know each other. There is no semicolon before the `and`. The semicolon at the end indicates that `let rec` is finished.

Let's use these functions:

```
# even(11);
- : bool = false
# even(2);
```

```
- : bool = true
# odd(11);
- : bool = true
# odd(2);
- : bool = false
```

10.3.2 Defining self-recursive functions

You also need `let rec` for functions that call themselves recursively, because when the recursive call is made, the binding does not exist, yet. For example:

```
let rec factorial = (x) =>
  if (x <= 2) {
    x
  } else {
    x * factorial(x - 1)
  };

factorial(3); /* 6 */
factorial(4); /* 24 */
```

10.4 Terminology: arity

The arity of a function is how many (positional) parameters it has. The arity of `factorial()` is 1. The following adjectives describe functions with arities from 0 to 2:

- A *nullary* function is a function with arity 0.
- A *unary* function is a function with arity 1.
- A *binary* function is a function with arity 2.
- A *ternary* function is a function with arity 3.

Beyond arity 3, we talk about 4-ary functions, 5-ary functions etc. Functions whose arity can vary are called variadic functions. These are also called *varargs* in some programming languages.

10.5 The types of functions

Functions are the first time that we get in contact with complex types: types built by combining other types. Let's use `rtop` to determine the types of a two functions.

10.5.1 Types of first-order functions

First, a function `add()`:

```
# let add = (x, y) => x + y;
let add: (int, int) => int = <fun>;
```

Therefore, the type of `add` is:

```
(int, int) => int
```

The arrow indicates that `add` is a function. Its parameters are two ints. Its result is a single int.

The notation `(int, int) => int` is also called the *(type) signature* of `add`. It describes the types of its inputs and its outputs.

10.5.2 Types of higher-order functions

Second, a higher-order function `callFunc()`:

```
# let callFunc = (f) => f(1) + f(2);
let callFunc: ((int) => int) => int = <fun>;
```

You can see that the parameter of `callFunc` is itself a function and has the type `(int) => int`.

This is how `callFunc()` is used:

```
# callFunc(x => x);
- : int = 3
# callFunc(x => 2 * x);
- : int = 6
```

10.5.3 Type annotations and type inference

Type annotations are optional in ReasonML, but they improve the accuracy of type checking. The most extreme is to annotate everything:

```
let add = (x: int, y: int): int => x + y;
```

We have provided type annotations for both parameters `x` and `y` and for the result of the function (the last `: int` before the arrow).

You can omit the annotation for the return type and ReasonML will *infer* it (deduce it from the type of the parameters):

```
# let add = (x: int, y: int) => x + y;
let add: (int, int) => int = <fun>;
```

However, *type inference* is more sophisticated than that. It doesn't only work top-down, it can also infer the types of the parameters from the `int`-only plus operator (`+`):

```
# let add = (x, y) => x + y;
let add: (int, int) => int = <fun>;
```

If you want, you can also annotate only some of the parameters:

```
let add = (x, y: int) => x + y;
```

10.5.4 Type annotations: best practice

The coding style I prefer for functions is to annotate all parameters, but to let ReasonML infer the return type. Apart from improving type checking, annotations for parameters are also good documentation.

10.6 There are no functions without parameters

ReasonML doesn't have nullary functions, but you can use it without ever noticing that.

Recall that `()` is roughly similar to `null` in many C-style languages. It is the only element of type `unit`. When calling functions, omitting parameters is the same as passing the unit value as a single parameter. That is, the following two expressions are equivalent.

```
func()
func(())
```

The following example demonstrates this phenomenon: If you call a unary function without parameters, `rtop` underlines `()` and complains about that expression having the wrong type. It does not complain about not enough parameters being provided (it doesn't partially apply either – details later).

```
# let id = (x: int) => x;
let id: (int) => int = <fun>;
# id();
Error: This expression has type unit but
an expression was expected of type int
```

If you define a function that has no parameters, ReasonML adds a parameter for you, whose type is `unit`:

```
# let f = () => 123;
let f: (unit) => int = <fun>;
```

10.6.1 Why no nullary functions?

Why doesn't ReasonML have nullary functions? That is due to ReasonML always performing *partial application* (explained in detail later): If you don't provide all of a function's parameters, you get a new function from the remaining parameters to the result. As a consequence, if you could actually provide no parameters at all, then `func()` would be the same as `func` and neither would actually call `func`.

10.7 Destructuring function parameters

Destructuring can be used wherever variables are bound to values. That is, it also works in parameter definitions. Let's look at a function that adds the components of a tuple:

```
let addComponents = ((x, y)) => x + y;
let tuple = (3, 4);
addComponents(tuple); /* 7 */
```

The double parentheses around `x, y` indicate that `addComponents` is a function with a single parameter, a tuple whose components are `x` and `y`. It is not a function with the two parameters `x` and `y`. Its type is:

```
addComponents: ((int, int)) => int
```

When it comes to type annotations, you can either annotate the components:

```
# let addComponents = ((x: int, y: int)) => x + y;
let addComponents: ((int, int)) => int = <fun>;
```

Or you can annotate the whole parameter:

```
# let addComponents = ((x, y): (int, int)) => x + y;
let addComponents: ((int, int)) => int = <fun>;
```

10.8 Labeled parameters

So far, we have only used *positional parameters*: the position of an actual parameter at the call site determines what formal parameter it is bound to.

But ReasonML also supports *labeled parameters*. Here, labels are used to associate actual parameters with formal parameters.

As an example, let's examine a version of `add` that uses labeled parameters:

```
let add = (~x, ~y) => x + y;
add(~x=7, ~y=9); /* 16 */
```

In this function definition, we used the same name for the label `~x` and the parameter `x`. You can also use separate names, e.g. `~x` for the label and `op1` for the parameter:

```
/* Inferred types */
let add = (~x as op1, ~y as op2) =>
  op1 + op2;

/* Specified types */
let add = (~x as op1: int, ~y as op2: int) =>
  op1 + op2;
```

At call sites, you can abbreviate `~x=x` as just `~x`:

```
let x = 7;
let y = 9;
add(~x, ~y);
```

One nice feature of labels is that you can mention labeled parameters in any order:

```
# add(~x=3, ~y=4);
- : int = 7
# add(~y=4, ~x=3);
- : int = 7
```

10.8.1 Compatibility of function types with labeled parameters

There is one unfortunate caveat to being able to mention labeled parameters in any order: function types are only compatible if labels are mentioned in the same order.

Consider the following three functions.

```
let add = (~x, ~y) => x + y;
let addxy = (add: ((~x: int, ~y: int) => int)) => add(5, 2);
let addyx = (add: ((~y: int, ~x: int) => int)) => add(5, 2);
```

`addxy` works as expected with `add`:

```
# addxy(add);
- : int = 7
```

However, with `addyx`, we get an error, because the labels are in the wrong order:

```
# addyx(add);
Error: This expression has type
(~x: int, ~y: int) => int
but an expression was expected of type
(~y: int, ~x: int) => int
```

10.9 Optional parameters

In ReasonML, only labeled parameters can be optional. In the following code, both `x` and `y` are optional.

```
let add = (~x=?, ~y=?, ()) =>
  switch (x, y) {
  | (Some(x'), Some(y')) => x' + y'
  | (Some(x'), None) => x'
  | (None, Some(y')) => y'
  | (None, None) => 0
  };
```

Let's examine what the relatively complicated code does.

Why the `()` as the last parameter? That is explained in the next section.

What does the `switch` expression do? If you declare a parameter as optional, it always has the type `option(t)`, where `t` is whatever type the actual values have. `option` is a *variant* (which is explained in a separate chapter). For now, I'll give a brief preview. The definition of `option` is:

```
type option('a) = None | Some('a);
```

It is used as follows:

- Omit `~x` and `x` will be bound to `None`.
- Provide the value `123` for `~x` and `x` will be bound to `Some(123)`.

In other words, `option` wraps values and the `switch` expression in the example unwraps them.

10.9.1 With optional parameters, you need at least one positional parameter

Why does `add` have a parameter of type `unit` (an empty parameter, if you will) at the end?

```
let add = (~x=?, ~y=?, ()) =>
  ...
```

The reason has to do with partial application and is explained in more detail later. In a nutshell, two things are in conflict here:

- With partial application, if you omit parameters, you create a function that lets you fill in those remaining parameters.
- With optional parameters, if you omit parameters, they should be bound to their defaults.

To resolve this conflict, ReasonML fills in all defaults for missing optional parameters when it encounters the first positional parameter. Before it encounters a positional parameter, it still waits for the missing optional parameters. That is, you need a positional parameter to trigger the call and since `add()` doesn't have one, we added an empty one.

The advantage of this slightly weird approach is that you get the best of both worlds: you get partial application and optional parameters.

10.9.2 Type annotations for optional parameters

When you annotate optional parameters, they must all have `option(···)` types:

```
let add = (~x: option(int)=?, ~y: option(int)=?, ()) =>
  ...
```

The type signature of `add` is:

```
(~x: int=?, ~y: int=?, unit) => int
```

It's unfortunate that the definition differs from the type signature in this case. But it is the same as for parameters with default values (which are explained next) where it makes sense. The idea is to hide the implementation detail of how optional parameters are handled.

10.9.3 Parameter default values

Handling missing parameters can be cumbersome:

```
let add = (~x=?, ~y=?, ()) =>
  switch (x, y) {
  | (Some(x'), Some(y')) => x' + y'
  | (Some(x'), None) => x'
  | (None, Some(y')) => y'
  | (None, None) => 0
  };
```

In this case, all we want is for `x` and `y` to be zero if they are omitted. ReasonML has special syntax for this:

```
let add = (~x=0, ~y=0, ()) => x + y;
```

10.9.4 Type annotations with parameter default values

If there are default values, type annotations are more intuitive (no `option()`):

```
let add = (~x: int=0, ~y: int=0, ()) =>
  x + y;
```

The type signature of `add` is:

```
(~x: int=?, ~y: int=?, unit) => int
```

10.9.5 Passing `option` values to optional parameters (advanced)

Internally, optional parameters are received as elements of the `option` type (`None` or `Some(x)`). Until now, you could only pass those values by either providing or omitting parameters. But there is also a way to pass those values directly. Before we get to use cases for this feature, let's try it out first, via the following function.

```
let multiply = (~x=1, ~y=1, ()) => x * y;
```

`multiply` has two optional parameters. Let's start by providing `~x` and omitting `~y`, via elements of `option`:

```
# multiply(~x = ?Some(14), ~y = ?None, ());
- : int = 14
```

The syntax for passing `option` values is:

```
~label = ?expression
```

If `expression` is a variable whose name is `label`, you can abbreviate: the following two syntaxes are equivalent.

```
~foo = ?foo
~foo?
```

So what is the use case? It's one function forwarding an optional parameter to another function's optional parameter. That way, it can rely on that function's parameter default value and doesn't have to define one itself.

Let's look at an example: The following function `square` has an optional parameter, which is passes on to `multiply`'s two optional parameters:

```
let square = (~x=?, ()) => multiply(~x?, ~y=?x, ());
```

`square` does not have to specify a parameter default value, it can use `multiply`'s defaults.

10.10 Partial application

Partial application is a mechanism that makes functions more versatile: If you omit one or more parameters at the end of a function call `f(···)`, `f` returns a function that maps the missing parameters to `f`'s final result. That is, you *apply* `f` to its parameters in multiple steps. The first step is called a *partial application* or a *partial call*.

Let's see how that works. We first create a function `add` with two parameters:

```
# let add = (x, y) => x + y;
let add: (int, int) => int = <fun>;
```

Then we partially call the binary function `add` to create the unary function `plus5`:

```
# let plus5 = add(5);
let plus5: (int) => int = <fun>;
```

We have only provided `add`'s first parameter, `x`. Whenever we call `plus5`, we provide `add`'s second parameter, `y`:

```
# plus5(2);
- : int = 7
```

10.10.1 Why is partial application useful?

Partial application lets you write more compact code. To demonstrate how, we'll work with a list of numbers:

```
# let numbers = [11, 2, 8];
let numbers: list(int) = [11, 2, 8];
```

Next, we'll use the standard library function `List.map`. `List.map(func, myList)` takes `myList`, applies `func` to each of its elements and returns them as a new list.

We use `List.map` to add 2 to each element of `numbers`:

```
# let plus2 = x => add(2, x);
let plus2: (int) => int = <fun>;
# List.map(plus2, numbers);
- : list(int) = [13, 4, 10]
```

With partial application we can make this code more compact:

```
# List.map(add(2), numbers);
- : list(int) = [13, 4, 10]
```

Let's compare the two versions more directly:

```
List.map(x => add(2, x), numbers)
List.map(add(2), numbers)
```

Which version is better? That depends on your taste. The first version is – arguably – more self-descriptive, the second version is more concise.

Partial application really shines with the pipe operator (`|>`) for function composition (which is explained later).

10.10.2 Partial application and labeled parameters

So far, we have only see partial application with positional parameters, but it works with labeled parameters, too. Consider, again, the labeled version of `add`:

```
# let add = (~x, ~y) => x + y;
let add: (~x: int, ~y: int) => int = <fun>;
```

If we call `add` with only the first labeled parameter, we get a function that maps the second parameter to the result:

```
# add(~x=4);
- : (~y: int) => int = <fun>
```

Providing only the second labeled parameter works analogously.

```
# add(~y=4);
- : (~x: int) => int = <fun>
```

That is, labels don't impose an order here. That means that partial application is more versatile with labels, because you can partially apply any labeled parameter, not just the last one.

10.10.2.1 Partially application and optional parameters

How about optional parameters? The following version of `add` has only optional parameters:

```
# let add = (~x=0, ~y=0, ()) => x + y;
let add: (~x: int=?, ~y: int=?, unit) => int = <fun>;
```

If you mention only the label `~x` or only the label `~y`, partial application works as before, with one difference: The additional positional parameter of type `unit` must also still be filled in.

```
# add(~x=3);
- : (~y: int=?, unit) => int = <fun>
# add(~y=3);
- : (~x: int=?, unit) => int = <fun>
```

However, as soon as you mention the positional parameter, there is no more partial application; the defaults are now filled in:

```
# add(~x=3, ());
- : int = 3
# add(~y=3, ());
- : int = 3
```

Even if you take one or two intermediate steps, the `()` is always the final signal to evaluate. One intermediate step looks as follows.

```
# let plus5 = add(~x=5);
let plus5: (~y: int=?, unit) => int = <fun>;
# plus5(());
- : int = 5
```

Two intermediate steps:

```
# let plus5 = add(~x=5);
let plus5: (~y: int=?, unit) => int = <fun>;
# let result8 = plus5(~y=3);
let result8: (unit) => int = <fun>;
# result8(());
- : int = 8
```

10.10.3 Currying (advanced)

Currying is one technique for implementing partial application for positional parameters. Currying a function means transforming it from a function with an arity of 1 or more to a series of unary function calls.

For example, take the binary function `add`:

```
let add = (x, y) => x + y;
```

To curry `add` means to convert it to the following function:

```
let add = x => y => x + y;
```

Now we have to invoke `add` as follows:

```
# add(3)(1);
- : int = 4
```

What have we gained? Partial application is easy now:

```
# let plus4 = add(4);
let plus4: (int) => int = <fun>;
# plus4(7);
- : int = 11
```

And now the surprise: all functions in ReasonML are automatically curried. That's how it supports partial application. You can see that if you look at the type of the curried `add`:

```
# let add = x => y => x + y;
let add: (int, int) => int = <fun>;
```

On other words: `add(x, y)` is the same as `add(x)(y)` and the following two types are equivalent:

```
(int, int) => int
int => int => int
```

Let's conclude with a function that curries binary functions. Given that currying functions that are already curried is meaningless, we'll curry a function whose single parameter is a pair.

```
let curry2 = (f: (('a, 'b)) => 'c) => x => y => f((x, y));
```

Let's use `curry2` with a unary version of `add`:

```
# let add = ((x, y)) => x + y;
let add: ((int, int)) => int = <fun>;
# curry2(add);
- : (int, int) => int = <fun>
```

The type at the end tells us that we have created a binary function.

10.11 The reverse-application operator (|>)

The operator `|>` is called *reverse-application operator* or *pipe operator*. It lets you chain function calls: `x |> f` is the same as `f(x)`. That may not look like much, but it is quite useful when combining function calls.

10.11.1 Example: piping ints and strings

Let's start with a simple example. Given the following two functions.

```
let times2 = (x: int) => x * 2;
let twice = (s: string) => s ++ s;
```

If we use them with traditional function calls, we get:

```
# twice(string_of_int(times2(4)));
- : string = "88"
```

First we apply `times2` to 4, then `string_of_int` (a function in the standard library) to the result, etc. The pipe operator lets us write code that is closer to the description that I have just given:

```
let result = 4 |> times2 |> string_of_int |> twice;
```

10.11.2 Example: piping lists

With more complex data and currying, we get a style that is reminiscent of chained method calls in object-oriented programming.

For example, the following code works with a list of ints:

```
[4, 2, 1, 3, 5]
|> List.map(x => x + 1)
|> List.filter(x => x < 5)
|> List.sort(compare);
```

These functions are explained in the chapter on lists. For now, it is enough to have a rough idea of how they work.

The three computational steps are:

```
# let l0 = [4, 2, 1, 3, 5];
let l0: list(int) = [4, 2, 1, 3, 5];
# let l1 = List.map(x => x + 1, l0);
let l1: list(int) = [5, 3, 2, 4, 6];
# let l2 = List.filter(x => x < 5, l1);
```

```
let l2: list(int) = [3, 2, 4];
# let l3 = List.sort(compare, l2);
let l3: list(int) = [2, 3, 4];
```

We see that in all of these functions, the primary parameter comes last. When we piped, we first filled in the secondary parameters via partial application, creating a function. Then the pipe operator filled in the primary parameter, by calling that function.

The primary parameter is similar to `this` or `self` in object-oriented programming languages.

10.12 Tips for designing function signatures

These are a few tips for designing the type signatures of functions:

- If a function has a single primary parameter, make it a positional parameter and put it at the end. That supports the pipe operator for function composition.
- Some functions have multiple primary parameters that are all similar. Turn these into multiple positional parameters at the end. An example would be a function that concatenates two lists into a single list. In that case, both positional parameters are lists.
- All other parameters should be labeled.
- If there are two or more primary parameters that are different, all of them should be labeled.
- If a function has only a single parameter, it tends to be unlabeled, even if it is not strictly primary.

The idea behind these rules is to make code as self-descriptive as possible: The primary (or only) parameter is described by the name of the function, the remaining parameters are described by their labels.

As soon as a function has more than one positional parameter, it usually becomes difficult to tell what each parameter does. Compare, for example, the following two function calls. The second one is much easier to understand.

```
blit(bytes, 0, bytes, 10, 10);
blit(~src=bytes, ~src_pos=0, ~dst=bytes, ~dst_pos=10, ~len=10);
```

I also like optional parameters, because they enable you to add more parameters to functions without breaking existing callers. That helps with evolving APIs.

Source of this section: Sect. “Suggestions for labeling” in the OCaml Manual.

10.13 Single-argument match functions

ReasonML provides an abbreviation for unary functions that immediately switch on their parameters. Take, for example the following function.

```
let divTuple = (tuple) =>
  switch tuple {
  | (_, 0) => (-1)
```

```
  | (x, y) => x / y
  };
```

This function is used as follows:

```
# divTuple((9, 3));
- : int = 3
# divTuple((9, 0));
- : int = -1
```

If you use the `fun` operator to define `divTuple`, the code becomes shorter:

```
let divTuple =
  fun
  | (_, 0) => (-1)
  | (x, y) => x / y;
```

10.14 (Advanced)

All remaining sections are advanced.

10.15 Operators

One neat feature of ReasonML is that operators are just functions. You can use them like functions if you put them in parentheses:

```
# (+)(7, 1);
- : int = 8
```

And you can define your own operators:

```
# let (+++) = (s, t) => s ++ " " ++ t;
let ( +++ ): (string, string) => string = <fun>;
# "hello" +++ "world";
- : string = "hello world"
```

By putting an operator in parentheses, you can also easily look up its type:

```
# (++);
- : (string, string) => string = <fun>
```

10.15.1 Rules for operators

There are two kinds of operators: infix operators (between two operands) and prefix operators (before single operands).

The following *operator characters* can be used for both kinds of operators:

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

Infix operators:

First character	Followed by operator characters
`= < > @ ^ | & + - * / $ %`	0+
`#`	1+

Additionally, the following keywords are infix operators:

```
* + - -. == != < > || && mod land lor lxor lsl lsr asr
```

Prefix operators:

First character	Followed by operator characters
`!`	0+
`? ~`	1+

Additionally, the following keywords are prefix operators:

```
- -.
```

Source of this section: Sect. "Prefix and infix symbols" in the OCaml Manual.

10.15.2 Precedences and associativities of operators

The following tables lists operators and their associativities. The higher up an operator, the higher its precedence is (the stronger it binds). For example, * has a higher precedence than +.

Construction or operator	Associativity
prefix operator	–
`. .( .[ .{`	–
`[ ]` (array index)	–
`#···`	–
applications, `assert`, `lazy`	left
`- -.` (prefix)	–
`**··· lsl lsr asr`	right
`*··· /··· %··· mod land lor lxor`	left
`+··· -···`	left
`@··· ^···`	right
`=··· <··· >··· |··· &··· $··· !=`	left
`&&`	right
`||`	right
`if`	–
`let switch fun try`	–

Legend:

- `op···` means `op` followed by other operator characters.

- Applications: function application, constructor application, tag application

Source of this table: Sect. "Expressions" in the OCaml manual

10.15.3 When does associativity matter?

Associativity matters whenever an operator is not *commutative*. With a commutative operator, the order of the operands does not matter. For example, plus (+) is commutative. However, minus (-) is not commutative.

Left associativity means that operations are grouped from the left. Then the following two expressions are equivalent:

```
x op y op z
(x op y) op z
```

Minus (-) is left-associative:

```
# 3 - 2 - 1;
- : int = 0
```

Right associativity means that operations are grouped from the right. Then the following two expressions are equivalent:

```
x op y op z
x op (y op z)
```

We can define our own right-associative minus operator. According to the operator table, if it starts with an `@` symbol, it is automatically right-associative:

```
let (@-) = (x, y) => x - y;
```

If we use it, we get a different result than normal minus:

```
# 3 @- 2 @- 1;
- : int = 2
```

10.16 Polymorphic functions

Recall the definition of *polymorphism*: making the same operation work for several types. There are multiple ways in which polymorphism can be achieved. OOP languages achieve it via *subclassing*. *Overloading* is another popular kind of polymorphism.

ReasonML supports *parametric polymorphism*: so-called *type variables* indicate that any type can be filled in. (Such variables are universally quantified.) A function that uses type variables is called a *generic function*.

10.16.1 Example: `id()`

For example, `id` is the *identity function* that simply returns its parameter:

```
# let id = x => x;
let id: ('a) => 'a = <fun>;
```

The type for `id` that ReasonML infers is interesting: It can't detect a type for `x`, so it uses the type variable `'a` to indicate "any type". Type variables always start with a straight apostrophe. ReasonML also infers that the return type of `id` is the same as the type of its parameter. That is useful information.

`id` is generic and works with any type:

```
# id(123);
- : int = 123
# id("abc");
- : string = "abc"
```

10.16.2 Example: `first()`

Let's look another example: a generic function `first` for accessing the first component of a pair (a 2-tuple).

```
# let first = ((x, y)) => x;
let first: (('a, 'b)) => 'a = <fun>;
```

`first` uses destructuring to access the first component of that tuple. Type inference tells us that the return type is the same as the type of the first component.

We can use an underscore to indicate that we are not interested in the second component:

```
# let first = ((x, _)) => x;
let first: (('a, 'b)) => 'a = <fun>;
```

With a type-annotated component, `first` looks as follows:

```
# let first = ((x: 'a, _)) => x;
let first: (('a, 'b)) => 'a = <fun>;
```

10.16.3 Example: `ListLabels.map()`

As a quick preview, I'm showing the signature of another function that I explain properly in the chapter on lists.

```
ListLabels.map: (~f: ('a) => 'b, list('a)) => list('b)
```

10.16.4 Overloading vs. parametric polymorphism

Note how overloading and parametric polymorphism are different:

- Overloading provides different implementations for the same operation. For example, some programming languages let you use + for arithmetic, string concatenation and/or array concatenation.
- Parametric polymorphism specifies a single algorithm that works with several types.

10.17 ReasonML does not support variadic functions

ReasonML does not support variadic functions (varargs). That is, you can't define a function that computes the sum of an arbitrary number of parameters:

```
let sum = (x0, ···, xn) => x0 + ··· + xn;
```

Instead, you are forced to define one function for each arity:

```
let sum2(a: int, b: int) = a + b;
let sum3(a: int, b: int, c: int) = a + b + c;
let sum4(a: int, b: int, c: int, d: int) = a + b + c + d;
```

You have seen a similar technique with currying, where we couldn't define a variadic function `curry()` and had to go with a binary `curry2()`, instead. You'll occasionally see it in libraries, too.

An alternative to this technique is to use lists of ints.

Chapter 11

Basic modules

In this chapter, we explore how modules work in ReasonML.

11.1 Installing the demo repository

The demo repository for this chapter is available on GitHub: `reasonml-demo-modules`. To install it, download it and:

```
cd reasonml-demo-modules/
npm install
```

That's all you need to do – no global installs necessary.

11.2 Your first ReasonML program

This is where your first ReasonML program is located:

```
reasonml-demo-modules/
    src/
        HelloWorld.re
```

In ReasonML, each file whose name has the extension is `.re` is a module. The names of modules start with capital letters and are camel-cased. File names define the names of their modules, so they follow the same rules.

Programs are just modules that you run from a command line.

`HelloWorld.re` looks as follows:

```
/* HelloWorld.re */

let () = {
  print_string("Hello world!");
  print_newline()
};
```

This code may look a bit weird, so let me explain: We are executing the two lines inside the curly braces and assigning their result to the pattern `()`. That is, no new variables are created, but the pattern ensures that the result is `()`. The type of `()`, `unit`, is similar to `void` in C-style languages.

Note that we are not defining a function, we are immediately executing `print_string()` and `print_newline()`.

To compile this code, you have two options (look at `package.json` for more scripts to run):

- Compile everything, once: `npm run build`
- Watch all files and incrementally compile only files that change: `npm run watch`

Therefore, our next step is (run in a separate terminal window or execute the last step in the background):

```
cd reasonml-demo-modules/
npm run watch
```

Sitting next to `HelloWorld.re`, there is now a file `HelloWorld.bs.js`. You can run this file as follows.

```
cd reasonml-demo-modules/
node src/HelloWorld.bs.js
```

11.2.1 Other versions of `HelloWorld.re`

As an alternative to our approach (which is a common OCaml convention), we could have also simply put the two lines into the global scope:

```
/* HelloWorld.re */

print_string("Hello world!");
print_newline();
```

And we could have defined a function `main()` that we then call:

```
/* HelloWorld.re */

let main = () => {
  print_string("Hello world!");
  print_newline()
};
main();
```

11.3 Two simple modules

Let's continue with a module `MathTools.re` that is used by another module, `Main.re`:

```
reasonml-demo-modules/
    src/
```

```
Main.re
MathTools.re
```

Module `MathTools` looks like this:

```
/* MathTools.re */

let times = (x, y) => x * y;
let square = (x) => times(x, x);
```

Module `Main` looks like this:

```
/* Main.re */

let () = {
  print_string("Result: ");
  print_int(MathTools.square(3));
  print_newline()
};
```

As you can see, in ReasonML, you can use modules by simply mentioning their names. They are found anywhere within the current project.

11.3.1 Submodules

You can also nest modules. So this works, too:

```
/* Main.re */

module MathTools = {
  let times = (x, y) => x * y;
  let square = (x) => times(x, x);
};

let () = {
  print_string("Result: ");
  print_int(MathTools.square(3));
  print_newline()
};
```

Externally, you can access `MathTools` via `Main.MathTools`.

Let's nest further:

```
/* Main.re */

module Math = {
  module Tools = {
    let times = (x, y) => x * y;
    let square = (x) => times(x, x);
  };
};
```

```
let () = {
  print_string("Result: ");
  print_int(Math.Tools.square(3));
  print_newline()
};
```

11.4 Controlling how values are exported from modules

By default, every module, type and value of a module is exported. If you want to hide some of these exports, you must use *interfaces*. Additionally, interfaces support *abstract types* (whose internals are hidden).

11.4.1 Interface files

You can control how much you export via so-called *interfaces*. For a module defined by a file `Foo.re`, you put the interface in a file `Foo.rei`. For example:

```
/* MathTools.rei */

let times: (int, int) => int;
let square: (int) => int;
```

If, e.g., you omit `times` from the interface file, it won't be exported.

The interface of a module is also called its *signature*.

If an interface file exists, then docblock comments must be put there. Otherwise, you put them into the `.re` file.

Thankfully, we don't have to write interfaces by hand, we can generate them from modules. How is described in the BuckleScript documentation. For `MathTools.rei`, I did it via:

```
bsc -bs-re-out lib/bs/src/MathTools-ReasonmlDemoModules.cmi
```

11.4.2 Defining interfaces for submodules

Let's assume, `MathTools` doesn't reside in its own file, but exists as a submodule:

```
module MathTools = {
  let times = (x, y) => x * y;
  let square = (x) => times(x, x);
};
```

How do we define an interface for this module? We have two options.

First, we can define and name an interface via `module type`:

```
module type MathToolsInterface = {
  let times: (int, int) => int;
  let square: (int) => int;
};
```

That interface becomes the type of module `MathTools`:

```
module MathTools: MathToolsInterface = {
  ...
};
```

Second, we can also inline the interface:

```
module MathTools: {
  let times: (int, int) => int;
  let square: (int) => int;
} = {
  ...
};
```

11.4.3 Abstract types: hiding internals

You can use interfaces to hide the details of types. Let's start with a module `Log.re` that lets you put strings "into" logs. It implements logs via strings and completely exposes this implementation detail by using strings directly:

```
/* Log.re */

let make = () => "";
let logStr = (str: string, log: string) => log ++ str ++ "\n";

let print = (log: string) => print_string(log);
```

From this code, it isn't clear that `make()` and `logStr()` actually return logs.

This is how you use `Log`. Note how convenient the pipe operator (`|>`) is in this case:

```
/* LogMain.re */

let () = Log.make()
  |> Log.logStr("Hello")
  |> Log.logStr("everyone")
  |> Log.print;

/* Output:
Hello
everyone
*/
```

The first step in improving `Log` is by introducing a type for logs. The convention, borrowed from OCaml, is to use the name `t` for the main type supported by a module. For example: `Bytes.t`

```
/* Log.re */

type t = string; /* A */

let make = (): t => "";
```

```
let logStr = (str: string, log: t): t => log ++ str ++ "\n";

let print = (log: t) => print_string(log);
```

In line A we have defined `t` to be simply an alias for strings. Aliases are convenient in that you can start simple and add more features later. However, the alias forces us to annotate the results of `make()` and `logStr()` (which would otherwise have the return type `string`).

The full interface file looks as follows.

```
/* Log.rei */

type t = string; /* A */
let make: (unit) => t;
let logStr: (string, t) => t;
let print: (t) => unit;
```

We can replace line A with the following code and `t` becomes *abstract* – its details are hidden. That means that we can easily change our minds in the future and, e.g., implement it via an array.

```
type t;
```

Conveniently, we don't have to change `LogMain.re`, it still works with the new module.

11.5 Importing values from modules

There are several ways in which you can import values from modules.

11.5.1 Importing via qualified names

We have already seen that you can automatically import a value exported by a module if you qualify the value's name with the module's name. For example, in the following code we import `make`, `logStr` and `print` from module `Log`:

```
let () = Log.make()
  |> Log.logStr("Hello")
  |> Log.logStr("everyone")
  |> Log.print;
```

11.5.2 Opening modules globally

You can omit the qualifier "`Log.`" if you open `Log` "globally" (within the scope of the current module):

```
open Log;

let () = make()
  |> logStr("Hello")
  |> logStr("everyone")
  |> print;
```

To avoid name clashes, this operation is not used very often. Most modules, such as `List`, are used via qualifications: `List.length()`, `List.map()`, etc.

Global opening can also be used to opt into different implementations for standard modules. For example, module `Foo` might have a submodule `List`. Then `open Foo;` will override the standard `List` module.

11.5.3 Opening modules locally

We can minimize the risk of name clashes, while still getting the convenience of an open module, by opening `Log` locally. We do that by prefixing a parenthesized expression with `Log.` (i.e., we are qualifying that expression). For example:

```
let () = Log.(
  make()
    |> logStr("Hello")
    |> logStr("everyone")
    |> print
);
```

11.5.3.1 Redefining operators

Conveniently, operators are also just functions in ReasonML. That enables us to temporarily override built-in operators. For example, we may not like having to use operators with dots for floating point math:

```
let dist = (x, y) =>
  sqrt((x *. x) +. (y *. y));
```

Then we can override the nicer `int` operators via a module `FloatOps`:

```
module FloatOps = {
  let (+) = (+.);
  let (*) = (*.);
};
let dist = (x, y) =>
  FloatOps.(
    sqrt((x * x) + (y * y))
  );
```

Whether or not you actually should do this in production code is debatable.

11.5.4 Including modules

Another way of importing a module is to *include* it. Then all of its exports are added to the exports of the current module. This is similar to inheritance between classes in object-oriented programming.

In the following example, module `LogWithDate` is an extension of module `Log`. It has the new function `logStrWithDate()`, in addition to all functions of `Log`.

```
/* LogWithDateMain.re */
```

```
module LogWithDate = {
  include Log;
  let logStrWithDate = (str: string, log: t) => {
    let dateStr = Js.Date.toISOString(Js.Date.make());
    logStr("[" ++ dateStr ++ "] " ++ str, log);
  };
};
let () = LogWithDate.(
  make()
    |> logStrWithDate("Hello")
    |> logStrWithDate("everyone")
    |> print
);
```

`Js.Date` comes from BuckleScript's standard library and is not explained here.

You can include as many modules as you want, not just one.

11.5.5 Including interfaces

Interfaces are included as follows (`InterfaceB` extends `InterfaceA`):

```
module type InterfaceA = {
  ...
};
module type InterfaceB = {
  include InterfaceA;
  ...
}
```

Similarly to modules, you can include more than one interface.

Let's create an interface for module `LogWithDate`. Alas, we can't include the interface of module `Log` by name, because it doesn't have one. We can, however, refer to it indirectly, via its module (line A):

```
module type LogWithDateInterface = {
  include (module type of Log); /* A */
  let logStrWithDate: (t, t) => t;
};
module LogWithDate: LogWithDateInterface = {
  include Log;
  ...
};
```

11.5.6 Renaming imports

You can't really rename imports, but you can alias them.

This is how you alias modules:

```
module L = List;
```

This is how you alias values inside modules:

```
let m = List.map;
```

11.6 Namespacing modules

In large projects, ReasonML's way of identifying modules can become problematic. Since it has a single global module namespace, there can easily be name clashes. Say, two modules called `Util` in different directories.

One technique is to use *namespace modules*. Take, for example, the following project:

```
proj/
    foo/
        NamespaceA.re
        NamespaceA_Misc.re
        NamespaceA_Util.re
    bar/
        baz/
            NamespaceB.re
            NamespaceB_Extra.re
            NamespaceB_Tools.re
            NamespaceB_Util.re
```

There are two modules `Util` in this project whose names are only distinct because they were prefixed with `NamespaceA_` and `NamespaceB_`, respectively:

```
proj/foo/NamespaceA_Util.re
proj/bar/baz/NamespaceB_Util.re
```

To make naming less unwieldy, there is one *namespace module* per namespace. The first one looks like this:

```
/* NamespaceA.re */
module Misc = NamespaceA_Misc;
module Util = NamespaceA_Util;
```

`NamespaceA` is used as follows:

```
/* Program.re */

open NamespaceA;

let x = Util.func();
```

The global open lets us use `Util` without a prefix.

There are two more use cases for this technique:

- You can override modules with it, even modules from the standard library. For example, `NamespaceA.re` could contain a custom `List` implementation, which would override the built-in `List` module inside `Program.re`:

  ```
  module List = NamespaceA_List;
  ```

- You can create nested modules while keeping submodules in separate files. For example, in addition to opening `NamespaceA`, you can also access `Util` via `NamespaceA.Util`, because it is nested inside `NamespaceA`. Of course, `NamespaceA_Util` works, too, but is discouraged, because it is an implementation detail.

The latter technique is used by BuckleScript for `Js.Date`, `Js.Promise`, etc., in file `js.ml` (which is in OCaml syntax):

```
...
module Date = Js_date
...
module Promise = Js_promise
...
module Console = Js_console
```

11.6.1 Namespace modules in OCaml

Namespace modules are used extensively in OCaml at Jane Street. They call them *packed modules*, but I prefer the name *namespace modules*, because it doesn't clash with the npm term *package*.

Source of this section: "Better namespaces through module aliases" by Yaron Minsky for Jane Street Tech Blog.

11.7 Exploring the standard library

There are two big caveats attached to ReasonML's standard library:

- It is currently work in progress.
- Its naming style for values inside modules will change from snake case (`foo_bar` and `Foo_bar`) to camel case (`fooBar` and `FooBar`).
- At the moment, much functionality is still missing.

11.7.1 API docs

ReasonML's standard library is split: most of the core ReasonML API works on both native and JavaScript (via BuckleScript). If you compile to JavaScript, you need to use BuckleScript's API in two cases:

- Functionality that is completely missing from ReasonML's API. Examples include support for dates, which you get via BuckleScript's `Js.Date`.
- ReasonML API functionality that is not supported by BuckleScript. Examples include modules `Str` (due to JavaScript's strings being different from ReasonML's native ones) and `Unix` (with native APIs).

This is the documentation for the two APIs:

- ReasonML API docs
- BuckleScript API docs

11.7.2 Module `Pervasives`

Module `Pervasives` contains the core standard library and is always automatically opened for each module. It contains functionality such as the operators `==`, `+`, `|>` and functions such as `print_string()` and `string_of_int()`.

If something in this module is ever overridden, you can still access it explicitly via, e.g., `Pervasives.(+)`.

If there is a file `Pervasives.re` in your project, it overrides the built-in module and is opened instead.

11.7.3 Standard functions with labeled parameters

The following modules exist in two versions: an older one, where functions have only positional parameters and a newer one, where functions also have labeled parameters.

- `Array`, `ArrayLabels`
- `Bytes`, `BytesLabels`
- `List`, `ListLabels`
- `String`, `StringLabels`

As an example, consider:

```
List.map: ('a => 'b, list('a)) => list('b)
ListLabels.map: (~f: 'a => 'b, list('a)) => list('b)
```

Two more modules provide labeled functions:

- Module `StdLabels` has the submodules `Array`, `Bytes`, `List`, `String`, which are aliases to `ArrayLabels` etc. In your modules, you can open `StdLabels` to get a labeled version of `List` by default.
- Module `MoreLabels` has three submodules with labeled functions: `Hashtbl`, `Map` and `Set`.

11.8 Installing libraries

For now, JavaScript is the preferred platform for ReasonML. Therefore, the preferred way of installing libraries is via npm. This works as follows. As an example, assume we want to install the BuckleScript bindings for Jest (which include Jest itself). The relevant npm package is called `bs-jest`.

First, we need to install the package. Inside `package.json`, you have:

```
{
  "dependencies": {
    "bs-jest": "^0.1.5"
  },
  ...
}
```

Second, we need to add the package to `bsconfig.json`:

```
{
  "bs-dependencies": [
    "bs-jest"
  ],
  ...
}
```

Afterwards, we can use module `Jest` with `Jest.describe()` etc.

More information on installing libraries:

- BuckleScript's build system is explained in Chap. "Build system support" of the BuckleScript Manual.
- ReasonML's docs explain how to find ReasonML libraries on npm.
  - Useful npm keywords include: `reason`, `reasonml`, `bucklescript`

Chapter 12

Variant types

Variant types (short: *variants*) are a data type supported by many functional programming languages. They are an important ingredient in ReasonML that is not available in C-style languages (C, C++, Java, C#, etc.). This chapter explains how they work.

12.1 Variants as sets of symbols (enums)

Variants let you define sets of symbols. When used like this, they are similar to enums in C-style languages. For example, the following type `color` defines symbols for six colors.

```
type color = Red | Orange | Yellow | Green | Blue | Purple;
```

There are two elements in this type definition:

- The name of the type, `color`, which must start with a lowercase letter.
- The names of *constructors* (`Red`, `Orange`, …), which must start with uppercase letters. Why constructors are called constructors will become clear, once we use variants as data structures.

The names of constructors must be unique within the current scope. That enables ReasonML to easily deduce their types:

```
# Purple;
- : color = Purple
```

Variants can be processed via `switch` and pattern matching:

```
let invert = (c: color) =>
  switch c {
  | Red => Green
  | Orange => Blue
  | Yellow => Purple
  | Green => Red
  | Blue => Orange
```

```
  | Purple => Yellow
  };
```

Here, constructors are used both as patterns (left-hand sides of `=>`) and values (right-hand sides of `=>`). This is `invert()` in action:

```
# invert(Red);
- : color = Green
# invert(Yellow);
- : color = Purple
```

12.1.1 Tip: replacing booleans with variants

In ReasonML, variants are often a better choice than booleans. Take for example, this function definition. (Remember that in ReasonML, the main parameter goes at the end, to enable currying.)

```
let stringOfContact = (includeDetails: bool, c: contact) => ···;
```

This is how `stringOfContact` is invoked:

```
let str = stringOfContact(true, myContact);
```

It's not clear what the boolean at the end does. You can improve this function via a labeled parameter.

```
let stringOfContact = (~includeDetails: bool, c: contact) => ···;
let str = stringOfContact(~includeDetails=true, myContact);
```

Even more self-descriptive is to introduce a variant for the value of `~includeDetails`:

```
type includeDetails = ShowEverything | HideDetails;
let stringOfContact = (~levelOfDetail: includeDetails, c: contact) => ···;
let str = stringOfContact(~levelOfDetail=ShowEverything, myContact);
```

Using the variant `includeDetails` has two advantages:

- It is immediately clear what "not showing details" means.
- It is easy to add more modes later on.

12.1.2 Associating variant values with data

Sometimes, you want to use variant values as keys for looking up data. One way of doing so is via a function that maps variant values to data:

```
type color = Red | Orange | Yellow | Green | Blue | Purple;
let stringOfColor = (c: color) =>
  switch c {
  | Red => "Red"
  | Orange => "Orange"
  | Yellow => "Yellow"
  | Green => "Green"
  | Blue => "Blue"
```

```
  | Purple => "Purple"
  };
```

12.2 Variants as data structures

Each constructor can also hold one or more values. These values are identified by position. That is, individual constructors are similar to tuples. The following code demonstrates this feature.

```
type point = Point(float, float);
type shape =
  | Rectangle(point, point)
  | Circle(point, float);
```

Type `point` is a variant with a single constructor. It holds two floating point numbers. A `shape` is another variant. It is either:

- a `Rectangle` defined by two corner points or
- a `Circle` defined by a center and a radius.

With multiple constructor parameters, them being positional and not labeled becomes a problem – we have to describe elsewhere what their roles are. *Records* are an alternative in this case (they are described in their own chapter).

This is how you use the constructors:

```
# let bottomLeft = Point(-1.0, -2.0);
let bottomLeft: point = Point(-1., -2.);
# let topRight = Point(7.0, 6.0);
let topRight: point = Point(7., 6.);
# let circ = Circle(topRight, 5.0);
let circ: shape = Circle(Point(7., 6.), 5.);
# let rect = Rectangle(bottomLeft, topRight);
let rect: shape = Rectangle(Point(-1., -2.), Point(7., 6.));
```

Due to each constructor name being unique, ReasonML can easily infer the types.

If constructors hold data, pattern matching via `switch` is even more convenient, because it also lets you access that data:

```
let pi = 4.0 *. atan(1.0);

let computeArea = (s: shape) =>
  switch s {
  | Rectangle(Point(x1, y1), Point(x2, y2)) =>
    let width = abs_float(x2 -. x1);
    let height = abs_float(y2 -. y1);
    width *. height;
  | Circle(_, radius) => pi *. (radius ** 2.0)
  };
```

Let's use `computeArea`, continuing our previous interactive `rtop` session:

```
# computeArea(circ);
- : float = 78.5398163397448315
# computeArea(rect);
- : float = 64.
```

12.3 Self-recursive data structures via variants

You can also define recursive data structures via variants. For example, binary trees whose nodes contain integers:

```
type intTree =
  | Empty
  | Node(int, intTree, intTree);
```

`intTree` values are constructed like this:

```
let myIntTree = Node(1,
  Node(2, Empty, Empty),
  Node(3,
    Node(4, Empty, Empty),
    Empty
  )
);
```

`myIntTree` looks as follows: 1 has the two child nodes 2 and 3. 2 has two empty child nodes. Etc.

```
1
  2
    X
    X
  3
    4
      X
      X
    X
```

12.3.1 Processing self-recursive data structures via recursion

To demonstrate processing self-recursive data structures, let's implement a function `computeSum`, which computes the sum of the integers stored in the nodes.

```
let rec computeSum = (t: intTree) =>
  switch t {
  | Empty => 0
  | Node(i, leftTree, rightTree) =>
    i + computeSum(leftTree) + computeSum(rightTree)
  };

computeSum(myIntTree); /* 10 */
```

This kind of recursion is typical when working with variant types:

1. A limited set of constructors is used to create data. In this case: `Empty` and `Node()`.
2. The same constructors are used as patterns to process the data.

That ensures that we handle whatever data is passed to us properly, as long as it is of type `intTree`. ReasonML helps by warning us if `switch` doesn't cover `intTree` exhaustively. That protects us from forgetting cases that we should consider. To illustrate, let's assume we forgot `Empty` and wrote `computeSum` like this:

```
let rec computeSum = (t: intTree) =>
  switch t {
  /* Missing: Empty */
  | Node(i, leftTree, rightTree) =>
    i + computeSum(leftTree) + computeSum(rightTree)
  };
```

Then we get the following warning.

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Empty
```

As mentioned in the chapter on functions, introducing catch-all cases means that you lose this protection. That's why you should avoid them if you can.

12.4 Mutually recursive data structures via variants

Recall that with `let`, we had to use `let rec` whenever recursion was involved:

- A single self-recursive definition was done via `let rec`.
- Multiple mutually recursive definitions were done via `let rec` and connected via `and`.

`type` is implicitly `rec`. That allowed us to do self-recursive definitions such as `intTree`. For mutually recursive definitions, we also need to connect those definitions via `and`. The following example again defines int trees, but this time with a separate type for nodes.

```
type intTree =
  | Empty
  | IntTreeNode(intNode)
and intNode =
  | IntNode(int, intTree, intTree);
```

`intTree` and `intNode` are mutually recursive, which is why they need to be defined within the same `type` declaration, separated via `and`.

12.5 Parameterized variants

Let's recall our original definition of int trees:

```
type intTree =
  | Empty
  | Node(int, intTree, intTree);
```

How can we turn this definition into a generic definition for trees whose nodes can contain any type of value? To do so, we have to introduce a variable for the type of a `Node`'s content. *Type variables* are prefixed with apostrophes in ReasonML. For example: `'a`. Therefore, a generic tree looks as follows:

```
type tree('a) =
  | Empty
  | Node('a, tree('a), tree('a));
```

Two things are noteworthy. First, the content of a Node, which previously had the type `int`, now has the type `'a`. Second, the type variable `'a` has become a parameter of the type `tree`. `Node` passes that parameter on to its subtrees. That is, we can choose a different node value type for each tree, but within a tree, all node values must have the same type.

We can now define a type for int trees via a type alias, by providing `tree`'s type parameter:

```
type intTree = tree(int);
```

Let's use `tree` to create a tree of strings:

```
let myStrTree = Node("a",
  Node("b", Empty, Empty),
  Node("c",
    Node("d", Empty, Empty),
    Empty
  )
);
```

Due to type inference, you do not need to provide a type parameter. ReasonML automatically infers that `myStrTree` has the type `tree(string)`. The following generic function prints any kind of tree:

```
/**
 * @param ~indent How much to indent the current (sub)tree.
 * @param ~stringOfValue Converts node values to strings.
 * @param t The tree to convert to a string.
 */
let rec stringOfTree = (~indent=0, ~stringOfValue: 'a => string, t: tree('a)) => {
  let indentStr = String.make(indent*2, ' ');
  switch t {
  | Empty => indentStr ++ "X" ++ "\n"
  | Node(x, leftTree, rightTree) =>
    indentStr ++ stringOfValue(x) ++ "\n" ++
    stringOfTree(~indent=indent+1, ~stringOfValue, leftTree) ++
    stringOfTree(~indent=indent+1, ~stringOfValue, rightTree)
  };
};
```

This function uses recursion to iterate over the nodes of its parameter `t`. Given that `stringOfTree` works with arbitrary types `'a`, we need a type-specific function to convert values of type `'a` to strings. That is what parameter `~stringOfValue` is for.

This is how we can print our previously defined `myStrTree`:

```
# print_string(stringOfTree(~stringOfValue=x=>x, myStrTree));
a
  b
    X
    X
  c
    d
      X
      X
    X
```

12.6 Useful standard variants

I will briefly show two commonly used standard variants.

12.6.1 Type `option('a)` for optional values

In many object-oriented languages, a variable having type `string` means that the variable can be either `null` or a string value. Types that include `null` are called *nullable*. Nullable types are problematic in that it's easy to work with their values while forgetting to handle `null`. If – unexpectedly – a `null` appears, you get the infamous `null` pointer exceptions.

In ReasonML, types are never nullable. Instead, potentially missing values are handled via the following parameterized variant:

```
type option('a) =
  | None
  | Some('a);
```

`option` forces you to always consider the `None` case.

ReasonML's support for `option` is minimal. The definition of this variant is part of the language, but the core standard library has no utility functions for working with optional values, yet. Until they are, you can use BuckleScript's `Js.Option`.

12.6.2 Type `result('a)` for error handling

`result` is another standard variant for error-handling in OCaml:

```
type result('good, 'bad) =
  | Ok('good)
  | Error('bad);
```

Until ReasonML's core library supports it, you can use BuckleScript's `Js.Result`.

12.6.3 Example: evaluating integer expressions

Working with trees is one of the strengths of ML-style languages. That's why they are often used for programs involving syntax trees (interpreters, compilers, etc.). For example, the syntax checker Flow by Facebook is written in OCaml.

Therefore, as a concluding example, let's implement an evaluator for simple integer expressions.

The following is a data structure for integer expressions.

```
type expression =
  | Plus(expression, expression)
  | Minus(expression, expression)
  | Times(expression, expression)
  | DividedBy(expression, expression)
  | Literal(int);
```

This is what an expression encoded with this variant looks like:

```
/* (3 - (16 / (6 + 2)) */
let expr =
  Minus(
    Literal(3),
    DividedBy(
      Literal(16),
      Plus(
        Literal(6),
        Literal(2)
      )
    )
  );
```

And finally, this is the function the evaluates integer expressions.

```
let rec eval(e: expression) =
  switch e {
  | Plus(e1, e2) => eval(e1) + eval(e2)
  | Minus(e1, e2) => eval(e1) - eval(e2)
  | Times(e1, e2) => eval(e1) * eval(e2)
  | DividedBy(e1, e2) => eval(e1) / eval(e2)
  | Literal(i) => i
  };

eval(expr); /* 1 */
```

Chapter 13

Where are the remaining chapters?

You are reading a preview of this book:

- The full version of this book is available for purchase.
- You can take a look at the full table of contents (also linked to from the book's homepage).